Issues and Experiences in Implementing a Distributed Tuplespace

James B. Fenwick Jr. *Lori L. Pollock*

Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716
(302) 831-1953 fax: (302) 831-8458

{fenwick,pollock}@cis.udel.edu

Abstract

Distributed memory multiprocessors and network clusters are being used increasingly as parallel computing resources due to their scalability and cost/performance advantages. However, it is generally believed that shared memory parallel programming is easier than explicit message passing programming. Although the generative communication model provides scalability like message passing and the simplicity of shared memory programming, it is a challenge to effectively implement this model on machines with physically distributed memories. This paper describes the issues involved in implementing the essential component of generative communication, the shared data space abstraction called tuplespace, on a distributed memory machine. The paper gives a detailed description of Deli, a UNIX-based distributed tuplespace implementation for a network of workstations. This description, along with discussions of implementation alternatives, provides a detailed basis for designers and implementors of shared data spaces, not currently available in the literature.

Keywords: *distributed memory parallel architecture, generative communication model, distributed tuplespace*

INTRODUCTION

Because organizations have begun to realize that their LAN-connected workstations and PCs constitute a distributed memory parallel machine, systems providing efficient, yet easy, access to the power of this parallel computing resource are becoming increasingly important. Moreover, distributed memory parallel architectures offer better scalability than shared memory machines. However, it is generally agreed that shared memory parallel programming is easier than explicit message passing programming.

The *generative communication*[23] paradigm of parallel programming offers the simplicity of shared memory programming and only a small number of primitives for coordination, while also providing flexibility, power, and the potential to scale like message passing. Rather than sharing variables, processes share a data space. Messages are not sent from one process to another, but rather are placed in the shared data space for other processes to access. The data placed into the shared space are termed *tuples* and thus the shared space is called *tuplespace*. The actual implementation of the parallel program on the target architecture is hidden from the programmer, and the architecture can be any number of platforms ranging from shared or distributed memory to networks of workstations. The two distinguishing characteristics of tuplespace that give rise to its power and flexibility are communication uncoupling and the associative nature of the logically shared tuplespace.

Implementing the shared tuplespace on a distributed memory architecture has raised concerns regarding efficiency and performance [17]. While researchers have demonstrated that distributed tuplespace implementations can be efficient for a wide variety of "real" applications encompassing a large scope of parallel algorithm classifications [19, 34, 11], there remains opportunity for improvement through compiler analysis targeting the underlying message passing [9, 21, 29, 20]. Such research requires a tuplespace implementation that allows experimentation and modification. In order to gain this flexibility in experimentation, an implementation of distributed tuplespace for a network of workstations has been developed. The experiences of this implementation effort have found existing distributed tuplespace implementations [6, 12, 16, 31, 44, 49] to be incomplete in their descriptions of the implementation issues faced and the potential solutions and pitfalls in addressing these issues. This paper focuses on filling this gap, by sharing our experiences in designing and implementing a distributed tuplespace, focusing on the design issues faced, the potential implementation alternatives investigated, and the reasons behind our implementation choices. This paper does not intend to introduce an improved tuplespace design, but rather seeks to provide sufficient detail of issues and an actual implementation to enable a software developer to quickly implement a tuplespace.

Our implementation of the generative communication model is an implementation of Linda tuplespace, the best-known implementation of the generative communication model [23, 10]. We have built a Linda optimizing compiler based on the SUIF compiler infrastructure[45], and a distributed tuplespace runtime system. This paper focuses more directly on the distributed tuplespace runtime system, called Deli (University of **De**laware **Li**nda). More details on the Linda optimizing compiler can be found in [20, 21, 22].

The following sections provide background for the tuplespace model, discuss a number of implementation issues and the solutions taken by other researchers, the details of our own implementation, and a comparison of prominent implementations.

LINDA TUPLESPACE

Linda is a coordination language consisting of a small number of primitive operations which are added into existing sequential languages [23, 10]. These operations perform the communication and synchronization necessary for parallel programming. The following subsections describe the basic Linda model, and an improvement that addresses the efficiency of the associative tuplespace memory.

Basic Linda Model

Communication between processes is achieved through tuples in an associative, global memory known as tuplespace. A tuple is an ordered collection of typed fields which are either data elements or placeholders. For example,

```
float f;
("tuple", 3, f)
```

is a tuple with three fields. The first field contains the constant string "tuple". The second field is the constant integer value 3, while the third field contains the value of the floating point variable f. The field types of tuples are dependent on the underlying sequential language. The six Linda coordination operations (OUT, EVAL, IN, RD, INP, RDP) manipulate tuplespace.

The OUT is a non-blocking operation which asynchronously inserts a tuple into tuplespace. The IN operation functions as a blocking operation which synchronously extracts a tuple from tuplespace. Usually the IN operation has one or more fields which act as placeholders. The values of those fields from a tuple in tuplespace are copied into the placeholders of the IN operation. This is how data is passed between processes. The RD operation is another synchronous receive which functions like the IN, only it does not remove the tuple from tuplespace. The INP and RDP operations are predicate versions of their counterparts. That is, they do not block if no matching tuple is present in tuplespace but rather return a false value.

The placeholders of IN, RD, INP, and RDP operations are termed *formal* fields because they receive the values of the corresponding field of a matching tuple, much as formal procedure arguments receive values from the actual arguments of a call site. Prefixing a field with a question mark (?) syntactically denotes a formal field. A field that is not formal is termed an *actual* field.

Lastly, the EVAL operation creates an *active* tuple in tuplespace. This simply means that some field(s) of the tuple is currently being evaluated and the tuple is unavailable for matching until this field has completed evaluation. At that time, the tuple becomes a *passive* tuple, like those created by the OUT operation. New processes are created to evaluate the fields of the EVAL operation[1]. This is how the programmer explicitly creates parallelism.

This style of explicit parallelism should be differentiated from other types of parallelism. Functional, task, or DAG parallelism is the parallelism that can be exploited among different operations in a program by using data and control dependences [38] targeting a parallelism that is more coarsely grained than loop-level parallelism. A Linda compiler may apply other research [38, 25] to extract these finer grained kinds of parallelism present in individual, explicitly specified Linda processes.

Tuplespace is an associative memory, thus tuples are accessed not by an address but rather by their contents. Accessing a tuple therefore requires a description of the tuple desired. While OUT and EVAL operations create tuples, the IN, RD, INP, and RDP operations actually create *templates* (sometimes called *anti-tuples*) which are descriptions of desired tuples. For a template to *match* a tuple in tuplespace, the template and tuple must:

- agree on the number of fields,
- agree on the types of corresponding fields,
- agree on the values of corresponding actual fields,
- and have no corresponding formal fields.

Table 1 shows various combinations of tuples and templates and whether or not they match.

[1] In response to the high cost of process creation, many Linda implementations only create processes for the function-valued fields of an EVAL.

int i=3, x[3], y[2,3];
float f;

Tuple	Template	Match?	Comments
(”semaphore”)	(”semaphore”)	yes	values match
(2)	(3)	no	values do not match
(3)	(i)	yes	values and types match
(2)	(i)	no	types match, values do not
(2	(?i)	yes	types match (i←2)
(2, 5.6)	(?i, ?f)	yes	types match
(2, 5.6)	(?f, ?i)	no	types do not match in order
(1,2,3)	(?x)	yes	types and shapes match
(1,2,3,4)	(?x)	no	shapes do not match
(1,2,3)	(?y[0])	yes	types and shapes match

Table 1: Tuple Matching Examples

Tuplespace Partitioning Improvement

The earliest work on making Linda more efficient was directed at minimizing the amount of search required for the associative matching of tuplespace [12]. The idea is to partition tuplespace at compile time, and implement each partition with a data structure that makes searching the partition fast. While tuples and templates are runtime objects, the operations which produce tuples (OUT, EVAL) and templates (IN, INP, RD, RDP) are available at compile time. The process of partitioning is to group these operations into disjoint sets, such that no tuple produced by an operation of one set can match a template produced by an operation of any other set. Thus, only those tuples in the same partition as a template need to be searched for a match at runtime. The partitioning process involves matching Linda operations at compile time or preprocessor time, much as tuples and templates are matched at runtime. The salient difference between partitioning and runtime tuple matching is that the values of actual fields are not known before runtime. A tuple producing operation matches a template producing operation if they:

- agree on the number of fields,
- agree on the types of corresponding fields,
- agree on the values of corresponding *constants*,
- and have no corresponding formal fields.

It is important to realize that the compile time matching of operations assumes that corresponding actual fields match and that a corresponding constant and actual field match. Thus, two operations may match at compile time, but the runtime tuple and template may not match. Tuplespace partitioning is an architecture independent phase, a very important phase for efficient associative matching.

IMPLEMENTATION ISSUES

In this section, we describe the major issues that need to be addressed in order to implement tuplespace on a distributed memory parallel machine. In this environment, tuplespace provides a shared memory abstraction on top of a message-passing, distributed memory machine. Because tuplespace is a logically shared memory, it encounters implementation issues similar to other distributed shared memories (DSM) [33, 35, 5, 27]. Nitzberg and Lo [36] present a survey of the important issues of implementing DSMs and discuss how the developers of several DSMs faced these design choices. However, because of its special nature, a tuplespace implementation must address several additional issues [44]. Combining our own implementation experiences with the issues reported in the DSM literature, we establish the following list of major implementation issues for a distributed tuplespace each of which are discussed below.

- Structure and granularity of the shared data space
- Processor location of a tuple
- Data structures for efficient access to a tuple
- Tuplespace coherence protocol
- Tuple transfer protocol
- EVAL operator implementation
- Low-level Transport Protocol
- Heterogeneity
- Simultaneous Invocation
- System Extensions

Tuplespace Structure and Granularity

A key issue in implementing a shared distributed data space like tuplespace is how the data space is structured and what granularity is used for sharing among processors. Some DSMs structure their memory in the familiar form of a linear array of words. In contrast, tuplespace is structured as a collection of user-defined data objects. Thus, tuplespace is not addressable in the typical sense as an offset into a linear array of words and existing kernel code handling memory references cannot be utilized.

Granularity refers to the size of the unit of sharing. For some DSMs, this unit is the size of an operating system page [33], while others use a smaller block size [18]. One DSM uses a combination of a large and small unit of transfer [39]. For tuplespace, the unit of sharing is the tuple. However, because tuple sizes vary, the tuplespace implementor is not afforded the simplicity of predetermined fixed length transmissions. An interesting benefit to tuplespace's granularity is the elimination of the false sharing problem experienced by most DSMs. False sharing occurs between processes that do not share variables, but unknowingly share the page that holds the distinct variables. Severe performance degradation can result as the page is transmitted back and forth.

Processor Location of a Tuple

For tuplespace (or any DSM) to allow the sharing of data, a process must be able to locate and retrieve the data it requires. This issue is termed *tuplespace organization and distribution* in [44]. One choice, taken by Cline and Arthur [16], is to centralize tuplespace on a single node of the parallel machine. All tuples are stored on the one processor, and locating a tuple is simplified because all processes simply route the request to this node. While this approach does provide parallelism by allowing the computation to be divided across multiple nodes of the machine, there are several drawbacks. Data access is unnecessarily serialized reducing parallelism, a communication bottleneck appears if the single node is unable to keep pace with requests, and the available power of the parallel machine is being under-utilized.

An alternative is to distribute the task of tuple storage over some subset (possibly all) of the nodes of the system. However, locating a tuple becomes more difficult. There are two primary approaches to dealing with the distribution problem, each having several variations. They are classified as hash based tuple distribution and operator based tuple distribution.

Hash-based Tuple Distribution

This method deterministically maps tuples and templates to a specific node using a hash function. Two questions become immediately apparent: "Is the hash function effective? What is the hash function input?" The effectiveness of the hash function is measured by its ability to equalize the memory requirements and tuple search load on each node by evenly distributing tuples across nodes in the system. The input to the hash function can be a dedicated key field in every tuple and template (hence its partition identifier) [12], or a combination of the tuplespace partition identifier and the values of actual fields in the tuple/template [6]. Bjornson [6] provides a significant amount of information regarding the effectiveness of many hash functions. A potential disadvantage with a hash based scheme lies in the observation that the number of distinct partition identifiers for many applications is typically small. In this case, the scheme may not scale as the number of nodes used for storing the tuples of an application does not change even as the number of available computation nodes is increased. However, if the number of tuplespace partitions is greater than the number of available nodes, hashing is one of the best methods for efficient tuple distribution [44].

Operator-based Tuple Distribution

This method uses the Linda operator itself to indicate the destination of the tuple/template. One possibility is for tuples to be stored on the node executing the OUT or EVAL[2]. Templates, generated by the IN, RD, INP, and RDP operators, are sent to all nodes. Bjornson termed this scheme *negative broadcast*. This is the method utilized in the implementations of [31, 3]. Another possibility is *positive broadcast* which is the inverse of negative broadcast. Tuples are broadcast to all nodes, and the node executing the IN, RD, INP, and RDP operator locally handles the templates. Carriero's S-NET implementation [12] used this scheme. A hybrid

[2] Here, we are speaking of the resultant passive tuple generated by an EVAL.

of these two methods is to multicast tuples and templates to a subset of nodes. The intersection of these subsets must not be null (or else templates can not find matching tuples), but restricting the intersection to a single node is advantageous. This scheme is best visualized by considering a machine organized as a mesh of processors. Tuples are multicast to all processors in a machine's row, and templates are sent to all processors in the machine's column. This scheme was adopted by Krishnaswamy for his Linda machine [2]. Methods using broadcast (or multicast) require a type of coherence protocol. Consider the negative broadcast scheme, for example. A template is broadcast to all nodes and more than one node has a matching tuple. A protocol is required to ensure that only one of the matching tuples is removed from tuplespace.

Conflicting measures of success

One measure of the "goodness" of the distribution of tuples in tuplespace is how evenly the load of managing the tuples is distributed because managing tuples requires CPU time and memory on the local nodes. Another measure of tuple distribution "goodness" is the number of tuple requests that do not require network accesses because the tuple is found locally. Unfortunately, achieving one of these goals often conflicts with the achievement of the other. The hash-based distributions tend to succeed when measured by the first metric, but it is unclear how they fare with the second. The operator-based distributions, particularly negative broadcast, would seem to succeed as measured by the second metric because a process that deposits a tuple and later requests a tuple of the same type stands a reasonable chance of finding it locally. In fact, this form of locality of reference was precisely the reasoning for Leichter's defense of using a negative broadcast distribution [31].

Without any empirical evidence of such locality of reference patterns, it would appear that a hash-based decision on where to store tuples has a better chance of succeeding by the first metric than operator-based methods do by the second. However, with knowledge of the underlying distribution scheme, sophisticated compiler analysis estimating the tuplespace access patterns of processes seems possible. This information could be used to "suggest" which nodes should manage which partitions, or on which nodes processes should run, or a combination of these. Another possibility is a dynamically tunable distribution method. In fact, Bjornson [6] describes two runtime heuristics (described below as system extensions) that use actual tuplespace access patterns in an attempt to discover and correct "bad" hash-based distribution decisions. Having the compiler perform analysis aimed at "alerting" the runtime of potentially bad decisions may even be able to speed up the time to discover such scenarios.

Tuple Access

When the node containing a desired tuple has been identified, the tuple must be accessed in that processor's memory. This introduces questions regarding appropriate data structures for representing tuplespace, in particular, finding data structures for supporting the content addressability feature of tuplespace. Unfortunately, the implementation approaches regarding this issue are lacking in most of the relevant literature. There are several efficient data structure paradigms for tuple storage (hence, tuple access): trees, hash tables, queues, counters, and lists.

The tuplespace implementor can use a single data structure for all of tuplespace, or use different paradigms for different tuplespace partitions. Both hash tables and trees require a key in each tuple. For some implementations, this is a requirement for all tuples, while others use compiler information to identify specific tuplespace partitions that contain a key in each tuple. Trees provide ordering properties that appear unnecessary in a tuplespace context, and are typically not used. If the key consists of every actual field in the tuple, then access is $\mathcal{O}(1)$. However, most implementations use a subset of the tuple's actual fields for the key. In this case, different tuples may have the same key and thus reside in the same bucket necessitating a linear search of the bucket's chain. Note that this is not considered a collision because the key is identical.

The worst case for representing tuplespace is as a list. Every tuple in the list must be examined to determine no match exists in tuplespace. Partitioning tuplespace is an improvement as it divides a single list into smaller sublists, only one of which needs to be searched, but the potential for expensive tuple access remains. Sophisticated compilers may perform analysis revealing that a tuplespace partition never requires runtime matching; that is, any tuple may satisfy any template. In this case, the list can be viewed as a queue, and the access becomes constant. Compiler analysis can also determine that not only is no runtime matching required, but also there is no data copying necessary. In this situation, no data needs to be stored and a simple counter provides efficiency.

While using efficient data structures can allow a template to find a matching tuple in constant time for a pending template, the converse is not true. This is because a single tuple satisfies an arriving template, but an arriving tuple can satisfy multiple templates. Consider tuplespace containing several RD and IN templates when a matching tuple arrives. The tuple can satisfy at most one of the IN templates and a (possibly null) subset of the RD templates. This requires special attention by the tuplespace implementor.

Tuplespace Coherence Protocol

A form of memory coherence is required for all DSMs that replicate the shared data. For implementations of tuplespace that do not replicate tuples, memory coherence is trivial. However, implementations that select positive broadcast as the method of distributing tuples must also select a coherence protocol as tuples are duplicated on all nodes. The available protocols are essentially the same as those developed by the cache coherency research (for background one can start with [28, 13, 1, 15], and for more current research see [14, 30]). Bjornson [6] noted the advantages of replicating "read-only" tuples, thus implementations which provide this optimization must also decide on an appropriate coherence protocol.

Tuple Transfer Protocol

Tuple transfer is the movement of a tuple among nodes of the machine either to store the tuple or upon a tuple match or both. The tuplespace implementor must choose a tuple transfer protocol that ensures the validity of tuple movement while simultaneously attempting to minimize unnecessary movement. Tuple movement validity refers to providing the atomicity of the tuplespace operations. The selection of a tuple distribution scheme and the target communication architecture have effects on appropriate tuple transfer protocols. As an example of the issues faced by an implementor choosing a transfer protocol, two possible protocols for the negative broadcast tuple distribution method are considered. The protocols seem rather straightforward initially, but each decision creates new issues. There are certainly other possible protocols, but the point is to understand that the tuple transfer protocol must attempt to balance the sometimes conflicting parameters of interconnection bandwidth, transmission startup costs, processing time required to implement the protocol, etc.

Let the system consist of three nodes, A, B, C, and let a tuple of type T reside on nodes A and B. At this point, node C requests a tuple of type T by broadcasting a template.

Example Protocol 1

If a node has a tuple that matches a received template, then send the tuple to the requesting node.

In our example, node C would receive two tuples while only needing one, and thus must either store the extra tuple or send it back. If the protocol says to store the extra tuples, then the required communication includes one broadcast for the template and n transmissions of the tuple (where n is the number of nodes with a matching tuple). $n - 1$ of the tuple transmissions can be considered unnecessary. Since the size of tuples can be large, this unnecessary traffic may be a problem. Another problem is that tuples that were nicely distributed throughout the machine may tend to become concentrated on a single node thereby serializing their access. If the protocol says to send the extra tuples back, then the required communication includes one broadcast for the template and $2n - 1$ transmissions of the tuple. This variation does not concentrate tuples, but causes even more unnecessary communication.

Example Protocol 2

If a node has a tuple that matches a received template, then send a positive acknowledgment. If a node's positive acknowledgment is acknowledged, then send the tuple.

In our example, node C would receive two positive acknowledgments. Node C selects one of the positive acknowledgments, say the one from node A, requesting its tuple. This protocol requires one broadcast for the template, n positive acknowledgments, one acknowledged acknowledgment, and one tuple transmission. Because the acknowledgments are typically much smaller than tuples, this method seems to reduce the amount of unnecessary traffic, but increases the number of distinct communication steps possibly increasing the overall time to satisfy the template. The protocol as specified is incomplete. To see why, consider if node A receives a request from another process for the tuple. What does node A do? It does not yet know that it has been selected by node C because that acknowledgment is currently enroute. One solution is to positively acknowledge this second template as well. If also selected by the second process, node A must back out of one of the transactions requiring one node to re-broadcast its template. Such a protocol could create a large amount of interconnection traffic. An alternative solution is for each of the n node which positively acknowledged the initial template from node C to be locked until that template is satisfied, but this unnecessarily delays the $n-1$ nodes that are not selected.

No protocol is perfect

It is hopefully clear that selection of a tuple transfer protocol is not a simple decision, and one that attempts to balance several system characteristics. However, the designer should take some solace in the fact that there is no perfect protocol. Just as the effectiveness of the tuple distribution method depends upon the actual tuplespace access patterns, any reasonable protocol will work well for some tuple access patterns, and poorly (even terribly) for others. Again, sophisticated compiler analysis estimating tuplespace access patterns and/or runtime statistics gathering of actual access patterns may be able to allow the selection of alternative protocols. Shekhar and Srikant [44] suggest that because each application has its own tuplespace access patterns, the tuplespace implementation itself should be reconfigurable for each application so as to minimize inefficiencies. However, it seems the selected configuration remains in effect for the lifetime of the application, thus as an application's tuplespace access patterns evolve and change, their technique may be unable to avoid these inefficiencies.

Eval Operator Implementation

The EVAL operator is the programmer's vehicle for explicitly expressing parallelism through the creation of a new process. This section examines the semantic difficulties of the EVAL operator, followed by implementation alternatives looking in particular at what node should execute the new process and how it establishes communication with other nodes.

Eval semantics

The EVAL operator is used to create concurrent threads of control. As such, it is often the implementation issue requiring the most machine-specific solution. In addition, the semantics of the EVAL operator are not well-specified, making this a potential source of non-portability of applications across different implementations. Specifically, the model does not define whether the new process shares global variables with its parent, or if a process can modify parameters passed by reference [44, 49]. In the absence of a clear semantic definition, implementations seem to generally let the machine's available process creation primitives dictate the definition of their semantics. For example, on a shared memory machine, access to the values of the parent process' global variables is easily achieved. However, in a network of workstations, this is not the case. Leichter [31] and Bjornson [6] also discuss this semantic issue. Due to the semantic confusion, it would seem prudent for implementors to select the most restrictive interpretation, allowing the new process to only have access to r-values of explicitly passed parameters.

Implementation alternatives

One possibility for implementation is through system calls that create new threads (e.g., the UNIX *fork()* system call). However, such system calls may not exist for all machines. Another possibility is for the compiler to package together the code making up the processes into separate executables. The appropriate executable could be shipped to the selected node at runtime, but this appears unattractive because the executable code is typically large, and heterogeneity among nodes is infeasible. Providing access to these executables for each node via a common file system could eliminate the complication of runtime transmission of large codes[3].

A general, machine-independent approach is to use an *eval server* [6, 41]. The EVAL servers, which run on every node, monitor tuplespace for active tuples. A particular EVAL server can evaluate all processes spec-

[3] In fact, the wide utilization of NFS makes this rather straightforward.

ified within the active tuple, or only one such process. Some implementations require only one process per active tuple for simplification reasons. One possible disadvantage of the EVAL server approach is that each server must have the code of all processes that can be evaluated in parallel. The availability of a common file system can again remedy this problem.

Node selection and communication

Another issue to address when implementing the EVAL operation is deciding on an available node to execute the process. Shekhar and Srikant [44] discuss the importance of process creation in the context of system load balance. Combining knowledge of the tuple distribution method, computational requirements of processes, information relating processes, and tuple usage could lead to a more balanced load by placing processes on appropriate nodes. However, most implementations seem to use a straightforward cyclic placement approach. Indeed, using an EVAL server easily achieves such a placement by means of the other tuplespace operations.

The EVAL operator creates new processes that most likely will need to communicate with other processes. How and when this communication is established becomes an implementation characteristic. *Static communication binding* establishes communication channels between all nodes at program (or system) initiation. *Dynamic communication binding*, on the other hand, establishes channels the first time two nodes need to communicate. Static binding delays the start of parallel program execution as all the communication channels are established prior to the user's program execution. Because dynamic binding is a type of lazy binding, communication channels that are not needed are not established. However, evaluating performance may become more difficult because it is not as easy to separate program execution from this system overhead.

Low-level Transport Protocol

At the heart of a distributed tuplespace lies the efficient, yet reliable, transfer of data among the nodes of the machine. Because there seems to be an almost unique interconnection network for each class of distributed machine, there will likely be no consensus on a best method to provide efficient reliability of data transfer. We will discuss the low-level transport protocol in terms of the Internet protocol suite (TCP/IP) because it is widely supported by many systems, and the methodologies of TCP/IP (i.e., reliable, circuit-based transmission vs. unreliable datagram transmission) are adopted by other protocols (e.g., XNS).

Reliable, connection-oriented protocol

A connection-oriented protocol involves the establishment of a (virtual) circuit between the sender and receiver, which is typically a costly step. Often, it is possible to amortize this cost by reusing the circuit, but this may involve additional processing to maintain the connection. Utilizing such a circuit for the message transfer, the protocol automatically provides for message fragmentation and reassembly, packet sequencing, flow control, and error checking. There is usually no restriction on the size of messages that can be transmitted using a connection-oriented protocol. Because the protocol provides these services, the implementor is relieved of these low-level details and is free to concentrate on decisions regarding the tuplespace-specific issues. The cost of this freedom is the possibly significant amount of time to establish the logical connection between the endpoints.

Unreliable, connection-less protocol

A datagram protocol eliminates the cost of the connection-oriented scheme by not setting up a path from sender to receiver in advance. Rather, each message is independently transmitted possibly taking different paths. Generally, a connection-less protocol does not support the useful services offered by the connection-oriented protocols. This requires the programmer to explicitly implement the services that are needed. Often, there is a ceiling on the message size supported by a connection-less protocol, meaning the implementor may need to design fragmentation, sequencing, and reassembly services into their low-level protocol. The sender of a datagram does not know if the datagram reaches its destination. Ensuring delivery of a datagram requires the implementor to provide this extra processing using timeouts, acknowledgments, retransmissions, etc. (see description of various data link layer protocols in [47]).

Which to use?

Of course, this decision is up to each individual implementor who must consider the tuple distribution method selected and characteristics of the particular interconnection network. The choice seems to come down to the performance advantages of connection-less protocols versus the simplicity of the connection-oriented techniques. A compromise would seem to defer the implementation complexity of this issue by having an initial version select a connection-oriented protocol, while a later version can address performance by replacement with a connection-less protocol.

Heterogeneity

Today's local area networks are increasingly being populated with nodes of different capabilities and architectures. Network systems such as LAN-based tuplespace implementations should accommodate this heterogeneity. Allowing heterogeneous nodes to participate in the distributed, parallel execution of a single application involves addressing several items. The first problem concerns data formats. Some machines will be little endian and others big endian; some nodes will have 64 bit words and others will use a word size of 32 bits. A common solution to the data format problem taken by many systems is to utilize XDR routines to convert specific data formats to/from a canonical format. The use of the SCI data formats is also a possible technique to solve the data format problem.

Another problem is execution of processes on heterogeneous nodes. Linda programs are typically compiled on one node generating an executable image for that node architecture. Thus this image is not runnable on another node of a differing architecture. One solution is to have the Linda compiler act as a cross-compiler for every type of node in the system. Then the appropriate image is referenced by each node. A similar alternative would have the Linda compiler pass an intermediate form of the program to a native compiler on a node of each architecture for code generation.

The above discussion assumes a tuplespace accessible by operations of processes in a single application image[4]. [8] terms this a *closed* tuplespace. An *open* tuplespace would permit access by any Linda operation regardless

[4]While multiple instances of the image may be active due to the EVAL operator, it remains the same image.

of application. Because the tuplespace model decouples the communicating processes, there is already support for heterogeneity between applications written in different languages. There are no tuplespace calling convention issues to worry about. [8] discusses an approach to providing truly heterogeneous support through the use of multiple and persistent tuplespaces, where communication applications utilize an open tuplespace, and a closed tuplespace is restricted to processes of a single application. This structure permits the optimizations possible on a closed tuplespace because all the tuplespace operations of the application are known in advance by the compiler, as well as opening the door to coordinating applications via the open tuplespace.

Simultaneous Invocation

In an academic environment, the extra time involved to implement a system supporting multiple, simultaneous invocations may not be justifiable as evaluation of system characteristics is achievable even on a single-user system. However, designing a system to support such invocations often does not incur significant additional effort. As the system matures, the additional implementation effort may be warranted and with a foresighted design, this effort can be minimized. Therefore, a system such as Linda which can appeal to a broad class of parallel programmers should at the very least be designed to accommodate multiple and simultaneous invocation. This essentially comes down to the avoidance of using "hardcoded" system resources. As best as possible, the availability of system resources should be dynamically determined and allocated. In our own implementation, TCP/IP communication ports are determined at the time of invocation at only a slight increase in complexity as opposed to assuming a predetermined port must be available[5].

System Extensions

This paper describes many issues faced by a distributed tuplespace implementor wanting to build a minimal system. This basic system can be extended in a number of interesting ways. The first two enhancements described remain within the confines of the fundamental tuplespace model as explained, while the latter two

[5]Nevertheless, Deli currently does not support simultaneous invocation as described in the implementation section.

focus on extending the model itself.

Runtime optimizations

Bjornson added a number of enhancements to his basic tuplespace implementation [6]. *Tuple broadcasting* sends tuples to all nodes hoping to eliminate the cost of performing a RD operation. An IN operation requires additional processing to remove the replicated tuples. This optimization is most effective with some preprocessor/compiler assistance indicating those partitions that contain RD operations. The rendezvous protocol uses a hashing function to deterministically assign a node the responsibility of managing a partition of tuples and templates. Performance is increased through reduced network accesses when a process that is heavily consuming the tuples of a partition happens to reside on the node assigned to manage that partition. *Tuple rehashing* is a runtime technique that will reassign partition management to a node containing a process that is heavily consuming tuples of the partition thus making accesses local.

The *inout collapse* optimization combines an IN operation with an OUT operation where preprocessor/compiler analysis indicates that it is safe to do so. Prior to the transformation, the two operations account for three network accesses, two due to the IN to send the tuple request and receive the tuple, and the OUT operation's redepositing of the tuple. Collapsing the operations entails two accesses, a request for a tuple and receiving the matching tuple. The tuplespace manager is now responsible for updating the tuple in place in tuplespace also saving memory allocation/deallocation. The normal rendezvous protocol maps all the tuples of a queue partition to a single node which may become a bottleneck. The *randomized queue set* will spread the tuples of such a partition over a subset of nodes thereby increasing concurrency and eliminating a bottleneck at the cost of additional management overhead.

Advanced compiler analysis

Two of the runtime optimizations just described require compiler support. The others require the runtime system to constantly monitor and gather statistical information regarding tuple access, and only after a period of time of inefficient accesses are the dynamic optimizations performed. Aggressive compiler analyses

[20, 21] can improve the situation by finding more opportunities for inout collapse and making more informed judgments for tuple broadcast (i.e., estimating number of RD's versus IN's). Compiler analysis could also suggest to the runtime system which partitions may require rehashing thus reducing (possibly eliminating) the time period of inefficiency. Advanced compiler techniques are also able to move Linda operations so as to achieve a form of latency hiding [29].

Multiple tuplespaces

The basic tuplespace model utilizes a single tuplespace equally accessible by any operation in any process in the application. Some researchers have explored allowing multiple tuplespaces thus treating a tuplespace as a fundamental object of the model. [24] describes a new tuplespace data type and an operation to create an object of this data type (i.e., a tuplespace). Interesting motivation and examples of applicability of multiple tuplespaces is also described. [32, 4] also utilize flavors of the multiple tuplespace concept. Presently, there seems to be no clear semantic definition of multiple tuplespaces or how they can/should be implemented.

Persistent tuplespaces

Another interesting extension is having the tuplespace persist beyond the lifetime of an application. This seems to have a pleasing symmetry with the basic model which already decouples processes in time within an application. Allowing a tuplespace to exist beyond applications would seem to require a persistent tuplespace manager, one that is not inherently coupled with the application itself. This separation may make certain communication optimizations infeasible. Saving tuplespace to disk will ensure its continual operability; however, in a distributed setting, it is not clear how this should be done. Leichter [31] suggests two approaches but does not recommend one over another. Each distinct tuplespace manager can save its local portion of tuplespace, and then reactivation of the tuplespace would require all portions to be present; or designate some node to collect and combine the various portions and save it in a single location. The last problem is that there does not exist any agreed upon programming interface for supporting a persistent tuplespace. How does an application connect to an existing persistent tuplespace? How is the persistent tuplespace created initially? How can a persistent tuplespace support a separate but simultaneous invocation of an application?

DESIGN GOALS

Deli was developed for the purpose of validating and experimenting with our compiler analysis and optimization of Linda programs [21, 20]. To make this Linda tuplespace implementation a robust research tool, the following goals were established. These goals are presented here as they affect implementation choices described in the following section.

Software-based approach. No specialized hardware or specific interconnection topology should be required. Although workstations connected by Ethernet has been the initial target architecture, the implementation should not assume any features (e.g., broadcast).

Distributed tuplespace. Tuples in tuplespace should reside on more than one node of the parallel machine. Centralizing tuplespace while distributing the computation is a valid implementation choice, but one that under-utilizes the available power of the entire parallel machine and can also lead to bottlenecks in tuplespace access.

Compatibility. An important consideration is the ability to run existing Linda programs with no, or very little, source code changes. Linda, because it augments existing languages, has a short learning curve. An implementation that requires additional syntactic information contradicts this important feature of Linda.

Flexibility. The implementation should easily accommodate changes, for a variety of reasons including,

- *Optimizations.* Some optimizations require alternative handling of tuples and templates.
- *Architecture simulation.* The implementation should support the inclusion of a cost model (such as [48]) to simulate architecture parameters such as processor speed and interconnection bandwidth.
- *Heterogeneity.* The implementation should be able to accommodate varying processor architectures.

Reliability. The implementation must guarantee reliable message passing. Providing fault tolerance is not a present focus[6].

Portability. The implementation should port to new systems in a straightforward manner.

Reconfigurability. The implementation should be easily reconfigured in order to examine different approaches to several of the implementation issues, such as tuple transfer protocol, and EVAL operator implementation. Shekhar and Srikant [44] argue that reconfigurability provides individual applications with efficiency improvements over a fixed implementation.

IMPLEMENTATION

Deli (University of **De**laware **Li**nda) is an implementation of Linda tuplespace for a distributed memory parallel machine. In particular, Deli successfully executes on a network of SUN 4 workstations connected via Ethernet. Deli is implemented in C on a UNIX platform utilizing the Berkeley socket API for network communication. Deli was modeled after Bjornson's distributed Linda implementation [6] which has been used in recent versions of production Linda systems [7], and is also based on our experiences with the *ntsnet* utility from the SCA Inc. distribution of Linda.

Deli is a program that facilitates, transparently to the user, both the utilization of available network nodes and the preparation of these nodes for participation in the execution of the user's Linda parallel program. These operations are collectively referred to as *bootstrapping* the Deli system. As shown in the left side of Figure 1, the user invokes Deli directly at the host node specifying the Linda executable program to run and the number of additional desired nodes. Deli then attaches the desired number of nodes, starts up the tuplespace managers and EVAL servers, and begins execution of the Linda program. This situation is depicted in the right side of Figure 1.

The following subsections describe in detail the Deli bootstrapping process, the responsibilities of the Linda compiler, the modification of the user's program in order to run inside the Deli environment, the tasks and

[6]Providing fault-tolerance introduces a series of additional implementation issues; moreover, a fault-tolerant implementation must first satisfy the issues described here. See [4, 26, 50] for in-depth treatments of fault-tolerant Linda implementations.

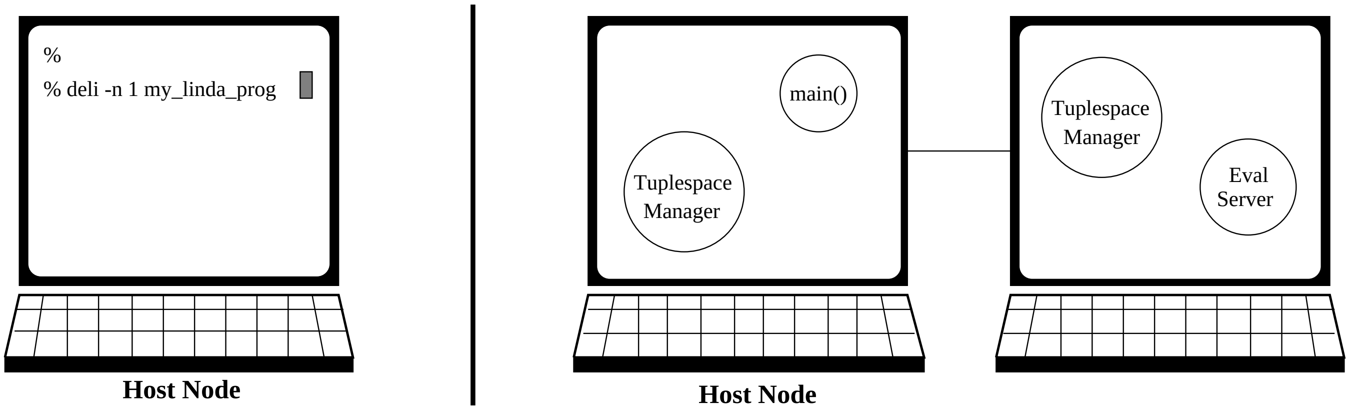

Figure 1: High level system state before and after Deli bootstrapping.

implementation of tuplespace managers, tuple and tuplespace data structures, tuplespace access functions, and implementation of the EVAL operation. Providing a thorough treatment of the implementation details requires examining and discussing source code. In the source code pieces throughout this section, irrelevant technicalities have been omitted to improve clarity. For example, error checking code has been removed.

Bootstrapping

This section details how Deli transparently progresses from a user invocation on one network node to an environment consisting of multiple network nodes poised to commence execution of the user's Linda parallel program. Deli executes in one of two modes, host or remote mode. There is only one execution of Deli in host mode, and it is on the host node. However, there may be many concurrent executions of Deli in remote mode, one on each remote node. In either mode, Deli first parses the command line to verify a valid invocation and determine the number of desired nodes, called *num_nodes*. Then Deli creates and attaches a shared memory segment. This region provides an interprocess communication channel for the local processes that Deli is about to create. Appropriate signal handlers are also installed at this time. Currently, Deli traps the interrupt signal (SIGINT) at the host node to initiate an orderly shutdown of the entire Deli system. Lastly, Deli obtains two consecutively numbered and available communication ports. The first of these ports is referred to as the Deli *base* port. Figure 2 shows these first steps of Deli's *main()* function.

Initially, there are no executions of Deli anywhere in the system. When a user interactively invokes Deli from

```
#include "deli.h"

extern int host_mode,        /* Execution mode */
           server, client;   /* process ids */

main(int argc, char *argv[])
{
  parse_command_line(argc, argv);
  open_share();
  signal(SIGINT, deli_intr);
  get_ports();

  if (host_mode)
    host_bootstrap(argc, argv);
  else
    remote_bootstrap(argc, argv);

  /* Deli is bootstrapped. */

  if ( (server = fork()) == 0)
    Server(argc, argv);
  else
    if ( (client = fork()) == 0)
      Client(argc, argv);
    else
      Wait();

  deli_shutdown();
}
```

Figure 2: Deli main function

a node, this node becomes the host node. After the mode independent operations are performed, the host mode Deli (on the host node only) begins a sequence of operations to bootstrap the system. The host node initiates the bootstrapping procedure when it is still the only node in the system. Information regarding the names of available nodes is retrieved from a file. By default, the file is expected to be in the current directory although a command line argument specifying an alternate location would be a straightforward enhancement. Using this information, the host mode Deli execution creates a node table and initializes the table with *num_nodes* remote node names. A loop is then performed *num_nodes* times with each iteration creating a child process. These child processes then determine a remote node to contact by using their iteration value to index into the node table. The intent is to begin execution of a remote mode Deli on each of these remote nodes. Deli uses the UNIX *rsh()* utility to login to the remote node and begin an execution of Deli on that node. To invoke these Deli executions in remote mode, a secret argument is added to the Deli command line. This secret command line argument also encodes the host name and base port so that

```
#include "deli.h"

extern int num_nodes;
extern int *remote_pids;
extern NODE *node_table;

host_bootstrap(int argc, char *argv[])
{
   MSG *msg;

   init_node_table();

   /* Create processes to spawn Deli on remote nodes. */
   remote_pids = (int *)malloc(sizeof(int)*num_nodes);
   for (i=0; i < num_nodes; i++) {
      remote_pids[i] = fork();
      if (remote_pids[i] == 0)
         rsh(i, argc, argv);
   }

   /* Build node table, node_table[0] is host */
   for (i=1; i <= num_nodes; i++) {
      msg = get_message();
      for (j=0; j <= num_nodes; j++)
         if (node_table[j].name == get_sender(msg))
            node_table[j].port = msg->orig_node;
   }

   send_table();
}
```

Figure 3: Deli host_bootstrap function

the remote node can establish communication with the host. This series of steps is depicted in Figure 3.

At this point in the bootstrapping procedure, remote nodes are asynchronously and in parallel being directed to invoke a remote mode execution of Deli. The host Deli execution then cooperates with the remote nodes to build a complete node table on the host which is then forwarded to the remote nodes. Building the node table is simply a loop to accept *num_nodes* messages on the base port, one from each remote node. The originator of a message is determined from the connection, and the message body contains the base port of the remote node. Figure 4 diagrams the bootstrapping communications between the host node and a remote node.

After bootstrapping, the Deli system consists of one node running in host mode and *num_nodes* remote mode nodes. Communication ports have been allocated, a shared memory segment has been created and signal handlers have been installed on all nodes. A node table, containing the name and base port for each node, resides in the shared memory region on each node. A process has been created on each node to act as the

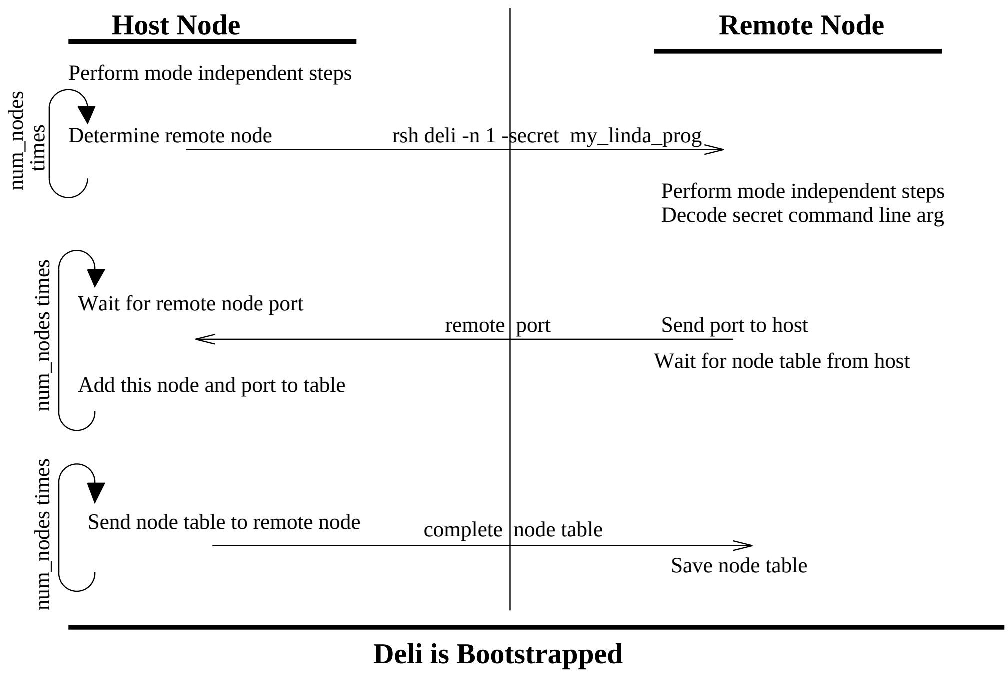

Figure 4: Deli bootstrapping communication timeline

tuplespace manager, and a client process has also been created. On nodes running in remote mode, the client process quiescently turns into an EVAL server. On the host node, the client process begins execution of the Linda program. The parent Deli process then waits for the termination of its child processes, at which time an orderly shutdown is performed.

Compiler Support

Consistent with production Linda systems, Deli requires programmers to name their top level function *real_main()* rather than *main()*. The user's program Linda is linked with a *main()* function supplied by the compiler to perform various system related activities. Recall from Figure 2 that Deli creates two child processes to execute a "server" and a "client". Both of these functions perform a UNIX *exec()* system call to execute the user's Linda program specified on the Deli command line. In addition to passing whatever command line arguments are specified for the Linda program, Deli appends a secret command line argument. Obtaining this special argument and removing it from the command line is part of the system activities per-

```
#include "deli.h"

extern int host_mode,          /* Execution mode */
           port0;              /* Node's communication port */

Server(int argc, char *argv[])
{
  int i, j, linda_prog_arg;
  char **args;

  /* Get Linda program command line args by skipping Deli specific args. */
  prog_arg = linda_prog_arg_num(argc argv);
  args = (char **)malloc(sizeof(char *) * (argc-prog_arg+2));

  /* Create command line argument array for the exec() syscall. */
  for (i=prog_arg,j=0; i < argc; i++,j++)
    args[j] = strdup(argv[i]);

  /* Append the Deli secret command line argument. */
  if (host_mode)
    args[j] = encode(HOST, SERVER, num_nodes, port0);
  else
    args[j] = encode(REMOTE, SERVER, num_nodes, port0);

  args[++j] = NULL;

  /* Run Linda program who decodes secret arg above to become TS manager. */
  execvp(args[0], args);
}
```

Figure 5: Deli's server child invoking user program as a tuplespace manager

formed by the compiler supplied *main()* function. The special command line argument encodes information directing the process to become a tuplespace manager, an EVAL server, or to invoke the Linda program's *real_main()* function.

Figure 5 illustrates how the "server" child of Deli invokes the user's Linda program with an extra command line argument indicating that the tuplespace manager should be executed. The "client" child works in a similar fashion. Figure 6 depicts the Linda compiler supplied *main()* function. Other system operations include installing signal handlers, attaching the shared memory region created by the Deli parent, and executing a function generated by the Linda compiler to initialize a data structure containing static tuple information (this data structure is described in greater detail below). A shutdown routine is used to perform an orderly termination with the other processes and nodes. The names of all Deli data and functions that are in the scope of the user's Linda program are prepended with "_deli_" to avoid duplication with any user names.

```
#include "deli.h"

main(int argc, char *argv[])
{
  int action, rtnval=0;

  _deli_initialize();
  action = _deli_decode(argv[argc-1]);

  argv[argc-1] = NULL;   argc--;          /* Hide secret arg from user. */

  switch (action) {
    case TS_MANAGER :
      _deli_tuplespace();   break;

    case HOST_CLIENT :
      rtnval = real_main(argc, argv);   break;

    case REMOTE_CLIENT :
      _deli_eval_server();   break;
    }

  _deli_shutdown();
  return rtnval;
}
```

Figure 6: *main()* function supplied by Linda compiler

The Linda compiler replaces all Linda operations in the user's program by code segments that make procedure calls to a runtime library. These procedures are responsible for details such as marshaling data and performing the physical communication. The compiler also imperceptibly attaches two additional pieces of software to the user's program. These two extra modules are the EVAL server and the tuplespace manager. The EVAL server is included because the server requires the code for all of the user specified processes, and these are contained within the user's program. Details regarding the implementation aspects of the EVAL are below. The tuplespace manager is included with the user program to allow certain communication optimizations. While increasing the size of the executable, this organization allows the compiler to generate code based on the user's program that is visible to the tuplespace manager. The *inout collapse* optimization is implemented in this fashion.

Lastly, our compiler implements the tuplespace partitioning described by Carriero. We also perform inout collapse analysis, and analysis that identifies tuples eligible for tuple broadcasting (i.e., shared variable tuples).

Tuplespace Managers

Each node in the Deli system runs a tuplespace manager process. This process initializes its portion of tuplespace, then awaits requests arriving on the base port and appropriately acts upon them. Deli currently uses three types of request messages: KILL, EXTRACT, and INSERT. The host node tuplespace manager sends the KILL message to the remote nodes informing them to shutdown. When a KILL message is received, the tuplespace manager simply exits, and this exit is trapped by the waiting Deli parent on the remote node who terminates its remaining children. The EXTRACT message is issued in response to a Linda program process performing an IN, RD, INP, or RDP operation; similarly, the INSERT message is in response to an OUT or EVAL Linda operation.

The data structures for tuples and tuplespace are described first followed by the tuplespace access functions that the tuplespace manager uses to handle these messages. Of course, optimizaitons requiring runtime support may alter the data structures and access functions as described here. The data structures described contain enough informaiton to support the inout collapse optimization; however, the access functions have been simplified and thus do not indicate the alternative handling that inout tuples receive.

Deli is modeled after Bjornson's tuplespace description [6] to allow fairer comparison. Thus Deli utilizes essentially the same data structures as Bjornson to represent tuples and tuplespace. To save the reader from referring back and forth to Bjornson's work, these data structures are described below. In particular, we make use of several of Bjornson's figures which serve to clearly depict the structures described.

Tuple data structures

Two data structures to hold tuple and template information. A static structure (called an ST) contains information about the tuple/template partition and operation, information that does not change during the course of execution. For example, number of fields, field polarities, type of operation (IN or OUT), and partition identifier are all part of the ST. A dynamic structure (called a PTP for *proto-tuple-packet*) maintains information about an individual tuple/template, including the values of actual fields, runtime

```
#define NUM_FIELDS 32

/* Static tuple structure. Initialized by code generated by the compiler */
typedef struct {
   short polarity;        /* formal or actual */
   short type;          /* data type of field */
   short anon;          /* anonymous field? */
} ST_FIELD;

typedef struct {
   short set_id;          /* set id number */
   short op_type;          /* type of operation */
   void (*inout_func)();          /* pointer to inout function */
   short num_fields;          /* number of fields in tuple */
   short length;          /* len of used part of ptp, in bytes (send length) */
   short set_type;          /* storage paradigm for tuple */
   long hash_field;          /* field in tuple used for hashing */
   ST_FIELD field[NUM_FIELDS];
} ST_TYPE;
```

Figure 7: ST static tuple data structure (from Bjornson [6])

length, and node of origin. The Linda compiler produces an ST for each Linda statement in the original program. The structure of an ST is shown in Figure 7. The *_deli_initialize()* function of Figure 6 performs the runtime initialization of the ST structures. This provides savings in both bandwidth and memory by allowing only the PTP to be sent to transmit a tuple and permitting the sharing of a single ST by all tuples generated by the same Linda statement.

A PTP can exist in two forms, having to do with the representation of aggregates (e.g., structs and arrays). In one form, an address to the aggregate is used, and in the other form, an offset relative to the start of the PTP indicates the position of the aggregate which has been packed to the end of the PTP. The layouts of these two forms of a PTP are illustrated in Figure 8. This strategy trades extra copying for higher communication costs through the need for multiple communications. The structure of a PTP, shown in Figure 9, follows the Bjornson model.

Tuplespace data structures

The data structures that organize PTPs into a coherent tuplespace are described in this section. Recall that because of the tuplespace partitioning, each tuple and template belongs to a partition set identified by an integer value. In practice, the number of partitions is typically small (on the order of tens of disjoint sets).

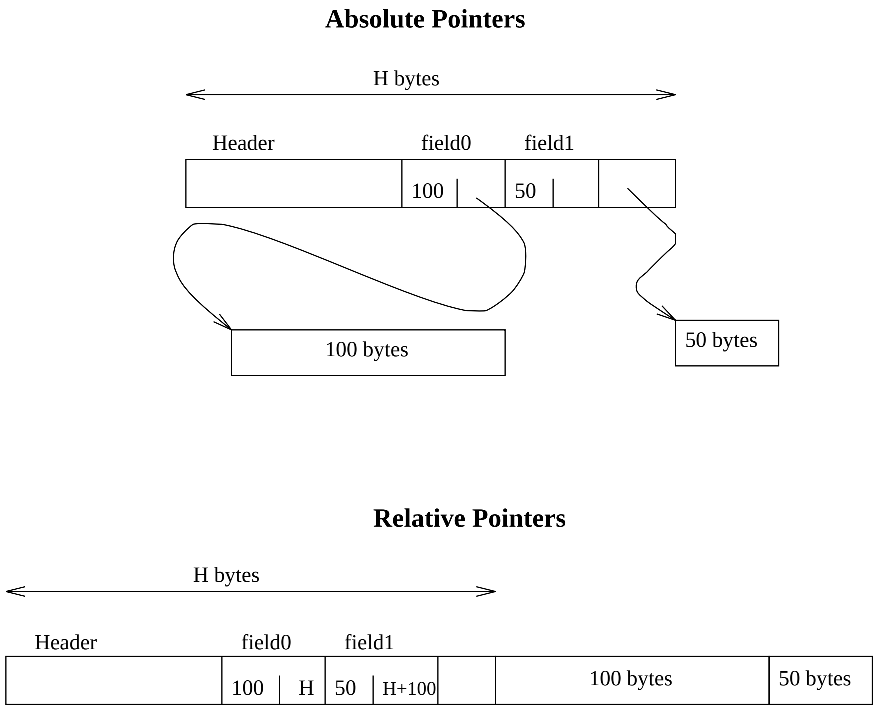

Figure 8: PTP using absolute vs. relative pointers to aggregates (from Bjornson [6])

Thus, tuplespace can be viewed as an array of buckets such that the index into the array is the partition identifier. Each bucket contains two singly-linked chains, one of tuples and the other of templates. All the tuple storage paradigms discussed in the IMPLEMENTATION ISSUES section are implemented using chains, with the exception of counters. However, even in the case of counters, template PTPs must be maintained in a chain so that the originating node information is available when a matching tuple arrives. Figure 10 shows the tuplespace structure. New PTPs are added to the head of the chains resulting in a LIFO access. Although semantically consistent, LIFO access to templates can cause unfairness in satisfying requests. To provide FIFO access to templates, a pointer to the end of the template chain is also maintained. In the case of the partitions that are stored as hash tables, one pointer in the tuplespace bucket points to the hash table. The hash table buckets are represented and manipulated in exactly the same way as the tuplespace buckets. In the absence of tuple chains in the case of counter partitions, the tuple chain pointer actually points to an integer location maintaining the tuple count.

```
#define NUM_FIELDS 32

/* Aggregate data structure used for struct and array elements. */
typedef struct {
   long size;
   union {
      char *abs;          /* absolute data pointer */
      long rel;          /* relative data pointer */
   } data;
} AGGREGATE;

/* Used by system to send data whose type is unknown at compile time. */
typedef struct {
   AGGREGATE overlap;          /* overlaps largest item in ptp_field */
   int type;
} TYPE;

/* The value of a particular tuple element. */
typedef union ptp_field {
   double d;          /* double (actual) */
   double *dp;          /* double (formal) */
   float f;          /* float   (actual) */
   float *fp;          /* float   (formal) */
   char ic;          /* char    (actual) */
   char *icp;          /* char    (formal) */
   short is;          /* short   (actual) */
   short *isp;          /* short   (formal) */
   int ii;          /* int     (actual) */
   int *iip;          /* int     (formal) */
   long il;          /* long    (actual) */
   long *ilp;          /* long    (formal) */
   AGGREGATE a;          /* Aggregate  (actual or formal) */
} PTP_FIELD;

/* Dynamic tuple structure, filled each time operation is performed. */
typedef struct ptp_type {
   long scratch;          /* volatile field - usually the request type */
   struct ptp_type *succ;
   long tid;          /* unique id for this tuple */
   short orig_node;          /* id of origination node (of the ptp) */
   short orig_pid;
   short dest_node;          /* id of destination node (of the ptp) */
   short dest_pid;
   ST_TYPE *st_ptr;          /* key into statement array */
   long length;          /* runtime length of ptp (incl. long fields) */
   long hash_value;
   PTP_FIELD field[NUM_FIELDS];
} PTP_TYPE;
```

Figure 9: PTP dynamic tuple data structure (from Bjornson [6])

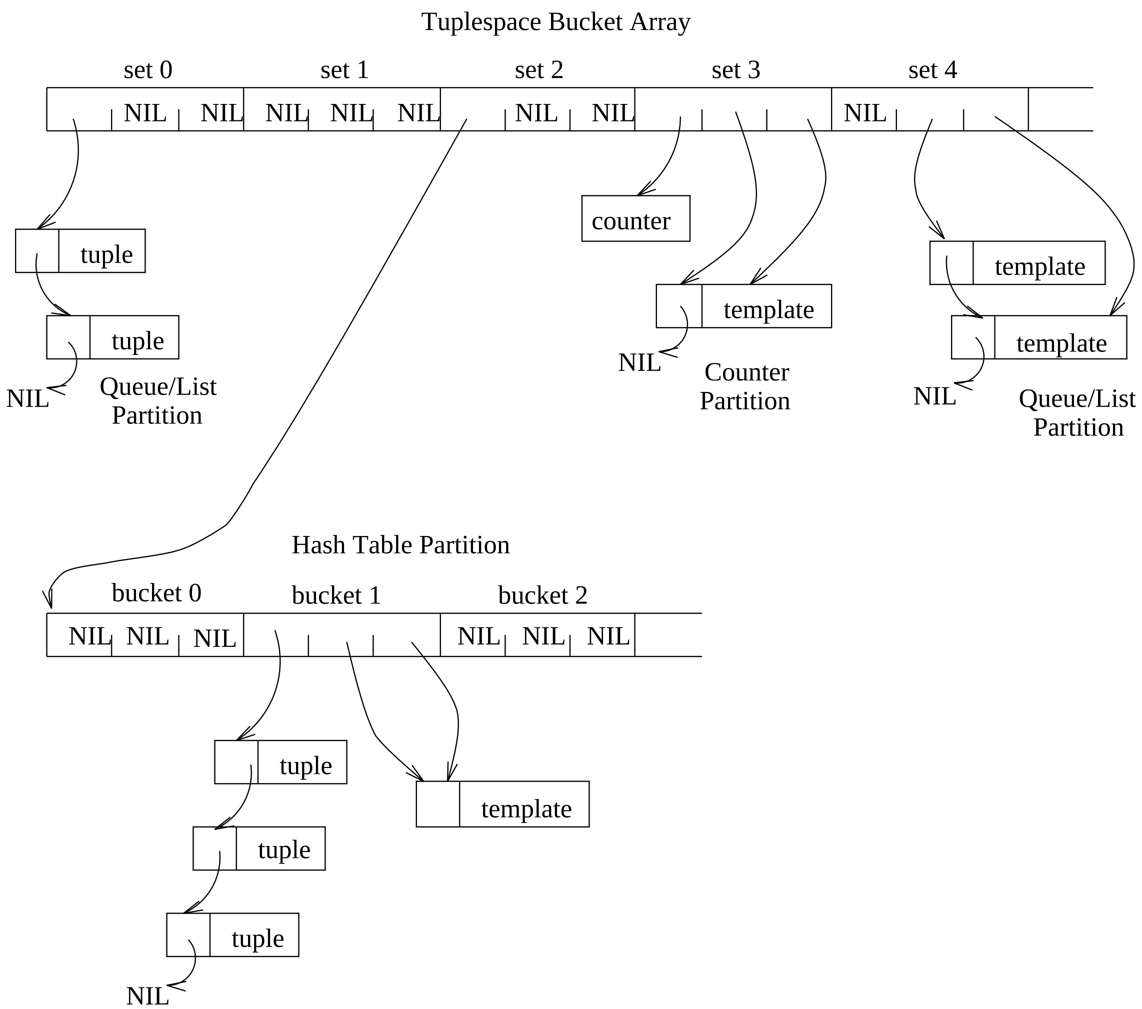

Figure 10: Structure of tuplespace (from Bjornson [6])

Tuplespace access functions

Recall that the tuplespace manager receives EXTRACT and INSERT request messages in response to a program's Linda statements that remove or add a tuple in tuplespace, respectively. These request messages are nothing more than the header portion of a PTP, and the message type is retrieved from the *scratch* field (see Figure 9). Through the *st_ptr* field, the PTP header indirectly provides information regarding the partition class of the tuple/template allowing the selection of class-specific access routines. While each partition class may be implemented with a different data structure (e.g., queues, counters, etc.), there is still a great deal of similarity in the access functions themselves due to the chaining feature common to all the tuplespace data structures.

The basic steps needed to insert a tuple into tuplespace are outlined in Figure 11. Depending on the type of partition, the appropriate tuplespace bucket is identified and the template chain is traversed. If no templates exist for this partition, then the tuple needs to be stored. Otherwise, each template encountered in the chain is matched against the tuple by the *match* function. For queue and counter partitions, this function always returns a boolean true value. For lists and hash tables, the values of actual fields are compared. If the tuple matches a stored template, the template is removed from its chain[7] and the tuple PTP is sent to the blocked process. Consistent with the observed behavior of production Linda systems, the template chain traversal terminates after either examining all templates or upon encountering a matching template from an IN Linda operation[8]. The tuple must also be stored if the template traversal encounters no matching IN templates. A tuple from a counter partition is stored by incrementing the associated integer variable. A tuple from any other partition is saved by adding it to the tuple chain.

Figure 12 outlines the basic steps of removing a tuple from tuplespace. Again, the proper bucket is identified and the chain of tuples is traversed. As with tuple insertion, the "match" function is partition type specific, as are the "satisfy_incoming_tmplt" and "remove_tuple" functions. If the current template is the result of

[7]The function to remove a template from its chain returns the successor of the removed template.

[8]Because the order of templates in the template chain does not necessarily indicate the order that processes actually made their requests, an interesting semantic question arises. Should RD templates *after* an IN template in the chain be satisfied? SCA's version 2.5.2 does not satisfy these RD templates, but it does not appear invalid to satisfy all pending RD templates in the chain regardless of position and one, if any, IN template.

```
extern TS_BUCKET *TupleSpace;
extern ST_TYPE *_deli_ST;

int insert(PTP_TYPE *ptp)
{
  ST_TYPE *st = &_deli_ST[ptp->st_ptr];
  TS_BUCKET *hash_table, *bucket, *ts_bucket = &TupleSpace_set[st->set_id];
  PTP_TYPE *tmplt, **tmplt_chain, **tuple_chain, **last_tmplt;
  int stop = 0;

  if (st->set_type == HASH_TABLE) {
    hash_table = (TS_BUCKET *) ts_bucket->tuples;
    bucket = &hash_table[ptp->hash_value];
  }
  else
    bucket = ts_bucket;

  tmplt = bucket->templates;
  tmplt_chain = &bucket->templates;
  last_tmplt = &bucket->last_template;
  tuple_chain = &bucket->tuples;

  while (tmplt && ! stop) {
    if (match(ptp, tmplt)) {
      satisfy_waiting_tmplt(ptp, tmplt, sockfd);
      if (_deli_ST[tmplt->st_ptr].op_type == IN_OP)
        stop = 1;
      tmplt = remove_tmplt(tmplt_chain, tmplt, last_tmplt);
    }
    else
      tmplt = tmplt->succ;
  }

  if (! tmplt || stop) {
    if (st->set_type == COUNTER) {
      long *counter = (long *) *tuple_chain;
      *counter += 1;
    } else {
      ptp->succ = *tuple_chain;
      *tuple_chain = ptp;
    }
  }
}
```

Figure 11: Tuplespace insertion access function

a RD Linda statement, then the tuple is *not* removed from the chain. The traversal of the tuple chain is terminated upon finding a matching tuple or until the entire chain has been traversed. In the latter case, the template must be appended to the template chain if it is not due to an INP or RDP operation. If a matching tuple is not found, then the client process issuing the EXTRACT request becomes blocked by waiting for a matching tuple to be delivered on the second communication port (i.e., $base\_port + 1$).

An important item omitted from the functions in Figures 11 and 12 is the possibility of collisions in the hash table. Collisions due to tuples and templates having the same key are chained together in the same bucket of the hash table. However, because the hash table is finite, the possibility remains for tuples/templates with different keys to map to the same bucket. Bjornson [6] simply adds these differing-key, colliding tuples/templates to the bucket's chain. This approach is justified, in large part, by noting that the hash table will never require expansion (and subsequent reorganization) which can be a time-consuming step. However, the chains will be longer and thus require more time spent searching for a match in the chain. Since the hash table itself is nothing more than a table of pointers to the heads of the chains, a rather large table may not be prohibitive. Therefore, Deli currently uses a simple linear probe collision resolution protocol when collisions of different keys occur, and chains together tuples/templates with identical keys. In experiments to date, Deli has not filled a hash table.

EVAL Implementation

Each remote node in the Deli system executes an EVAL server process. The purpose of the EVAL server is to execute pieces of the Linda program that the programmer explicitly requests via the EVAL statement. This execution is performed asynchronously and in parallel with the host node's *real_main()* function and any other EVAL servers. The Deli implementation of EVAL servers is modeled after Bjornson's description. The EVAL servers of Deli currently support the execution of a single Linda process per node at a time. (See the SUMMARY AND FUTURE EXTENSIONS section for remarks on how to support multiple processes per node.)

Multiple EVAL servers may be utilized to evaluate a single active tuple. For example, the EVAL statement

```
extern TS_BUCKET *TupleSpace
extern ST_TYPE *_deli_ST;

int extract(PTP_TYPE *ptp)
{
  ST_TYPE *st = &_deli_ST[ptp->st_ptr];
  TS_BUCKET *hash_table, *bucket, *ts_bucket = &TupleSpace[st->set_id];
  PTP_TYPE *tuple, **tuple_chain, **tmplt_chain, **last_tmplt;

  if (st->set_type == HASH_TABLE) {
    hash_table = (TS_BUCKET *)ts_bucket->tuples;
    bucket = &hash_table[ptp->hash_value];
  }
  else
    bucket = ts_bucket;

  tuple = bucket->tuples;
  tuple_chain = &bucket->tuples;
  tmplt_chain = &bucket->templates;
  last_tmplt = &bucket->last_template;
  if (tuple && st->set_type == COUNTER && *((long *)tuple) == 0)
    tuple = NULL;

  while (tuple) {
    if (match(tuple, ptp)) {
      satisfy_incoming_tmplt(tuple, ptp);
      if (st->op_type == IN_OP)
        remove_tuple(tuple_chain, tuple);
      break;
    }
    else
      tuple = tuple->succ;
  }

  if (! tuple) {
    /* Either no tuples, or no matching tuples, so save template ptp*/
    if (*tmplt_chain)
      (*last_tmplt)->succ = saved;
    else
      *tmplt_chain = saved;
    *last_tmplt = saved;
  }
}
```

Figure 12: Tuplespace extraction access function

```
eval(3, i, foo(i), bar())
```

creates an active tuple with the latter two fields requiring evaluation,[9] both of which may be independently evaluated by any idle EVAL server. More parallelism is obtained using this approach than having a single EVAL server evaluate all fields of an active tuple.

The Linda compiler replaces each EVAL operation with code that creates tuples used by the EVAL server. These tuples belong to two reserved tuplespace partitions. One holds the values of the resultant tuple, the number of currently unevaluated values, and a unique identifier that differentiates executions of the same textual EVAL statement. The other type of tuple reuses the unique identifier, indicates the process to execute and its arguments, and the position of its return value in the resultant tuple. For example, the EVAL statement above would translate into code resembling,

```
eid = unique_eval_id();
e_values[0] = 3;
e_values[1] = i;
out("resultant values", eid, 2, e_values);
e_args[0] = i;
out("eval process", eid, 2, FOO, e_args);
out("eval process", eid, 3, BAR, e_args);
```

The EVAL server repeatedly performs the following steps. An "eval process" tuple, specifying the process to invoke, is obtained from the tuplespace manager responsible for this tuple's partition by issuing an IN operation resembling

```
in("eval process", ?eid, ?position, ?process, ?args)
```

After obtaining the associated "resultant values" tuple from its tuplespace manager with

```
in("resultant values", eid, ?count, ?values)
```

the return value of the invoked process is placed into the indicated position of the resultant values. The number of unevaluated values is decremented, and if the count is now zero, the resultant values are output

[9] Deli, as with most tuplespace implementations, locally evaluates fields that are not function-valued to avoid the high cost of process creation.

as a new tuple. For example, let the unique eval identifier *eid* be 101, and the value of i be 37. Also, let the "eval process" tuple associated with FOO be completely processed before the BAR eval process tuple. Lastly, let *foo(37)* return the value 5.6 and let *bar()* return the value 97. Then the "resultant values" tuple would take on the following values:

```
("resultant values", 101, 2, {3,37,?,?})          initial tuple
("resultant values", 101, 1, {3,37,5.6,?})        tuple after foo() done
("resultant values", 101, 0, {3,37,5.6,97})       tuple after bar() done
```

Because the "count" field is now zero, the resultant tuple

```
(3, 37, 5.6, 97)
```

is inserted into tuplespace via an OUT Linda operation.

Summary

To put each component into perspective within the Deli system, Figure 13 details the Deli process structure, fleshing out the high level view of Figure 1. The UNIX shared memory segment of Figure 13 contains the node table specifying the nodes and base ports for all machines participating in this execution. Participating nodes have access to the Deli and user executable programs by means of a common file system. The user program, *real_main()* is linked with a compiler supplied *main()* function, a tuplespace manager, and an EVAL server. The *main()* function selects which of the components to execute. The tuplespace manager and EVAL server are both classified as iterative[10] because they accept one request and fully process it before accepting another request. Deli utilizes a dynamic communication binding strategy because the system's communication channels are not established during the bootstrapping phase. The tuplespace manager's port (the base port) is passively opened for connections, but for each communication with a tuplespace manager, a client must first obtain an ephemeral port and then establish a communication channel. All Deli communications utilize the TCP reliable transport protocol. This additional overhead to establish the

[10] This is terminology borrowed from Stevens' excellent text [46].

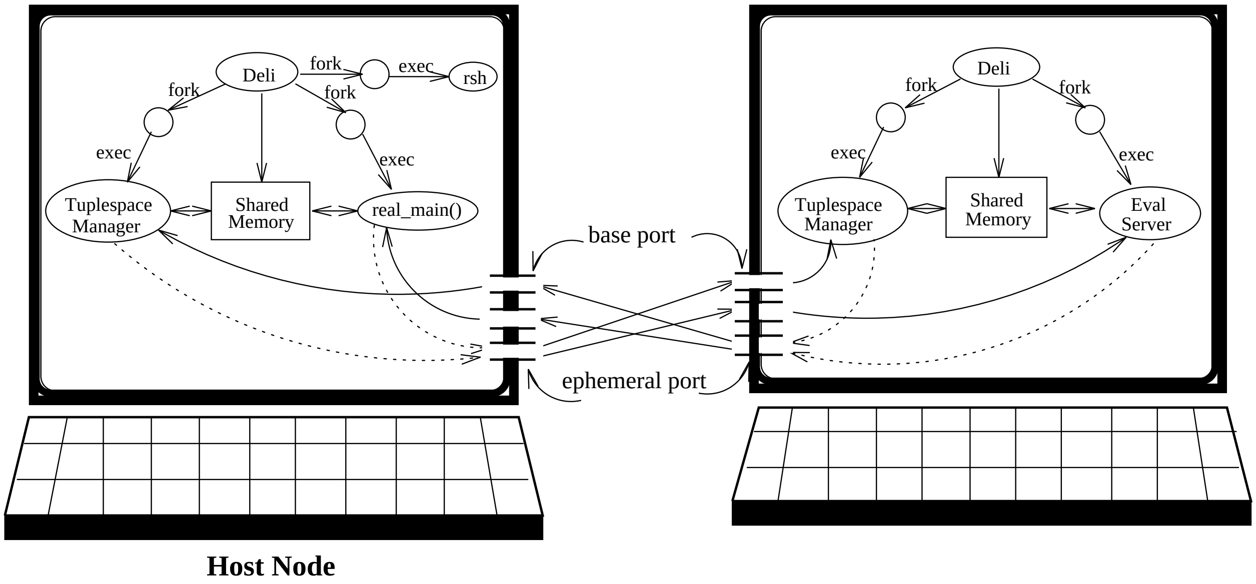

Figure 13: Deli environment detailed diagram

communication circuit is incurred at the cost of reduced implementation time and complexity. The TCP connection is not maintained after the tuple or template is transmitted. Future versions of Deli plan to improve performance by utilizing the UDP datagram service and explicitly ensuring reliable delivery of data.

Allowing the simultaneous invocation of more than one Linda application with Deli requires each invocation to obtain and utilize separate system resources (e.g., network access points, memory, etc.). Currently Deli supports the selection and utilization of non-conflicting communication ports. However, each node utilizes a single shared memory segment whose identifier is hardcoded. Thus a node can only be a part of one Linda application at a given time. A second Deli invocation is possible provided that the host and remote nodes are not participating in any other Deli invocation, but this is deemed unsatisfactory even though a selection of proper nodes could be automated. A better solution is planned and involves simply adjusting the identifier for the shared memory segment if a previous attempt was denied.

IMPLEMENTATION COMPARISONS

There have been several other published implementations of the Linda tuplespace targeting distributed memory machines [6, 44, 3, 16, 7, 31]. The differences and similarities of these systems are summarized in Table 2. Blank entries in the table are due to inadequate information in the publication relating to that characteristic.

The *Tsnet* publication [3] is no longer in print and we were unable to obtain a copy of this report. However, some information is available indirectly from Bjornson's thesis [6]. The two-level tuple transfer protocol used by [44] is a variation of the *Example Protocol 2* described in the IMPLEMENTATION ISSUES section. The rendezvous node tuple transfer protocol [6] dictates that all tuples/templates of a tuplespace partition be sent to one node. Because this rendezvous node is solely responsible for the tuples/templates of the partition, no coordination with other nodes is required to forward tuples to requesting nodes. The Linda-LAN system essentially employs an EVAL server although it is in a slightly different form because they utilize a persistent control subsystem that is independent of the running of any particular application. Special handling of large tuples increases efficiency by reducing the number of times the data is transmitted. Bjornson describes a method of storing a large tuple locally at the generating node and having the rendezvous node inform the generator node of the intended recipient node upon a match.

Bjornson's implementation [6] performs a number of optimizations which are applied due to dynamic analysis, and also some optimizations due to limited static (preprocessor) analysis. These include rendezvous node remapping and tuple broadcast. The Linda-LAN system supports the instructional footprinting optimization of Landry and Arthur [29]. Deli currently supports static communication optimizations similar to Bjornson but using a more informed and comprehensive analysis within the compiler [21, 20]. The transputer implementation of Shekhar and Srikant [44] allows the application to specify among a set of alternative implementation parameters such as tuple transfer protocol. However the application does not provide these alternative functionalities directly, but rather indicates a choice from the set of alternatives already present within the system. Leichter's tuple transfer protocol is very similar to the initial *Example Protocol 1* de-

	DMM-Linda	Tsnet	Linda-LAN	Transputer	Deli	Linda	VAX Linda-C
Authors	Bjornson	Arango, Berndt	Cline, Arthur	Shekhar, Srikant	Fenwick, Pollock	SCA, Inc.	Leichter
Institution	Yale University	Yale University	Virginia Tech.	Indian Inst. of Science	University Delaware		Yale University
Distributed Tuplespace	yes		no	yes	yes	yes	yes
Processor Location	hash on partition identifier	negative broadcast	single fixed node	hash on reserved string field	hash on partition identifier	hash on partition identifier	negative broadcast
Data Structures	queues hash tables lists			hashed array	queues hash tables lists counters	queues hash tables lists counters	hash table
Tuplespace coherence	delete protocol		no replicated data	no replicated data	no replicated data		
Tuple transfer	rendezvous node			two level protocol	rendezvous node	rendezvous node	(see text)
Eval operator	eval server		eval server	eval server	eval server	eval server	none
Efficient handling of large tuples	yes		no	no	no	yes	yes
Application configurable	no		no	yes	no	no	no
Supports optimizations	yes		yes		yes	yes	

Table 2: Summary of distributed memory Tuplespace implementations

scribed in the IMPLEMENTATION ISSUES section. It is claimed that this protocol takes advantage of a type of tuplespace locality of reference [31], but it would seem this becomes less the case when multiple distinct nodes concurrently execute the same process - a common occurrence in Linda programs. Leichter does not implement the EVAL operator but describes an implementation that essentially is an EVAL server.

Several publications were not included in the table and preceding discussion because they focus on implementing Linda not for application level programming, but rather for low-level system programming. As such, there are a multitude of additional issues encountered that require altering or extending the tuplespace model as we have described. Leler [32] adapts the tuplespace model as the basis for a new distributed operating system. His implementation includes a number of interesting extensions including multiple tuplespaces, passing tuplespaces as tuple fields, and language-independent data types. This work also guarantees the ordering of tuples requiring network support or increased implementation complexity. Pinakis [40] similarly uses Linda as the basis for a distributed operating system microkernel. To support certain operating system requirements, the Linda system is modified to allow both heavyweight and lightweight process creation, and a special *ticket* data type is introduced to prevent a process from examining tuples destined to another process. A separate process is utilized to register “tuple types” (tuplespace partitions using our notation) and determine the node responsible for managing a particular tuple type. This seems to add an extra layer of processing, communication, and implementation complexity.

Two other systems include POSYBL [42] and Glenda [43]. Both of these systems have little or no compiler support, thereby requiring the programmer to indicate information to the runtime system through increased programming complexity. POSYBL also changes the semantics of the EVAL operator and does not support the INP, RDP operators at all. Glenda appears to use a centralized tuplespace, requiring the user to explicitly start the tuple server on the host node prior to executing the application program. Glenda does not provide the simplicity of the EVAL operator for process creation. Reflecting the utilization of PVM [37] as its underlying communication layer, Glenda instead provides a spawning capability requiring the programmer to manually break the application into separate executable files and to explicitly specify in the spawn invocation which node executes which executable. To take advantage of optimized PVM communication capabilities, Glenda adds additional tuplespace operations.

SUMMARY AND FUTURE EXTENSIONS

The goal of the Deli implementation was the design and implementation of an experimental testbed for compiler analysis and optimization of Linda programs running on distributed memory parallel systems. For this system, the following design goals were selected: a software-based approach, a distributed tuplespace, compatibility with existing Linda programs, flexibility, reliability, portability, and reconfigurability. This paper described the details of our design and implementation, pointing out the major issues and alternative approaches.

The testbed is a software-based, distributed tuplespace implementation that runs existing Linda programs with little and often no change thus achieving the first three of our goals. The testbed already supports various communication optimizations, and coupled with the planned inclusion of various cost models to allow the simulation of varying machine characteristics implies the accomplishment of the flexibility goal. Reliability is attained through the use of the TCP communication protocol and extensive error checking throughout the source code. Portability is realized by not assuming any specific interconnection, and utilizing UNIX system calls and the Berkeley socket API network facilities. Reconfigurability is possible, but only through modification of source code. Allowing the application to specify certain reconfigurable parameters remains a future goal.

In addition to detailing a tuplespace implementation targeting a network of workstations, this paper makes the following contributions:

- A clear statement of the many issues faced by an implementor of a distributed tuplespace.
- A detailed comparison of prominent distributed tuplespace implementations.
- Comprehensive coverage of implementation aspects either unavailable or missing from current literature including:
  - How multiple nodes are bootstrapped into a single parallel machine.
  - How to handle an arriving tuple matching multiple pending templates.

An extension to Deli is planned, allowing more than a single Linda process per node. This enhancement is deemed necessary to improve performance and eliminate a form of livelock resulting from more processes than processors. Currently, a Deli EVAL server repeatedly obtains a tuple indicating a process to invoke and then executes the process. A modification whereby a child process is created to relieve the EVAL server of the process execution would allow the server to obtain another process tuple concurrently with the child's execution. Currently, each node has a predetermined port for the transmission of a tuple to a blocked Linda process. Allowing multiple Linda processes per node may result in more than one such process being blocked simultaneously. Rather than using a single predetermined port for a blocked process, this responsibility could move to the blocked process itself, having each process encode its own port into the template. When a matching tuple enters tuplespace, the tuplespace manager sends it not to the predetermined port, but to the port specified in the template.

Another amelioration under consideration would increase the parallelism of the system by allowing the tuplespace manager to service requests concurrently. Presently, a tuplespace manager dedicates itself to one request at a time, even if the requests are for different segments of the tuplespace, in effect, locking *all* the tuplespace segments for which the tuplespace manager is responsible. Allowing the concurrent servicing of multiple requests by the tuplespace manager requires creating a child process to service a request while the parent tuplespace manager loops back to accept another request. Additionally, the tuplespace data structures will now require locking mechanisms to ensure that access of shared structures among multiple processes is mutually exclusive.

References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, June 1998.

[2] Sudhir Ahuja, Nicholas Carriero, David Gelernter, and Venkatesh Krishnaswamy. Matching language and hardware for parallel computation in the linda machine. *IEEE Transactions on Computers*, 37(8):921–929, Aug 1988.

[3] Mauricio Arango and Donald Berndt. Tsnet: A linda implementation for networks of unix-based computers. Technical Report YALEU/DCS/TR-739, Yale University, Department of Computer Science, September 1989.

[4] David E. Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, March 1995.

[5] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, March 1990.

[6] Robert D. Bjornson. *Linda on Distributed Memory Multiprocessors.* PhD thesis, Yale University, November 1992.

[7] Leigh D. Cagan and Andrew H. Sherman. Linda unites network systems. *IEEE Spectrum*, 30(12):31–35, December 1993.

[8] N. Carriero, D. Gelernter, and T.G. Mattson. Linda in heterogeneous computing environments. In *Workshop on Heterogeneous Processing '92 Proceedings*, pages 43–46, 1992.

[9] Nicholas Carriero and David Gelernter. A foundation for advanced compile-time analysis of linda programs. In *Languages and Compilers for Parallel Computing*, pages 389–404. Springer-Verlag, 1992.

[10] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs, A First Course.* The MIT Press, 1992.

[11] Nicholas Carriero and David Gelernter. Learning from our successes. In Janusz S. Kowalik and Lucio Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F: Computer and Systems Sciences*, pages 37–45. Springer-Verlag, 1992.

[12] Nicholas John Carriero, Jr. *Implementation of Tuple Space Machines.* PhD thesis, Yale University, December 1987.

[13] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, Dec 1978.

[14] David Chaiken. *Mechanisms and Interfaces for Software-Extended Coherent Shared Memory.* PhD thesis, Massachusetts Institute of Technology, 1994. MIT/LCS/TR-644.

[15] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[16] George E. Cline and James D. Arthur. Linda-lan: A controlled parallel processing environment. Technical Report TR 92-37, Virginia Polytechnic Institute and State University, 1992.

[17] C. Davidson. Technical correspondence on *linda in contex*. *Communications of the ACM*, 32(10):1249–1252, October 1989.

[18] Gary S. Delp, David J. Farber, Ronald G. Minnich, Jonathan M. Smith, and Ming-Chit Tan. Memory as a network abstraction. *IEEE Network*, 5(4):34–41, Jul 1991.

[19] Ashish Deshpande and Martin Schultz. Efficient parallel programming with linda. In *Supercomputing '92 Proceedings*, pages 238–244, Minneapolis, Minnesota, November 1992.

[20] James B. Fenwick, Jr. and Lori L. Pollock. Global static analysis for optimizing shared tuple space communication on distributed memory systems. In *Proceedings of 8th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1996.

[21] James B. Fenwick, Jr. and Lori L. Pollock. Identifying tuple usage patterns in an optimizing linda compiler. In *Proceedings of Mid-Atlantic Workshop on Programming Languages and Systems (MASPLAS '96)*, 1996.

[22] James B. Fenwick, Jr. and Lori L. Pollock. Implementing an optimizing linda compiler using suif. In *Proceedings of SUIF Compiler Workshop*, Stanford, California, January 1996.

[23] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[24] David Gelernter. Multiple tuple spaces in linda. In *Proceedings Parallel Architectures and Languages Europe (PARLE 89)*, volume 2, pages 20–27. Springer Verlag LNCS 366, June 1989.

[25] Milind Girkar and Constantine D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[26] Karp Joo Jeong and Dennis Shasha. Plinda 2.0: A transaction/checkpointing approach to fault-tolerant linda. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, pages 96–105, 1994.

[27] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, Jan 1994.

[28] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept 1979.

[29] Kenneth Landry and John D. Arthur. Achieving asynchronous speedup while preserving synchronous semantics: An implementation of instructional footprinting in linda. In *The 1994 International Conference on Computer Languages*, pages 55–63, Toulouse, France, May 1994.

[30] Alvin R. Lebeck. *Tools and Techniques for Memory System Design and Analysis.* PhD thesis, University of Wisconsin at Madison, Computer Sciences Department, expected 1995.

[31] Jerrold Sol Leichter. *Shared Tuple Memories, Shared Memories, Buses and LAN's – Linda Implementations Across the Spectrum of Connectivity.* PhD thesis, Yale University, July 1989.

[32] Wm Leler. Linda meets unix. *Computer*, pages 43–54, February 1990.

[33] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov 1989.

[34] Timothy G. Mattson. The efficiency of linda for general purpose scientific programming. *Scientific Programming*, 3(1):61–71, 1994.

[35] Ronald G. Minnich and Daniel V. Pryor. Mether: Supporting the shared memory model on computing clusters. *38th Annual IEEE Computer Society International Computer Conference (COMPCON SPRING '93)*, 1993.

[36] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[37] Oak Ridge National Laboratory, Oak Ridge, TN 37831. *PVM 3.0 User's Guide and Reference Manual*, February 1993. (http://www.netlib.org/pvm3/).

[38] Santosh S. Pande, Dharma P. Agrawal, and Jon Mauney. Compiling functional parallelism on distributed-memory systems. *IEEE Parallel & Distributed Technology*, pages 64–76, Spring 1994.

[39] Joerg Thomas Pfenning, Achim Bachem, and Ronald Minnich. Virtual shared memory programming on workstation clusters. *Future Generation Computer Systems*, 11(4-5):387–399, Aug 1995.

[40] James Nicholas Pinakis. *Using Linda as the basis of an Operating System Microkernel*. PhD thesis, University of Western Australia, 1993.

[41] Patrick G. Robinson and James D. Arthur. Distributed process creation within a shared data space framework. Technical Report TR 94-13, Virginia Polytechnic Institute and State University, 1994.

[42] G. Schoinas. Issues on the implementation of programming system for distributed applications. (part of POSYBL distribution documentation located at ftp://nic.funet.fi/pub/unix/parallel/POSYBL.TAR.Z).

[43] Benjamin R. Seyfarth, Jerry L. Bickham, and Mangaiarkarasi Arumughum. *Glenda Installation and Use*. University of Southern Mississippi, November 1993. (part of Glenda distribution documentation located at http://sushi.st.usm.edu/ seyfarth/research/glenda.html).

[44] K.H. Shekhar and Y.N. Srikant. Linda sub system on transputers. *Computer Languages*, 18(2):125–136, 1993.

[45] Stanford SUIF Compiler Group. *The SUIF Parallelizing Compiler Guide*. Stanford University, 1994. Version 1.0.

[46] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[47] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, second edition, 1988.

[48] Andrew P. Wack. Practical communication cost formula for users of message passing systems. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 418–425, 1994.

[49] Gregory V. Wilson. *Practical Parallel Programming*. Scientific and Engineering Computation Series. The MIT Press, 1995.

[50] Andrew Xu and Barbara Liskov. Design for a fault-tolerant, distributed implementation of linda. In *Digest of Papers - FTCS (Fault-Tolerant Computing Symposium)*, pages 199–206, 1989.