Distributed Data Structures in Linda

Nicholas Carriero, David Gelernter and Jerry Leichter

Yale University
Department of Computer Science
New Haven, Connecticut

Abstract. A *distributed data structure* is a data structure that can be manipulated by many parallel processes simultaneously. Distributed data structures are the natural complement to parallel program structures, where a *parallel program* (for our purposes) is one that is made up of many simultaneously active, communicating processes. Distributed data structures are impossible in most parallel programming languages, but they are supported in the parallel language Linda and they are central to Linda programming style. We outline Linda, then discuss some distributed data structures that have arisen in Linda programming experiments to date. Our intent is neither to discuss the design of the Linda system nor the performance of Linda programs, though we do comment on both topics; we are concerned instead with a few of the simpler and more basic techniques made possible by a language model that, we argue, is subtly but fundamentally different in its implications from most others.

This material is based upon work supported by the National Science Foundation under Grant No. MCS-8303905. Jerry Leichter is supported by a Digital Equipment Corporation Graduate Engineering Education Program fellowship.

1. Introduction

A *distributed data structure* is a data structure that can be manipulated by many parallel processes simultaneously. Distributed data structures are the natural complement to parallel program structures, where a *parallel program* (for our purposes) is one that is made up of many simultaneously active, communicating processes. Despite this natural relationship, distributed data structures are impossible in most parallel programming languages. Most parallel languages are based instead on what we call the *manager process* model of parallelism, which requires that shared data objects be encapsulated within manager processes; operations on shared data are carried out by the manager process on the user's behalf.

The manager-process model has important advantages. It represents a safe and convenient borrowing from the conventional, sequential environment: Since only one process deals with a given data object, it may do so in the conventional way. Programmers never face the potentially complicated logic of many processes simultaneously manipulating one object. But manager processes have disadvantages as well. All processes in this model must funnel their shared-data manipulations through the manager, and there are potential costs in parallelism and in runtime interprocess communication and process management overhead. Operations that might safely have been carried out by many user processes in parallel are performed by the (single) manager process one at a time; every operation on a shared object entails a conversation with its manager-process chaperone, and creating a new sharable data object requires either the creation of a new process or an increase in the load on an existing manager. Harder to quantify but perhaps of greater importance, the manager-process model prejudices the development of a truly parallel programming style by forcing parallel programs into conventional, sequential molds. We illustrate these claims with examples in the sections following.

Linda ([Gel85], [CG85]) consists of a small set of communication and process-control operators that support the creation and manipulation of distributed data structures. When they are injected into a host language *h*, these operators turn *h* into a parallel programming language. Whether the result is better viewed as a new language or as an old one with added system calls depends on the compiler or pre-processor we use. Linda is a new language to the extent that the compiler, among other things, recognizes the Linda operations, checks and rewrites them on the basis of symbol table information, and can optimize the pattern of kernel calls that result based on its knowledge of constants and loops.

Most of our programming experiments so far have been conducted in C-Linda, but we have recently implemented a Fortran-Linda pre-processor, at the request of Yales's Numerical Analysis group. Linda runs on AT&T Bell Labs' S/Net multi-computer [Ahu83] and on an Ethernet-based Micro-Vax network; implementations for two different hypercube multicomputers[1] are now in design. (The S/Net is a collection of up to sixty-four computer nodes -- currently MS-68000's -- communicating over a fast, word-parallel broadcast bus. On both the S/Net and, of course, on the Vax network, processors are memory-disjoint -- no memory is physically shared among them.)

In the following we outline Linda, then discuss some distributed data structures that have arisen in Linda programming experiments to date. In closing we briefly discuss some related work. Our

[1]A 128-node Intel iPSC and a 64-node cube designed and built by Eric DeBenedictis of AT&T Bell Labs.

the matched tuple itself remaining in TS in either case. `eval(t)` is the same as `out(t)`, except that `eval` adds an *un*evaluated tuple to TS. (`eval` is not primitive in Linda; it will be implemented on top of `out`. We haven't done this yet in S/Net-Linda, so we omit further mention of `eval`.)

The parameters to an `in()` or `read()` statement needn't all be formals. The leading parameter is always an actual, but any others may be actuals as well. All actuals must be matched by corresponding actuals in a tuple for tuple-matching to occur. Thus the statement

```
in("P", int i, 15)
```

may withdraw tuple `("P", 6, 15)` but not tuple `("P", 6, 12)`. When a variable appears in a tuple without a type declarator, its value is used as an actual. The annotation `var` may precede an already-declared variable to indicate that the programmer intends a formal parameter. Thus, if `i` and `j` have already been declared as integer variables, the following two statements are equivalent to the preceding one:

```
j = 15;  in("P", var i, j)
```

Linda's extended naming convention -- it resembles the select operation in relational databases -- is referred to as *structured naming*. Structured naming makes TS content-addressable, in the sense that processes may select among a collection of tuples that share the same first component on the basis of the values of any other component fields. Any parameter to `out()` or `eval()` except the first may likewise be a formal; a formal parameter in a tuple matches any type-consonant actual in an `in` or `read` statement's template.

The implementation and performance of the Linda kernel are discussed briefly in section 4.

3. Programming examples

The distributed data structures we've experimented with so far have occured mainly in the context of replicated-worker parallelism. In *network-style* parallelism (the more common variety), a program is partitioned into n pieces, where n is determined by the logic of the algorithm or the form of the data; each of the n logical pieces is implemented by a process, and each process keeps its attention demurely fixed on its own conventional, local data structures. In the *replicated worker* model, we don't partition our program at all; we *replicate* it r times, where r is determined by the number of processors we have available. All r processes clamber simultaneously over a *distributed* data structure, seeking work where they can get it. The replicated worker model is interesting for a number of reasons:

1. *It scales transparently.* Once we have developed and debugged a program with a single worker process, our program will run in the same way, only faster, with ten parallel workers or a hundred. We need be only minimally aware of parallelism in developing the program, and we can adjust the degree of parallelism in any given run to the available resources.

2. *It eliminates logically-pointless context switching.* Each processor runs a single process. We add processes only when we add processors. The process-management burden per node is exactly the same when the program runs on one node as when it runs on a thousand. (This is not true, of course, in the network model. A network program always creates the same number of processes. If many processors are available, the processes spread out; if there are only a few, they pile up.)

3. *It balances load dynamically, by default.* Each worker process repeatedly searches for a task to execute, executes it, and loops. Tasks are therefore evenly divided at runtime among the available workers.

It is *not* the case that distributed data structures are interesting only in the context of replicated workers. Of the programs we discuss, none is a pure example of this type -- all include some partitioning as well as replication of duties -- and it's easy to describe purely network-style programs that rely on distributed data. But these examples represent our experience to date, and the link between distributed data structures and replicated workers is significant.

The examples discuss two basic classes of distributed structures, *unordered* and then *ordered* ones. Some examples (a distributed array is one) have conventional analogs; others -- for example task bags, broadcast streams or "negative" data structures asssembled out of blocked processes instead of data -- don't.

Unordered structures. It has been remarked[2] that unordered structures like sets seem to have found no natural place in computation. One possible explanation is that, while we can entertain the *abstraction* of an unordered structure, a conventional machine provides no way to represent such a structure. All data stored in a conventional memory is ordered (if only implicitly) by the inherently ordered medium. Since the machine has no way to take advantage of the fact that it is *logically* simpler to store a set than an array or a list, and will always take the trouble to store an array or a list anyway, the programmer might as well be aware of the data ordering and make use of it.

Where distributed structures are involved, though, the situation is different. A structure that exists in many different places simultaneously is in fact unordered (though each "place" -- each local memory -- may be ordered internally). Manager process models, by restricting each data structure to a single process's context, narrow the horizon right back down to the conventional uniprocessor's. Linda does not; it encompasses unordered structures and is a good match to problems that use them.

As a first example of an unordered distributed data structure, consider the task bag. A *task bag* holds next-task assignments in many kinds of replicated-worker programs. Workers repeatedly draw their next assignment from a task bag, carry out the specified assignment and drop any new tasks generated in the process back into the task bag. The program completes when the bag is empty. Note that the tasks in the bag are *unordered* -- we assume that the order in which tasks are processed doesn't matter, so long as they are all done eventually.

A distributed multiset is exactly the right structure for implementing a task bag. It is easily implemented. The elements of the task bag will be tuples of the form

`("Task", `*task-descriptor*`);`

to add a new task workers execute

`out("Task", `*task-descriptor*`),`

and to remove one for processing,

```
in("Task", var NextTask).
```

Note that we can interpret this `in` statement as meaning either "choose a tuple whose first element is "Task", or "choose *any* tuple from a sub-tuple-space named "Task" -- we can regard all tuples with the same prefix as constituting a sub-tuple-space in themselves.

For example: LU matrix decomposition is a problem that we have programmed in several ways using S/Net-Linda; one version of the algorithm illustrates the use of a task bag. These experiments are interesting in their own right for several reasons. LU

[2] Alan Perlis, personal communication.

decomposition is the method of choice for solving dense linear systems. Parallel LU algorithms have non-trivial communication and control requirements. Finally, since there are many ways to solve this problem in Linda, it serves as a good vehicle for exploring parallel programming methodology.

The following is an outline of the LU algorithm, with a modification to aid numerical stability (partial pivoting):

```
for (each column c of matrix M) {
        /* Choose pivot row */
        find row r such that M[r,c] is maximum for
        r in [c..DIM]
        exchange rows c and r;

        for each row r in (c..DIM] {
                /* Reduce row r */
                subtract frow row r a multiple of
                row c such that the c'th
                        entry of row r is 0;
                record the multiplier in L[r,c];
        }
}
```

U is the resulting matrix. (For simplicity we have omitted some details, in particular those pertaining to recording the permutations made to the matrix.)

We lack space to describe in detail the several Linda versions of this algorithm we've tested. In outline, though, one task-bag approach works as follows. We need a control process and one or more workers. The control process fills the task bag with a collection of tuples holding the rows to be reduced on this iteration (one row per tuple), and adds one additional tuple that holds the pivot row. Each worker repeatedly withdraws a row and reduces it, using `read` to check the pivot-row tuple; then it sends the result back to the control process and repeats, until all rows are done. The control process checks the incoming reduced rows as they arrive; when they are all in, it refills the task bag with the rows to be reduced on the next iteration, outputs the new pivot row, and the process repeats.

In timing tests on the S/Net, two workers and a control process finish faster than a version of the same algorithm in uniprocessor C, and the Linda program shows linear speedup up to the small number of processors -- currently 8 -- available to us. Other Linda versions in which rows are distributed only once, at the start of the program, show speedup which is close to ideal linear speedup of the comparable C program -- adding processors results in a progressive doubling, tripling and so on of the C program's speed.

The LU task bag is a rather tame member of the rowdy distributed-data-structure set: All workers withdraw from it, but only the control process refills it. Free-for-all task bags, added-to and deleted-from by every process in the program, occur as well. Consider a program that finds all paths between an arbitrary origin and destination node in a graph. To do so, it creates a bug and places it on the origin node. The bug trudges outwards through the graph, generating new bugs whenever more than one path is available. The program maintains a task-bag of bugs; "bug-trudger" processes (the workers) repeatedly remove a bug, toss it forward one edge, update the bug's path-taken and length-of-path logs, then squash it and generate new bugs for each possible next-edge in the path. The old bug's path-taken and length-of-path logs are implanted in each new bug; new bugs are dropped into the task bag. In outline, then, each bug-trudger process executes

```
loop {
        in("bug", var PathSoFar, var LengthSoFar);
        update Path and Length;
        for each next edge
          out("bug", UpdatedPath, UpdatedLength);
     }
```

The graph itself is a distributed data structure as well. Each node is represented by a tuple of the form

    ("node", *node-id, adjacent-nodes, adjacent-edge-lengths*)

For example,

    ("node", 12, [1,13,14,-1], [30,2,2,-1])

Here we use "-1" to signal the end of a variable-length array[3]. To find node 12's description, we use `read` with a structured name:

    read("node", 12, var AdjacentNodes, var AdjacentEdges).

Note that we can create any number of "bug_trudger" processes with no change to any part of the program except for the process-creating statement itself. Any number of parallel processes can safely share direct access to the task queue and the graph.

The task-bag model generalizes to a dataflow model. The task bag holds templates, one template per tuple; each template includes "ready" and id fields (and any others that are needed). Workers are general evaluators; they use structured naming to select "ready" templates, execute the specified task step, and update any templates to which a newly-developed value should be propagated. A template is marked "ready" by the worker that fills its last value slot. Thus:

```
loop {
        in("DataFlowBag", "ready", var id,
           other-task-describing-fields);
        execute the task step;
        for each template N to be updated {
          in("DataFlowBag", var flag, N, ... );
          if N is now ready
           out("DataFlowBag","ready",N,updated_fields);
          else
           out("DataFlowBag","not-yet",N,updated_fields);
         }
        }
```

We are using this control mechanism to develop in collaboration with researchers at AT&T Bell Labs a parallel VLSI simulator in Linda. In one version of the simulator, each node in the circuit graph is represented by a task-description tuple, where a task-description consists of a pointer to the function that simulates this node and a series of slots for input values to the function. A task-description with all of its slots filled in is "ready". This Linda solution will only run well, of course, if the average execution time for each task step is sufficiently long relative to an `out` or an `in` to amortize communication overhead. We don't claim by any means that this approach will work for arbitrarily fine degrees of parallelism. Preliminary evidence suggests that the simulator, in which an average task step requires about 9 ms. will meet our coarseness criterion; we can't yet say for sure. If not, (1) there are other replicated worker solutions to this problem which require fewer TS operations, and we'll try those, (2) future versions of the S/Net architecture, now in design, will allow our kernel to run faster, and make the grain-size of the dataflow-style program we can handle correspondingly finer.

As a final example, consider Linda's tuple space itself: it is a data structure whose semantics are those of an unordered collection or multiset. The semantics of tuple space reflect a physically unordered underlying structure. The details of the structure depend on the particular implementation, but is is assumed always to be unordered.

In S/Net Linda, for example, tuple space is stored as a replicated set of hash tables. Each processor stores all tuples in tuple space, but the order in which replicated tuples are stored varies from processor to processor. The replication has both advantages and costs. Other implementations may store tuples only once. Bjornson [4] has proposed a *distributed hash table*: a hash code computed from a tuple determines a unique node which will have responsibility for that tuple. [Gel84] proposes an implementation, appropriate on hyper-cube-shaped communication networks, in which each tuple is stored by all members of some row or sub-cube of each node. See [Leic85] for a further discussion of these and related algorithms.

[3]The bug-trudger we implemented used strings instead of arrays for "adjacent-nodes" and "adjacent-edge-lengths".

[4]Rob Bjornson, personal communication.

What all these algorithms share is an actual physical distribution of the data in tuple space, and the lack of any ordering on tuple space elements. No iterator over tuple space is provided by Linda; no iterator is present in the support kernel; and if a Linda program iterates over tuples -- by simply doing repeated in's -- it can expect to see tuples in different orders at different nodes[5].

Ordered structures

We can build distributed versions of conventional ordered structures like arrays in tuple space by including index fields in tuples and using structured names to pick elements out. One of our experiments with S/Net Linda, for example, involved a matrix multiplication program that consisted of an initialization process, a cleanup process, and at least one but ordinarily many worker processes. Each worker is repeatedly assigned some element of the product matrix to compute; it computes this assigned element and is assigned another, until all elements of the product matrix have been filled in. If A and B are the matrices to be multiplied, then the initialization process uses a succession of out statements to dump A's rows and B's columns into TS. When these statements have completed, TS holds

```
("A", 1, A's-first-row)      ("B", 1, B's-first-column)
("A", 2, A's-second-row)     ("B", 2, B's-second-column)
    ...                           ...
```

Indices are included as the second element of each tuple so that worker processes, using structured naming, can select the *ith* row or *jth* column for reading. Workers repeatedly in a NextAssigmnent tuple that holds the row and column of the product element to be computed next, then out it again, suitably incremented, for some other worker to find. The row and column read from the NextAssignment tuple are used to select the appropriate vectors and to label the product tuple: to compute element (i, j) of the product, a worker executes

```
read(A, i, var row);
read(B, j, var col);
out(prod, i, j, DotProduct(row, col));
```

Thus each element of the product is packed in a separate tuple and dumped into TS. (Note that the first read statement picks out a tuple whose first element is "A" and second is the value of i: this tuple's third element is assigned to the formal row.)

Although the product matrix is stored in tuple space as an unordered structure, a cleanup process may treat the product as a bag and reel in its elements in arbitrary order:

```
for (row = 1; row <= NumRows; row++)
  for (col = 1; col <= NumCols; col++) {
    in(prod, var row, var col, var result);
    prod[row][col] = result;
  }
print prod;
```

(A similar program's performance on the S/Net resembles the LU program's performance; again versions with less communication exist. For example, we had each worker compute an entire row at a time; the resulting Linda program's speedup is close to ideal linear speedup over uniprocessor C [CG85].)

A distributed tuple space array is closely analogous to a conventional array (although it may, of course, expand and contract dynamically, and it is accessible to many processes simultaneously). Ordered distributed data structures are also possible that have no close sequential analogs. Consider the following programming problems. In a distributed mailer utility, any number of user processes may wish to keep abreast of a continually-updated bulletin board. In a simulator program, we might create a set of table-management processes, each one of which stores one partition of a global state table. Worker processes send state updates to the table processes, and each table processe needs to scan every update in order to determine what it should ask workers to do next. (Such a structure arises in one version of the VLSI simulator mentioned previously.)

We can handle these problems in Linda by using a distributed data structure called a *broadcast stream*. Consider a stream called S; to implement S, we need an S-counter tuple,

```
("S", "counter", current-val)
```

where *current-val* is initially 0. In appending to S, processes manipulate the S-counter using the utility routine IncrCount, in outline

```
IncrCount(N)
{
    in(N, "counter", int i);
    out(N, "counter", i + 1);
    return i;
}
```

Stream S exists as a numbered series of tuples:

```
("S", 0, Val0), ("S", 1, Val1), ("S", 2, Val2) ...
```

To append *NewVal* to S, a process executes

```
out("S", IncrCount("S"), NewVal)
```

Any number of processes may share append-access to S; since the end-of-stream counter inhabits a single tuple, the count is maintained consistently no matter how many processes try to update it simultaneously.

A process that wishes to *read* the broadcast stream will execute

```
index = 0;
loop {
        read("S", index++, var NewVal);
        ...
    }
```

The first time read is executed, it awaits a tuple whose first component is "S", second is 0 and third is any value that is type-compatible with NewVal; the second read awaits ("S", 1, ...), the third ("S", 2, ...) and so on. So data elements are read in stream order, as desired.

The broadcast stream structure has interesting properties. It resembles in some ways a plain, naive broadcast, in which senders regularly flash messages to a set of receivers: but there are important differences. The broadcast stream's readers are completely decoupled from its writers: each reading process proceeds entirely at its own pace, without having to keep up with the writing processes and without reference to any other reader. If a reading process consumes messages faster than the writing processes append them, it repeatedly blocks pending the next message's arrival -- as it would be expected to do under naive broadcast. But a reading process may also scan the stream and receive each broadcast in sequence long after every writing process has terminated.

It's also interesting to note, again, that the broadcast stream is a fully distributed data structure: Any number of cooperating processes may read it, any number may append to it, and no one process is any more responsible for its storage and upkeep than any other.

If processes in the stream's head tuple instead of reading it (using a second shared counter for coordination), the *broadcast stream* becomes a *task queue*. We can append and remove using other ordering disciplines as well -- a task *stack* is as easy as a task queue, for example.

There is an interesting symmetry between the templates that are parameters to in and the tuples that are parameters to out; tuples and templates are structurally identical. This fact has implications for the implementor: tuples can be dispatched through the network searching for matching templates, or templates can be sent in search of tuples. Of concern here are the logical implications of the symmetry, the fact that we can think of distributed data structures that are constructed out of *templates* as well as tuples. Consider a replicated-worker task bag program in a state in which all workers but one are blocked awaiting a task tuple. As the workers wait at their

```
in("Task", var TaskDescriptor)
```

[5]Note that two simultaneous-executing processes executing such a series of in's would actually split the matching tuples between them in some unpredictable way. Allowing each to see all the tuples requires some additional structure. The broadcast stream structure discussed below is an example.

statements, we may think of the `("Task", var TaskDescriptor)` templates as constituting a "negative task bag" -- a bag of blocked processes.

We may build ordered sets of blocked processes as well (ordered "negative distributed data structures"). We can use such a negative structure to build an unconventional solution to the classic readers-writers problem. Suppose many processes share access to a complex data object which is too large, we assume, to be conveniently stored in a single Linda tuple. Processes are permitted to access the shared object directly, but only after waiting until it is permissible for them to do so. The rules of access specify that many readers or a single writer may have access to the data, but not both; a constant stream of read-requests must futhermore not be allowed to postpone satisfaction of a write request indefinitely, nor may a stream of write-requests indefinitely postpone reading.

The simplest and clearest way to solve this problem is to to append each new read or write request to the end of a single queue. If the queue's head request is a read request, the requestor is permitted to proceed as soon as no writer is active; if the head request is a write request, the requestor may proceed as soon as neither readers nor a writer are active. When a reader or writer is given permission to proceed, its request is removed from the head of the queue. The requesting process reads or writes directly, and notifies the system when it is done.

The following is this solution as it is expressed in Ada (from [Hab83]). The Ada version is a typical manager process solution. When user processes need to read or write, they send a "START(read)" or "START(write)" request to a manager process called "RWScheduler"; when they are done, they inform the manager by executing "STOPREAD" or "STOPWRITE". Thus a reader process executes

```
START(read);
read;
STOPREAD;
```

and a writer similarly. User messages are queued automatically in FIFO order for inspection by RWScheduler. There is one queue for "START" requests, one for "STOPWRITE" notifications and one for "STOPREAD" notifications. The manager uses its internal state variables "nrreaders" (the number of active readers) and "writing" (TRUE if there is an active writer) to accept requests from these queues, and thereby allow the requesting processes to proceed at the logically correct points.

```
task body RWscheduler is
    nrreaders: INTEGER := 0;
    writing: BOOLEAN := FALSE;
begin
    loop
        select
            when not writing =>
            accept START(r: in request) do
                   if r = read then
                          nrreaders := nrreaders + 1;
                   else
                          for i in 1 .. nrreaders loop
                                accept STOPREAD;
                          end loop;
                          nrreaders := 0;
                          writing := TRUE;
                   end if;
            end START;
        or
            when writing =>
            accept STOPWRITE do
                writing := FALSE
            end STOPWRITE;
        or
            when not writing =>
            accept STOPREAD do
                   nrreaders := nrreaders - 1;
            end STOPRREAD;
        end select;
    end loop;
end RWScheduler;
```

The Linda solution implements the same algorithm in an entirely different way. Linda reader processes again execute

```
startread();
read;
stopread();
```

and writers similarly. But no manager process administers a request queue. Instead, the queue is a negative distributed data structure -- an ordered set of templates, or blocked processes. A process queue may, in other words, be distributed just as a data queue may; we can assemble blocked processes into the same range of structures that are possible with tuples.

To implement the queue we use two counters, `rw-head` and `rw-tail`. When a process needs to read or write, it consults (and increments, using the `IncrCount` routine discussed above) the value of the `rw-tail` counter; suppose j is the value returned. The requesting process now waits until the value of the `rw-head` counter is j. When it is, this requestor is first on line and will be the next process permitted access to the shared data object. The process queue exists as a numbered series of templates, each corresponding to a blocked process:

("rw-head", n), ("rw-head", n+1), ("rw-head", n+2)...

When the value of "rw-head" reaches n, the head template is matched and the head process continues; when it reaches n+1 the next blocked process continues, and so on. We use two other counters to allow the head process to determine when it is safe to take the plunge and start reading or writing. Before proceeding, the head process must wait for the value of a "writers" counter to be 0 (at which point no writers are active), and, if it is itself a writer, it must also wait for a "readers" counter to be 0 as well. Finally it increments either the "readers" or the "writers" counter (depending on whether it intends to read or write), and increments the "rw-head" counter to give the next waiting process a chance to await access:

```
startread()
{
    read("rw-head", IncrCount("rw-tail"));
    read("writers", 0);
    IncrCount("readers");
    IncrCount("rw-head");
}
```

`startwrite` differs only in waiting for *both* `readers` and `writers` to be zero, and in incrementing `writers` in place of `readers`:

```
startwrite()
{
    read("rw-head", IncrCount("rw-tail"));
    read("readers", 0); read("writers", 0);
    IncrCount("writers");
    IncrCount("rw-head");
}
```

`stopread` and `stopwrite` are simply

```
stopread()
{
    DecrCount("readers")
}

stopwrite()
{
    DecrCount("writers")
}
```

The read-write request queue exists in the Linda solution as it does in the Ada solution -- but in Linda, the queue is a *distributed* data structure instead of a conventional non-distributed one. The program is simpler at runtime as a result: no manager process and no user-manager conversations are needed. (Linda requires each process to execute several TS operations, of course, but these are simpler than message-exchange operations -- they involve the user process and the kernel, not a sender, the kernel and a receiver.) Linda's readers-writers code is also shorter and simpler than Ada's. Note that Linda's runtime advantage grows with the number of objects to be managed. Suppose many objects are independently readable and writeable: providing more Ada-style manager processes increases the process-maintenance and inter-process communication burden; keeping manager-processes constant and giving each more to do increases each manager's bottleneck potential.

Note finally that we can build arbitrary linked data structures in tuple space as well as sequential ones. The language Symmetric Lisp [Gel85a], a high-level parallel programming language meant to be implemented using Linda primitives, allows any number of processes simultaneous access to distributed lists; we can implement a *cons* cell with a tuple by using logical names where a conventional implementation uses physical addresses. Thus

```
(C1, "cons", C17, C23)
```

is a *cons* cell whose *car* is the tuple beginning C17 and *cdr* is C23.

4. The S/Net implementation

Linda has often been regarded as posing a particularly difficult implentation problem. The following paragraphs summarize the way in which we implemented Linda on the S/Net, for the edification of the curious. The S/Net implementation is discussed in detail in [CG85].

Our implementation buys speed at the expense of communication bandwidth and local memory; the reasonableness of this trade-off was our starting point. (Variants are possible that are more conservative with local memory [Leic85].)

Executing out(*t*) causes tuple *t* to be broadcast to every node in the network; every node stores a complete copy of TS. Executing in(*s*) triggers a local search for a matching *t*. If one is found, the local kernel attempts to delete *t* network-wide using a procedure we discuss below. If the attempt succeeds, *t* is returned to the process that executed in(). (The attempt fails only if a process on some other node has simultaneously attempted to delete *t*, and *it* has succeeded). If the local search triggered by in(*s*) turns up no matching tuple, all newly-arriving tuples are checked until a match occurs, at which point the matched tuple is deleted and returned as before. read() works in the same way as in(), except that no tuple-deletion need be attempted -- as soon as a matching tuple is found, it is immediately returned to the reading process.

The delete protocol must satisfy two requirements: All nodes must receive the "delete" message; if many processes attempt to delete simultaneously, only one must succeed. The manner in which these requirements are met will depend, of course, on the available hardware.

When some node fails to receive and buffer a broadcast message, a negative-acknowledgement signal is available on the S/Net bus. One possible delete protocol has two parts: The sending kernel re-broadcasts repeatedly until the "negative acknowledgement" signal is *not* present. It then awaits an "ok to delete *t*" message from the node on which *t* originated. In this protocol the kernel on the tuple's origin node is responsible for allowing one process, and only one, to delete it. (We have implemented other protocols as well. Processes may use the bus as a semaphore to mediate multiple simultaneous deletes, for example, and avoid the use of a special "ok to delete" message.)

In order to estimate the time required to perform in's and out's we ran the following programs on separate processors:

```
PING:
count = 0;
while(TRUE) {
        in("ping");
        if (++count == LIMIT) break;
        out("pong");
}
print elapsed time;

PONG:
while(TRUE) {
      out("ping");
      in("pong");
}
```

Since we wanted to measure basic communication cost, we moved calls to Linda support routines out of the loops. This is equivalent to assuming the existence of a compiler that is able to recognize that the strings "ping" and "pong" are constants, and that these support routines (make_ptp and ptp_tb) return constant values when called with constant arguments[6]. Elasped time was measured using the 68000's clock via routines supplied by the existing operating system. Evidence from this test suggests that a minimal out-in transaction, from kernel entry on the out side to kernel exit on the in side, excluding -- as noted -- the cost of packetizing, takes about 1.4 msec. Other experiments support these general figures.

As a communications kernel for a bus-connected network, S/Net-Linda falls generally within the same category as several others that have been reported in recent years. Examples include the Birrell and Nelson RPC kernel [BN84], Cheriton and Zwaenpoel's V Kernel [CZ83] and Spector's Remote Operations kernel [Spec82], among others. Linda differs fundamentally from all three in what it offers the user: The V kernel provides RPC-like synchronous message passing (in addition to an efficient inter-node file transfer service), and Spector provides flexible systems-level protocol-construction tools to the systems programmer. It is nonetheless worth pointing out that Linda's performance, allowing for all the obvious incomparabilities, is roughly in league with the others (assuming the software and not the microcoded version of Spector's system). In the V kernel, the synchronous send of a short message, from send-message until the sender receives a reply, requires 2.56 ms[7]; a generally comparable operation in Linda -- the sender executes an out to send parameters, then an in to retrieve results -- requires roughly 2.6 ms with a null message. (The figure for short messages is about the same). Birrell and Nelson's reported 1.1 ms for remote invocation of a procedure of no arguments that returns no results -- the figure represents elapsed time from invocation through remote procedure execution and return -- is a little over twice as fast as Linda and V; but Linda and the V Kernel both run on MC-68000's, the RPC kernel on the much faster Dorado.

5. Conclusions.

Most parallel languages and programming systems don't support distributed data structures. Some do, and numerous experiments with replicated-worker programs have been reported -- for example on the BBN Butterfly [Deu84], the NYU Ultracomputer [Gott83] and the Denelcor HEP [Mull84]. But these systems have relied as a rule on the existence of physically shared memory among processor nodes, and have supported distributed data structures only by means of relatively low-level system calls. Parallel languages have in fact been proposed (Qlambda, for example [GM84]) that depend explicitly on the existence of a shared-memory parallel architecture, and *still* rely on manager processes or Hoare monitors instead of on distributed data structures in Linda's sense. Manager processes are good tools in many cases -- but our experiments to date have led us to hypothesize that distributed data structures are an inherently better match to many kinds of parallel algorithm. At the very least they are an alternative that, although it has been largely ignored in parallel-language work to date, is worth investigating.

[6]This is conceptually equivalent to a compiler recognizing that subscripting with a constant subscript produces a constant address.

[7]A second paper [CZ85] quotes a higher figure for a modified system, but the lower number reflects a kernel that is closer to ours.

References

[Ahu83] S. Ahuja, "S/Net: a high-speed interconnect for multiple computers," *IEEE Selected Areas in Communication*, (Nov. 1983):751-756.

[BN84] A.D. Birrell and B.J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comp. Sys.* 2,1 (1984):39-59.

[CG85] N. Carriero and D. Gelernter, "The S/Net's Linda kernel," in *Proc. Symp. Op. Sys. Principles* (Dec. 1985) (Program Committee *ACM TOCS nominee*) (to appear).

[CZ83] D.R. Cheriton and W. Zwaenepoel, "The distributed V kernel and its performance for diskless workstations," in *Proc. Ninth ACM Symp. Op. Sys. Princ.* (Oct. 1983):128-139.

[CZ85] D.R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Trans. Comp. Sys.* 3,2(1985):77-107.

[Deu84] J.T. Deutsch and A.R. Newton, "MSPLICE: A multiprocessor-based circuit simulator," in *Proc. 1984 Intl. Conf. Parallel Processing,* (Aug. 1984):207-214.

[Gel84] D.Gelernter, "Dynamic global name spaces on network computers," in *Proc. Intl. Conf. Parallel Processing,* (Aug. 1984).

[Gel85] D. Gelernter, "Generative communication in Linda," *ACM Trans. Prog. Lang. Sys.* 1(1985):80-112.

[Gel85a] D. Gelernter, "Symmetric Programming Languages," Yale Univ. Dept. Comp. Sci. Tech. Report yaleu/dcs/rr#253, Aug. 1985.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran and S. Chang, "Parallel programming in Linda," in *Proc. Int. Conf. Parallel Processing,* (Aug. 1985) (to appear).

[GM84] R.P. Gabriel and J. McCarthy, "Queue-based multi-processing Lisp," in *Proc. ACM Symp. Lisp. and Funct. Prog.* (1984):25-44.

[Gott83] A. Gottlieb, R. Grishman, C.P. Kruksal, K.P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comput.* C-32, 2(1983):175-189.

[Leic85] J. Leichter, "Implementing the Unimplementable -- Algorithms for Linda's Tuple Space". Yale University Dept. Comp. Sci. internal report.

[Mull84] R. Mullin, E. Nemeth and N. Weidenhofer, "Will public key crypto systems live up to their expectations? HEP Implementation of the discrete log codebreaker," in *Proc. 1984 Intl. Conf. Parallel Processing,* (Aug. 1984):193-195.