The S/Net's Linda Kernel

NICHOLAS CARRIERO and DAVID GELERNTER
Yale University

Linda is a parallel programming language that differs from other parallel languages in its simplicity and in its support for distributed data structures. The S/Net is a multicomputer, designed and built at AT&T Bell Laboratories, that is based on a fast, word-parallel bus interconnect. We describe the Linda-supporting communication kernel we have implemented on the S/Net. The implementation suggests that Linda's unusual shared-memory-like communication primitives can be made to run well in the absence of physically shared memory; the simplicity of the language and of our implementation's logical structure suggest that similar Linda implementations might readily be constructed on related architectures. We outline the language, and programming methodologies based on distributed data structures; we then describe the implementation, and the performance both of the Linda primitives themselves and of a simple S/Net-Linda matrix-multiplication program designed to exercise them.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.4 [**Operating Systems**]: Communication Management—*message sending*.

General Terms: Languages

Additional Key Words and Phrases: Parallel programming languages

1. INTRODUCTION

A parallel programming language is a language that supports process-forking and interprocess-communication (in one form or another) in addition to the normal computation and control operations that all programming languages need. Parallel languages are tools for parallel programming, and parallel programming in turn is useful in two ways. In domains where logically-concurrent algorithms are available (numerical problems, system simulation and AI are three such domains) it is a technique for making programs run faster. On local area networks, it is a method for constructing integrated operating systems and distributed utilities.

Linda [13] consists of four simple operators that, when injected into a host language *h*, turn *h* into a parallel programming language. A Linda-based parallel language is in fact a new language, not an old one with added system calls, to the extent that the Linda compiler or preprocessor recognizes the Linda operations,

This work was supported by the National Science Foundation, Grant No. MCS-8303905.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1986 ACM 0734-2071/86/0500-0110 $00.75

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, among other things. (Most of our programming experiments so far have been conducted in C-Linda, but we have recently implemented a Fortran-Linda preprocessor as well, at the request of Yale's Numerical Analysis group.) The S/Net [1] is a multicomputer that can also function as the backbone of a local area net. Each S/Net is a collection of not more than sixty-four memory-disjoint computer nodes communicating over a fast, word-parallel broadcast bus. We have implemented a Linda-supporting communication kernel on an S/Net at AT&T Bell Laboratories (where the machine was designed and built). This implementation is of interest—we will argue—for two reasons. It demonstrates, first, that Linda's usually powerful and flexible communication primitives can be made to run well; the language's shared-memory-like semantics can in fact be supported efficiently in the *absence* of physically shared memory. Second, although Linda and the S/Net are particularly well-matched, the simplicity of the language *and* of the implementation's design *and* of the S/Net logical structure suggest to us that Linda implementations like ours might readily be constructed on similar architectures elsewhere. Such implementations would promise, as ours does, to synthesize some of the advantages of shared-memory programming on the one hand and of unshared-memory network architectures on the other.

We have argued elsewhere that Linda's operators are in many cases substantially more powerful and expressive than comparable ones in other languages; that Linda is often cleaner, simpler and easier to use; and that the distributed data structures Linda supports and most other parallel languages forbid are often the most natural complement to distributed algorithms. In Section 2 we outline the language and briefly rehash some of these arguments. We compare Linda in particular to the remote procedure call protocol, which is of special interest because it has become the most widely discussed interprocess communication technique, and has been implemented, tested and used (Birrell and Nelson [3]). In Section 3 we describe our general strategy for implementing Linda on bussed networks, and in Section 4 we discuss the S/Net implementation specifically. Section 5 presents performance results, and Section 6, conclusions.

2. LINDA

Processes in Linda communicate through a globally-shared collection of ordered tuples called tuple space or TS. The four operators that Linda provides (1) add tuples to this shared collection, (2) remove tuples, (3) read tuples, (4) add unevaluated tuples whose evaluation begins as soon as they enter tuple space.

The four operations defined over TS are `out( )`, `in( )`, `read( )` and `eval( )`. `out(`*t*`)` causes tuple *t* to be added to TS; the executing process continues immediately. `in(`*s*`)` causes some tuple *t* that matches template *s* to be withdrawn from TS; the values of the actuals in *t* are assigned to the formulas in *s*, and the executing process continues. If no matching *t* is available when `in(`*s*`)` executes, the executing process suspends until one is, then proceeds as before. If many matching *t*'s are available, one is chosen arbitrarily. `read(`*s*`)` is the same as `in(`*s*`)`, with actuals assigned to formals as before, except that the

matched tuple remains in TS. `eval(t)` is the same as `out(t)`, except that `eval` adds an *unevaluated* 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`.)

For example: executing

```
out("P", 5, false)
```

causes the tuple `("P", 5, false)` to be added to TS. The first component of a tuple serves as a logical name, here `"P"`; the remaining components are data values. Subsequent execution of

```
in("P", int i, bool b)
```

might cause tuple `("P", 5, false)` to be withdrawn from TS; 5 would be assigned to `i` and `false` to `b`. Alternatively it might cause any other matching tuple—any other, that is, whose first component is `"P"` and whose second and third components are an integer and a Boolean respectively—to be withdrawn and assigned. Executing

```
read("P", int i, bool b)
```

when `(P, 5, false)` is available in TS may cause 5 to be assigned to `i` and `false` to `b`, or equivalently may cause the assignment of values from some other type consonant tuple, with the matched tuple itself remaining in TS in either case.

The parameters to an `in( )` or `read( )` statement need not all be formals. Any actuals among them 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)`. This 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.

2.1 Linda versus Remote Procedure Call

The remote procedure call model—to communicate with R, process Q sends invocation parameters to an entry in R and then blocks until the remotely-invoked entry sends result parameters back—is ubiquitous in parallel-language work, for an obvious reason. Since procedure invocation is the best tool for intraprocess communication, it seems natural to propose it as the basis for communication *between* processes as well. The Ada entry call [8] and the Qlambda process-closure invocation [10] are both variants of this basic protocol, and

Birrell and Nelson implement remote procedure call in its pure form.[1] It is obviously of interest, then, to compare Linda to remote procedure call.

RPC can easily be implemented on top of Linda. The invoker uses `out` to send invocation parameters and a subsequent `in` to retrieve results, including a unqiue name (supplied by the system as the value of `"me"`) as a parameter in the `out` tuple; the remote procedure prefaces the result tuple with this name. Thus the invoker executes

`out(ProcName, me,` *invocation-params*`); in(me,` *result-params*`).`

Linda, of course, can be implemented on top of RPC as well[2]—so which kernel is preferable, where should we start? Obviously there is no absolute answer, but it may not be obvious that, for better or worse, Linda lends itself to a style of programming that is *different* from RPC style. Linda programs exist that would be so awkward under RPC that they are effectively ruled out of consideration. These idiomatic Linda programs depend, by and large, on *distributed data structures*—data structures that may be manipulated by many parallel processes simultaneously. Such structures are illegal in most remote-procedure-call languages, which require instead that shared data objects be encapsulated in manager processes; operations on shared objects are carried out, on request, by the manager process on the user's behalf. Manager processes are safe and reasonable, but there are important cases where distributed data structures seem by far more natural and more efficient; a useful parallel language (to our way of thinking) will support both.

Distributed data structures are interesting for a number of reasons. In the manager-process model, all processes must funnel their shared-data manipulations through the manager, and there are potential costs in parallelism and in run time 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. It would not be surprising if distributed data structures proved, in many cases, to be the most natural compliment to distributed algorithms.

---

[1] The CSP-Occam output statement [15, 16] is another remote-procedure-call variant of sorts: executing a CSP output statement forces the sender to suspend until its message is received by the target process; the target process may thereupon return only an "okay-to-proceed" synchronization signal to the sender, where a remotely-invoked procedure would have been free to return whatever it liked. In any case, both the remote invocation and the CSP output statement deliver a message to an explicitly-designated receiver and suspend until the receiver gets it, both of which distinguish them from Linda.

[2] With (most likely) prohibitive inefficiency, however: we would need to provide a central server to manage tuple space.

The simple matrix-multiplication program whose performance is discussed in Section 5 is a good illustration. The program consists 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 specifically—*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, "row", A's-first-row)    ("B", 1, "col", B's-first-column)
("A", 2, "row", A's-second-row)   ("B", 2, "col", 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. The initializer then adds the tuple

```
("Dot", 1, dim, "A", "B", "C")
```

to TS, and terminates. Here 1 indicates the next element to be computed, `dim` is the dimension of the input matrices, `"A"` and `"B"` are the inputs and `"C"` is the name of the product.

Each worker process repeatedly decides on an element to compute, then computes it. To select a next element, the worker removes the "Dot" tuple from TS, determines from its second field the indices of the product element to be computed next, and reinserts "Dot" with an incremented second field:

```
in("Dot", var NextElem, var dim, var mat1, var mat2, var
   prod);
if (NextElem < dim*dim)
  out("Dot", NextElem + 1, dim, mat1, mat2, prod);
i = (NextElem - 1)/dim + 1;
j = (NextElem - 1)%dim + 1;
```

The worker will now proceed to compute the product element whose index is (i, j). Note that if (i, j) is the *last* element of the product matrix, the "Dot" tuple is not reinserted. When the other workers attempt to remove it, they will block. A Linda program terminates when all processes have terminated or have blocked at `in` or `read` statements. (Blocked workers can also be restarted on a new problem simply by dropping in a new "Dot" tuple.)

To compute element (i, j) of the product, the worker executes

```
read(mat1, i, var row);
read(mat2, 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.)

The cleanup process reels in the product-element tuples, installs them in the result matrix prod, and prints prod:

```
for (row = 1; row ⇐ NumRows; row++)
  for (col = 1; col ⇐ NumCols; col++)
    in(prod, row, col, var prod[row][col]);
print prod;
```

This simple program was easy to write and strikes others (in our limited experience) as easy to understand. It has the nice property of scaling transparently to accommodate any number of worker processes; it might be developed and debugged with a single worker, but thereafter it will run just as well with ten workers or a hundred. Unlike the "quasi-systolic" matrix program described by Shapiro [17], it does *not* require the system to fork a new process for every element in the result matrix; the user creates as many processes as seem reasonable given the available resources at any particular run. Finally, as we discuss in Section 5, it performs well. Note that many types of algorithms may be programmed within this general task-queue model [14].

This little matrix program illustrates many of the important differences between distributed data structures and manager processes. *The input matrices are distributed data structures*; all worker processes may read them simultaneously. In the manager-process model, processes would send read-requests to the appropriate manager and await its reply. *The "Next" tuple is a distributed data structure*: all worker processes share direct access to it. In the manager process model, again, worker processes would read and update the "Next" counter indirectly via a manager. *The product matrix is a distributed data structure*, which all workers participate in building simultaneously.

Notice how poor a conceptual model procedure-invocation provides for the interraction between the workers and the cleanup process. Workers don't need to suspend processing, each time they send a message to the printer, until the message is received. To do so would be a waste of time. Of course we can fix this problem, in the remote-procedure context, by forking a proxy process to perform the remote call and be suspended until it returns, or by providing a special asynchronous-invocation operator, as Qlambda does. Neither fix addresses the conceptual problem: remote-procedure advocates rely on the naturalness of their model, but the relationship between the workers and the cleanup process is *not* that of a caller to a callee. The same holds of interprocess relationships in many other simple parallel program structures—consider the relation between one segment and the next in a parallel pipeline, for example.

Remote procedure calls (as noted) are simple to build in Linda. What is interesting, though, is the number of cases in a single short program where they are not needed and not wanted—where direct dealings with a distributed structure are at least as natural and efficient, and arguably much more so. And note that matrix multiplication was singled out, not for anything special in its relationship to Linda, but simply because it was a readily-understood application, easy to write and to test.

3. LINDA ON BUSSED NETWORKS

Linda has been widely regarded as posing a difficult implementation problem. Thus Andrews and Schneider [2], after noting that Linda's global-naming scheme is well suited to programming client-server interactions (as discussed in [11]), write as follows in discussing the TS operations:

> Unfortunately, implementing mailboxes can be quite costly without a specialized communications network ... When a message is sent, it must be relayed to all sites where a **receive** could be performed on the destination mailbox; then, after a message has been received, all these sites must be notified that the message is no longer available for receipt. [9]

While the authors misunderstand the nature of Linda's primitives (which resemble mailbox operations only superficially), their misgivings are easy to credit; the distributed, globally-accessible character of tuple space allows a naive implementation great latitude for running poorly. Fortunately, on the S/Net, as on most bus and ring interconnects, it takes no longer to send to all n network nodes than to one. Thus it is in fact no more time-consuming (in the sense of elapsed clock time) to relay a tuple "to all sites where a **receive** could be performed" than it is to relay a message to any single site, nor is it more time-consuming to inform *all* sites that a tuple has been received, and should be deleted, than it is to inform the sender alone. There are indeed inherent costs in implementing Linda instead of plain send-message and receive-message, notably because the broadcast protocols we use require *every* processor on the bus to handle every message. On networks where there are no front-end communication processors (like our current S/Net), processors are therefore interrupted far more frequently than they would be under plain message-passing—but we *still* get good performance from our kernel. (These interrupt-handling costs will in any case largely disappear on the next-generation S/Net, now being tested; it provides communication coprocessors to absorb bus interrupts.)

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; we discuss one below.)

In the simplest version of the scheme—the version we implemented—executing `out(`t`)` causes tuple t to be broadcast to every node in the network; thus 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 later; 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 returned immediately 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. The first delete protocol we implemented therefore had two parts: the sending kernel rebroadcasts a "delete t" message repeatedly until the negative acknowledgement signal is *not* present; it then awaits a message, reliably transmitted point-to-point from the node on which t originated, informing it either that "t has been assigned to you: proceed," or "t has not been assigned to you: wait." In this protocol, then, the kernel on the tuple's origin node is responsible for allowing one process, and only one, to delete it.

The following is an informal argument that this protocol works correctly. We know, once node k has completed its "broadcast the delete message" phase, that every node in the network has been informed that t is gone, no longer available either for reading or removing. If some node *hadn't* gotten the word (if it had failed to receive k's broadcast message), a negative acknowledgement would have been present on the bus and k would have retried the broadcast. (Note that delete messages are idempotent—instructions to delete a tuple that isn't there are ignored.) Given that all nodes have been informed that t should be deleted, suppose k needs to rebroadcast its delete message several times, and the origin node's response arrives *during* this period—will `in` complete correctly? It will, so long as we insure that the kernel is prepared to respond properly to arbitrary messages received during the delete phase, and then to resume the delete phase—which we have done. Suppose other nodes are simultaneously attempting to claim t—will only one of them be told to proceed? Yes. The origin node can easily make sure that it returns only one "proceed" message for each tuple in its custody. Will at least one node receive a "proceed" message? We can be sure that the origin node received the delete request, because if it hadn't, a negative acknowledgment would have been raised during phase one. The "proceed" message itself is sent via a reliable point-to-point protocol: It is retried until the intended receiver gets it.

We have noticed that broadcast failures on the S/Net are very rare; in fact, we've never seen one. If broadcast were *necessarily* reliable, the delete protocol would be simpler: a "delete t" message is broadcast; if the broadcasting node reads its own message back off the bus with no other "delete t" message intervening, the delete attempt has succeeded. If some other node's "delete t" arrives first, the attempt fails. When several nodes attempt to delete simultaneously, in other words, the kernel that grabs the bus first succeeds. We have used this procedure in combination with our ability to detect failure to develop a second delete protocol. We assume that broadcast is reliable, and use "reliable bus" delete; a failed-broadcast signal, should one ever occur, triggers the execution of a higher-level (potentially complicated) recovery routine.

The protocols outlined above depend heavily on the availability of a negative broadcast acknowledgment on the S/Net bus. We are now in the process of completing a Linda kernel for an Ethernet-based Micro-Vax network, where there is no such signal. One approach in this new environment therefore requires (1) that each node remember the sequence number of the last message received from each other node, and send a back-order request if and when it notices a gap (this simple message-logging scheme is related to the technique described by Chang and Maxemchuck [5]); (2) that `reads` as well as `ins` get clearance from

the tuple's origin node—in the case of `read`, if node *k* misses a delete message, a tuple that it believes is available may already have been deleted; only the tuple's origin node (which alone may authorize a delete) knows for sure. Many other protocols are possible, and we expect to experiment with several.

The S/Net's kernel is costly in storage required, because tuple space is replicated. We currently allocate approximately 300 kbytes worth of tuple storage space on each node, out of total RAMs ranging from 600 kbytes to 1.2 Mbytes. As noted, other kernel designs are possible that are more conservative with storage space; we are now implementing one. In the new protocol, designed by Jerry Leichter, `out` requries only a local install; `in(`*s*`)` causes template *s* to be broadcast to all nodes in the network. Whenever a node receives a template *s*, it checks *s* against all of its locally-stored tuples. If there is a match, it sends the matched tuple off to the templates's node; if not, it stores the template for *x* ticks (checking all tuples newly-generated within this period against it), then throws it out. If the template's origin node hasn't received a matching tuple after *x* ticks, it rebroadcasts the template. More than one node may respond with a matching tuple to a template-broadcast; when a template-broadcaster receives more that one tuple, it simply installs the extras alongside its locally-generated tuples and sends them onward when they are needed. This scheme doesn't require reliable broadcast, and it doesn't require tuples to be replicated on each node, so per-node storage requirements are much lower.

In the current S/Net kernel, each node's copy of tuple space is hashed on a tuple or template's first component (which must, recall, be an actual). Tuple matching (that is, the mapping of newly-arrived tuples to waiting templates, or new templates to stored tuples) is guided by control words stored in each tuple or template's header. This hashing scheme is less than ideal and will eventually be replaced. The problem is a common and useful Linda program pattern in which a large number of tuples have the same first field and are distinguished by some other field; consider the matrix program, for example. Hashing breaks down under these circumstances; for efficiency, we generally list the index *first*: thus `(1, "A",` *A's-first-row*`)`. (Tuples are arranged in this fashion in the S/Net matrix program we tested.) The user should not have to worry about the kernel's hashing scheme, though, and we are now investigating a table organization in which each tuple field is hashed separately, and the shortest hash chain guides the tuple-match search.

4. DETAILS OF THE S/NET LINDA KERNEL

The S/Net we worked on consisted of 8 MC-68000's with local memory ranging between 650 to 1200 kbytes, and a VAX 11/750, all connected by a word-parallel bus whose capacity is about 80 Mbits per second. There are no DMA channels or front ends (although in future iterations of this hardware there will be). The VAX handles Unix systems calls for the 68000's; Unix calls are trapped by monitors on the 68000's and sent to the VAX over the S/Net bus.

The S/Net Linda kernel currently consists of a set of systems calls that allow a user to write C programs that run on several processors and communicate using `ins`, `outs` and `reads`. The Apollo Linda compiler shields the user from these low-level calls; we have yet to port it to the S/Net, but this will be done soon.

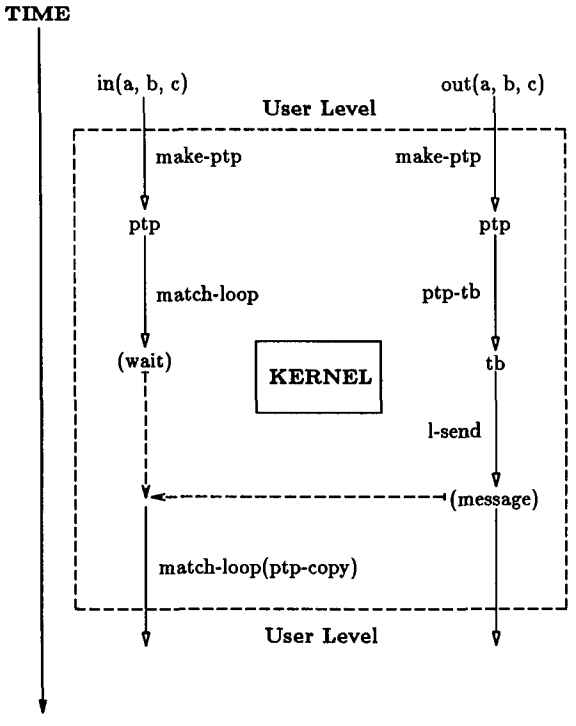

Fig. 1. Linda S/Net kernel system calls.

(The Apollo Linda compiler is simply a C preprocessor, so the move will be easy.) The kernel is written in C with the exception of one assembly routine tha sends data over the bus, and the few lines of code that invoke the Linda interrupt handler (written in C) upon the arrival of any Linda-format message. This dispatch-to-Linda routine is necessary because Linda processes and messages may coexist with others on the S/Net; our interrupt handler is installed above an S/Net monitor that invokes it when appropriate.

Figure 1 illustrates the ways in which the kernel routines interact. Two data structures are fundamental A `ptp` (proto-tuple packet) is a "tuple descriptor" that contains, for any tuple or template (recall that tuples and templates are structurally identical), the value or a pointer to the value of each actual element, and the type of each formal element. `ptp`'s are fixed-size structures, because our implementation imposes a limit of six fields per tuple beyond the first field—which is required to be either a string of at most sixteen characters or a long integer. (The tuples or templates themselves are not of fixed size, though. We currently support tuple elements of type integer, string, and "block"—where a block is an array of longwords and may be used to store reals, arrays and so forth.) `tb`'s (tuble blocks) hold tuple packets in a form that is suitable for transmission across the bus. They come in two sizes: 20 bytes of header information plus either 100 or 512 bytes of data. When the routine `ptp_tb( )` converts a tuple descriptor to a series of tuple blocks, it attempts to fit the tuple into the data area of a small `tb`. If the tuple spills over, additional large `tb`'s are linked on until the tuple fits. (Here and throughout, these numbers were chosen

by intuition. After we have gained some experience working with the implementation, we will be in a position to refine them.)

Matching of `in` templates to `out` tuples is done by `match_loop`. Tuples are hashed on their first element into one of 256 linked tuple lists. Executing an `in` causes an invocation of `match_loop( )`, which searches the appropriate hash list for a tuple to receive. If it finds one it attempts to delete it, as discussed in Section 3. If it doesn't find one, or if its delete attempt fails, it blocks pending arrival of a matching tuple. (We have not yet implemented multiprocessing within each S/Net node; for the time being, then, since each node runs only a single Linda process, a Linda process may block simply by spinning on a flag that is eventually toggled by the kernel in response to an interrupt. The Apollo Linda kernel does implement context-switching, though, and since the Apollo kernel is also a C program for the MC-680000 processor, adding per-node multiprocessing on the S/Net should be simple and will be undertaken soon.) A new tuple's arrival triggers a search for a matching template, and if a match is found the kernel proceeds as above.

It is worth noting that, although the kernel's organization may seem a bit complex, it has been designed to allow flexibility in deciding at some future time on an optimal division of work between the compiler and the run time kernel. The work done at run time by `make_ptp( )` (which builds tuple descriptors) can in fact be done almost entirely at compile time. The same holds in some cases for `ptp_tb( )`, which converts descriptors into tuple packets—although how frequently this is so depends on the compiler's sophistication. Tuples whose elements are all constants, for example, can obviously be packetized at compile time. A slightly more sophisticated compiler might also move packetizing out of a loop when tuple elements are loop invariant, and so on.

5. PERFORMANCE

Our first goal in experimenting with the S/Net kernel was to establish how long the basic TS operations, `out` and `in`, take to perform. We then attempted to refine our understanding of the kernel's performance by studying a simple Linda application program.

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 the support calls `make_ptp` and `ptp_tb` out of the loops, which is equivalent to assuming the existence of a compiler that is able to recognize that the strings `"ping"` and

"pong" are constants. Elasped time was measured using the 68000's clock via routines supplied by the existing operating system.

Running with LIMIT equal to 20000 (i.e., 40000 out_in pairs) we measured a rate of 720 pairs per second (where a pair is out("ping") plus in("ping") or out("pong") plus in("pong")) with the first delete protocol and 770 pairs per second with the second, simpler one. So the evidence suggests that a maximally-simple 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 1.3 ms with the fast delete protocol and 1.4 ms with the slow one. Other similar experiments support these general figures. The experiments were repeated various times with similar results, ordinarily with little or no nonLinda traffic sharing the S/Net bus with us.

We sought a better idea of the kernel's performance by implementing the matrix program discussed above. Figures 2 and 3 show measured execution time for Linda programs using one through four worker processes (one through six in the 32 × 32 case) in addition to one other process that first initializes and then cleans up. Each process runs on a separate processor, so two processors were active in the "one worker process" case, three in the "two workers" case and so forth, up to seven processors in the "six workers" case. We measured run time from the startup of the initializer to the point where the cleanup routine has removed the last product element from TS. (Cleanup then goes on to print the product matrix, and printing time is excluded from our measurement.) We are assuming an especially stupid compiler: we are measuring at run time tuple-formatting computations that would be avoided or done at compile time given a more sophisticated compiler. The graphs also show the time a standard C program (running on a single processor, or course) required to do the multiplication. Even for a matrix as small as 16 × 16, 3 Linda workers were enough to beat the C program. For large matrices, just two workers were enough.

Figures 2 and 3 show another interesting datum as well. In every case there existed an a and b such that a curve of the form $(a/n) + b$ would fit our datapoints *precisely*; our performance curves, in other words, have the shape of linear speedup curves. We interpret this data as suggesting that a represents *parallelizable* time while b represents *fixed* time. Execution time of the Linda program, in other words, can be divided into two parts: processing that can be split among the workers (a) and processing that is inherently sequential (b). Inherently sequential processing includes initialization time and time spent in kernel interrupts. Parallelizable time is composed of the time needed for the computations and for that portion of the communication burden that occurs outside of interrupt-handling and can therefore be carried out in parallel.

Note that our goals in this experiment were to learn something about the performance of our kernel, not to build a matrix multiplier that is fast in absolute terms. The boards we are working with do not have floating point chips—floating point multiplications are done in software. It is interesting to note, though, that when we ran a version of this routine using long integer instead of floating-point multiplies, the speedup curved that resulted were almost identical in shape to the ones in Figures 2 and 3, although all times are scaled down in absolute terms.

We are now in a position to make some further estimates of low-level communication costs. The fixed, sequential cost for a 64 × 64 multiplication is

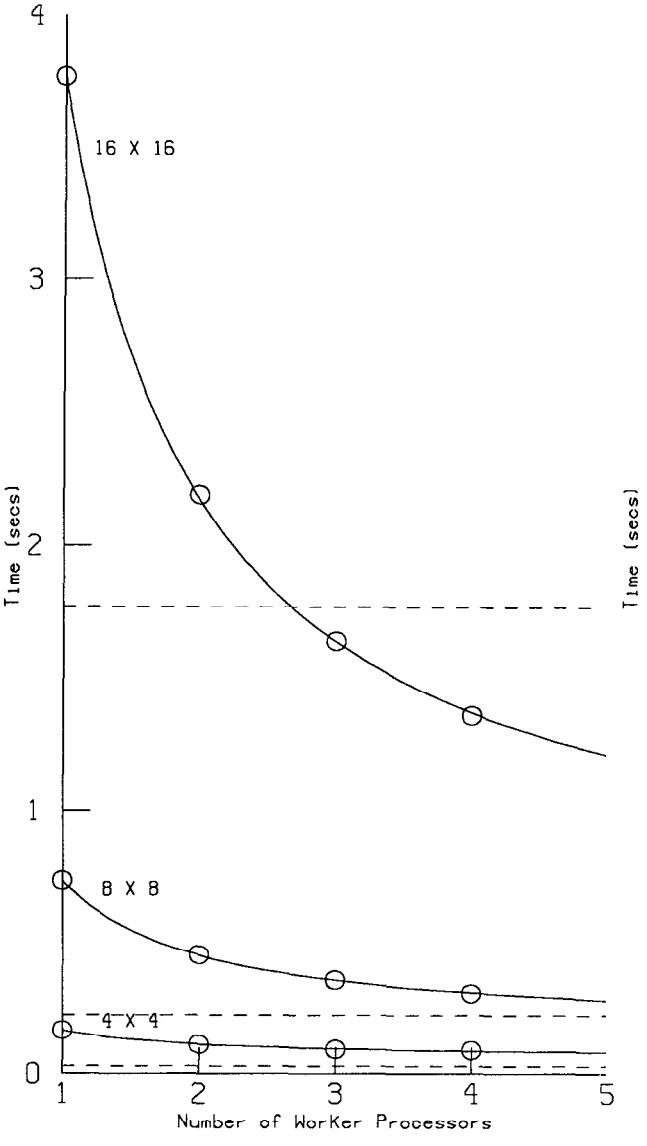

Fig. 2. Execution time versus number of worker (DOT computing one element of result matrix). O = measured value; — = $a/n + b$; --- = uniprocessor C.

roughly 8.2 seconds (s) (b above). This is largely made up of the 0.6 s we have measured as required initialization time (the time needed to dump the input matrices into TS), plus the time needed, for each element of the product, to handle interrupts for two `out-in` pairs, one for the "Dot" tuple and one for the result. We ran this program with the longer delete protocol only; under this protocol, three interrupts are handled by each kernel in the course of handling an `out-in` pair. The net result is 0.3 ms per interrupt-driven invocation of the Linda kernel.

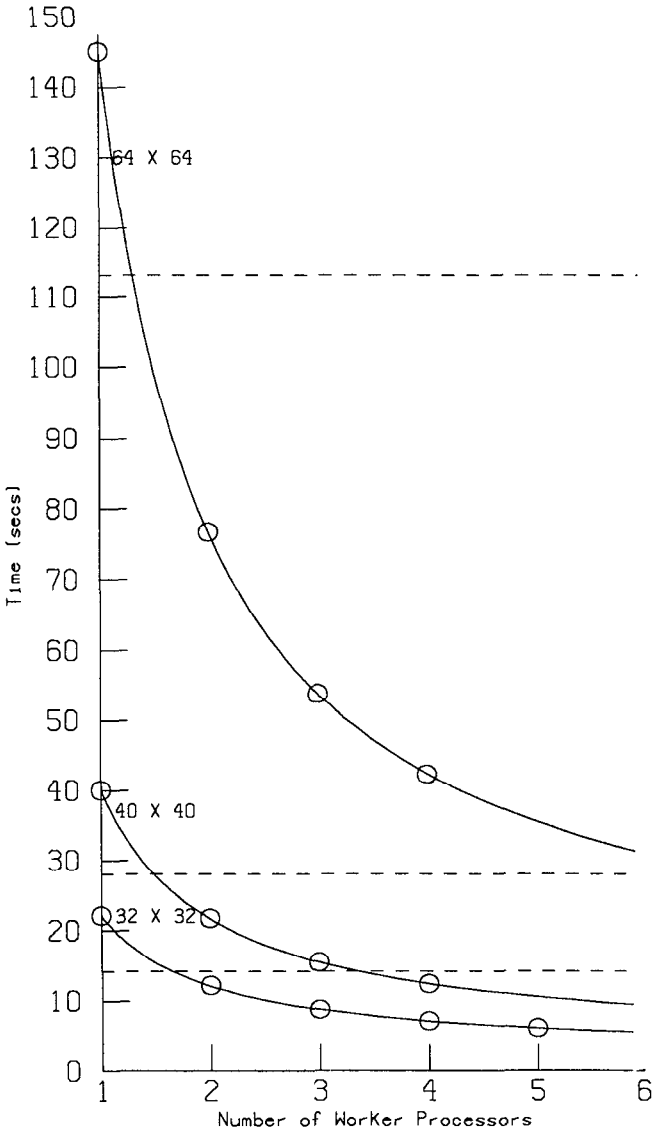

Fig. 3. Execution time versus number of workers (DOT computing one element of result matrix). O = measured value; — = $a/n + b$; --- = uniprocessor C.

We went on to test a second version of the algorithm in which the granularity of parallelism is coarser and communication costs are correspondingly lower. In this version, worker processes compute an entire row of the product, not just a single element, in each task step. Figure 4 shows results for matrices with dimension 16, 32, and 64 using 1, 2, 3, 4 and 5 worker processors. (Data for 16×16 has been scaled up by a factor of twenty.) Here we have plotted our data against curves that represent *ideal speed-up of the C program.* The solid curves intersect the ordinate at a point which represents *measured* performance of the

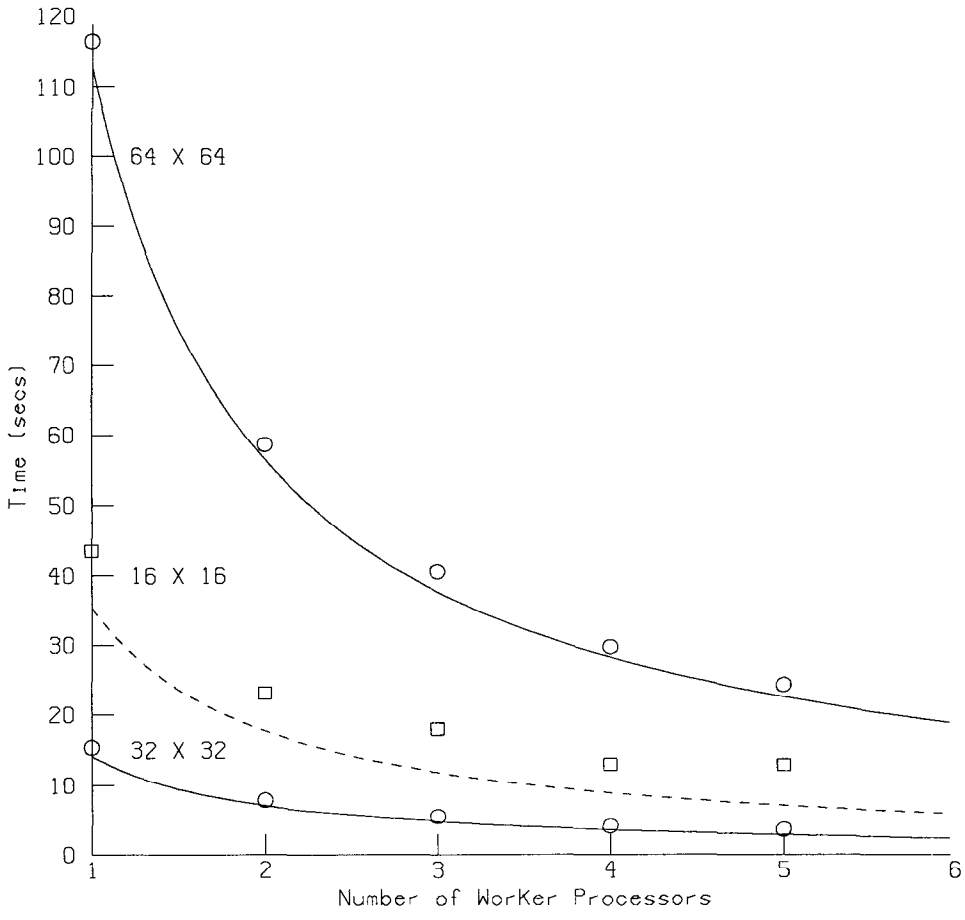

Fig. 4. Execution time versus number of workers (DOT computing one row of the result matrix). O = measured value; □ = measured value * 20; — = (time of C)/n; --- = (time of $C * 20$)/n.

C version: thus in the 64 × 64 case, for example, C took 113 s (Linda with a single worker required 116 s); the curve represents the effect of simply turning up C's speed linearly—doubling it, tripling it and so on. For the two larger dimensions we observe a good fit, for all five processors, of Linda time to the idealized C speedup curve. 16 × 16 is not as good, but even so the addition of just one extra processor is sufficient for this version to finish faster than the uniprocessor C program.

The astute reader may have noticed that the data points for odd numbers of processors are slightly worse than for even numbers. This is most pronounced in the 16 × 16 case for five processors. The explanation lies in the relatively large granularity of parallelism in this program. Consider the extreme case. For the 16 × 16 problem, there are 16 rows of the result matrix to compute. Given four workers, each computes four rows. Given five, four will need to compute only three rows, but the fifth still needs to compute four, exactly as in the four-worker case. Despite the extra worker we must still wait, then, for a full four rows to be

computed in sequence—and so it is not surprising that the four and five worker versions take equal amounts of time to finish. In the finer-grained, element-by-element version of this problem, we have 256 tasks to distribute instead of 16, with a maximum of 64 tasks for four workers and 52 for five—an appreciable difference. Communications costs, on the other hand, are much greater in the fine-grained version, and the net result is that row-by-row ran twice as fast as element-by-element.

One more interesting fact about the row-by-row version: if we were chaining matrix multiplications together—if we were waiting to multiply the product of A and B by C—then row-by-row multiplication of A and B produces results in the optimal sequence and the correct format for pipelining the two multiplications; multiplication of $A \times B$ by C can begin as soon as the first $A \times B$ worker-task completes.

Is matrix multiplication a reasonable test case? Note that the Linda solution has more generality than we need. The matrix program assigns tasks to workers dynamically, but in a problem as simple and regular as matrix multiplication, we could as well have assigned each of n workers $1/n$ of the product matrix to compute. (It's interesting to note, however, that even with a problem as orderly as matrix multiplication, dynamic scheduling might be the technique of choice if we were running on an inhomogeneous network, on which processors vary in speed and in run time loading. We've been studying just such a network—a collection of Vaxes ranging from Micro-Vax I's to 8600's.) What is interesting, then, is the fact that we measured good speedups despite (unnecessary) dynamic scheduling; a generalization of this same solution framework will work on irregular problems where dynamic scheduling *is* important. (We discuss such problems in [4].)

5.1 Limitations of our Test Results

The Linda kernel has only recently been completed, and we have much more testing to do. Most important, we can not yet say how our kernel will run on more than eight nodes; eight is the maximum available to us on our machine. We will have some basis for prediction once we know what fraction of the S/Net's total communication capacity our running programs consume; we are now developing tests that will measure this. Several points are worth mentioning, though. Note first that, given programs like the matrix multipliers, total bytes transmitted over the bus depends on the size of the input matrices only, *not* on the number of parallel workers. Whether we run a single worker or a hundred, the "Dot" tuple is `in`ed and `out`ed exactly once for each element of the product, and so fourth. We don't increase total bytes to be transmitted, then, when we add processors; we increase offered load insofar as we attempt to send the *same* number of bytes within a shorter interval. Test programs measured by other researchers at Bell have not succeeded, however, in using more than about 20 percent of the S/Net's available capacity even when doing nothing but repeated sends. This figure suggest to us that there is room for much more speedup as we add workers beyond the number we have been able to test so far, and that the TS primitives will continue to run well as total processors increase. We won't know for sure, though, until we have access to a larger S/Net.

6. CONCLUSIONS

Linda offers parallel programmers a new way of looking at network communications systems. Standard communication protocols require that information be handed around from process to process; no process can unburden itself of new data without first determining where the data should go, and then handing it along explicitly. Linda processes, on the other hand, are anonymous drones sharing access to one data pool. Shared memory has long been regarded as the most flexible and powerful way of sharing data among parallel processes—but a naive shared memory is hard to implement without hardware support, and requires the addition of synchronization protocols if it is to be safely accessed in parallel. In Linda, however, the shared memory's cell-size is the logical tuple, not the physical byte, and so it is coarse-grained enough to be supported efficiently without special hardware. And because, in Linda's shared memory, data may not be altered *in situ*—it is accessible via read, remove and add instead of the standard read and write—it may safely be shared by any number of parallel processes.

Several limitations and one nonlimitation of this work should be noted. A major limitation is the cursory way in which we have dealt with questions of reliability and failure. A Linda environment is not a particularly vulnerable one; in some ways, failure or unreliability of the nodes or the interconnect are easier to handle in a Linda environment than in a conventional one. Regardless, we have not yet dealt carefully with failure, and we will clearly need to before our implementation is complete. There is another limitation of sorts in the fact that our focus in the S/Net project to date has been on the Linda primitives themselves, not on parallel applications that use them; applications programming using our Linda kernel is a current research topic both for ourselves and for a parallel-application project at Bell.

One further point that will perhaps be taken as a limitation is the limited extensibility of the S/Net architecture and the limited generalizability, in this sense, of our results. The current S/Net bus will not support more than sixty-four nodes: A VLSI-based reimplementation will accomodate no more than 256. The S/Net and its Linda kernel will not suffice for thousand or multithousand node super-computer networks. We have in fact studied the implementation of Linda on large hypercube-connected linked networks [12], and a Linda implementation for such a machine is now in design. More important, the limited extensibility of the S/Net is irrelevant to our interests and goals in this project, and we do *not* regard it as a limitation. This is so for three reasons. First, we believe that a working fifty- or one hundred-node multicomputer that application programmers could actually *use* (one whose potential power was conveniently accessible to a wide community of programmers, not limited to those with intimate knowledge of the machine), would be highly powerful and desirable tool. Such a machine has certainly *not* (to our knowledge) been achieved to date. Second, smaller networks will continue to be of interest in the design of advanced workstations; investigation of a parallel workstation (one that uses the Linda kernel to support a parallel interpreter for Symmetric Lisp [14]) is a major future goal of the S/Net-Linda project. Finally, the techniques we are investigating on the S/Net should be applicable to much larger bussed networks as well.

Linda is intended as a general-purpose programming vehicle both for parallel applications and for distributed systems. Our S/Net work to date has included, besides the matrix multiplication routines, experiments with parallel LU decomposition with partial pivoting, and with a VLSI simulator; some of this ongoing work is discussed in [4]. The Micro-Vax Linda kernel is intended mainly for distributed systems programming; previous work with an Apollo-workstation-based Linda simulator included concurrent-system problems like Dijkstra's dining philosophers and the readers-writers problem, the Jacobi iterative method for differential equations, pipeline programs (a square-root approximator, for example), parallel sorting and prime-number-generating routines that build process pipelines dynamically in outward-telescoping fashion, and a concurrent version of the Apollo Linda preprocessor itself.

As a communications kernel for a bus-connected network, S/Net-Linda falls generally within the category of several others that have been reported in recent years, including the Birrell and Nelson RPC kernel, Cheriton and Zwaenpoel's V Kernel [7] and Spector's Remote Operations kernel [18], 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 internode 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, within latitude of 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[3]; a generally comparable operation in Linda 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 considerably faster; but Linda and the V Kernel both run on MC-68000's, the RPC kernel on the much-faster Dorado.

Since Linda provides a form of logically shared memory, it might be deemed reasonable to compare it to physically shared memory systems. Two separate questions are possible: Is Linda as expressive as the programming systems provided with physically shared memory machines? Does it run as well? For now, the second question is unanswerable, because we haven't implemented Linda on a shared memory machine. Comparing Linda on the S/Net to something else on a shared memory architecture tells us nothing, because we have not controlled for hardware: In general, we expect communication to be cheaper on physically shared memory systems; we expect them to be capable of realizing speedups from programs that are far more communication-intensive than anything we can run efficiently using S/Net-Linda—but of course, we'd also expect *Linda* on a shared-memory machine to run faster than S/Net-Linda. Regarding the first question, programming primitives developed for shared-memory architectures cannot on the whole be expected to run, as Linda does, on network

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

machines. Once again, we are dealing with incomparables. For the sake of argument, the programming tools that have been made available on shared-memory architectures seem generally to be conceptually lower-level than Linda's primitives. Consider, for example, the Chrysalis operating system for the BBN Butterfly [9]. Global names in Chrysalis may refer to memory segments, events, "dual queues," or processes. (In Linda, global names refer only to tuples; Linda processes never need to deal directly with memory segments or with other processes.) Chrysalis provides operations on segment attribute registers for purposes of memory sharing, and events (with associated event handlers and event blocks), together with "post" operations, for synchronization and communication. (Events are always associated with processes; "post" may send either to an event or to a dual-queue.) There are doubtless cases in which Chrysalis primitives are more appropriate than Linda primitives; in most cases, we will prefer Linda's simplicity.

The S/Net architecture has features that suit it particularly well to our kernel, and we have made significant use of them, as noted. In planning new Linda kernels we are not restricted, however, either to the S/Net in particular or to bus-based networks generally. The Micro-Vax kernel (which we have already used for some preliminary matrix-multiplication experiments) is very similar to the S/Net's, but Linda kernels for two different hypercube multicomputers[4] are also in design, and their lower-half communication routines are very different (though the upper-half routines that set up and manage tuple space are largely the same). Linda does not require shared memory, but it is a natural match to multicomputers that happen to provide it; we believe that Linda will be easy to implement on shared memory machines, and that it will prove valuable as a clean, simple way of parcelling out access to the shared memory resource. This is potentially complex to control in a parallel-programming environment.

Despite the varied architectures we are now dealing with, the S/Net's special importance to the Linda project can not be denied. Some of our most interesting current collaborative work involves developments in S/Net hardware that will allow the Linda kernel to run better. In the short term this work involves communication coprocessors that will soak up the heavy bus-interrupt load our kernel generates, leaving the host processors free to compute in peace. For the longer term, we are investigating the design of a VLSI "Linda chip" that executes a good part of the kernel in hardware. All of our work to date tends to the conclusion that the tuple-space operators are in fact a good basis for a parallel machine language—a simple set of flexible, powerful operators to be supported directly by a multicomputer's communication hardware. These basic operators are accessed most directly via Linda, but higher-level languages may be implemented above them as well. Our goal of a hardware Linda Machine is likely to be realized first in the S/Net context.

ACKNOWLEDGMENTS

This work was performed primarily at AT&T Bell Laboratories, Holmdel; Sid Ahuja, Erik DeBenedictis, Robert Gaglianello, Howard Katseff and Thomas London, all of Bell Laboratories, were our research collaborators.

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

REFERENCES

1. S. AHUJA. S/Net: A high-speed interconnect for multiple computers. *IEEE Selected Areas in Communication* (Nov. 1983), 751–756.
2. ANDREWS, G. R. AND SCHNEIDER, F. B. Concepts and notations for concurrent programming. *ACM Comp. Surv. 15*, 1 (Mar. 1983), 3–44.
3. BIRREL, A. D. AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comp. Syst. 2*, 1 (Feb. 1984), 39–59.
4. CARRIERO, N., GELERNTER, D. AND LEICHTER, J. Distributed data structures in Linda. In *Proceedings of the ACM Symposium on Principles Programming Languages* (Jan. 13–15, St. Petersburg, Fla.), 1986, ACM, N.Y.
5. CHANG, J. M. AND MAXEMCHUCK, N. F. Reliable broadcast protocols. *ACM Trans. Comp. Syst. 2*, 3 (May 1984), 251–273.
6. CHERITON, D. R. AND ZWAENEPOEL, W. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operatings Systems Principles.* (Oct. 11–13, Bretton Woods, N.H.), 1983, ACM, N.Y., 128–139.
7. CHERITON, D. R. AND ZWAENEPOEL, W. Distributed process groups in the V kernel. *ACM Trans. Comp. Syst. 3*, 2 (May 1985), 77–107.
8. DEPARTMENT OF DEFENSE. *Reference Manual for the Ada Programming Language*, U.S. Dept. of Defense, July 1982.
9. DEUTSCH, J. T. AND NEWTON, A. R. MSPLICE: A multiprocessor-based circuit simulator. In *Proceedings of the 1984 International Conference on Parallel Processing*, (Aug. 1984), 207–214.
10. GABRIEL, R. P. AND MCCARTHY, J. Queue-based multi-processing Lisp. In *Proceedings of the ACM Symposium on Lisp. and Functional Programming* (Aug. 6–8, Austin, Tex.), 1984, ACM, N.Y., 25–44.
11. GELERNTER, D. AND BERNSTEIN, A. Distributed communication via global buffer. In *Proceedings ACM Symposium Principles of Distributed Computing* (Aug. 18–20, Ottawa, Ont.), 1982, ACM, N.Y., 10–18.
12. GELERNTER, D. Dynamic global name spaces on network computers. In *Proceedings International Conference Parallel Processing*, (Aug. 1984).
13. GELERNTER, D. "Generative communication in Linda. *ACM Trans. Prog. Lang. Syst. 7*, 1 (Jan. 1985), 80–112.
14. GELERNTER, D., CARRIERO, N., CHANDRAN, S. AND CHANG, S. "Parallel programming in Linda," In *Proceedings of the International Conference on Parallel Processing*, (Aug. 1985), 255–263.
15. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 11 (Aug. 1978), 666–677.
16. INMOS LTD. *OCCAM Programming Manual.* Prentice-Hall (1984).
17. SHAPIRO, E. Systolic programming: A paradigm of parallel processing. Tech. Rep. CS84-21, Weitzmann Institute of Science, Dept. of Applied Mathematics, Rehovot, Israel (Aug. 1984).
18. SPECTOR, A. Performing remote operations efficiently on a local network. *Commun. ACM 25*, 4 (1982), 246–260.

Received July 1985; revised September 1985; accepted November 1985