Linda and Friends

Sudhir Ahuja, AT&T Bell Laboratories
Nicholas Carriero and David Gelernter, Yale University

Linda consists of a few simple primitives that support an "uncoupled" style of parallel programming. Implementations exist on a broad spectrum of parallel machines.

Linda consists of a few simple operators designed to support and simplify the construction of explicitly-parallel programs. Linda has been implemented on AT&T Bell Labs' S/Net multicomputer and, in a preliminary way, on an Ethernet-based MicroVAX network and an Intel iPSC hypercube. Although the implementations are new and need refinement, our early experiences with them have been revealing, and we take them as supporting our claim that Linda is a promising tool.

Parallel programming is often described as being fundamentally harder than conventional, sequential programming, but in our experience (limited so far, but growing) it isn't. Parallel programming in Linda is conceptually the same order of task as conventional programming in a sequential language. Parallelism does, though, encompass a potentially difficult problem. A conventional program consists of one executing process, of a single point in computational time-space, but a parallel program consists of many, and to the extent that we have to worry about the relationship among these points in time and space, the mood turns nasty. Linda's mission, however, is to make it largely unnecessary to think about the coupling between parallel processes. Linda's uncoupled processes, in fact, never deal with each other directly. A parallel program in Linda is a spatially and temporally unordered bag of processes, not a process graph. To the extent that process uncoupling succeeds, the difficulty of designing, debugging, and understanding a parallel program grows additively and not multiplicatively with the variety of processes it encompasses.

When the simple operators Linda provides are injected into a host language *h*, they turn *h* into a parallel programming language. A Linda system consists of the runtime kernel, which implements interprocess communication and process management, and a preprocessor or compiler. A Linda-based parallel language is in fact a new language, not an old one with added system calls, to the extent that the preprocessor or compiler 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, among other things. Most of our programming experiments so far have been conducted in C-Linda (and we use C-Linda for the examples below), but we have implemented a Fortran-Linda preprocessor as well. The kernel is language-independent. It will support *N*-Linda for any language *N*.

Associated with the Linda operators is a particular programming methodology, based on distributed data structures. (The language doesn't restrict programmers to this methodology. It merely allows the methodology, which most other languages don't.) The distributed-data-structure

0018-9162/86/0800-0026$01.00

methodology in turn suggests a particular strategy for dealing with parallelism. Most models of parallelism assume that a program will be parallelized by partitioning it into a large number of simultaneous activities. This partitioning appears, however, to be relatively difficult to do, especially when we consider large multicomputers that support thousand-fold parallelism and beyond. In the Linda framework, we can get parallelism by *replicating* as well as by partitioning. We anticipate that it will frequently be simpler to stamp out many identical copies of one process than to create the same number of distinct processes. So the final ingredient in the Linda framework is a strategy for coping with parallelism by replication rather than by partitioning.

In the following we discuss first what it seems to us that parallel programmers need. We then describe Linda, some programming experiments using Linda, the current implementation, and the project now underway to go beyond the current software implementation and build a hardware Linda machine. We go on to discuss some related higher-level parallel languages that can be implemented on top of the Linda kernel, particularly the symmetric languages. We close in a blaze of speculation.

What parallel programmers need

Many parallel algorithms are known and more are in development; many parallel machines are available and many more will be soon. But the fate of the whole effort will ultimately be decided by the extent to which working programmers can put the algorithms and the machines together. The needs of parallel programmers have not been accommodated very well to date.

A machine-independent and (potentially) portable programming vehicle. Designers of parallel languages generally hold that programming tools should accommodate a high-level programming model, not a particular architecture. But as parallel machines emerge commercially, there has been little effort spent on making high-level, machine-independent tools available on them. Young debutante machines are sometimes gotten-up in their own full-blown parallel languages; more often they come dressed in only a handful of idiosyncratic system calls that support the local variant of message-passing or memory-sharing. In either case, so long as each new machine is provided with its own parallel programming system, programs for multicomputer x will not only have to be recoded, they may need to be conceptually reformulated to run on multicomputer y. (This is particularly true if x is a shared-memory machine like a BBN Butterfly[1] or an IBM RP3[2] and y is a network, like an Intel iPSC.) But users need to be able to run parallel programs on a range of architectures, particularly now, when interesting designs of unknown merit proliferate. They need to be able to communicate parallel algorithms. Methodological knowledge can't grow when sources are cluttered with local dialect. Finally, they need programming tools suited to their needs, not to the machine's.

A programming tool that absolves them as fully as possible from dealing with spatial and temporal relationships among parallel processes. We referred to the general problem of uncoupling above. Uncoupling has both a spatial and a temporal aspect. Spatially, each process in a parallel program will usually develop a result or a series of results that will be accepted as input by certain other processes. Uncoupling suggests that process q should not be required to know or care that process j accepts q's data as input. Instead of requiring q to execute an explicit "send this data to j" statement, we would rather that q be permitted simply to tag its new data with a logical label (for example, "new data from q") and then forget about it, under the assumption that any process that wants it will come and get it. At some later point in program development, a different process may decide to deal with q's data. Under the spatially-uncoupled scheme, this won't matter to q.

Temporal uncoupling involves similar though perhaps slightly more subtle issues. If q is forced to send to j explicitly, the system is constrained to have both processes loaded simultaneously (or at least to have buffer space allocated for j when q runs). Further, most parallel languages attach some form of synchronization constraint to send. A synchronized send operation like Ada's entry call or CSP-Occam's output statement forces the system not merely to load but to run the receiving process before the sender can continue. We would rather that our parallel programs be largely free of scheduling implications like these. Not only do they constrain the system in ways that may be undesirable, but they force programmers to think in simultaneities. As far as possible, we would like programmers to be able to develop q's code without having to envision other simultaneous execution loci. To achieve this, we would like q to be allowed to take each new datum it develops and heave it overboard without a backwards glance. (We make this a bit more concrete below.)

A programming tool that allows tasks to be dynamically distributed at runtime. Generally there is more logical parallelism in a parallel algorithm than physical parallelism in a host multicomputer, which means that at runtime there are more ready tasks than idle processors. Good speedup obviously requires that tasks be evenly distributed among available processors. Many systems require that this distribution be performed statically at load time. Sometimes, finding a good static distribution is easy, notably when the program's logical structure matches the machine's physical structure. As the program's logical structure grows more irregular, the task gets harder, and when the program's computational focus develops dynamically at runtime, finding a good static mapping may be impossible. Many important applications and program structures fall into the first, easily-handled category, but many more do not. For those that don't, dynamic distribution of tasks is essential.

A programming tool that can be implemented efficiently on existing hardware. Obviously. Parallel language research has produced far more designs than implementations. Elegant language ideas will always be interesting regardless of the existence of good implementations, but parallel programmers, as opposed to language researchers, require implementable elegance.

Linda

Linda centers on an idiosyncratic memory model. Where a conventional memory's storage unit is the physical byte (or

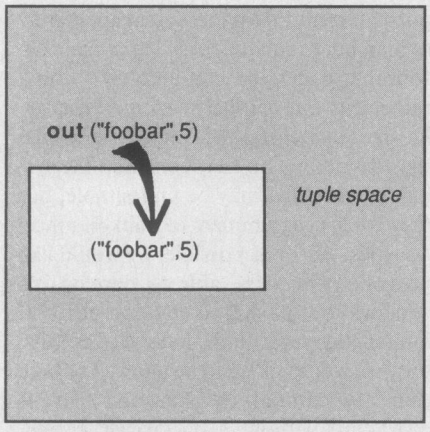


Figure 1. out statement: drop it in.

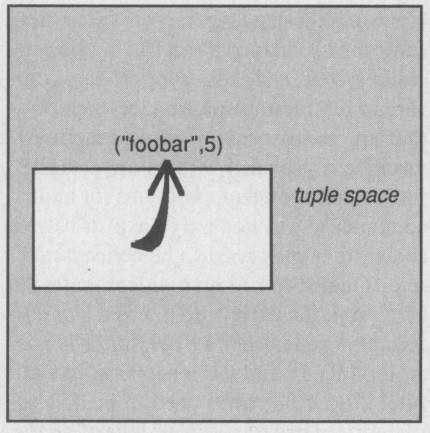


Figure 2. in statement: haul it out.

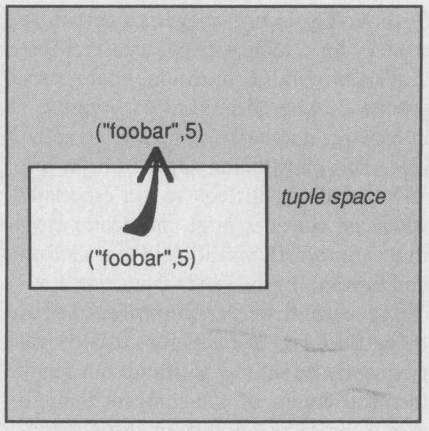


Figure 3. read statement: read it and leave it.

something comparable), Linda memory's storage unit is the logical tuple, or ordered set of values. Where the elements of a conventional memory are accessed by address, elements in Linda memory have no addresses. They are accessed by *logical name*, where a tuple's *name* is any selection of its values. Where a conventional memory is accessed via two operations, read and write, a Linda memory is accessed via three: read, add, and remove.

It is a consequence of the last characteristic that tuples in a Linda memory can't be altered *in situ*. To be changed, they must be physically removed, updated, and then reinserted. This makes it possible for many processes to share access to a Linda memory simultaneously; using Linda we can build distributed data structures that, unlike conventional ones, may be manipulated by many processes in parallel. Furthermore, as a consequence of the first characteristic (a Linda memory stores tuples, not bytes), Linda's shared memory is coarse-grained enough to be supported efficiently without shared-memory hardware. Shared memory has long been regarded as the most flexible and powerful way of sharing data among parallel processes, but a naive shared memory requires hardware support that is complicated, expensive to build, and suitable only for multicomputers, not for local area networks. Linda's variant of shared memory, on the other hand, runs both on the S/Net and on a MicroVAX network, neither of which provides any physically shared memory. (Of course, Linda may be implemented on shared-memory multicomputers as well, as we discuss below.)

Linda's shared memory is referred to as *tuple space*, or TS. Messages in Linda are never exchanged between two processes directly. Instead, a process with data to communicate adds it to tuple space and a process that needs to receive data seeks it, likewise, in tuple space. There are four operations defined over TS: **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 formals 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.

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. **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**.) See Figures 1, 2, and 3.

The parameters to an **in()** or **read()** statement needn't all be formals. Any or all 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 **formal** 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", formal i, j)

This extended naming convention (it resembles the **select** operation in relational

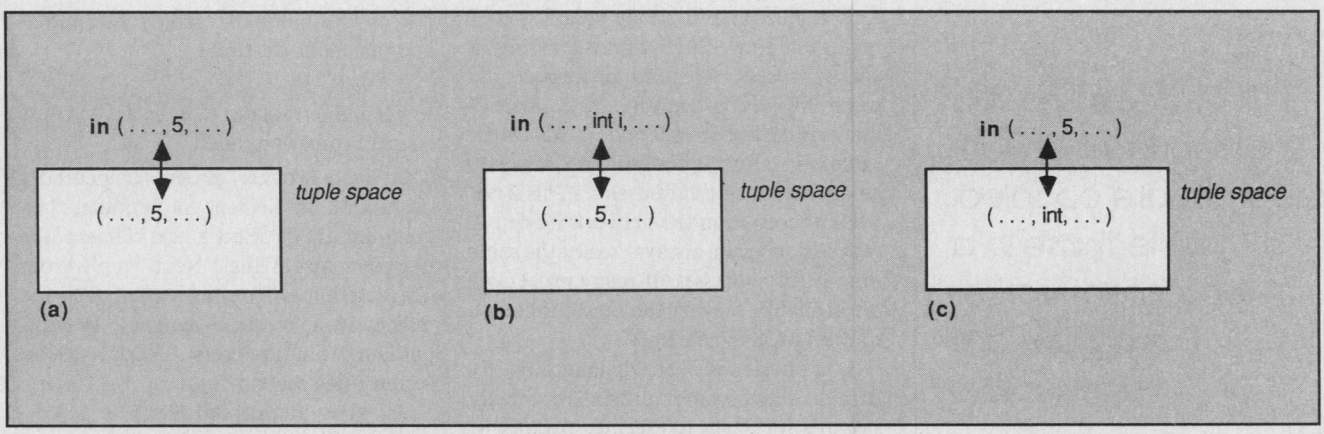


Figure 4. Structured naming: legal matches.

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. See Figure 4.

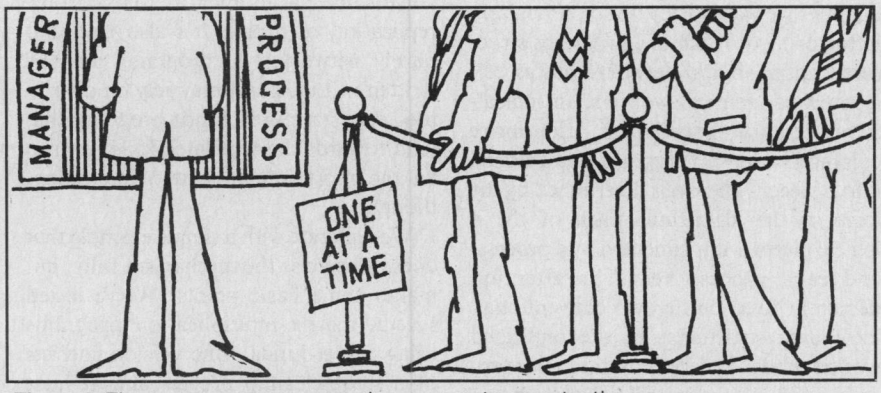


Figure 5. The manager process services requests one at a time.

Programming in Linda

Linda accommodates the needs for uncoupling and dynamic scheduling we listed above by relying on distributed data structures. As noted, a distributed data structure is one that may be manipulated by many parallel processes simultaneously. Distributed data structures are the natural complement to parallel program structures, but 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, on request, by the manager process on the user's behalf. See Figures 5 and 6.

The manager-process model has important advantages, and manager-process programs are easy to write in Linda. What

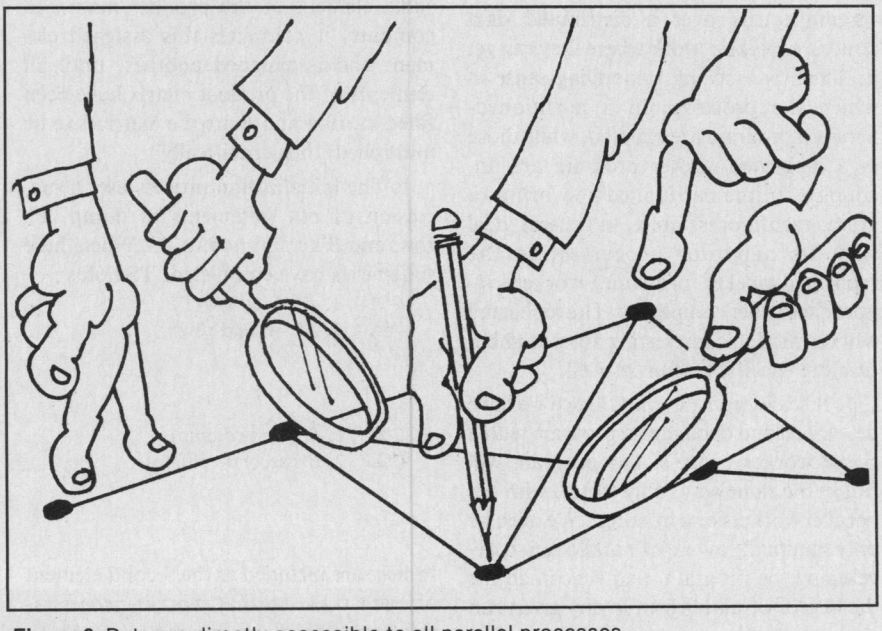


Figure 6. Data are directly accessible to all parallel processes.

> The processes in a partitioned-network program are coupled, while those in a replicated-worker program are uncoupled.

is significant, though, is the number of cases in which distributed data structure programs come closer to achieving the qualities we want. They do so particularly in the context of parallel programs structured not as logical networks but as collections of identical workers. In logical-network-style parallelism (the more common variety), a program is partitioned into n pieces, where n is determined by the logic of the algorithm. 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 scramble simultaneously over a distributed data structure, seeking work where they can get it. There is a strong underlying sense in which the processes in a partitioned-network program are coupled, while those in a replicated-worker program are uncoupled. In the partitioned network program, each process must, in general, deal with its neighbor processes; in the replicated-worker program, workers ignore each other completely. The replicated worker model is interesting for a number of more specific reasons as well.

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-managment 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, they 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 divided at runtime among the available workers.

It's important to note that most of the programs we've experimented with are not pure replicated-worker examples; they involve some partitioning as well as some replication of duties. It's also true that purely network-style programs may be written in Linda and may rely on distributed data structures. Linda programs that tend towards the replicated style seem to be the most idiomatic and interesting, though.

We illustrate with a simple example that doesn't exercise the mechanism fully, but makes some basic points. We've tested several matrix multiplication programs using S/Net-Linda. One version consists of a setup-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

1. 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)
("A", 2, A's-second-row)
.
.
.
("B", 1, B's-first-column)
("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. The initializer then adds the tuple

("Next", 1)

to TS and terminates. 1 indicates the next element to be computed.

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

```
in("Next", formal NextElem);
if(NextElem < dim * dim)
  out("Next", NextElem + 1);
i = (NextElem - 1)/dim + 1;
j = (NextElem - 1)%dim + 1;
```

The worker now proceeds to compute the product element whose index is (i,j). Note that if (i,j) is the last element of the product matrix, the "Next" 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.

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

```
read("A", i, formal row);
read("B", j, formal col);
out("result", 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*.)

3. 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 ("result", row, col, formal prod
      [row][col]);
print prod;
```

This simple program depends entirely on distributed data structures. 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.

We discuss the performance of this program, and of another version that assigns coarser-grained tasks that compute an entire row rather than a single inner product, elsewhere.[3] Both versions show good speedup as we add processors up to the limited number available to us on our S/Net (currently eight). The version discussed above requires only two parallel workers and one control process to beat a conventional uniprocessor C program on 32×32 matrices, and continues to show linear speedup as we add workers. The coarser-grained version, with its lower communication overhead, shows speedup close to ideal linear speedup of the uniprocessor C version: our figures show close to a progressive doubling, tripling, and so on of the C program's speed as we add Linda workers.

The matrix program displays the uncoupling and dynamic-scheduling properties that we claimed above were important. Uncoupling: no worker deals directly with any other. Dynamic task scheduling: the matrix program assigns tasks to workers dynamically. But of course, in a problem as simple and regular as matrix multiplication, dynamic scheduling isn't important. We could just 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 a nonhomogeneous network, on which processors vary in speed and in runtime loading. We've been studying just such a network—a collection of VAXes ranging from MicroVAX I's to 8600's.)

Dynamic scheduling becomes important when tasks require varying amounts of time to complete. In the general case, moreover, new tasks may be developed dynamically as old ones are processed. Linda techniques to deal with this general problem are based on a distributed data structure called a *task bag*. Workers repeatedly draw their next assignment from the 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. The scheme is easily implemented in Linda. Elements of the bag will be tuples of the form

("Task", task descriptor)

Each worker executes the following loop:

```
loop {
  /* withdraw a task from the bag: */
in("Task", formal NextTask);
process "NextTask";
for (each NewTask generated in the
     process)
  /* drop the new task into the bag: */
  out("Task", NewTask);
    }
```

We've experimented with programs of this sort to perform LU decomposition with pivoting and to find paths through a graph, among others. Note that, if it were necessary to process tasks in a particular order rather than in arbitrary order, we would build a task queue instead of a task bag. The technique would involve numbered tuples and structured naming.

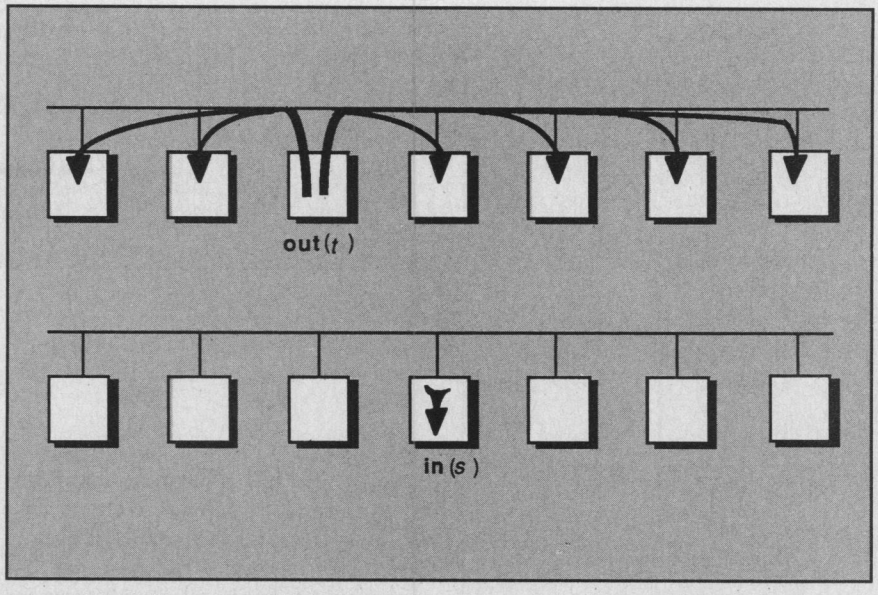


Figure 7. An S/Net kernel. (a) shows **out**: broadcast, while (b) shows **in:** check locally. (The inverse of this scheme is also possible.)

The S/Net's Linda kernel

Linda has often been regarded as posing a particularly difficult implementation problem. The difficulty lies in the fact that, as noted above, Linda supplies a form of logically-shared memory without assuming any physically-shared memory in the underlying hardware. The following paragraphs summarize the way in which we implemented Linda on the S/Net (the S/Net implementation is discussed in detail elsewhere[3]); there are many other possible implementations as well.

Our implementation buys speed at the expense of communication bandwidth and local memory. The reasonableness of this trade-off was our starting point. (Possible variants are more conservative with local memory.)

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 suceeds, 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 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. See Figure 7.

The delete protocol must satisfy two requirements. First, all nodes must receive

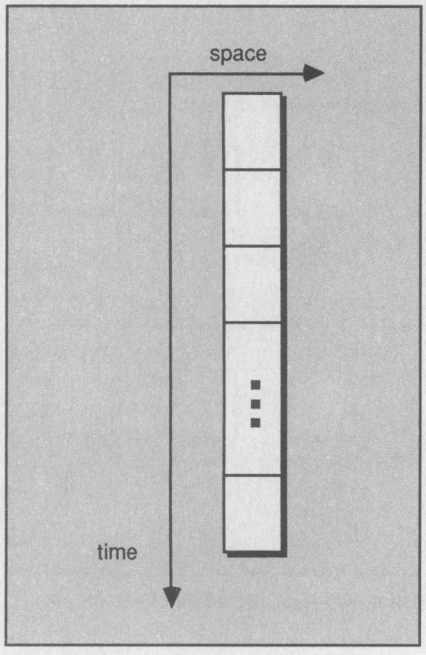


Figure 8. Execute sequentially.

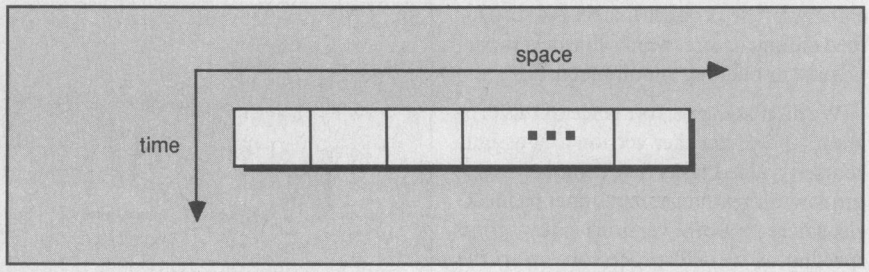


Figure 9. Execute in parallel.

the "delete" message, and second, 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 rebroadcasts repeatedly until the negative-acknowlegement 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.)

Evidence suggests that a minimal **out-in** transaction, from kernel entry on the **out** side to kernel exit on the **in** side, takes about 1.4 ms.

We are working on other implementations as well. The VAX-network Linda kernel (which was designed and is being implemented by Jerry Leichter) uses a technique that is in a sense the inverse of the existing S/Net scheme. We're in the process of trying this new technique on the S/Net also. In the new protocol, **out** requires 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 template'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 than one tuple, it simply installs the extras alongside its locally-generated tuples and sends them onward when they're needed. (In a more elaborate version, we can forestall the arrival of unneeded tuples by having potential senders monitor the bus, or by broadcasting an "I've got one, enough already" message at the appropriate point.) This scheme doesn't require hardware support for reliable broadcast and it doesn't require tuples to be replicated on each node, so per-node storage requirements are much lower.

The Linda kernel for the Intel iPSC hypercube, designed and implemented by Rob Bjornson, relies on point-to-point rather than broadcast communication. His scheme implements tuple space as a hash table distributed throughout the network. Each tuple is hashed on **out** to a unique network node and is sent there for storage. Templates are hashed and stored in the same way.

Finally, several of us (Bjornson, Carriero, Leichter, and Gelernter) have begun, in conjunction with Scientific Computing Associates, to design and implement a Linda kernel for the Encore Multimax. Nodes on the Multimax have direct access to physically shared memory. The Multimax Linda kernel should therefore be faster and simpler than the kernels described above, and in fact it is. The relationship between Linda and shared-memory multiprocessors like the Multimax is roughly similar to what holds between block-structured languages and stack architectures. The architecture strongly supports the language; the language refines the power of the architecture and makes it accessible to programmers. Of course, for all its promise, the Encore doesn't end our interest in networks like the S/Net. Shared memory seems ideal for small or medium-sized collections of processors. S/Net-like architectures, particularly the Linda machine we describe below, may well scale upwards to enormous sizes.

We have referred to Linda as a programming language, but it really isn't. It is a new machine model, in the same sense in which dataflow or graph-reduction may be regarded as machine models as much as programming methodologies. The kernels described above are software realizations of a Linda machine, but Ahuja and Venkatesh Krishnaswamy of Yale are designing a hardware Linda machine as well, based on the S/Net. The heart of the Linda machine is a box to be interposed between each processor and the S/Net bus. The box implements the Linda communication kernel in hardware, turning an ordinary bus into a tuple space. The current box is designed for the S/Net exclusively, but we are interested in general

versions that will connect arbitrary nodes and communication media as well. Installation of either the software Linda kernel or the hardware Linda boxes has the effect of uniting many physically-disjoint nodes into one logically-shared space.

Friends

Linda may be regarded as machine language for the Linda machine. We can in fact compile higher-level parallel languages into Linda. Higher-level languages may, for example, support shared variables that are directly accessible to parallel processes. If *v* is a shared variable, the compiler might translate

```
v: = expr
```

to

```
in("v", formal v_value);
out("v", expr)
```

and

```
f(...v,...)
```

to

```
read ("v", formal v_value);
f(...v_value,...)
```

We can support data objects like streams on top of Linda in the same general way.

The higher-level parallel languages that interest us particularly are the so-called symmetric languages. Symmetric languages are based on the proposition that, just as we can give names to arbitrary statement sequences and nest their execution in arbitrary ways, we should be able to name arbitrary horizontal combinations, or environments, and nest them in arbitrary ways.

Consider an arbitrary "execute sequentially" statement:

```
s1; s2; ...; sn
```

We can represent the execution of this sequence at runtime as as in Figure 8. Each box represents the execution of one statement in the sequence. The boxes are stacked on top of each other; the evaluations of successive elements occupy disjoint intervals of time, but they may successively occupy the same space. (Thus if each *si* is a block that creates local variables, we can always reuse the previous block's storage space for the next block's variables, once evaluation of the previous block is complete.) Now suppose we transpose this structure around the time-space axis, as in Figure 9. Again, each box represents the execution of a separate statement. In the resulting structure, which we refer to as an *alpha*, the evaluations of successive elements occupy disjoint regions of space and share one time; that is, they execute concurrently. If we added alphas to a programming language and wrote them

```
s1& s2& ...& sn,
```

the resulting statement calls for the execution of all *si* in parallel.

Suppose we add one more element to the alpha's definition. In most programming languages, a local variable's scope is specified explicitly and its lifetime is inferred from its scope by the following simple rule: A variable must live for at least as long as the statements that refer to it, so that they may be assured of finding it when they look. In symmetric languages we reverse this rule and infer scope from lifetime: If a variable is guaranteed to live for at least as long as a group of statements, then those statements may refer to it because they are assured of finding it when they look. Now consider the alpha: Execution of an alpha as a whole can't be complete until each of its components has executed to completion. (The same rule holds for the standard "execute sequentially" form.) Because alpha execution isn't complete until every component has been fully evaluated, no box in the alpha representation above will disappear until they all do. It follows that, if we store a named variable instead of an executable statement inside some box, then that variable should be accessible to statements in adjacent boxes, because the variable and the statement live for the same interval of time. The statement is therefore assured of finding the variable when it looks for it. Hence, symmetric languages will use alphas to create blocks as well as to create parallel-execution streams. For example, the Pascal block

```
var i: real; j: integer; begin ... end
```

becomes

```
i: real& j: integer& begin ... end
```

in Symmetric Pascal.

The alpha can in fact be used as a flexible computational cupboard. We can store any assortment of named values and active processes in its slots. Symmetric languages use alphas to serve the purpose of a Pascal record, of a Simula class or Scheme closure, of a package or a module, and in fact of an entire program or environment. All symmetric languages naturally encompass interpreted as well as compiled execution. A symmetric-language interpreter simply builds an alpha incrementally, repeatedly tacking on new elements at the end. This incrementally-growing alpha is the interpreter's environment. Because the elements of an alpha may be evaluated in parallel, the symmetric interpreter is a parallel interpreter: Each new expression the user enters is evaluated in a separate process, concurrently with all previous expressions. The values returned by all these concurrent evaluations coalesce into a single shared naming environment.

> We'd like to be able to encompass whole networks, even physically-dispersed ones, within Linda systems.

This is a mere sketch. Symmetric languages are discussed in detail elsewhere.[4] We are particularly interested in Symmetric C and Symmetric Lisp; either may be implemented on top of the runtime environment provided by the Linda kernel.

The future

We have many future plans.

The semantics of a tuple space allow it, like a file, to exist independently of any particular process or program. A tuple space might in the abstract outlive many invocations of the same program. What we'd like, then, is for tuple spaces to be regarded as a special sort of file (or equivalently, for files to be special tuple spaces). We'd like to be able to keep tuple spaces along with files in hierarchical directories. With many tuple spaces to choose from, Linda processes must be given a way to indicate which one is the current one. Once some such mechanism

has been provided, the availability of multiple tuple spaces greatly expands the system's capabilities. We can associate different protection attributes with different tuple spaces, just as we do with different files. We can use tuple spaces to support communication between user and system processes by making tuple spaces available for **out** only, **read** only, and so on. We can also allow **in** operations that remove whole tuple spaces at one blow and **outs** that add whole tuple spaces. The design and implementation of such an extension is a goal for the immediate future.

It's clear that Linda can be an interpreted command language as well as a compiled one. It would be useful to allow users to add, read, and remove tuples from active tuple spaces interactively. We've taken some preliminary steps towards implementing such a system. We'd like, too, to be able to encompass whole networks, even physically-dispersed ones, within Linda systems. We can then use Linda to write distributed network utilities like mailers and file systems. Our work on the VAX-network implementation is leading toward experiments of this sort.

Finally, we imagine, as an object of prime interest for the future, an enormous Linda machine highly optimized to support Linda primitives. We don't yet know how to build such a machine, but it's hard not to notice that very large networks with small diameters might be constructed out of multidimensional grids of S/Net-like buses. Having built such a machine, we imagine tuple space itself as the machine's main memory. (Outside of tuple space, only registers and local caches exist.) As the Linda primitives become faster and more efficient, such an architecture looks more and more like a sort of dataflow machine, but with a crucial difference. As in a dataflow machine, we can create task templates (stored in tuples), update them with new values as they become available, and mark them "enabled" when all values are filled in. General-purpose evaluator processes, much like the replicated workers discussed above, use **in** to pick out task tuples marked "enabled." But unlike the token space of a dataflow machine, a Linda machine's tuple space can store data structures as well as task descriptors. Processes are free to build whatever (distributed) data structures they want, and manipulate and side-effect them as they choose. We might even use such a Linda machine to store large databases operated upon in parallel.

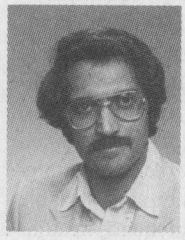
Such work is for the future. We still lack a polished Linda implementation on any machine. We hope to have one soon. And clearly we can learn a great deal by continuing to refine and to experiment with Linda kernels for present-generation architecture. This is what we plan to do.□

Acknowledgments

Rob Bjornson, Venkatesh Krishnaswamy, and Jerry Leichter are our collaborators in the Yale Linda group. Thanks also to Erik DeBenedictis, Robert Gaglianello, Howard Katseff, and Thomas London of AT&T Bell Labs.

References

1. J. T. Deutsch and A. R. Newton, "MSplice: A Multiprocessor-based Circuit Simulator," *Proc. 1984 Int'l Conf. Parallel Processing,* Aug. 1984, pp. 207-214.
2. G. F. Pfister et al., "The IBM Research Parallel Processor (RP3): Introduction and Architecture," *Proc. 1985 Int'l Conf. Parallel Processing,* Aug. 1985.
3. N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *Proc. Symp. Operating System Principles,* Dec. 1985, and *ACM TOCS*, May 1986.
4. D. Gelernter, "Symmetric Programming Languages," Yale Univ. Dept. Comp. Sci. tech. report yaleu/dcs/ rr#253, Dec. 1984.



Sudhir Ahuja obtained his MS and PhD in electrical engineering from Rice University in 1974 and 1977, respectively. He has been with AT&T Bell Laboratories, Holmdel, NJ since 1977. He is currently the head of the System Architectures Research Department. His earlier work involved associative memories, pipelining, and parallel processing. He has been involved in the design and implementation of an associative processor, high-speed buses, and multiprocessor systems. His current interests are in the field of multiprocessor architectures, concurrent programming, local networking, and the use of VLSI to implement specialized processor architectures.

Readers may write to Gelernter at the Dept. of Computer Science, PO Box 2158, Yale Station, New Haven, CT 06520-2158.

Nicholas Carriero is a graduate student in the Yale University Department of Computer Science. He received a BS from Brown in 1980 and an MS in computer science from SUNY at Stony Brook in 1983. Distributed programming languages and operating systems are his research interests.

David Gelernter's biography and photo appear following the Guest Editor's Introduction, on page 16.