Artificial Intelligence and Language Processing

Jacques Cohen Editor

LINDA IN CONTEXT

How can a system that differs sharply from all currently fashionable approaches score any kind of success? Here's how.

NICHOLAS CARRIERO and DAVID GELERNTER

Linda consists of a few simple operations that embody the tuple space model of parallel programming. Adding these tuple-space operations to a base language yields a parallel programming dialect. Use of Linda is growing slowly but steadily as we distribute versions of the system for beta test, and as several manufacturers work on their own implementations. But Linda remains stubbornly outside the research mainstream. In the research community, discussion of parallel programming models focuses mainly on message-passing, concurrent object oriented programming, concurrent logic languages, and functional programming systems. Linda bears little resemblance to any of these. How can a system that differs sharply from all currently fashionable approaches score any kind of success, even a tentative and preliminary one (which is all we claim for Linda)? We'll argue that, on balance, Linda is a simpler, more powerful and more elegant model than any of these four alternatives. The claim doesn't hold for all problems and includes, obviously, a subjective element; but we hope to convince readers that any consideration of how to write parallel programs is incomplete without considering this heterodox model also. Our method is a series of head-to-head comparisons. First we introduce the Linda model and compare it to the concurrent object model (or actually the concurrent object non-model, as we explain). Then we compare sample programs, contrasting Linda with examples coded in Parlog86, a concurrent logic language prominently featured in a recent issue of *Communications of the ACM* [30], and with a pure functional language.

THE MODEL AND THE SYSTEM'S STATUS

To write parallel programs, programmers must be able to create and coordinate multiple execution threads.

Linda is a model of process creation and coordination that is *orthogonal* to the base language in which it's embedded. The Linda model doesn't care *how* the multiple execution threads in a Linda program compute what they compute; it deals only with how these execution threads (which it sees as so many black boxes) are created, and how they can be organized into a coherent program.

The model is based on generative communication. If two processes need to communicate, they don't exchange messages or share a variable; instead, the data producing process generates a new data object (called a tuple) and sets it adrift in a region called tuple space. The receiver process may now access the tuple. Creating new processes is handled in the same way: a process that needs to create a second, concurrently executing process generates a "live tuple" and sets it adrift in tuple space. The live tuple carries out some specified computation on its own, independent of the process that generated it, and then turns into an ordinary, data object tuple.

This simple scheme has a series of important implications. First, communication and process creation are two facets of the same operation. To create processes, we generate live tuples, which turn into data object tuples; to communicate, we generate data object tuples directly. The result in both cases is the same: a new object is added to tuple space, where any interested party may access it. Second, data is exchanged in the form of persistent objects, not transient messages. The receiver may remove the tuple generated by the data-producing process, but may also leave it in tuple space, where many other processes may read it too. We can organize collections of tuples into distributed data structures, structures that are accessible to many processes simultaneously. The unification of process and data creation means that we can organize collections of processes in the same way, into "live data structures": each process in the live data structure computes and then turns into one element of the passive data structure that is returned as the result. Roughly speaking, the fact that we can write programs that use messages, distributed data structures or live data structures means that this simple model encompasses coarse, medium and fine-grained approaches to parallelism.

In fact, the implications of generative communication extend beyond parallel programming. If we think of communication as a transaction between separate programs or processes—a transaction that can't rely on standard intra-program mechanisms like shared variables and procedure calls—then communication is a fundamental problem for which few unified models exist. Two processes in a parallel program may communicate; a program in one language may use communication mechanisms to deal with a program in another language; a user program may communicate with the operating system; or a program may need to communicate with some future version of itself, by writing a file. Most systems classify these events under separate and unrelated mechanisms, but the tuple space model covers them all. Tuple space exists outside of (encompasses) the black-box programs that do the computing. Accordingly it can be used to pass information between black boxes in different languages (its own simple semantics is independent of any one language's), between user and system black boxes and between past black boxes and future ones (its lifetime isn't bounded by the programs it encompasses). Although implemented Linda systems focus only on communication within parallel programs, the generative communication model encompasses all of these forms of communication, as will future Linda systems.

Linda, to summarize, deals only with process creation and coordination. If a modular language is embedded in the Linda model, Linda becomes part of a "modular" approach to parallelism; likewise with an object oriented language, or a logic language, or an interpreted language or anything else. By not meddling in computation issues, Linda wins the freedom to coexist peacefully with any number of base languages and computing models, and to support clean, simple, and powerful operations within its own domain.

Current Status

Linda has been implemented and tested in a broad range of environments. In our group we have added the Linda operations to C and Fortran, and other groups are working on C++, PostScript (see [27]) and Scheme as base languages; [7] describes a Modula-2 Linda, [28] an object-oriented derivative. Linda runs on a wide range of parallel machines: shared-memory multi-computers like the Encore Multimax, Sequent Balance and Symmetry and Alliant FX/8; distributed-memory multicomputers like the Intel iPSC/2 and the S/Net; and Vax/VMS-based local area nets. Linda applications have shown good speedup through 64 nodes on our iPSC/2, which is the biggest machine we have used to date [19]; given the nature of the Linda implementation, these applications are very likely to continue to speed up on larger machines as well. (Linda implementations for machines like the Intel hypercube are based on distributed hash tables that scale with machine size.) Ports to other machines are in progress. A Linda simulator runs on Sun workstations, and a "Linda Machine" that supports tuple space operations in hardware is under construction [3, 25]. The system has been used for a variety of parallel programming experiments, including matrix multiplication and LU decomposition [13], DNA sequence comparison and parallel database search [11], traveling salesman, expert systems [16], charged particle transport, finite element equation solvers [34], linear programming and others. Particularly interesting from a numerical algorithms point of view is new work on a sparse system solver in Linda [5]; ongoing work includes a neural net simulator.

Several independent commercial implementations now underway will expand the range of supported architectures. In some contexts the system is now seeing what is essentially production as opposed to experimental use; Linda is used at Yale in the generation of ray-tracing displays of fractal images (by Ken Musgrave of Benoit Mandelbrot's group), and at Sandia National Labs in executing a parameter sensitivity analysis for rocket plume simulations over a local area network [34].

State of the Art?

Turn now to current research in computer science: how should parallel programs be written? At present, three approaches make up the most widely discussed group: concurrent object oriented programming, concurrent logic programming, and functional programming.[1] Linda falls into none of these categories. The tuple space model is strictly *sui generis*. Where does it fit, though, and how does it relate to the Big Three?

In the following, we will discuss each of the three named approaches in turn and compare each to Linda. In their enthusiasm for object oriented, logic based, or functional programming, proponents have argued that these models are ideal not only for computing but for parallel computing. We will argue that in fact, the strengths of all three models are irrelevant to parallelism, and generally unhelpful in dealing with process creation and coordination in parallel programs.

The following three sections vary in structure. In the first, we describe Linda's tuple space model; we contrast it with concurrent object oriented programming, and deal briefly with message passing and the Actors model as well. In the next section, we discuss parallel logic programming. We focus here on the programming examples presented in a recent article on Parlog86 [30], contrasting them with Linda's solutions to the same

[1] This list was proposed by Ehud Shapiro in a talk at the 1988 Hypercube Multicomputer conference.

problems. In the last section we discuss pure functional programming, again comparing a Linda and a purely functional solution to the same problem, and discussing the implications of the comparison.

A Note on Parallelizing Compilers

It used to be widely argued that, with sufficiently good parallelizing compilers, programmers could code in standard sequential languages and leave the compiler to worry about parallelism. Smart compilers, it was thought, could produce good parallel programs automatically, and this was important for two reasons: not only would it allow old programs to be recycled for parallelism without any programmer intervention, but it would spare programmers the unknown and presumably gruesome horrors of writing and debugging explicitly parallel programs.

Much progress has indeed been made on parallelizing compilers. But two things have become clear as well. First, compilers can't find parallelism that isn't there; the algorithms that work best on parallel machines are often new algorithms, different from the ones relied on in sequential programs. Second, evidence to date (some of which we will discuss here) suggests that writing explicitly parallel programs isn't so terribly difficult after all. A growing (if admittedly still small) number of applications programmers write parallel programs on a regular basis. Granted, parallelizing compilers are the only way to parallelize existing programs without rewriting them. Nonetheless, few researchers today seem to disagree with the contention that programmers confonting a new, compute-intensive application should have a well-designed and efficient parallel language at hand. The question is, *which* parallel language?

TUPLE SPACES AND CONCURRENT OBJECTS

We concentrate here on two topics: explaining the tuple space model, and contrasting it with concurrent object oriented systems. We briefly discuss Actor systems as well.

As a point of departure for explaining tuple space, consider the best-known of parallel programming techniques, message passing. To build a parallel program using message passing, we create many processes, all executing concurrently and asynchronously; to communicate—to disperse input data, collect final results and exchange intermediate results—processes send messages to each other. This model or something similar underlies parallel programming in systems as diverse as the native communication system supplied by Intel with the iPSC hypercube, the CSP language fragment and the Occam language that is based on it [29], and the Mach distributed operating system [35].

Message passing systems rely on three basic operations: `create-process`, `send-message`, and `receive-message`. If a sending process S has a message for a receiver R, S uses the `send-message` operation and R uses `receive-message`. In the simplest case, a single process was created automatically when we started the program; this initial process used `create-process` to create S and R.

Linda on the other hand provides four basic operations, `eval` and `out` to create new data objects, `in` and `rd` to remove and to read them respectively. If a Linda sending process S has data for a receiver R, it uses `out` to generate a new tuple. Then R uses `in` to remove the tuple. A tuple, unlike a message, is a data object in its own right. In message-sending systems, a message must be directed to some receiver explicitly, and only that receiver can read it (unless the programmer uses some special broadcast message operation, which some message systems supply and some don't). Using Linda, any number of processes can read a message (i.e., a tuple); the sender needn't know or care how many processes or which ones will read the message. Processes use the `rd` operation to read a tuple without removing it.

To create processes S and R, the initial process used `eval`. Processes can also communicate by using `eval` if they choose; when a sending process S has data for a receiver R, it can use `eval` to generate a new process M. M executes some assigned piece of code; when it terminates, it turns into a tuple. Then R uses `in` to remove the tuple.

The fact that senders in Linda needn't know anything about receivers and *vice versa* is central to the language. It promotes what we call an uncoupled programming style. When a Linda process generates a new result that other processes will need, it simply dumps the new data into tuple space. A Linda process that needs data looks for it in tuple space. In message passing systems, on the other hand, a process can't disseminate new results without knowing precisely *where* to send them. While designing the data generating process, the programmer must think simultaneously about the data consuming process or processes. In our experience, parallel programming needn't be terribly difficult, but this kind of "thinking in simultaneities" seems calculated to make it difficult.

A tuple exists independently of the process that created it, and in fact many tuples may exist independently of many creators, and may collectively form a data structure in tuple space. It's convenient to build data structures out of tuples because tuples are referenced associatively, in many ways like the tuples in a relational database. A tuple is a series of typed fields, for example `("a string", 15.01, 17, "another string")`, or `(0, 1)`. Executing the `out` statements

```
out("a string", 15.01, 17, "another
string")
```

and

```
out(0, 1)
```

causes these tuples to be generated and added to tuple space. (`out` statements don't block: the process executing out continues immediately.) An in or rd statement specifies a template for matching: any values included in the in or rd must be matched identically; formal parameters must be matched by values of the same type. (It's also possible for formals to appear in tuples, in which case a matching in or rd must have a type consonant value in the corresponding position. Values are not communicated from the in statement "backward" to the tuple, however. Formals in tuples serve only as wildcards, expanding the range of possible matches.) Consider the statement

```
  in("a string", ? f, ? i, "another
string")
```

Executing this statement causes a search of tuple space for tuples of four elements, first element "a string" and last element "another string", middle two elements of the same types as variables f and i, respectively. When a matching tuple is found it is removed, the value of its second field is assigned to f and its third field to i. If there are no matching tuples when in executes, the in statement blocks until a matching tuple appears. If there are many, one is chosen nondeterministically. The read statement, for example

```
  rd("a string", ? f, ? i, "another
string")
```

works in the same way, except that the matched tuple is not removed. The values of its middle two fields are assigned to f and i as before, but the tuple remains in tuple space.

It's now easy to see how to build data structures in tuple space. Consider one simple but important case: we can store an n-element vector V as n tuples of the form

```
  ("V", 1, FirstElt)
  ("V", 2, SecondElt)
   ...
  ("V", n, NthElt)
```

To read the jth element of the vector and assign it to x, processes use

```
  rd("V", j, ? x);
```

to change the ith element,

```
  in("V", i, ? OldVal);
  out("V", i, NewVal)
```

We discuss some more elaborate cases in the next section.

We can also use Linda to build fine grained live data structure programs. A live data structure program takes its shape from the result it is intended to yield. If the result is a vector, the program is a vector of processes, and so on. Each process in the live data structure computes and then turns into one element of the passive data structure yielded as result. Consider, for example, a program that yields an $n \times n$ matrix whose jth counter-diagonal depends (only) on the preceding counter-diagonal. The computation can proceed wave-front–wise: as soon as we know a counter-diagonal, we can compute in parallel all elements of the next counter-diagonal. (We describe a real and slightly more complicated example later.) It is easy to express such a program in Linda: we use eval statements of the form

```
  eval("M", i, j, compute(i, j))
```

to create one process for each element of the result. The function compute(i, j) uses rd to examine the values of the preceding counter-diagonal—for example,

```
  rd("M", i - 1, j, ? value).
```

As soon as the processes along the kth counter-diagonal have completed computing, they turn into passive tuples and become visible to the processes along the $(k + 1)$st counter-diagonal. Thus the computation proceeds in stages, as active processes turn into passive tuples along a wave-front from upper-left to lower-right.

Such fine-grained programs are generally impractical given our current implementations. The point is, however, that Linda can express them cleanly, and implementations on future generation machines can be expected to support them efficiently.

Linda versus Concurrent Objects and Actors

Although there is no single canonical concurrent object model, the Emerald system [8, 24] appears to be a state-of-the-art example. A concurrent Emerald program consists of a collection of objects, some passive and some with embedded active processes. New objects are generated by executing an object constructor. An object defines methods which may be invoked procedure-style by other objects. An object that is accessible to many others must protect its internal data structures against unsafe concurrent access. To do so, it embeds its methods in a monitor (the structure proposed by Hoare in [25]). By definition, only one process may be active inside a given monitor's protected routines at any one time. A second process that attempts to enter the monitor is automatically blocked on a monitor queue until the monitor becomes free. When processes enter a monitor prematurely—they needed access to some resource, an empty buffer, say, that is at present unavailable—they block themselves explicitly on a condition queue. A process that frees resources associated with condition queue j signals j, allowing some process blocked on j to continue. (In addition to these mechanisms, Emerald includes some sophisticated operations designed to relocate objects in a network. They are not germane here, though; we are interested in how Emerald can be used to express parallel programs.)

Monitors were designed originally for concurrent processes within a single address space. In Emerald, a process may access a monitor via a remote procedure call across address spaces and hence, potentially, across machine boundaries. The effect is not to alter in any way the expressivity of monitors, but rather to make them available in a new environment.

The strengths and weaknesses of monitors were explored extensively in languages like Concurrent Pascal [6], Modula [33] and Mesa [26]. We prefer Linda to monitors for several reasons. We can divide them roughly into matters of simplicity and of flexibility.

In the monitor approach, process creation, inter-process communication and synchronization fall into three separate categories: process forking or object creation, the monitor's shared state variables, and the condition queue and signal mechanism, respectively. Linda unifies the three within the context of tuple-space operations. It follows that support for monitor-based concurrency requires significant changes to a base language. Linda requires that we add new operations, and so do monitors (monitors require wait and signal operations on condition queues). But monitors also require a new envelope structure within the language, namely the monitor construct itself. Linda is simpler, and its greater flexibility is related to its simplicity.

Communication in monitor-based systems is based on procedure call (or method invocation, in the terminology of object oriented monitor systems). The same holds for distributed operating systems that are based on remote procedure call [e.g., 9]. Procedure call is inherently a synchronous operation: a procedure call blocks until a result is returned. This characteristic is problematic in the programming environment for which Linda is designed. In our experience, processes in a parallel program usually don't care what happens to their data, and when they don't, it is more efficient and conceptually more apt to use an asynchronous operation like Linda's `out` than a synchronous procedure call. Using `out`, a process can dump a result into tuple space and continue immediately; invoking a monitor procedure requires blocking until the monitor is free, and then until the requested transaction has been performed and a result is available. It's always possible to patch up a procedure-call system so that that it supports some form of asynchronous communication also. But in our experience, asynchronous communication is far more common than synchronous. This is hardly surprising; in parallel programming, the goal is to compute a result fast. In order to work efficiently, the processes that make up such a program will avoid blocking as far as possible. It's trivial, in Linda, to implement a synchronous remote-procedure-call-like operation in terms of `out` and `in`. There is no reason we know of, however, to base an entire parallel language on this one easily programmed but not crucially important special case.

Monitors are not as flexible as tuples for building distributed data structures. All elements of the distributed structure must be stored within a single monitor, which restricts access to one process at a time, or they must be stored in separate monitors, which requires that a new monitor and monitor queue be created for each new element or group of elements. Monitors don't support the transparent transition from process to data object that live data structures require.

Monitors do have points in their favor: most important, they allow all operations on a particular shared structure to be encapsulated in a simple and language-enforced way. But Linda strikes us as more powerful, simpler, and easier to integrate unobtrusively into a base language.

Where Are the "Objects"?

What does the preceding discussion have to do with object oriented programming? Curiously enough, nothing. Object oriented programming, which originated with Simula 67 and reached a cathartic climax in Smalltalk, is a model in which objects are instantiated from templates or classes. Classes may be created according to an interesting scheme that allows a new class to inherit and augment (or override) the properties of preexisting classes. The model is powerful and attractive, but irrelevant to parallelism. In the object model, each object is accessible only by way of the *methods* it defines; but what if two methods are invoked by separate processes simultaneously? We can insist that each object be an active process, that it accept messages (of the form "invoke method M") one at a time, and return reply messages to each invoker in turn. So far as parallelism goes, this is merely a message passing model. We can implement monitors, as described, but then we're left with a monitor-based model of parallelism. Alternatively, we might combine an object oriented language with Linda. Objects are generated using `out` (for passive objects) or `eval` (active ones). Passive objects are immutable; processes get their own copies by using `rd`, then invoke the methods directly. All communication with active objects goes through tuple space. Again, objects *per se* don't figure in this model's approach to parallelism. We believe in sum that the current widespread enthusiasm for "concurrent object oriented programming" is to some extent misinformed. If you are designing an object oriented parallel language, you face exactly the same design choices you would face if you were designing any other kind of parallel language. "Objects" in and of themselves do not help.

"Actors" [22] is related in some ways to concurrent object systems. The proposal is fairly old, but it continues to attract interest (see particularly Agha [2]). In an Actors system, processes (called actors) communicate by sending each other messages (called tasks). A process may respond to a message by generating one or more new processes. This kind of system is readily expressed in Linda: processes can exchange messages in the form of tuples, and generate new processes by using `eval`. Linda, of course, can also be used for distributed and active data structure programming; Actors, which is basically a message passing model, seems less suitable for these purposes. It's interesting that the Actors model seems to cover fewer programming patterns than Linda does, but in a more complicated way. Thus a process and a message in the Actors model are separate structures; in Linda, a process becomes a tuple. An Actors message contains three separate parts, a tag, a target and a communication; a tuple is merely a set of fields, any or all of which may be tag (i.e., message id) or target (address) or communication (data). Actors doesn't support the associative addressing of tasks, nor does it allow messages to be sent to not-yet-created processes. (In Linda, a tuple may be read or removed by a process created long after the tuple itself was `out`ed.) The Actors model is, on the other hand, supported by a fairly elaborate formal framework. Linda is not. But Linda exists in practical implementations on a range of commercial systems, and we don't believe this holds (at least at present) for the Actors model.

CONCURRENT LOGIC PROGRAMMING

Concurrent logic programming is a booming field. The Japanese Institute for New Generation Computer Technology (ICOT) continues work on their Parallel Inference Machine, which is intended for concurrent logic programming. A recent *Communications of the ACM* article discussed the new concurrent logic language Parlog86 [30], and collected papers on one of the first of these languages, Concurrent Prolog, have just been published (complete in 1,178 pages) by MIT Press [31]. Several years ago we published a brief discussion contrasting Concurrent Prolog with Linda [17], but new developments make a comparison between Linda and some of the Parlog86 solutions recently featured in *Communications* seem desirable. The bulk of Ringwood's presentation of Parlog86-style concurrency deals with two examples, the client-server paradigm and the dining philosophers problem. We will discuss each in turn.

Concurrent logic programming takes several forms, but in most cases the basic idea is as follows: we can specify many parallel activities by use of a "parallel conjunction," which states that some result depends on a series of sub-results, all of which may be pursued simultaneously. A collection of "guarded clauses" allows us to specify that only one of a series of parallel conjunctions be performed, but the selection criteria (the "guards") can be evaluated simultaneously. Parallel execution threads communicate by means of shared logical variables, which are initially unbound ("uninstantiated") but can't be referenced until one party to the communication binds them to a value. Data streams are implemented by "partially instantiated" shared logic variables. The data producer successively appends (in effect) new elements to a list or stream of elements; the data consumer reads the stream. It all adds up to a parallel version of logic programming—to a parallel version (in other words) of what Ringwood [30, p. 11] and many others have called a very high level approach to programming. Very high level programs have mathematical properties that their lowlier brethren lack, but we focus here on a more concrete issue: presumably, very high level programs are more elegant and concise than other kinds.

In the client-server paradigm, many client processes must communicate with a single server process. The problem is usually associated with distributed operating systems (the server might provide a file service, mail, name-location or other resource management function); in our experience, it is equally or more important in parallel programming.

Consider the Parlog86 solution (Figure 1). It creates, in effect, three processes, one corresponding to each paragraph of code in the figure. The "server" process repeatedly reads a stream of requests. It accomplishes this by binding the name `Transaction` to the first element and the name `More-transactions` to the rest of some input stream. It proceeds to `respond` to `Transaction` (that is, it services the stream's head request), and then it applies the same procedure recursively to the remainder of the stream. When the `client` process needs to communicate with the server, it adds to the stream a message of the form `transaction(Reply)`. `Reply` is an "uninstantiated variable": it is forwarded to the server as (in effect) a name without a binding. When service to the client is complete, the server binds the client's `Reply` variable to the string `"Roger Roger"` (the code for this step isn't shown). By executing `wait("Roger Roger")`, the client blocks until this binding is accomplished.

The complete system entails more than a server and client processes, though. A Parlog86 stream may be appended to by a single process only, but ordinarily there are many clients. The solution is for each client to append messages to its own private stream, and for these multiple client streams to be merged to form the single request stream that the server expects. Merging is the job of the third process: it examines each client stream (the example shown assumes that there are exactly two, `Stream1` and `Stream2`), and merges them into a single `Outstream`, which the server scans in turn. (Invocations that cause names to be bound correctly—e.g., that cause the `merge` process's `Outstream` to be identified with the `More-transactions` stream read by the server—are required as well, but the code is omitted.)

A C-Linda version is also shown in Figure 1. It assumes again that the server will scan down a stream of requests. In Linda, the stream exists as a series of numbered tuples, of the form

```
("request", 1, FirstRequest)
("request", 2, SecondRequest)
...
```

By successively incrementing the `index` field that forms part of the identification template in the `in` statement, the server can `in` each of these tuples in sequence. To respond to the *j*th request, it places its response in a tuple labeled `j`. Whichever process made the *j*th request will be waiting (again using `in`) for a tuple so labeled.

Parlog86 version:

```
server([Transaction | More-transactions], Data,...) <--
  respond(Transaction, Data, Newdata, ...) :
  server(More-transactions, Newdata, ...).

client([transaction(Reply) | More-transactions], ... ) <--
  wait(Reply), ... .

wait("Roger Roger") <--.

merge([Item | Stream1]), Stream2, [Item | Outstream]) <--
  merge(Stream1, Stream2, Outstream).
merge(Stream1, [Item | Stream2],  [Item | Outstream]) <--
  merge(Stream1, Stream2, Outstream).
```

C-Linda version:

```
server()
{
   int index = 1;
   ...
   while (1) {
    in("request", index, ? req);
    ...
    out("response", index++, response);
   }
}

client()
{
  int index;
  ...
  in("server index", ? index);
  out("server index", index+1);
  ...
  out("server", index, request);
  in("response", index, ? response);
  ...
}
```

FIGURE 1. Server-Clients in Parlog86 and in C-Linda

Client processes must first establish where the end of the stream is. To do so, they read and update a tuple labelled "server index". If many clients attempt to update the server index simultaneously, they succeed one-at-a-time and the index is updated safely. Once a server has determined the current end-of-stream index, it adds its request-tuple to the stream. It then blocks using `in` until a response arrives.

How do the solutions compare? They are comparable in length (in both cases initialization code is omitted, amounting to several lines in C-Linda and probably about the same for Parlog86). We would argue that the C-Linda code is easier to understand. This contention is subjective, but surely the Linda version is no *harder* to understand. The real difference involves the flexibility of the two versions. The Parlog86 version requires an extra process to perform merging; the Linda version does not. More important, the code of Parlog86's `merge` process depends on how many and which streams are being merged. The `merge` process examines the head of each client stream explicitly. Consider how the Parlog86 code would look if there were 20 or 100 streams to be merged instead of two. The Linda code is insensitive to number of clients. Nothing in the code reflects this number, or needs changing when the number or identity of client processes change.

There is nothing unrealistic in imagining 20 or 100 input streams to some server. In our experience, this kind of pattern occurs often in master-worker–style parallel applications. Tasks (work assignments) are performed by some arbitrary number of identical worker processes in parallel. A master process sets up the computation, then reels in and coordinates the results. The workers are clients and the master is the server.

This master-worker version of the server-client paradigm raises an interesting and more subtle problem. The order in which task results are returned to the master may be irrelevant—so long as the master gets everything eventually, there's no need to deal with the results in FIFO order. If ordering isn't needed, much better to leave it out: it complicates the code and may require nontrivial processing at runtime. But in Parlog86, client messages *must* be ordered: there is no way to build an unordered stream. In Linda, clients can simply drop messages into tuple space using `out`, omitting the index field. The server withdraws tuples in arbitrary order using `in`.

The merging problem in Prolog-derived concurrent languages is well-known. Some researchers argue that the solution is simply to add a new *merge* primitive. This solution points to a deeper problem, however. The client-server example is important because it brings into play one of Linda's most important advantages, flexibility. One particular species of stream is canonical in Parlog86. A Parlog86 stream has one writer, and perhaps a series of readers. Linda has no canonical stream, but it's easy to build a variety of important types. The client-server example requires a one-reader many-writer stream. Other examples require one-writer, many-remover streams. This type of stream's first element is removed repeatedly by any one of a group of interested processes—a situation that arises where (for example) a master process generates an ordered list of tasks, and each worker repeatedly removes and carries out the head task. This kind of stream is again simple to build in Linda, but unidiomatic in Parlog86. (Note that this example's crucial characteristic is that many processes jointly disassemble, not merely read, the stream.)

We turn briefly to the main focus of the Parlog86 article, the dining philosophers problem. A round table is set with some number of plates (traditionally five); there is a single chopstick between each two plates, and a bowl of rice in the center of the table. Philosophers think, then enter the room, eat, leave the room and repeat the cycle. A philosopher can't eat without two chopsticks in hand; the two he needs are the ones to the left and the right of the plate at which he is seated. If the table is full and all philosophers simultaneously grab their left chopsticks, no right chopsticks are available and deadlock ensues. To prevent deadlock, we allow only four philosophers (or one less than the total number of plates) into the room at any one time.

What is the point of solving such a silly problem? According to Ringwood, dining philosophers "is a benchmark of the expressive power of new primitives of concurrent programming and stands as a challenge to proposers of these languages" [30, p. 11]. While we can't agree that the problem is as central as Ringwood makes it out to be, it has indeed been used as a benchmark of expressivity since Dijkstra first suggested it.

The Parlog86 solution (Figure 2) is too complicated to explain here, but generally speaking it uses a series of processes to implement each state that a philosopher can occupy—thinking, preparing to eat, and eating. Philosophers circulate along data streams among these process-states, passing along the way through intermediate processes that merge streams, match available chopsticks to hungry philosophers and so on. In the Linda solution, assume for concreteness that the number of place-settings and of philosophers is five. There are four "room tickets," represented by four tuples. A philosopher who is ready to enter the dining room uses `in` to grab a ticket. (If there are no free tickets, he will block until some other philosopher leaves and releases his ticket.) Once inside, he uses `in` to grab his left chopstick and then his right chopstick. Each chopstick is again represented by a separate tuple. When he's done eating, he replaces both chopsticks and his room ticket.

(Careful readers will notice that, if the Linda kernel is "unfair"—if it can repeatedly bypass one process blocked on `in` in favor of others—the Linda solution allows indefinite overtaking or livelock. A slow philosopher could remain blocked on an `in("room ticket")` statement while a speedy one repeatedly `out`s a room ticket and then grabs it again, leaving the slow philosopher still blocked. The Parlog86 solution suffers from a similar problem. Both solutions should be read, then, under a fair implementation assumption.)

The Linda solution is, in a sense, nothing special. It uses `in` and `out` as counting semaphore operations, and essentially the same solution is possible in any system that supports distributed semaphores. But here again, we believe that the contrast between Linda and Parlog86 makes an important point. When a problem has a simple solution, a useful system will give programmers access to the simple solution. Forcing complex solutions to simple problems makes us suspect that a language has chosen the wrong "abstraction level" for its primitives, chosen operations with too many policy decisions built-in and too few left to the programmer. Such languages are ostensibly "higher level" than ones with more flexible operations, but this kind of high-levelness dissipates rapidly when programmers step outside the (often rather narrow) problem spectrum that the language designer had in mind. A Parlog86 proponent would almost certainly call Parlog86 a "higher-level" language than C-Linda; but we've shown that it is somewhat easier to solve the client-server problem in C-Linda than in Parlog86, and much easier

Parlog86 solution:

The Parlog86 solution in Ringwood's 1988 paper involves 70 lines of code divided into processes **cloister**, **refec**, **table**, **cell**, **finder** and **merge**. Five diagrams help to explain the structure of the solution. But these 70 lines are only part of the program: "Not all clauses will be illustrated [*i.e.*, not all code will be reproduced], only those that have informative diagrams. The communication between calls is often much more easily comprehended by means of such illustrations," Ringwood writes [p.19].

C-Linda solution:

```
phil(i)
   int i;
{
   while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i+1)%Num);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1)%Num);
    out("room ticket");
   }
}

initialize()
{
   int i;
   for (i = 0; i < Num; i++) {
    out("chopstick", i);
    eval(phil(i));
    if (i < (Num-1)) out("room ticket");
   }
}
```

FIGURE 2. Using Parlog86 and C-Linda to Solve the Dining Philosophers Problem

to solve the dining philosophers problem. Each of the problems involves significant issues in concurrent programming. Both were chosen as showpieces for Parlog86 by a proponent of concurrent logic programming.

FUNCTIONAL PROGRAMMING

We have discussed problems for which Linda has a shorter and neater solution than some alternative, but we turn now to a comparison that cuts the other way. Figure 3 shows two versions of a program to compute the "similarity value" of two DNA sequences. One version is written using a pure functional language (the language shown is Crystal [14]); the other version uses C-Linda. Although the two programs do not differ by much, the C-Linda version is slightly longer.

Unsurprisingly, there's more to this comparison than meets the eye. Advocates of pure functional languages for parallel programming argue as follows: Programmers shouldn't decide how to parallelize their algorithms. Armed with a pure functional language, they should express their algorithms as a set of recursion equations. Pure functional languages impose a decisive limitation: they don't allow assignment statements, and hence no variable's binding can ever change. This restriction simplifies the task of finding an algorithm's implicit parallelism; programs expressed in functional languages are therefore good candidates for automatic

```
a = [<sequence data ... >],
b = [<sequence data ... >],
m=6,n=6,
?a,
?b,
h(i,j)=
  <<i=0 ->0,
    j=0 ->0,
    else ->\max[t1,t2,t3,0]>>
where(
t1=h(i-1,j-1)+s(x,y),
t2=max((h(i-1,j)-1.3333),(p(i-1,j)-.3333)),
t3=max((h(i,j-1)-1.3333),(q(i,j-1)-.3333)),
   s(x,y)=
     <<x=y->1,
       else -> -.3333>>,
              x=a[i-1],
              y=b[j-1]
)
where(
p(i,j)=
  <<i=0 ->0,
    j=0 ->0,
    else ->max((h(i-1,j)-1.3333),(p(i-1,j)-.3333))>>,
q(i,j)=
  <<i=0 ->0,
    j=0 ->0,
    else ->max((h(i,j-1)-1.3333),(q(i,j-1)-.3333))>>
),
? [[h(i,j) | 0 <= i <= m] | 0 <= j <= n],
```

```
extern char     a[], b[];
extern int      m, n;
extern float    max();

float h(i, j)
{
  float t1, tt1, t2, tt2, t3, tt3;

  if (i == 0 || j == 0) return 0.0;
  rd("H", i-1, j-1, ? tt1);
  rd("H", i-1, j, ? tt2);
  rd("H", i, j-1, ? tt3);
  t1 = tt1 + (a[i-1] == b[j-1]) ? 1 : -.3333;
  t2 = max(tt2-1.3333, p(i-1,j)-.3333);
  t3 = max(tt3-1.3333, q(i,j-1)-.3333);
  return max(t1, max(t2,  max(t3, 0)));
}

float p(i,j)
{
  float t;

  if (i == 0 || j == 0) return 0.0;
  rd("H", i-1, j, ? t);
  return max(t-1.3333, p(i-1,j)-.3333);
}

float q(i,j)
{
  float t;

  if (i == 0 || j == 0) return 0.0;
  rd("H", i, j-1, ? t);
  return max(t-1.3333, q(i,j-1)-.3333);
}

lmain()
{
  int   i, j;

  for (i = 0; i <= m; ++i)
    for (j = 0; j <= n; ++j)
      eval("H", i, j, h(i, j));
}
```

FIGURE 3. DNA Sequence Comparison Using Recursion Equations and C-Linda

parallelization by smart compilers or run-time systems. These arguments have been made for some time by proponents of dataflow functional languages [1], and also (for example) by proponents of Crystal [14, 15]. The promised advantages of the approach are substantial. Source programs are portable and completely machine independent. Programmers don't bother with parallelism; they produce mathematical expressions instead. Mathematical expressions are more conducive to formal verification than ordinary programming languages, they are easier to debug, and the advantages of mathematical expressions in terms of elegance and conciseness are well known. Thus Turner writes that "a basic difficulty" of programming languages that are not "functional" is that "they are very long winded, in terms of the amount one has to write to achieve a given effect . . . [32, p. 5]"

We can put these arguments in perspective by examining a concrete case. In The DNA-sequence comparison problem (Figure 3), we need to compute a similarity value for two sequences; this value is designed to capture a geneticist's qualitative judgement. In the algorithm shown, we compute a similarity matrix whose *i, j*th entry is the similarity between the first *i* elements of one sequence and the first *j* of the other. The matrix can be computed in wavefront fashion: first the upper-left element, then the second counter-diagonal, then the third and so on.

This is an ideal problem for any functional language. It's simple and convenient to express this algorithm as a series of equations giving each matrix element's dependence on previous elements. A smart compiler could establish that all elements of the similarity matrix can be computed in parallel, with the elements along the *j*th counter-diagonal blocking dataflow-style until all previous counter-diagonals have been computed.

The C-Linda version isn't a set of equations; it is a fine grained explicitly parallel program. It builds a live data structure called *H*. The *i, j*th element of *H* is a process that computes, then turns into, the corresponding element of the similarity matrix.

Having introduced the two versions, we can contrast them. The C-Linda version is longer, but the difference in compactness seems minor. One major underlying assumption of the functional-languages advocates—that explicitly-parallel programs are hard to design and to understand—gets no support from this comparison. The

C-Linda version was easy to design. Some readers will find the Crystal version easier to understand, but we doubt whether too many will find a dramatic difference in comprehensibility. The real difference between the two lies elsewhere The C-Linda version is a parallel program. The Crystal version is a specification that a compiler might turn into a parallel program. In fairness, then, the two versions are incomparable, one being a program and the other, a program specification. The contention that parallel *programs* are necessarily clunkier, more complex and less elegant than functional language program *specifications* is too often accepted without comment, and it should not be, but the real issue lies elsewhere. Are the operations that create and control parallelism best left in the programmer's hands, or should they be turned over to the compiler?

Leaving them to the compiler is attractive in some ways. It releases the programmer from any responsibility for the granularity of his computation. The intent of projects like Crystal is for the compiler to gather up the fragments of a parallel algorithm and bundle them together in any way that seems appropriate—into a few big bundles if the target architecture includes a few powerful processors, into many small bundles if the target machine includes many less-powerful processors. In principle, the kinds of analysis used in the Crystal compiler would probably work for a program like the C-Linda DNA comparison also, but we haven't investigated them and don't know for certain. It's also true that Crystal targets special purpose SIMD architectures like the Connection Machine, as well as asynchronous computers; Linda is designed for asynchronous machines only.

We aren't sold on the compiler alternative, though, for two reasons. One involves a matter of detail related to the point discussed earlier. The other concerns underlying concept.

On the general purpose parallel machines of which we are aware, it is far more efficient to create a smaller number of processes, and put each in charge of computing an entire sub-block of the comparison matrix, than to put one process in charge of each element. Even on machines that support cheap processes (lightweight tasking), creating, scheduling and passing data among a very large number of processes can be a considerable expense. The smart Linda programmer therefore starts with the program shown in Figure 3, but immediately restructures it for efficiency by using "interpretive abstraction" [12]—he replaces the live data structure with a passive one, and raises the processes one level in the conceptual scheme each process fills in many elements, rather than becoming a single element. We discuss the performance of this kind of solution to the problem in [11] (it shows good speedup). Interpretive abstraction is a fairly simple programming technique, but it's only possible if the parallelism tools are under the programmer's control. In a pure functional language, it appears to be impossible. Not to worry: the functional language compiler is designed to take care of this problem (to carry out the equivalent of "interpretive abstraction") automatically. The detail that bothers us is that this capability has yet to be demonstrated, and achieving it automatically and generally strikes us as a difficult problem. Consider one aspect of the difficulty for the DNA problem. A reasonable approach is to divide the matrix into horizontal bands; a band is filled in by a single process; in general, one process can fill in many bands. How many bands should there be? Fewer, deeper bands means loss of efficiency during startup and completion, because the delay in starting work on the second band depends on the first band's depth. A greater number of shallower bands increases inter-band communication overheads. Given the Linda program, it is easy to determine a good band size by running some experiments. The Crystal system (and all other approaches to automatic parallelization) takes it upon itself to determine this number *a priori* and automatically.

These issues of granularity will become less important over time. Newer machines will be capable of supporting increasingly fine-grained programs (although we don't expect that the optimal granularity issue will ever disappear entirely). But suppose we had a machine on which the finest grain programs run well. Is it then reasonable to surrender control over parallelism and luxuriate in the pure sunlight of recursion equations? Absolutely not, because in too many important cases, recursion equations have nothing to do with the programs we need to write.

Again, the DNA problem is a good starting point. The approach we've discussed so far is—in our experience—interesting but wrong. The goal of geneticists in practice is to compare a newly-discovered sequence against a large database of existing sequences. The problem is compute-intensive and parallelism is called for, but in our experience it is usually more efficient to run many conventional, sequential searches in parallel than to parallelize individual searches. One efficient approach to the problem is easy to express in C-Linda (Figure 4). The basic method is to create many identical searcher-processes, each initialized with a copy of the target sequence; a master process hands out sequences from the database; workers execute comparisons, and return the results to the master. There are two subproblems: (1) Sequences in the database vary greatly in length, so we need dynamic sequence-assignment; e.g., one worker may finish 10 short comparisons before another finishes one long one; and (2) The database is too big to fit in core and must be played out gradually. Both problems are solved easily using C-Linda. Each sequence in the database is dumped as a tuple into tuple space; searcher processes repeatedly grab a tuple, execute the comparison and dump a tuple holding the result back into tuple space, whence the master retrieves it. The program is simple to write and to understand because of the underlying physical model: a tuple is a quasi-physical thing that can be created, added to a pile, removed from a pile (Figure 5). Recursion equa-

```
master()
{
while (get(new_sequence)) {
    out("task", new_sequence);
    if (++tasks > HIGH_WATER_MARK)
        do {
             in("result", ?result);
             update best-results list;
        } while (--tasks > LOW_WATER_MARK)
    }
while (tasks--) {
    in("result", ?result);
    update best-results list;
    }
report results;
}

searcher()
{
do {
    in("task", ?seq);
    compare "seq" to target sequence, yielding "value";
    out("result", value);
} while (search is not complete)
}
```

FIGURE 4. Parallel Sequence Database Search in C-Linda

tions are in no way helpful in understanding this simple physical process, and functional languages are inappropriate for this kind of programming.[2]

In Figure 5, a tuple space visualization tool, built by Paul Bercovitz of our group, displays a series of windows onto tuple space, one for each "tuple class" in the program. (The tuples within a given class have the same type signature and have been determined by the link-time tuple analyzer to behave—roughly speaking—in the same way.) Each tuple within a class at any given time is represented by a sphere in the appropriate window; users can display the contents of a tuple by mousing on its sphere. The figure shows four snapshots of an executing DNA database-searcher that is similar to the program described in the text, executing with a master and five worker processes. The first snaphot (a), soon after start-up, shows the single tuple holding the target sequence (in the upper left-hand window), and the first of many tuples that will hold sequences from the database (in the wide window second from the top). In snapshot (b), about 100 sequence tuples are awaiting available workers, and about 20 result tuples are waiting for the master, who has been busy adding sequence tuples to tuple space. In snaphot (c), there are only a few sequence tuples remaining to be searched; in the last snaphot (d), the job is done. No sequence or result tuples remain. The worker processes were live tuples, created using `eval`; having completed, each worker has become an ordinary data tuple, and the five tuples corresponding to the five finished workers appear in the upper right-hand window. This version of the program uses a stream of sequence tuples rather than an unordered bag; the tuple that appears in the upper middle window is used to maintain a pointer to the current head-of-stream.

The tuple space visualizer is a promising debugging tool, but it points to a more basic issue as well. A useful programming language makes it easy for the programmer to embody a mental model directly in working code. A program that embodies a mental model simply and directly is easier to write initially, and easier to understand once it exists as a working application. The parallel database search described in the text is one example of a program that is at least as easy (we would argue far easier) to imagine in terms of objects to be manipulated than in terms of equations to be solved. In Linda, a mental model based on objects is easily embodied in working code. Merely by displaying the state of tuple space as it evolves, the tuple space visualizer gives the user a strong "feel" for an application—how it works and how it is progressing. (In this case, for example, how many sequences have been searched so far, how many results have been tabulated, whether the master is dumping tuples into tuple space or waiting for tuples to be cleared out, and so on.) In short, if the definition of a "higher level language" is a language that is relatively closer to the programmer's way of thinking, it is impossible for us to accept the claim that functional languages are necessarily higher-level than C-Linda.

Granted, this program is a single data point, but consider some others. Neither of the examples discussed in the previous section can be solved in a pure functional language. The solutions in both cases depend on nondeterminism (concretely, whichever process shows up first gets access to some resource), and nondeterminism is impossible in functional languages. A class of programs we're particularly interested in now can be described as process lattices [16], i.e., heuristic programs structured as hierarchies of concurrent expert processes, with more general or abstract decision procedures appearing at higher levels. This is a promising structure for use in a problem area that (we believe) will become increasingly important: the construction of real time expert monitors. Nodes on the bottom rank of the process lattice are wired directly to external data sources, and incoming data values filter upward through the lattice. This structure is conceived in terms of communicating processes; recursion equations are the wrong model (or at the very least, a strongly counter-intuitive one). Whiteside and Leichter have shown that Linda programs can be made to perform well on a collection of heterogeneous machines on a local area network [34]; if we break an application into modules, and run a compute-intensive back-end on a collection

[2] Recent work has lead to a refined approach and further performance improvements in the Linda DNA-search program, without invalidating the points made here; the new version is discussed in [5].

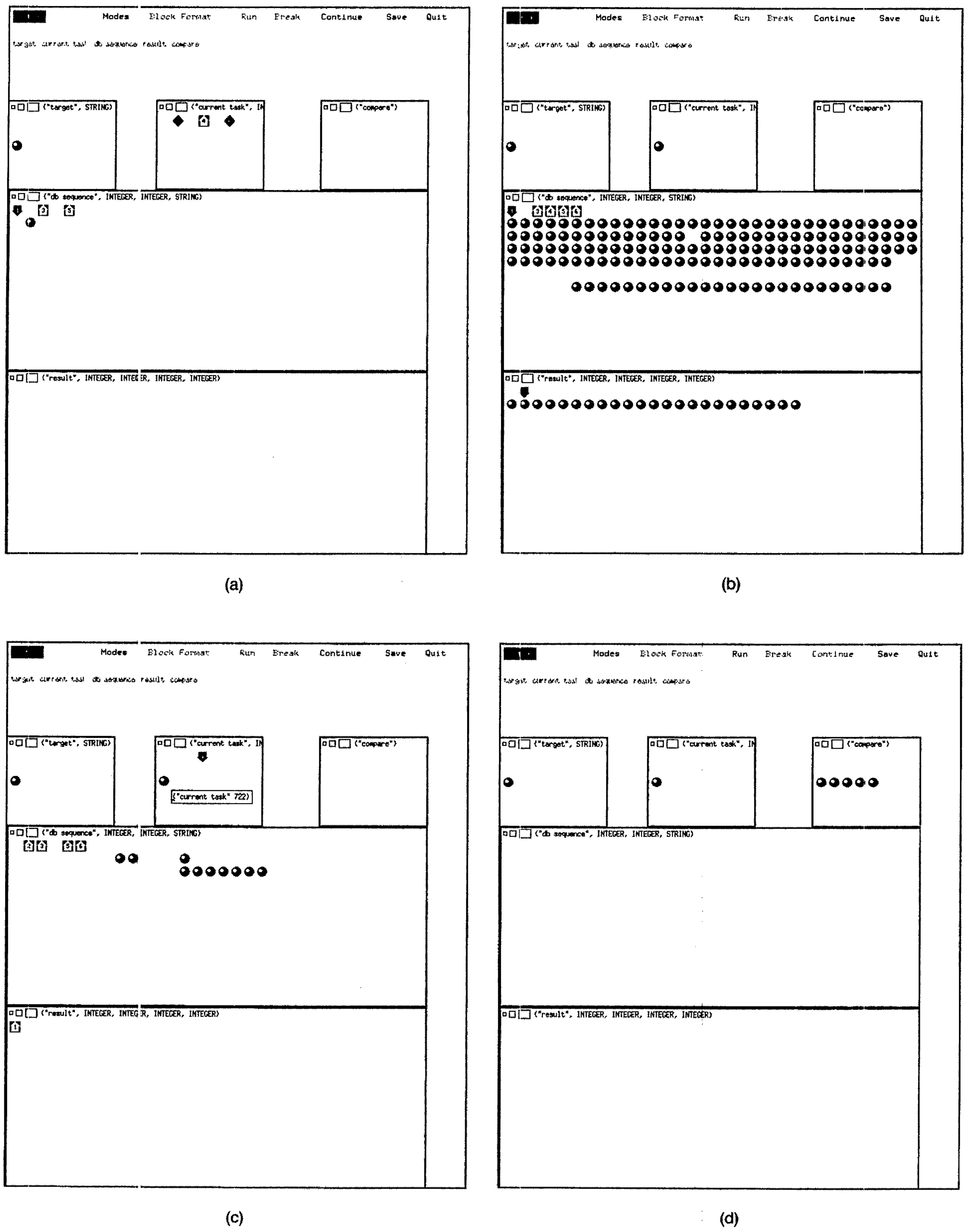
Modes
Block Format
Run
Break
Continue
Save
Quit
target current task db sequence result compare
("target", STRING)
("current task", IN
("compare")
("db sequence", INTEGER, INTEGER, STRING)
("result", INTEGER, INTEGER, INTEGER, INTEGER)
("current task" 722)
(a)
(b)
(c)
(d)

FIGURE 5. Tuples as Quasi-Physical Objects

of parallel machines and the display-manager front-end on a workstation, the programmer needs a way to specify inter-module (as well as intra-module) communication, and Linda gives him a way; here, once again, recursion equations would be out of place.

CONCLUSIONS

We set out to compare Linda to three leading models of parallel programming. We can now summarize the arguments and draw some conclusions.

We have no intention of closing the book on monitors, and we're sure that monitor research and programming will continue for quite awhile. On a different note, we don't doubt that object oriented programming is a powerful and attractive model, and we look forward to investigating a combination of the object model with Linda. *Concurrent object oriented programming* on the other hand is a phrase with a nice ring, but (in and of itself) little or no meaning. It strikes us as more of a marketing than a technical term, used mainly to freshen up respectable but slightly shelf-worn ideas like message passing and monitors. Discussion of parallelism models would be clearer and better-defined, we think, if this term disappeared.

Logic programming is obviously a powerful and attractive computing model as well. Notwithstanding, we continue to believe that the tools in concurrent logic languages are too policy-laden and inflexible to serve as a good basis for most parallel programs. Applications certainly do exist that look beautiful in concurrent logic languages, and we tend to accept the claim that virtually any kind of parallel program structure can be squeezed into this box somehow or other (although the spectacle may not be pretty). But we believe that, on balance, Linda is a more practical and a more elegant alternative.

Pure functional languages are a trickier issue. One point is immediate: these languages can't be the *whole* story, because in too many cases they fail to provide the expressivity we need. Many (if not most) of the programs we deal with are conceived in terms of objects to be manipulated, not equations to be solved. But restricting our attention to those problems that do have elegant functional solutions, like the DNA sequence-to-sequence comparison—shouldn't we be tempted to recode these programs in functional languages, flip the autopilot switch, lean back, have a beer, and leave the parallelizing to them? In fairness we *might* be tempted, under the right circumstances. We'd stand to gain remarkably little in terms of conciseness or elegance over C-Linda, and nothing in terms of machine independence, so far as asynchronous machines are concerned. (Again, Linda doesn't help in programming SIMD machines. We think it will be an excellent language for massively-parallel, fine-grained asynchronous machines, though.) It seems to us that, speaking in terms of the machines we deal with, the only good reason to "flip autopilot" is performance. If these systems can be shown to perform better than Linda and a competent programmer, we'll use them (or at any rate, we will gratefully appropriate their compiler technology for our Linda compilers). Constructing a high-performance autopilot will be difficult; for our part, we'd find the search more compelling if we didn't have good, demonstrated ways to deal with these programming problems right now. But any difficult research problem stands to yield unanticipated benefits if solved successfully, and we look forward to following these efforts.

On a positive note, Linda is prepared to absorb computational wisdom from any source that can provide some. Our own preference is to program in C or Lisp, and our numerical analyst colleagues are reasonably happy with Fortran; as we've emphasized, though, Linda can co-exist with almost any model of computing. We'd like to see the Linda model flourishing in a wide variety of machine environments and language models. We're actively working toward this end, and believe we're making progress.

We don't want to leave the impression, though, that Linda is a colorless, neutral system. The Linda model—a swarm of active tuples surrounded by a cloud of passive ones—is suggestive in itself. One program now in design is a real-time "tracking simulation" of a subset of patients in a hospital; patients (passive tuples) are monitored and updated by a succession of manager and scheduler processes (active tuples). The goal is to monitor anomalies and rationalize the scheduling of tests and treatments by maintaining a software microcosm of the external world. Another program involves a biologically-accurate simulation of neuron networks; each neuron is represented by a passive tuple; there is a collection of identical simulation engines (active tuples), and each engine repeatedly grabs a neuron, executes one step of the simulation and blows it out the back into a recirculating pipeline. The more simulation engines, the faster the program runs.

The Linda operating system environment we're now building accommodates multiple first class tuple spaces [20]. A tuple space is created with certain attributes—for example, some tuple spaces are persistent, and persistent tuple spaces constitute the file system. Whole tuple spaces can be treated as single objects: they can be suspended, archived, reactivated or snapshotted *en masse*. As always, a tuple space may contain active as well as passive tuples, which suggests that an ordinary, passive file may also contain active processes if we choose to toss some in. Flocks of passive tuples might be stored alongside active shepherd processes—a file of mail messages, for example, might contain a sorting-and-scanning daemon to add new messages to the file and keep things in order. We might approach any of these three possibilities in contexts other than Linda, but Linda suggested them. Progress on these (and other less esoteric) projects continues.

Acknowledgments. This work is supported by National Science Foundation SBIR grant ISI-8704025, by National Science Foundation CCR-8601920 and CCR-

8657615, and by ONR N00014-86-K-0310. The authors thank the Yale Linda group, and particularly its senior members, Jerrold Leichter, Robert Bjornson, and Venkatesh Krishnaswamy, for their vital contributions to this research.

REFERENCES

1. Agerwal, T., and Arvind, Data flow systems. *Computer 15*, 2 (Feb. 1982), 10–14.
2. Agha, G. *ACTORS: A model of concurrent computation in distributed systems.* The MIT Press 1986.
3. Ahuja, S., et al. Matching language and hardware for parallel computation in the Linda machine. *IEEE Trans. Computers 37*, 8 (Aug. 1988), 921–929.
4. Ahuja, S., Carriero, N., and Gelernter, D. Linda and friends. *IEEE Computer 19*, 8 (Aug. 1986), 26–34.
5. Ashcraft, C., Carriero, N., and Gelernter, D. Hybrid DB search and sparse LDL^T factorization using Linda. Yale University Department of Computer Science Research Report (Jan. 1989).
6. Brinch Hansen, P. The programming language Concurrent Pascal. *IEEE Trans. Softw. Eng.*, SE-1, 2 (1975), pp. 99–206.
7. Borrman, L., and Herdieckerhoff, M. Linda integriert in Modula-2—ein Sparchkonzept für portable parallel Software. In *Proceedings 10. GI/ITG-Fachtagung Architektur und Betrieb von Rechensystemen.* (Paderborn, March 1988).
8. Black, A., et al. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (Portland, Ore., Sept. 29–Oct. 2, 1986), pp. 78–86.
9. Birrel, A.D., and Nelson, B.J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), pp. 39–59.
10. Carriero, N., and Gelernter, D. The S/Net's Linda kernel. *ACM Trans. Comput. Syst. 4*, 2 (May 1986), pp. 110–129.
11. Carriero, N., and Gelernter, D. Applications experience with Linda. In *Proceedings of the ACM Symposium on Parallel Programming*, July 1988.
12. Carriero, N., and Gelernter, D. How to write parallel programs: A guide to the perplexed. Yale University Department of Computer Science Research Report 628 (Nov. 1988).
13. Carriero, N., Gelernter, D., and Leichter, J. Distributed data structures in Linda. In *Proceedings of the ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 13–15, 1986).
14. Chen, M. C. A parallel language and its compilation to multiprocessor architectures or VLSI. In *Proceedings of the ACM Symposium Principles of Programming Languages* (St. Petersburg, Fla., Jan. 13–15, 1986).
15. Chen, M. C., et al. The influence of machine architecture on programming style and performance. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications* (Pasadena, Calif., Jan. 19–20, 1988).
16. Factor, M., and Gelernter, D. The parallel process lattice as an organizing scheme for realtime knowledge daemons. Yale University Department of Computer Science Technical Memo (March 1988).
17. Gelernter, D. A note on systems programming in Concurrent Prolog. In *Proceedings of the International Symposium on Logic Programming* (Atlantic City, N.J., Feb. 6–9, 1984).
18. Gelernter, D. Generative communication in Linda. *ACM Trans. Prog. Lang. Syst. 7*, 1 (1985) pp. 80–112.
19. Galernter, D. Getting the job done. *Byte 13*, 12 (Nov. 1988), 301–310.
20. Gelernter, D. Elastic computing envelopes and their operators (Linda 3). Yale University Department of Computer Science Research Report (Jan. 1989).
21. Hoare, C.A.R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), pp. 549–557.
22. Hewitt, C. Viewing control structures as patterns on passing messages. *J. Art. Intell. 8*, 3 (June 1977), 323.
23. Ingalls, D. H. H. The Smalltalk-76 programming system design and implementation. In *Proceedings of the 1978 ACM Symposium Principles of Programming Languages*, pp. 9–15.
24. Jul, E., et al. Fine-grained mobility in the Emerald system. *ACM Trans. Comp. Sys. 6*, 1 (Feb. 1988), pp. 109–133.
25. Krishnaswamy, V., et al. The architecture of a Linda coprocessor. In *Proceedings of the ACM Conference on Computer Architecture* (1988).
26. Lampson, B.W., and Redell, D.D. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (Feb. 1980), pp. 105–117.
27. Leler, W. PIX, the latest NEWS. In *Proceedings of the COMPCON Spring '89* (San Francisco, Feb. 1989), IEEE.
28. Matsuoka, S., and Kawai, S. Using tuple space communication in distributed object-oriented languages. In *Proceedings of OOPSLA '88* (San Diego, Sept. 25–30, 1988), pp. 274–284.
29. May, M.D. Occam. *ACM SIGPLAN Notices 18*, 4 (1983), pp. 69–79.
30. Ringwood, G.A. Parlog86 and the dining logicians. *Commun. ACM 31*, 1 (Jan. 1988), pp. 10–25.
31. Shapiro, E. ed. *Concurrent Prolog collected papers.* Vols 1 and 2. The MIT Press, 1987.
32. Turner, D.A. Recursion equations as a programming language. In *Functional Programming and its Applications*, J. Darlington, P. Henderson and D.A. Turner, eds., Cambridge University Press (1982) pp. 1–28.
33. Wirth, N. Modula: A language for modular multiprogramming. *Softw. Pract. and Exp. 7* (1977) pp. 3–35.
34. Whiteside, R.A., and Leichter, J.S. Using Linda for supercomputing on a local area network. Yale University Department of Computer Science Technical Memo (March 1988).
35. Young, M. *et al.*, The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (Nov. 1987), pp. 63–76.

CR Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Language Classifications—*parallel languages*; D.3.3: Language Constructs
General Terms: Languages
Additional Key Words and Phrases: Concurrent object-oriented programming, monitors, concurrent logic programming, functional programming, Linda, parallelism

ABOUT THE AUTHORS:

NICHOLAS CARRIERO is currently an associate research scientist in the Department of Computer Science at Yale University and a research scientist at Scientific Computing Associates in New Haven, Connecticut. His research interests include parallelism, compiler techniques, and programming languages. He is a co-author with David Gelernter of a forthcoming book on parallel programming. Author's Present Address: Yale University, Department of Computer Science, New Haven, CT 06520-2158. carriero@cs.yale.edu.

DAVID GELERNTER is an associate professor of computer science at Yale. His research interests include parallelism, programming languages, and heuristic systems. He is a co-author with Nicholas Carriero of a forthcoming book on parallel programming, and with Suresh Jagganathan on a forthcoming book on programming language design. Author's Present Address: Yale University, Department of Computer Science, New Haven, CT 06520-2158. gelernter@cs.yale.edu.