Technical Correspondence

Calendar P's and Queues

The article, "Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem," by Randy Brown (*Communications*, Oct. 1988, pp. 1220–27) presents a mechanism whose time-had-come-to-be-invented. Working independently, we have both come up with essentially the same mechanism. In addition, it appears that some features of the mechanism were discovered independently by a third party [1].

In April of 1988, IBM released its VM/XA SP1 operating system which uses my implementation of the mechanism to do its timer-request-queuing. In most operating systems, components of the operating system may request to be notified when a specified time is reached. The software must queue each request until the hardware clock reaches the specified time. I thought it would be appropriate to write and describe IBM's experiences with this new mechanism for doing timer-request-queuing. I also want to congratulate Randy Brown for his independent discovery of the idea and for his excellent explanation of it.

In the past, all versions of IBM's VM operating system implemented the timer-request-queue as a simple linear chain. But with larger numbers of users on the systems in recent years, the $O(N)$ growth in the overhead of searching this chain began to be a problem. The solution used by the developers of VM/SP-HPO (a predecessor of VM/XA SP) was to find ways of reducing the number of timer-requests per user, while still keeping the $O(N)$ mechanism. Obviously, that was only a temporary solution. While running VM/XA SF (another predecessor of VM/XA SP) in the lab with experimental workloads (large N), there were hints that the $O(N)$ timer-request-queuing was becoming a problem. Sometime in 1986, I devised and then implemented and debugged a timer-request-queue using a "heap" mechanism. But we did not bother testing the heap in the lab with interesting workloads (large N) because before finishing the heap I had figured out how the timer-request-queue could be better implemented using a hash table. (I had toyed with the hash-table idea earlier, but at first could not see how to make it work.)

Normally, a practitioner might be satisfied to stay with the heap at that point since the heap could be expected to produce a dramatic overall reduction in overhead for the large N to be tested in the lab. But, in this case, there was reason to go beyond the heap. Although the heap would give a vast reduction in enqueuing overhead, nevertheless, there would be a moderate increase in dequeuing overhead. This was a concern because, due to a peculiarity of the VM/XA operating system, the overhead on the dequeuing side is more critical than the overhead on the enqueuing side.

So, I discarded the heap and implemented the new hash-table mechanism. This implementation was built upon VM/XA SF code, but too late to ship with VM/XA SF. The modified VM/XA SF system was tested in the lab in May, 1987, on an IBM 3090–600 (a system having six processors), with a workload consisting of five thousand simulated users. (More users than the shipped version of VM/XA SF could handle.) As we expected, the new mechanism reduced the overhead associated with the timer-request-queue dramatically. Subsequently, the new mechanism was included in VM/XA SP1 and was shipped in April, 1988. A spin-lock is held during all manipulations of the timer-request-queue. Therefore, the overhead under consideration is not only the direct overhead from manipulating the queue, but also overhead generated by other processors spinning while one processor holds the lock to manipulate the queue.) Here is a comparison of Randy Brown's mechanism and IBM's VM/XA SP mechanism. The remarkable similarities give credence to the idea that "there is only one way to do it right."

Similarities

1. Each "day" of the queue is represented by a sorted-linked list of the request scheduled for that day.
2. A table (or array) containing one pointer for each "day" of the "year" is used to find the linked list for a particular day.
3. There is no overflow list. Requests scheduled for future years (beyond the end of the table) are wrapped back into this year's table. They are placed in the table according to the day they come due, disregarding the year. They are placed in the sorted-linked list with the year taken into account in the sorting.
4. The size of the table (length of the year) and the width of a "day" are both adjusted dynamically.
5. The table size is always a power of 2. The width of the "day" is also a power of 2.
6. We use a special "direct search" to find the next request to dequeue whenever the forward scan fails to find anything in an entire "year."
7. We maintain a "current day" cursor to mark where we are in the scan. This cursor consists of several data fields, including "index to the table" (day number) and "time at end of current day."
8. We have similar targets for efficient operation. Randy Brown recommends that at least seventy-five percent of the requests should fall within the coming year; the VM/XA target is to get eighty and a half percent (seven eighths) within the coming year.

Randy Brown recommends that three or four requests from the current year should be found in the current day; whereas, our target is five.

Differences/Extensions

1. Our operating system requires a "cancel request" function. To support this function, our chains of requests are doubly linked (forward and backward) so that the chain can be easily closed after a request is removed. Also, each request contains a pointer to the anchor-word of its chain. The anchor may require updating when a request is removed, and this allows the anchor to be located quickly.
2. When inserting a new request in a queue we can scan the chain either forward or backward. We are careful about making this choice, as there is a possibility that there may be a few very long queues. If they exist, we do not want to be scanning them in the wrong direction. Such queues can exist if there are some favorite times that people request. There would be such times as "now plus ten seconds," "infinity minus one second," etc. ("Infinity," here, means the largest time the clock can handle: all 1-bits in the time field.) If the new request is one of these, then it is equal to or greater than the last such request in the chain. So, it makes sense to scan the chain backward from the far end. This is also true in the general (normal short queue) case. We would expect that the new request will most likely be later than most of the requests already in the queue. There is one case, however, where we do a forward scan when inserting into a queue. We scan forward if the last request in the chain is within one hour of infinity (suggesting that this queue may be one of the long ones) and the new request is not within one hour of infinity.
3. We do not do "trial dequeuing" to sample the overhead. Instead, we track the overhead continuously by counting the nodes that we pass whenever we scan a chain and also when we scan forward past empty anchors in the table. (In assembler, this takes just one instruction in the loops to increment a register.) We keep the following statistics:

   (a) Count of (recent) enqueue operations.
   (b) Count of nodes skipped (recently) while scanning chains to find the insert point.
   (c) Count of (recent) dequeues of lowest-valued request.
   (d) Count of empty anchors skipped (recently) while scanning forward through the table (to find lowest-valued request).
   (e) Count of requests currently in the queue.

   This data is checked once every thirty seconds to see if the width-of-day or size-of-table should be changed.
4. Our goal is to have the width-of-day set so that, on the average, we are dequeuing (for expiration) five requests from each "day" of the table before advancing to the next day. If the statistics show that we are above one hundred and eighty percent of that or below fifty-five percent of it then we consider halving or doubling the width.
5. We also keep statistics on how often we make a change. Re-organizing the table causes quite a bit of overhead, so we do not want to oscillate back and forth every thirty seconds between the same two day-widths. The decision to make a change is in two parts: first, to decide if changing the day-width or table-size might reduce overhead; and second, to decide if making the change is worth the effort.
6. We allow the table to grow, but not to shrink. After the load on the system has peaked, the table size no longer changes. However, we do continue to adjust the width of the day up or down as necessary. As the load falls off after a peak, the width can increase to prevent the critical part of the table (the part just ahead of the cursor) from becoming too sparsely populated. It is true that if the distribution of times requested is very erratic (e.g., bimodal), then we might benefit from allowing the table to shrink (to reduce the time spent in occasional long forward scans), but so far we have not found this to be necessary.
7. Before we will make the table bigger, two conditions must be met: (a) the density (queued requests divided by table size) must justify a bigger table (this is the only condition that Randy Brown requires); and (b) the overhead of enqueuing a request must have exceeded a threshold. (The threshold is based, in a complex way, upon the targets of eighty-seven and a half percent and criterium 5 mentioned earlier.) Rationale: there seems to be no point in making the table bigger—even when it is very dense—if the density is not causing overhead. And we can hope that our backward scan, when inserting into the sorted-linked lists, will keep overhead low even when the table is densely populated. (Whether or not these considerations have actually resulted in our table being smaller than it otherwise might be is unknown.)
8. In our multi-tasking, multiprocessing environment, we can find that when a request is received, it is already due to expire. In other words, we may receive requests that precede the "current day" cursor. These expired requests cannot be readily accommodated by the general mechanism. Fortunately, expired requests are very rare, so we just maintain a separate queue (a sorted-linked list) for them. We check this queue first whenever looking for the earliest event to dequeue. Except for the insignificant activity at this special queue, our implementation should run in $O(1)$ time.

Gerald A. Davison
IBM Corp., 48TA/914
Neighborhood Road
Kingston, NY 12401

REFERENCES
1. Varghese, G., and Lauch, T. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. *Operating Syst. Rev. 21*, 5 (Nov. 1987), 25–38.

AUTHOR'S RESPONSE

I applaud Gerald Davison's letter, although I would perhaps dispute his main contention. I applaud his letter because he describes the solution to a major weakness of the calendar queue algorithm: the mechanism for choosing calendar length and length of day. The calendar queue, as I described it, readjusts these parameters only when the queue size changes by a factor of two, which may be too seldom. One can visualize them needing change even in a situation where only hold operations are being performed.

In addition, gathering efficiency statistics, as he suggests, to aid in choosing the new values seems to me to be worthwhile. The mechanism I settled on seems to work, but it is essentially a guess. I do not know how to prove that it would always work well. Basing the new day length and calendar size on the number of empty days recently encountered and the number of chain nodes recently examined per item enqueued or dequeued seems like a much more solid mechanism. Finally, comparing the expected gain from a readjustment to the cost of readjustment before making one seems sensible.

Before commenting on whether it was inevitable that this algorithm be discovered soon, let me describe briefly how I came to create it. I was scanning through back issues of *Communications* one day (recreational reading) when I came across Jones's [1] excellent article on priority queues. I did not even know what a priority queue was until then, but I saw immediately that a desk calendar solves the event-set problem in $O(1)$ time, and my intuition was outraged by the thought that a computer could not also be made to do it in $O(1)$ time. I felt sure that if I applied myself to the problem, I could do better than was currently being done. Since I was seeking publications in order to acquire tenure, I began work on the problem immediately.

When I began work on the problem in earnest, however, I discovered it was more difficult than I had anticipated (as is frequently the case). The problems of keeping the number of days in a calendar and the length of day appropriate, and of handling overflow all in $O(1)$ time soon presented themselves. I prayed over the problem, asking the Lord's help, and began mentally examining possible solutions. After several days work, I arrived at essentially the published algorithm.

In my case at least, the discovery of the algorithm did not come from an extensive knowledge of the literature. Nearly everything I knew about priority queues came from Jones's article. The driving force behind my work was a strong intuition and a good understanding of data structure techniques in general. If calendar queues are an invention whose time has come, then the necessary foundation is the general state of a computer science plus the information in Jones's article. It is not obvious to me that these were sufficient to guarantee discovery of the algorithm. I will note, however, that an article such as the one written by Douglas Jones is a powerful aid and incentive to invention. I would like to take this opportunity to thank him for his article.

Since the algorithms Gerald Davison and I discovered are so nearly identical, in spite of having considerable complexity, it may be that the requirements of the problem do determine the structure of the solution rather precisely. I would like to complement Gerald Davison on his excellent work.

Randy Brown
Dept. of Electrical Engineering
3179 Bell Engineering Center
University of Arkansas
Fayetteville, AK 72701

REFERENCES
1. Jones, D. W. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM* 29, 4 (Apr. 1986), 300–311.

DISK FRAGMENTATION AND CACHE MACHINES

It is difficult to analyze disk (and disk cache) performance so that the results are repeatable and useful in the real world. It is my experience that the layout of directory blocks and files on a hard drive is both (seemingly) random and very important; often, performance testing does not take this into account. For example, performance improvements which show up in a test program may be much lower than the improvements which occur in real life in a highly fragmented and almost full hard disk drive. In such cases, seek time can be greatly exaggerated over the case of essentially contiguous files present on empty, unfragmented drives. It is my informal observation that in the real world, seek times are more important than I/O starts and I/O processing times. Please note that none of the commercially available de-fragmentation programs help with this problem: the newly contiguous files may still be many tracks away from each other.

The Flash cache software mentioned by Jalics and McIntyre (*Communications*, Feb. 1989, pp. 246–255) has some important features which can help reduce seeking enormously. Flash has a "sticky" mode, during which blocks read into the cache stick there and are never unloaded. Whenever I boot my PC, I do a complete directory scan with set Flash in sticky mode. Thereafter, I never pay seek penalties when accessing the directory structures on my disk. This speeds file access not only at file open time, but when reading largefiles (where additional file layout information must be accessed).

Another Flash feature with which Jalics and McIntyre seem unimpressed is the ability to specify files to be loaded into the cache. I often pre-load my editor, compiler, and linker at boot time, which speeds up the load times for these frequently used utilities. While the following comment may be somewhat outside of the context of their article, one thing which they seem to have missed about Flash is that it is not strictly a cache: it offers other, related features which can increase system performance.

Jalics and McIntyre seem to feel that write caching is not important, but it can help in three ways. First, in cases where a directory block is being written repeatedly (as when a series of files are being copied into a directory), that block is actually only written once. This one effect can be almost magical when copying to a floppy. Second, while deferred writes cannot eliminate the processing required to do the actual write, the program does not suffer performance penalties during the time required to seek the block to be written, only for the actual write. Finally, even though total processing time may be no less with write caching, the user of an interactive program may appreciate getting control back immediately even if the actual physical writes have not yet been performed.

Jalics and McIntyre's comments about cache size are useful, but then I would never consider using a cache smaller than the programs I run (I use a 1.5MB cache). This brings up a related problem in the latest crop of 80386 computers: they often offer huge main memories (RAM) with trivial caches (64KB) which cannot be expanded. I know of one text editor which performs text insertions by block-moving text to make room for the insertion. This single (and frequent) operation can clear the entire cache, eliminating its usefulness in an instant. The lessons which Jalics and McIntyre offer need to be better appreciated in other related arenas.

They seem to feel that caching does not help compilations significantly, but I disagree. For one thing, I preload my compiler into cache, as described above, insuring rapid loading. Another issue is language-related. I use Modula-2, a language which encourages the generation of many software modules. When you reference a tool module from within an application module, the compiler must get the interface specifications for the tool module; each such specification resides in a different file. Thus, if one is re-compiling a given application module repeatedly, having the interface specification files in the cache speeds up compilations. The key issue here is that in some languages, more files must be referenced than just the source and object files.

Thank you for printing an article such as that written by Jalics and McIntyre. The *Communications* is often both dry and marginally applicable to the "real world." I encourage you to keep printing articles with relevance to those of us "in the trenches."

Jon Bondy
President, JLB Enterprises, Inc.
Box 148
Ardmore, PA, 19003

AUTHORS' RESPONSE

We agree with Mr. Bondy that disk fragmentation can be a very substantial problem. Our goal was to repeat experiments in as repeatable an environment as possible and concentrate on only one aspect: caching. We do agree that I/O starts under a primitive operating system like DOS have relatively small overhead when compared to full-scale operating systems with multiprogramming.

While we believe caching mechanisms should be transparent to the user, we do not for a moment denigrate the performance potential of sticky modes for files, or preloading certain files into a cache.

We did not say and do not mean that write caching is unimportant. It certainly produces impressive results in the hardware cache and, as Mr. Bondy indicates, in some software cache applications as well. However, what we did say is that write caching with software caches is not as important as we had thought.

Mr. Bondy's comments regarding hardware caches on 80386 computers are well taken.

Regarding compilations, we do not say that there is no potential there. What we do say is that we tried it on one high-speed Cobol compiler and that the savings were unimpressive.

Paul J. Jalics
David R. McIntyre
James J. Nance College of Business Administration
Dept. of Computer and Information Science
Cleveland State University
2121 Euclid Avenue
Cleveland, OH 44115

LINDA IN CONTEXT

In the recent *Communications* [1] Carriero and Gelernter compare Linda to "the Big Three" families of parallel languages: concurrent object-oriented languages, concurrent logic languages, and functional languages. While I accept many of their statements concerning concurrent object-oriented languages and functional languages, I find the comparison with concurrent logic languages misguided.

Linda = Prolog − Logic + Concurrency

Carriero and Gelernter like to think of Linda as very different from all the three models they explore. I do not think this is so and would like to suggest that Linda incorporates fundamental aspects of the logic programming model, although perhaps in a degenerate form.

Specifically, observe that the content of a Tuple Space, i.e., a multiset of tuples, can be represented directly by a multiset of unit clauses (also called facts), and that the basic tuple space operations of Linda, *out* and *in*, are variants of Prolog's database manipulation operations *assert* and *retract* over a database of facts; *in* is a degenerate form of *retract* in the sense that it uses a home-brewed form of matching, rather than full-fledged unification. The *rd* operation on a Tuple Space similarly corresponds to querying a database of facts; it also uses matching rather than unification.

Assert and *retract* save concurrency and are the major extra-logical operations in Prolog. That is essentially all that Linda is offering; hence, the title of this section.

The remaining difference between Linda and the database manipulation component of (sequential) Prolog is concurrency: Linda allows concurrent Tuple Space operations, whereas, by definition, sequential Prolog is sequential. Concurrent execution in Linda means, first, that the *in* and *out* operations are atomic, and second, that *in* suspends if a matching tuple is not found. This is in contrast to *retract* in sequential Prolog, which fails in such a case.

Database-oriented versions of Prolog, such as LDL [3], are enhanced with the concept of transactions to support concurrency control of database update. However, to my knowledge no one has attempted to use *assert* and *retract* as a basic communication and synchronization mechanism, as is done in Linda.

Research in concurrent logic programming has taken a different approach. Concurrent logic languages do not incorporate extra-logical primitives like *assert* and *retract*. Instead, they specify concurrency and communication by pure logic programs, augmented with synchronization constructs.

The Dining Philosophers

Carriero and Gelernter did not have to set up a strawman for the purpose of comparing Linda with concurrent logic programming—Ringwood, in *Communications*, (January 1988) has done this job for them [4]. With its seventy lines of (incomplete) code, six procedures, and four illustrative figures, Ringwood's Parlog86 program for solving the dining philosophers problem is indeed an easy mark. However, this program is not an appropriate basis for comparison with concurrent logic languages. The reason is that a dining philosopher can be specified by a much simpler concurrent logic program:

```
phil(Id, [eating(LeftId, done) | Left], Right)
    ← phil(Id, Left, Right).
        % Left is eating, wait until he is done.
phil(Id, Left, [eating(RightId, done) | Right])
    ← phil(Id, Left, Right).
        % Right is eating, wait until he is done.
phil(Id, [eating(Id, Done) | Left] ↑,
    [eating(Id, Done) | Right] ↑) ←
  . . . eat, when done unify Done = done,
  then think, then become:
  phil(Id, Left, Right).
        % Atomically grab both forks
```

The program is independent of the number of philosophers dining. A dinner of n philosophers can be specified by the goal:

```
phil(1, Fork1, Fork2), phil(2, Fork2, Fork3),
                              . . . , phil(n, Forkn, Fork1)
```

whose execution results in each of the *Fork* variables being incrementally instantiated to a stream of terms *eating(Id, done)*, with the *Id*'s on each *Fork* reflecting the order in which its two adjacent philosophers use it. For example, a partial run of this program, augmented with "eating" and "thinking" components that take some random amount of time, on a dinner of 5 philosophers, provided the substitution:

```
Fork1 = [eating(1, done), eating(5, done),
                        eating(1, done), eating(5, _) | _]
Fork2 = [eating(1, done), eating(2, done),
                        eating(1, done), eating(2, _) | _]
Fork3 = [eating(3, done), eating(2, done),
                        eating(3, done), eating(2, _) | _]
Fork4 = [eating(3, done), eating(4, done), eating(4, done),
                    eating(3, done), eating(4, done) | _]
Fork5 = [eating(4, done), eating(4, done), eating(5, done),
                        eating(4, done), eating(5, _) | _]
```

The run was suspended midstream in a state in which *Fork4* is free and the second and fifth philosophers are eating using the other four forks. Up to that point, each of the philosophers ate twice, except the fourth, which ate three times. (This output was obtained by running this program under the Logix system [10], a concurrent logic programming environment developed at the Weizmann Institute.)

The program is written in the language FCP(↑) (a variant of Saraswat's FCP(↓, |) [5]). The unannotated head arguments are matched against the corresponding terms in the goal atom, as in Parlog86 and GHC [14]. In contrast, the ↑-annotated terms are unified with the corresponding terms, as in Prolog. A goal atom can be reduced with a clause if both the input matching and the unification succeed. The reduction of a goal with a clause is an atomic action: it either suspends or fails without leaving a trace of the attempt, or it succeeds with some unifying substitution, which is applied atomically to all the variables it affects.

The key to the simplicity of the program is indeed the ability of FCP(↑) to specify atomic test unification: the philosopher atomically tries to grab both forks, excluding other philosophers from grabbing them. The mutual exclusion is obtained by unifying the head of the *Fork* stream with a term containing the unique *Id* of the philosopher. (Atomic test unification is incorporated in all the languages in the FCP family, but not in Parlog86 or GHC. Programs exploiting atomic test unification cannot be written easily in the latter two languages.)

The deadlock-freedom of the program is guaranteed by the language semantics; specifically, if several processes compete to unify the same variable(s), one succeeds and the others will see the variable(s) instantiated (deadlock-free atomic unification can be implemented using a variant of the two-phase locking protocol; see [11, 12]). Like the Parlog86 and the Linda programs, this program is not starvation free. It should be noted, however, that the Linda and Parlog86 solutions can be made starvation free with the addition of standard fairness assumptions. This is not true for our solution, since two philosophers sharing a common neighbor can conspire against this neighbor so that his grabbing the two forks is never enabled, leading to his starvation. If this is to be avoided, the FCP(↑) program can be enhanced with the "tickets" method so that it achieves starvation freedom under similar fairness assumptions. The resulting program is very similar to the Linda one and still much simpler than the original Parlog86 one.

Assigning unique identifiers to philosophers can be avoided if the language is strengthened with read-only variables [6] or with a test-and-set primitive. We consider the second extension. In FCP(↑, !), an !-annotated term in the clause head denotes an output assignment of the term to a variable in the corresponding position in the goal. The assignment fails if the corresponding position is a non-variable term. It can be mixed (but not nested) with ↑-annotations, and all assignments and unifications should be done atomically. In FCP(↑, !), an anonymous dining philosopher can be specified by:

```
phil([eating(done) | Left], Right) ←
    phil(Left, Right).
      % Left is eating, wait until he is done.
phil(Left, [eating(done) | Right]) ←
    phil(Left, Right).
      % Right is eating, wait until he is done.
phil[eating(Done) | Left]!, [eating(Done) | Right]!) ←
    . . . eat, when done unify Done = done,
    then think, then become:
    phil(Left, Right).
      % Atomically grab both forks
```

A dinner of *n* anonymous philosophers:

```
phil(Fork1, Fork2), phil(Fork2, Fork3), . . . ,
                                    phil(Forkn, Fork1).
```

is of course not as interesting. Each *Fork* variable is bound to a stream of *eating(done)*.

Embedding Linda

Sure the picture of Linda versus concurrent logic languages looks different after rehabilitating the dining philosophers. But is this the whole story?

Both in Linda and in concurrent logic programs, processes communicate by creating a shared data structure. A stream of messages between two concurrent logic processes is just a shared data structure constructed incrementally from binary tuples. However, a shared structure in Linda cannot contain the equivalent of logical variables. The ability to construct and communicate structures with logical variables (incomplete terms, incomplete messages) is the source of the most powerful concurrent logic programming techniques. The proposed extension to Linda of nested Tuple Spaces is yet another approximation-from-below of the power of the logical variable, as will be demonstrated shortly.

Another difference between Linda and concurrent logic languages is that Linda incorporates no protection mechanism: any process can access any tuple in the Tuple Space by reinventing the appropriate keywords. In contrast, shared logical variables cannot be reinvented and are unforgeable; and, therefore, they provide a natural capabilities and protection mechanism [2]: concurrent logic processes can only access structures bound to variables they were given explicit access to.

Another difference is that in a concurrent logic language a shared structure can only be increased monotonically (by further instantiating variables in it) and can only be reclaimed by garbage collection when unreferenced. In Linda, tuples can be explicitly deleted from the shared Tuple Space, but the lack of discipline in Tuple-Space access makes garbage collection impossible.

The similarity between the cooperative incremental construction of logical terms and Linda Tuple Space operations is perhaps best illustrated by showing a concurrent logic program implementation of Linda's primitives. For simplicity, the concurrent logic program represents the Tuple Space as an incomplete list of terms of the form *tuple(OutId, InId, T)*. The operations are defined, of course, so that several processes can perform them concurrently on the same Tuple Space. Each *in* and *out* operation must be given a unique identifier for the purpose of mutual exclusion. An *out* operation with tuple *T* and identifier *OutId* inserts the term *tuple(OutId, _, T)* at the tail of the Tuple Space list.

```
% out(Id, T, Ts) ← Tuple T inserted to Tuple Space Ts
% with an out operation with identifier Id.
out(Id, T, [tuple(Id, _, T) | Ts] ↑).
        % Found tail of Ts, insert T.
out(Id, T, [_ | Ts]) ← out(Id, T, Ts),
        % Slot taken, keep looking.
```

An *in* operation with tuple template *T* and identifier *InId* finds a term *tuple(_, InId′, T′)* in the Tuple Space such that *T′* unifies with *T* and *InId′* unifies with *InId*.

```
% in(Id, T, Ts) ← Tuple T in Tuple Space Ts deleted by
% an in operation with identifier Id.
in(Id, T, [tuple(_, Id ↑, T ↑) | Ts]).
        % Found unifiable tuple, delete it.
in(Id, T, [tuple(_, Id′, T′) | Ts]) ←
   (Id, T), ≠ (Id′, T′) | in(Id, T, Ts).
        % Tuple deleted or doesn't unify, keep looking.
```

The precise interrelation between *in* and *rd* is not specified in the Linda article. Assuming it is defined so that a tuple deleted from the tuple space by *in* should not be visible to any subsequent *rd*, the corresponding concurrent logic program is:

```
rd(T, [tuple(_, InId, T ↑) | Ts]) ←
   var(InId) | true.
        % Found tuple that has not been deleted yet.
rd(T, [tuple(_, InId, T′) | Ts]) ←
   (foo, T) ≠ (InId, T′) | rd(T, Ts).
        % Tuple deleted or doesn't unify, keep looking.
```

The program assumes that *foo* is not used as an identifier. It uses the *var* guard construct to check that *InId* is still a variable, i.e., the tuple has not been deleted by a concurrent *in* process. If weaker synchronization is allowed between *in* and *rd*, then the corresponding logic program may use weaker synchronization as well.

These programs can be executed "as is" under the Logix system. A Logix run of the following goal, consisting of several concurrent operations on an initially empty Tuple Space *Ts*:

```
out(1, hi(there), Ts),   in(2, hi(X), Ts),   rd(hi(Y), Ts)
          in(3, hi(Z), Ts),   out(4, hi(ho), Ts)
```

results in the following substitution:

```
X = there
Z = ho
Ts = [tuple(1, 2, hi(there)), tuple(4, 3, hi(ho)) | _]
```

indicating that *hi(there)* was inserted to *Ts* with operation 1, deleted with 2 and *hi(ho)* was inserted with 4 and deleted with 3. In this particular run, *in*(3, *hi*(Z), *Ts*) reached the second tuple before *rd(hi*(Y), Z), so the *rd* process remains suspended.

If we execute an additional *out* operation:

```
out(5, hi(bye), Ts)
```

then both *out* and the suspended *rd* terminate, which results in the substitution:

```
Y = bye
Ts = [tuple(1, 2, hi(there)), tuple(4, 3, hi(ho)),
     tuple(5, _, hi(bye)) | _]
```

Note that the last tuple inserted has not been deleted yet, so its *InId* is still uninstantiated.

This implementation uses genuine unification, rather than the home-brewed matching mechanism of Linda. Hence, tuple templates and tuples in the Tuple Space may contain logical variables. Hence, this implementation can realize nested Tuple Spaces as is[1]. For example, the following goal sets up in *Ts* a Tuple Space consisting of two nested Tuple Spaces, called *ts1* and *ts2*:

```
out(1, tuple_space(ts1, Ts1), Ts

out(2, tuple_space(ts2, Ts2), Ts)
```

Tuple Space *ts1* can be retrieved into the variable *SubTs* using the goal:

```
rd(tuple_space(ts1, SubTs), Ts)
```

Once retrieved (detecting this requires *rd* to report successful completion; this is an easy extension), it can be operated upon:

```
out(1, hi(there), SubTs),  in(2, hi(X), SubTs),
          out(3, hi(ho), SubTs)
```

The result of these operations combined is:

```
Ts = [tuple(1, _, tuple_space(ts1, [tuple(1, 2, hi(there)),
        tuple(3, _, hi(ho)) | Ts1'])), tuple(2, _, tuple_
        space(ts2, Ts2)) | _]
Ts1 = SubTs = [tuple(1, 2, hi(there)), tuple(3, _,
        hi(ho)) | Ts1']
  X = there
```

Note that operations on a nested Tuple Space are independent of operations on the parent Tuple Space and, hence, may use the same set of identifiers.

Using test-and-set, *in* and *out* can be implemented even more simply in FCP(↑, !), without using unique identifiers. A Tuple Space is represented by an incomplete list of terms *tuple(T, Gone)*, where *Gone* is instantiated to *gone* when *T* is deleted.

```
out(T, [tuple(T, Gone) | Ts]!).
      % Found tail of Ts, insert T
out(T, [_ | Ts]) ←
    out(Id, T, Ts).
      % Slot taken keep looking

in(T, [tuple(T ↑, gone!) | Ts]).
      % Found unifiable tuple, delete it
in(T, [tuple(T', Gone) | Ts]) ←
    (T, foo) ≠ (T', Gone) | in(T, Ts).
      % Tuple deleted or doesn't unify keep looking
```

Supporting *eval* in a language that already supports dynamic process creation seems redundant, so we do not address it.

The concurrent logic programs for *out*, *in*, and *rd* are concise, executable specifications of the semantics of these operations. But they are inefficient since they use a naive data structure, i.e., a list, to realize a Tuple Space. Their efficiency can be increased by applying the same ideas to a more sophisticated data structure such as search tree or a hash table, as done in real implementations of Linda.

Embeddings of concurrent object-oriented languages and of functional languages in concurrent logic languages have been reported elsewhere. The trivial embeddings of Linda in FCP(↑) and in FCP(↑, !) are further evidence to the generality and versatility of concurrent logic programming.

Embedding as a Method of Language Comparison

Language comparison is a difficult task, with no agreed upon methods. There are many "soft" methods for comparing languages, and we have seen them being used in arguments in the sequential world ad nausia. No reason why we should make better progress with such arguments in the concurrent world. However, I know of one objective method for comparing and understanding the relations between languages and computational models: embedding ("compiling") one language in another and investigating the complexity of the embedding (it has been extensively used for comparing concurrent logic languages [cf. 6]).

In a response to an earlier draft of this letter, Carriero and Gelernter imply that embedding one language in

[1] We note that while multiple Tuple Spaces can be easily realized by multiple Prolog-like databases, nested Tuple Spaces cannot be easily mapped to Prolog-like *assert* and *retract* operations.

another is always such an easy and trivial task, that showing its possibility for two particular languages is hardly interesting. This is far from being true. Try embedding Linda in Occam, in a parallel functional language, or in an actor language, and see the mess you get. Conversely, try embedding these models (and some concurrent logic language) in Linda. I doubt if the results of this exercise will help in promoting Linda, but they will certainly increase our understanding of the relationship between Linda and the rest of the concurrent world.

The DNA Example

Another example in Carriero and Gelernter's article is the DNA sequence analyzer program. They show a functional Crystal program and a Linda program for solving it and argue (1) that the functional program is only slightly preferable on readability and elegance grounds, and (2) that the Linda program is better in giving explicit control of parallelism to the programmer and in not relying on a sophisticated compiler to decide on its parallel execution. I show (below) a concurrent logic program solving this problem that is shorter, clearer, and specifies a more efficient algorithm than the other two.

The program specifies a "soft-systolic" algorithm along the lines of the verbal description of the problem by Carriero and Gelernter. (See [8] for the concept of systolic programs and [13] for an analysis of their complexity.) An array process network is spawned, one *cell* process for each pair (i, j), with near-neighbors connections. Each *cell* process receives from the left the tuple (Hl, Pl) containing the h and p values from its left neighbor, and from the bottom the tuple (Hbl, Hb, Qb) containing the bottom h and q values, as well as the diagonal (bottom left) h value. It computes the local values of the functions p, q, and h and sends them in corresponding tuples to its top and right neighbors. The top neighbor is also sent Hl, the h value from the left.

The *matrix* and *row* procedures realize the standard concurrent logic programming method of spawning a process array network [8]. The matrix process is called with two vectors Xv and Yv and a variable Hm. It recursively spawns rows, with adjacent rows connected by a list of shared variables (the Ts and Bs of each row are its top and bottom connections). Each *row* process recursively spawns *cell* processes. A cell process has elements X and Y, one from each vector, and it shares with its left, right, top, and bottom *cell* processes in its *Left*, *Right*, *Top*, and *Bottom* variables, respectively. During the spawning phase, the matrix and row processes construct the skeleton of the output of the computation, namely, the similarity matrix Hm. They provide each element, H, of the matrix to its corresponding *cell* process, which instantiates H to its final value.

Although this schema for constructing recursive process networks may require some meditation the first time encountered, programs employing it can be constructed almost mechanically and understood with little effort once the schema is grasped.

```
% h(Xv, Yv, Hm) ← Hm is the similarity matrix of
vectors Xv and Yv.

h(Xv, Yv, Hm) ←
   matrix(Xv, Yv, Hm, Bs),
   initialize(Bs).

matrix([X | Xv], Yv, [Hv | Hm] ↑, Bs) ←
   row(X, Yv, Hv, (0, 0), Bs, Ts),
   matrix(Xv, Yv, Hm, Ts).
matrix([ ], Yv, [ ] ↑, Bs).

row(X, [Y | Yv], [H | Hv] ↑, Left, [Bottom | Bs] ↑,
   [Top | Ts] ↑) ←
   cell(X, Y, H, Left, Right, Bottom, Top),
   row(X, Yv, Hv, Right, Bs, Ts).
row(X, [ ], [ ] ↑, Left, [ ] ↑, [ ] ↑).

initialize([(0, 0, 0) ↑ | Bs]) ←
   initialize(Bs).
initialize([ ]).

cell(X, Y, H, (Hl, Pl), (H, P) ↑, (Hbl, Hb, Qb),
   (Hl, H, Q) ↑) ←
     compare(X, Y, D),
     P := max(Hl − 1.333, Pl − 0.333),
     Q := max(Hb − 1.333, Qb − 0.333),
     H := max(Hbl + D, max(P, max(Q, 0))).

compare(X, X, 1 ↑).
compare(X, Y, −0.3333 ↑) ← X ≠ Y | true.
```

For example, a run of the program on the initial goal $h([a, b, a, a, b, c, d], [a, b, c, d, e], Hm)$ produces the answer:

```
Hm = [[1, 0, 0, 0, 0],
      [0, 2, 0.667000, 0.334000, 0.001000],
      [1, 0.667000, 1.666700, 0.333700, 0.000700],
      [1, 0.666700, 0.333700, 1.333400, 0.000400],
      [0, 2, 0.667000, 0.334000, 1.000100],
      [0, 0.667000, 3, 1.667000, 1.334000],
      [0, 0.334000, 1.667000, 4, 2.667000]]
```

which shows that a match of length 4 is obtained at positions (7, 4).

The program is more efficient than the Crystal and Linda ones in two respects. First, it avoids recomputation of the auxiliary functions p and q. Although in principle, the Crystal program can be transformed into a more intricate program that avoids the recomputation, a compiler capable of performing such a transformation has yet to be demonstrated. Similarly, the Linda program can be transformed (manually, I presume) to store both the P and Q matrices in the Tuple Space, in addition to H, thus avoiding their recomputation. It seems that this transformation will adversely affect program length and readability. Second, the concurrent logic program specifies a regular process structure with local communication only. This structure can be mapped—manually, using Turtle programs [8], or automatically—to a concurrent multicomputer in a way that near processes reside in near processors (or even in the same processor). Efficiently mapping the other two programs, which are not specified by a process network and have no obvious notion of locality, is more difficult.

Conclusions

I have shown that the semantic gap between concurrent logic programming and the set of constructs offered by Linda can be closed with just six clauses. This, I believe, is not because Linda was consciously designed with the concurrent logic programming model in mind (or vice versa). Rather, finding that an ad hoc construct for concurrency is easily embedded in the concurrent logic programming model is a recurrent phenomenon. This was demonstrated in various contexts for streams, futures, actors, monitors, remote procedure calls, guarded synchronous communication, atomic transactions, and other constructs of concurrency [6, 7]. I believe that this is a consequence of, and the best kind of evidence for, the generality and versatility of concurrent logic programming.

Concurrent logic languages are not just abstract computational models, general and versatile as they may. They are real programming languages. This may be surprising given their fine-grained concurrency. For example, in the Logix session required to compile, test, debug, and produce the sample runs of the programs shown in this note, 283,711 processes were created and thirteen minutes of Sun/3 CPU were consumed. (Since the Logix compiler and programming environment are written entirely in FCP, most of these processes were created during system activities such as compilation, rather than during the actual runs of the programs.) See [11] for a report on the parallel performance of FCP. A fair estimate is that more than 50,000 lines of FCP code were developed using the Logix system, including the Logix system itself and numerous applications [7]; and Logix is just one of several practical concurrent logic programming systems. However, presently concurrent logic languages offer a gain in expressiveness at the cost of a loss in performance, compared to conventional approaches to concurrency. The time when this loss in performance is small enough to be easily compensated by the gain in expressiveness has yet to come.

Acknowledgments. I thank Nicholas Carriero, David Gelernter, Ken Kahn, Amir Pnueli, and Bill Silverman for their comments on an earlier manuscript.

Ehud Shapiro
Dept. of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

REFERENCES
1. Carriero, N., and Gelernter, D. Linda in context. *Commun. ACM 32*, 4 (Apr. 1989), 444–458.
2. Miller, M.S., Bobrow, D.G., Tribble, E.D., and Levy, J. Logical secrets. In *Proceedings 4th International Conference*. MIT Press, 1987, pp. 704–728. Also Chapter 24 in [7].
3. Naqvi, S., and Tsur, S. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.
4. Ringwood, G.A. PARLOG86 and the dining logicians. *Commun. ACM 31*, 1 (Jan. 1988), 10–25.
5. Saraswat, V.A. Partial correctness semantics for CP[↓, |, &]. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science* (New Delhi, India, Dec. 16–18). Indian Inst. Tech./Tata Inst. Fund. Res., 1985, pp. 347–368.
6. Shapiro, E. The family of concurrent logic programming languages. *Comput. Surveys*, 1989. To be published.
7. Shapiro, E. *Concurrent Prolog: Collected Papers*, Vols. I and II, MIT Press, 1987.
8. Shapiro, E. Systolic programming: A paradigm of parallel processing. In *Proceedings of the International Conference on Fifth Generation Computer Systems*. 1984, pp. 458–471. Also Chapter 7 in [7].
9. Shapiro, E. A subset of Concurrent Prolog and its interpreter. TR-003, *ICOT Tech. Rep.* Inst. for New Generation Comput. Tech., Tokyo, 1983. Also chapter 2 in [7].
10. Silverman, W., Hirsch, M., Houri, A., and Shapiro, E. The Logix system user manual. Version 1.21. Chapter 21 in [7].
11. Taylor, S. *Parallel Logic Programming Techniques*. Prentice-Hall, 1989.
12. Taylor, S., Safra, S., and Shapiro E. A parallel implementation of Flat Concurrent Prolog. *J. Parallel Programming 15*, 3 (1987), 245–275. Also Chapter 39 in [7].
13. Taylor, S., Hellerstein, L., Safra, S., and Shapiro, E. Notes on the complexity of systolic programs. *J. Parallel and Distributed Comput. 4*, 3 (1987). Also Chapter 8 in [7].
14. Ueda, K., Guarded Horn clauses. In *Logic Programming*, E. Wada, Ed. Springer-Verlag, 1986, pp. 168–179. Also Chapter 4 in [7].

I was not convinced that Linda [3] has any advantages over traditionally message-oriented systems or monitors on today's parallel systems. To convince the market that a product is superior it must provide either superior run-time performance or a far simplified programming model for not much less performance. In addition, it is becoming necessary to provide these capabilities over a large range of hardware platforms.

Many of the benefits attributed to the tuple space by Carriero and Gelernter exist in most buffered message passing systems:

> 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 language 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 [3, p. 445].

The occam/CSP model of communication encompasses all these features within the language, without resorting to a computation and communication-intensive superstructure [5, 6, 10]. Since occam is built around the concept of safe, reliable communication, all process-to-process, program-to-program, program-to-operating system, and program-to-file systems are handled through the built-in communication primitives. As for different types of languages communicating, the only way they are allowed to share data is through message passing. Since each language has its own requirements for data representation in stack frames, the easiest way to coordinate different languages is through message passing.

The basic messaging system defined in occam does not allow for buffering since all the message passing is handled at the hardware level. This restriction does require blocking-send communications with a result of well synchronized code. If the algorithm does not need to be synchronized, buffer processes may be introduced to allow non-blocking message sends.

Messaging environments such as Express, Trollius, or Helios provide the equivalent of the buffering and routing capabilities of Linda, with much lower overhead [2, 8, 9]. Express even offers parallel task invocation, in the form of an interrupt routine, on receipt of a message of a given type [7].

I am stressing the run-time overhead of Linda since performance is the key issue in production programs on parallel processing systems. Although the authors have dismissed monitors because the normal method of implementation requires a single address space and a synchronous procedure call to get access to the data protected by the monitor, they have effectively implemented a monitor in Linda. Linda only moves these problems to a different location but still implements a monitor to handle the tuple space.

On a distributed memory environment, such as is needed for a Transputer-based or hypercube system, each processor must have a monitor task and a message routing task operating in parallel to the user's tasks. The following description is from Cornell's implementation of Linda, called Brenda, and is based heavily on Carriero and Gelernter's work at Yale [1].

> Brenda operates by having a server process running on each processor that may have objects stored in its address space. The library functions send messages to these servers and receive replies. In other words, a Brenda call is a classic Remote Procedure Call (with some twists). The storage processor is determined by a randomizing (hashing) function of the tag. When an object is to be grabbed, the request is sent to the server on only one processor, as determined by the same hashing function. In a system with many parallel processors, the execution of one Brenda function typically involves the transmission of several messages over a path that passes through several processors. Some tricks are utilized to reduce the number of messages passed whenever possible . . . The speed of execution of Brenda functions, as compared with the speed of sequential computation, is a crucial parameter. It determines what algorithms should be chosen for a given task, or whether Brenda is suitable for the task at all. The execution of one Brenda function call is typically an order of magnitude slower than passing one message to the nearest neighbor. It takes on the order of 10 milliseconds in the current implementation.

In addition to the communication delay, the processor which is handling the tuple space is performing a non-trivial amount of work to search, update, garbage collection, and, otherwise, maintain the tuple space. This work must significantly reduce the overall performance of the system. If it did not, why build custom hardware to handle the tuple-space functions?

One aspect of the tuple space that was not discussed in great detail in the article by Carriero and Gelernter [3] was the reading of values from the tuple space. Whereas the insertion of values into the tuple space is non-blocking, retrieval of the data is and can become a bottleneck that has no equal in a well designed algorithm on a message-based system. With distributed memory, to read a tuple from the tuple space, a message must be sent to the appropriate processor; the tuple space must be searched for a match; the required tuple must then be sent to the requesting process which can then continue. In a message-based system, the algorithm can be constructed so that the required data is in a buffer in the processor when needed. The task need only input from the buffer, without the overhead of the request/response message and the database search. The difference in performance of the approaches may be best seen when some data must be shared globally, such as in a simulation.

In Linda, one process would insert a value into the tuple space, and all other processes would then read the value. With message passing, a broadcast would be used. Examining the message traffic alone on a 64-processor hypercube, we find that the broadcast requires only sixty-three messages and only six message transmission times. With Linda, each process would have to send a request and wait for a response. Assuming a simple implementation, this would generate, on average, three message hops to and from the processor with the data for each of the requesting processors. Each processor would have to wait six message hop times and a total of more than three-hundred-sixty messages would be generated. Each communication port of the destination processor would have to handle, on average, ten incoming and ten outgoing messages. All sixty-four requests would have to queue for access to the tuple space. The queuing may be reduced through a look-ahead mechanism, but the time-consuming message passing is not easily avoidable. Since local memory access times can be measured on the scale of microseconds and message access is two to three orders of magnitude larger for most systems [9], the message traffic time will dominate the system performance.

It must be remembered that the applicability of a parallel system to a wide range of problems is heavily dependent on the efficiency in which the processors can coordinate and share data. In distributed memory systems, this is the efficiency of the message system. Applications are already limited by the message system in many multi-processors; an order of magnitude slower message system would make the problem worse.

The authors' claim that C-Linda results in code which is more readable than Parlog86 or Crystal is true, but that it is different from message-based systems is not true. The `out` command of Linda can easily be replaced by a message send and the in by a message read. Compare the following occam code with the C-Linda example for the Dining Philosophers Problem. Note that in the occam program [11] the equivalent of the tuple space manager requires only twenty-four lines of code. The remaining program requires fewer lines of code in a lower-level language than C. Also, note that no call to a mysterious initialize () routine of unknown length is required.

```
PROC philosopher (CHAN OF INT left,
                  right, down, up)
  WHILE TRUE
    SEQ
      ... think
      down ! 0       — get your room ticket
      PAR
        left ! 0     — get the left
                       chopstick
        right ! 0    — and the right at
                       the same time
      ... eat
      PAR
        left ! 0     — put down the
                       chopsticks
        right ! 0
      up ! 0         — and leave the room
:

— The following 2 routines implement the
'tuple space' PROC chopstick (CHAN OF INT
left, right)
  WHILE TRUE
    INT any:
    ALT
      left  ? any       —phil to left
                         gets it
        left ? any
      right ? any       —phil to right
                         gets it
        right ? any
:

PROC security ([] CHAN OF INT down, up)
  VAL INT max IS 4:
  INT n.sat.down:
  SEQ
    n.sat.down := 0
    WHILE TRUE
      INT any:
      ALT i = 0 FOR 5
        ALT
          (n.sat.down < max) &
             down[i] ? any
            n.sat.down :=
      n.sat.down + 1
   up[i] ? any
     n.sat.down :=
      n.sat.down - 1
:
```

```
— the main part
PROC secure.phil ()
  [5] CHAN OF INT left, right, up, down:
  PAR
    security (up, down)
    PAR i = 0 FOR 5
      PAR
        philosopher(left[i], right[i],
    down[i], up[i])
    chopstick(left[i], right[(i + 1)\5])
:
```

The stated benefit of not having to know about receivers of messages and vice versa in Linda is potentially dangerous. Much of the required synchronization in message-based systems is to ensure complete and accurate execution of the program at all times. By requiring the programmer to think about the destination of the data and the overall data flow, algorithms can be better optimized, and loose tuples will not be left in the machine at program termination. When designing the individual modules, the physical source and destination of the data need not be defined, only the message type. At system integration, the task of allocating message flow is no different from assigning names to the tuples in the tuple space, and the routing can be accomplished through a standard messaging interface. The overall difference in the approach is that of designing a system instead of hacking it together.

Carriero and Gelernter also did not define what "good speedup" is when they ran Linda on a 64-node iPSC hypercube. How does their speedup compare with that obtained by systems such as CrOS III from CalTech or its commercial equivalent, Express? Fox et al. also claim high efficiency for a large number of problems and state the values as being between eighty and ninety-five percent [9].

Finally, what Gelernter and Carriero have not shown is that Linda offers either a run-time performance advantage over a message-based system, or that the algorithm design, coding, and implementation is significantly easier. What they have shown is that Linda does run on a number of hosts, but, to get adequate performance, custom (read expensive) hardware is needed. This is a problem in a commercial environment with customers who are very price sensitive. The currently available messaging systems offer higher performance, more portability, and surprisingly similar programming models to Linda, and, as such, they are currently more commercially viable.

Craig Davidson
Mechanical Intelligence
922 Grange Hall Rd.
Cardiff, CA 92007

REFERENCES
1. Braner, M. *Brenda—A Tool for Parallel Programming.* Cornell Theory Center, Ithaca, N.Y., 1989.
2. Braner, M. *Trollius User's Reference Manual.* Cornell Theory Center, Ithaca, N.Y., 1988.

3. Carriero, N., and Gelernter, D. Linda in context. *Commun ACM 32*, 4 (Apr. 1989), 444–458.
4. Fox, G., et al., *Solving Problems on Concurrent Processors*, Vol. I. Prentice-Hall, Englewood Cliffs, N.J. 1988.
5. Inmos Ltd. *occam® 2 Reference Manual.* Prentice-Hall, Cambridge, Mass., 1988.
6. Inmos Ltd. *Transputer Development System.* Prentice-Hall, Cambridge, Mass., 1988.
7. ParaSoft Corp. *Multitasking: Running Multiple Processes under Express.* Mission Viejo, Calif., 1988.
8. ParaSoft Corp. *Express Documentation.* Mission Viejo, Calif., 1988.
9. Perihelion Software. *Helios Technical Manual.* Shepton Mallet, United Kingdom, 1988.
10. 3L Ltd. *Transputer Pascal Documentation.* 3L Ltd., Edinburgh, United Kingdom, 1987.
11. Welch, P.H. Occam and transputer engineering class notes. Univ. of Kent, Canterbury, 1989.

I read Carrierro and Gelernter's article, "Linda in Context," in the April 1989 issue [2] with interest. As one of the first pragmatic, machine-independent, parallel programming languages, Linda is worthy of much praise. However, as the theme of the article is a demonstration of the superiority of Linda and the tuple-space approach over other extant approaches to parallel programming, I find it necessary to point out a few errors of omission.

One approach that was omitted in their discussion is logic programming. This statement may appear strangely superficial, as the authors devote a substantial portion of the paper to CLP (Concurrent Logic Programming). However, CLP is not at all the same as parallel execution of logic programs. The CLP approach involves logic-based, explicitly parallel languages which (a) allow specification of processes with tail-recursive predicates and (b) use streams represented as partially instantiated "Logical Variables" for specifying communication between processes. The logic-programming approach, in contrast, is an implicitly parallel one, much like the Functional Programming approach considered in their article. Logic-programming formulations are useful, among other domains, in symbolic-computation applications involving search, backtracking, and problem reduction. There are over a dozen research groups, not counting the CLP-related groups, working on efficient parallel-execution schemes for logic programs.The authors may wish to use the same arguments against logic programming as they used against functional programming, but this significant effort certainly deserves consideration.

Functional and logic languages may not be appropriate for expressing certain types of computations. However, the authors contend that even for domains where computations can be naturally expressed in such languages, they are not appropriate *parallel* programming languages because they rely on an "auto pilot"—the compiler and the run-time system that decide what to execute in parallel, when, and where. Such auto pilots, they argue, currently do not and probably can never yield as good a performance as an explicitly parallel program (written in Linda, say). However, this argument overlooks an important possibility: allowing the programmers to annotate the logic or functional programs. The types of annotations must be chosen carefully. In our experience, grainsize control and data dependences are two attributes that can be specified fairly easily by programmers. For example, based on such annotations, we have recently obtained excellent speedups on different multiprocessors, with uniprocessor speeds comparable with sequential compiled Prolog programs, in our parallel logic-programming system based on the REDUCE OR Process Model [6, 8]. Such annotations are qualitatively different than Linda-fying Prolog (to borrow the authors' phrase) because they are highly non-invasive. The annotated logic program still reads/feels very much like a logic program, and the programmer does not have to deal directly with management of parallel execution. The *futures* of MultiLisp [5] and *Qlets* of QLisp [4] are a few of the other examples of systems that use annotations effectively.

My third point concerns the authors' arguments against a rather broad collection of approaches they have categorized as "concurrent objects." It is difficult to rule out such a diverse set of approaches with arguments against a few members. The points regarding object orientation being orthogonal to parallel programming and monitors being of limited use for parallel processing are well taken. But, they do not constitute an argument against all the approaches in the vast spectrum of formulations with synchronous or asynchronous message-passing processes, threads, concurrent objects, Actors, etc. For example, consider a language which allows dynamic creation of many medium-grained processes, includes primitives for asynchronous communication between them, and includes primitives for dealing with specific types of shared data. The Chare Kernel parallel programming system we are developing [7] is an example of such a system. The shared data may be as simple as *read-only* variables or as general as dynamic tables. (Dynamic tables consist of entries with a key and data and allow asynchronous insertions, deletions, and lookups, with optional suspension on failure, to find an entry. These are similar, to some extent, to tuple spaces). None of the arguments in this section argues against such a system or point out Linda's advantage over such an approach. For example, the arguments against actors are mainly that (a) they do not support shared data structures and (b) do not allow messages to be sent to not-yet-created tasks. A system such as the Chare Kernel surmounts both these criticisms. Because it explicitly supports dynamic shared data-structures, it is possible to deposit data in such structures and to have another new process pick that data later on. Of course, the authors may not have known about this particular system. But they seem to be arguing that all of the possible approaches in this broad spectrum are "irrelevant for parallelism."

In fact, the Chare Kernel-like approaches may have some advantages over Linda on the basis of the authors' own criteria. The communication of data is much more specific in Chare Kernel. In particular, when a process A needs to communicate data to process B, it specifically directs the message to B; when many processes share a dynamic table, they may read and write to this shared and distributed data structure. In contrast, in Linda, when a tuple is *outed*, it is not immediately clear what sort of sharing and communication is to take place. The programmers usually know what the sharing is to be, but they are not allowed to express their knowledge in Linda, leaving the burden to the compiler. In a few cases that the programmers cannot specify the type of communication, it is reasonable to use the most general mechanisms. The compiler—their own auto pilot—cannot be expected to be perfect regarding this (and even if it were, it would be unnatural for programmers to "hide" this information from the system). Yet, knowledge of communication patterns is essential for efficient implementation on message passing machines. Although there is a large body of Linda applications reported in literature, the subset for which the performance data on iPSC/2 hypercube is available is fairly small and consists of simpler programs. This may simply be indicative of the (transient) state of development of that system. However, I expect that when they attempt to port all the other Linda programs to the hypercube and/or write more complex Linda programs with more complex communication patterns to run on hypercube, the difficulties mentioned above will surface. It is not clear whether the Linda approach can be made to be viable in that situation.

The work of Athas and Seitz [1] on Cantor and Dally's work [3] on Concurrent Smalltalk on the "Jelly-Bean machine" represents other interesting extensions to the actors and object-oriented models that are viable in a large fine-grained multi-computer (and they are both constructing such machines).

To summarize, (a) logic programming is an approach to parallel programming on par with functional programming and is distinct from concurrent logic programming; (b) the objections raised against efficiency of parallel execution schemes for functional (or logic) languages can be surmounted with simple annotations. So, such schemes present fairly effective approaches in application domains where they naturally apply; and (c) appropriate extensions of message passing or Actor-like systems do not suffer from the disadvantages attributed by the authors to specific systems within this broad class. In fact, the uniformity of tuple-space operations may be detrimental because it passes on the burden of identifying the type of communication to the compiler, while forcing the programmer to "hide" that information from the system.

Linda does make a significant contribution. The tuple space approach is a unique and interesting way of programming which, in addition, can lead to good speeds and speedups over the best sequential programs, on multiprocessors available today. Only, the "context" of other competing approaches to parallel programming is much richer than the authors argue.

L.V. Kalé
Dept. of Computer Science
University of Illinois at Urbana Champaign
1304 W. Springfield Ave.
Urbana, IL 61801

REFERENCES
1. Athas, W.C., and Seitz, C. Multicomputers: Message-Passing Concurrent Computers. *Computer*, (Aug. 1988), 9–24.
2. Carriero, N., and Gelernter, D. Linda in context. *Commun. ACM 32*, 4 (Apr. 1989), 444–458.
3. Dally, W.J. *A VLSI Architecture for Concurrent Data Structures*. 1987.
4. Gabriel, R.P., and McCarthy, J. Queue-based multi-processing Lisp. In *ACM Symposium on Lisp and Functional Programming*, August 1984.
5. Halstead, R. Parallel Symbolic computing. *Computer 19*, 8 (Aug. 1986).
6. Kale, L.V. Parallel execution of Logic Programs: The REDUCE_OR process model. In *Proceedings of the 4th International Conference on Logic Programming* (Parkville, Victoria, Australia, May 25–29). Univ. of Melbourne, 1987, pp. 616–632.
7. Kale, L.V., and Shu, W. The Chare Kernel language for parallel programming: A perspective. UIUCDCS-R-88-1451, Dept. of Computer Science, Univ. of Illinois, 1988.
8. Ramkumar, B., and Kale, L.V. Compiled execution of the Reduce-Or process model on multiprocessors. UIUCDCS-R-89-1513, Dept. of Computer Science, Univ. of Illinois, 1989.

The article, "Linda in Context," by Carriero and Gelernter in April's *Communications*, pp. 440–458, presents an interesting model for parallel computing. Much of the article is a comparison with other approaches. Much of this comparison looks like attacks on concurrent logic programming (along with actors, concurrent object-oriented programming, and functional programming). The following paragraph from the conclusions is milder than most of the body of the article.

> Logic programming is obviously a powerful and attractive computing model as well. Notwithstanding, we continue to believe that the tools in concurrent logic languages [sic] 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.

This article has generated a flurry of electronic discussions in which many participants argue for their own particular hobby horse and are deaf to what the others are saying. In this note, we attempt to step back from these religious platforms to extract the good ideas from the different approaches as well as to understand their relationships.

Different Problems

Linda and concurrent logic programming seem to be solving two different problems. Linda is best not thought of as a language—but rather as an extension that can be added to nearly any language to enable process creation, communication, and synchronization. (For very clean languages such as functional programming languages, adding Linda constructs may destroy their clean semantics.) Linda is a model of how to coordinate parallel processes. As a solution to this problem, Linda seems pretty good. We do not see how logic variables and unification can be added to Fortran and C as well as tuple spaces have been. The utility of Linda rests on two observations about the world we live in.

(1) There is a lot of programming language inertia. There are lots of programmers who do not want to (or cannot) leave the languages they presently program in. Organizations prefer to minimize the retraining of their programmers.
(2) More and more applications are multi-lingual. More and more programmers deal with multiple languages as they work.

Concurrent logic programming has traditionally been addressing another problem—namely, how can one design a single language which is expressive, simple, clean, and efficient for general purpose parallel computing? It does not mix concurrent and sequential programming but, instead, takes the radical stance that all one needs (and should ever want) is a good way to describe concurrent computations. The challenges are to design and understand such a language, to provide useful formal models of the language, to implement it well, and to discover important programming techniques. The great hope is that this combination of being both semantically clean and real will enable radically new tools (e.g., partial evaluators, verifiers, algorithmic debuggers, program animators, etc.).

The practicality of such tools may rest upon the simplicity of the language. In principle, such tools could be built for C-Linda, Fortran-Linda, etc., but I expect they would be too complex to do in practice. The tools the Linda folk have developed (such as the Tuple space visualizer) only deal with the *part* of the computation which occurs in Linda. This is rather incomplete. What is needed are tools that can deal with computation and communication together as an integrated whole.

Uniformity

Both camps are striving for uniformity but of different sorts. The CLP camp strives for uniformity within a language while the Linda camp strives for uniformity across languages. There is just one way to communicate in CLP, not as in Linda where there is the dialect specific way of communicating in the small (e.g., procedure calls) and the tuple space way of communicating between processes.

There has been research in embedding object-oriented, functional, and search-oriented logic programming languages into CLP by implementing them in a CLP language. This approach might be generalizable to deal with programming *paradigm* interoperability. At least programmers would be able to program in the style they wish even though existing programming languages would not be supported. Research in how these embedded languages might be inter-callable remains to be done.

Someone who frequently programs in different conventional languages (C, Fortran, Lisp, etc.) may value most highly the uniformity that Linda offers. As they move from language to language they can deal with process creation, communication, and synchronization in a uniform manner. From this point of view, Linda should be compared with distributed operating systems like Mach which also provide similar capabilities as system calls from different languages. On the other hand, someone looking for a simple, yet real, concurrent programming language may be more appreciative of the uniformity in concurrent logic programming.

Distributed Computing and Issues of Scale

The Linda model is, in a sense, very high level. The model provides a decoupling between clients and servers by providing a shared global tuple space. It is left up to clever compilers to avoid central bottlenecks and to compile much of the code as point-to-point communication and specialized data structures. The claim is that it is easier to think and program in this decoupled manner.

Linda cannot scale to large-scale distributed computing (open systems) without making yet another shift in computational model. It would seem that Linda's practicality rests upon global compiler optimizations. As systems get larger and more distributed, it seems implausible that one module which adds a tuple of a certain form could be compiled knowing all the other modules that read tuples of that form. A solution to this is to provide multiple tuple spaces and control their access. Indeed, at the end of the article, Carriero and Gelernter mention that they are building a version of Linda with multiple first-class tuple spaces. We discuss this further below.

It is common practice to compare programming languages on the kinds of aesthetic grounds that underly most of the electronic discussion sparked by the Linda article. Shapiro (in a network conversation) proposes that the ease and complexity with which one can embed one language within another is important. There are many subtle issues here. First, we do not see how a *naive* implementation of Linda in a CLP language demonstrates much. If the point is about how expressive CLP is *and how these are real languages for implementing systems in a serious way*, then one needs to present a serious implementation. C-Linda, for example, relies on some sophisticated compiler optimizations based upon a global analysis of the restricted ways in which classes of tuples are used. A good embedding should be capable of such optimizations.

There are several other objective means of comparing languages besides embeddings which we explored in the paper "Language Design and Open Systems" [1]. These other criteria do not correspond to "expressiveness" as normally defined. Rather, they relate to issues that appear in open systems (large distributed systems with no global trust, information, or control).

Some of these issues are quite important with respect to a form of expressiveness which gets surprisingly little attention: modularity. To quote from "Logical Secrets" [3]:

> Trust is a form of dependency between entities; by reducing dependencies the modularity of the system is increased . . . Thus the degree of cooperation possible among mutually mistrusted agents is one measure for how viable a language is in supporting [a certain kind of modularity].

In [1], we enumerate the following "hard" criteria:

(1) Be able to define servers that are immune from the misbehavior of others,
(2) Be able to survive hardware failure,
(3) Be able to serialize simultaneous events,
(4) Be able to securely encapsulate entities with state,
(5) Be able to dynamically create and connect services,
(6) And finally, it must not rely on global constructs.

These criteria are hard in the sense that languages which are Turing equivalent can formally differ in their abilities to provide security and to react to external events. A language which has no means of implementing robust servers, for example, cannot simulate a language which can. In addition, we give the following "soft" criteria (soft in the sense that these properties are not strictly necessary at the distributed language level since higher levels of language can provide them):

- Can reprogram running programs,
- Can use foreign services not implemented in the distributed language,
- Can define services which can be used by foreign computations,
- Is based upon computational principles which scale,
- Supports the expression of very highly concurrent programs,
- Supports small scale programming well.

In [1] we explain why we believe actors and concurrent logic programming languages meet these criteria well (except (2)), and why functional programming, Or-parallel Prolog, and CSP/Occam do not. With respect to assessing Linda for open-systems programming, there are really two Lindas: Linda in the small and medium which relies on a single shared tuple space; and Linda in the large which relies on multiple first class tuple spaces. Let us call the first S-Linda and the second M-Linda (for Single versus Multiple tuple spaces).

S-Linda does not satisfy properties (1) or (6). The reason is that any other process can "in" any of the tuples that a given server is trying to serve and, thereby, disrupt communication between that server and its client. This is clearly unacceptable for a secure distributed system but also makes reasoning about the correctness of programs a less local and modular affair. One cannot show that a given subset of a Linda system will proceed successfully without knowing at least something about the correctness of all computations participating in that Linda system.

M-Linda seems to satisfy all the criteria as well as Actors and CLP. In M-Linda, used for open systems, there would be at least one tuple space private to each trust boundary. Therefore, with regard to open system properties, we can regard all computation proceeding inside a single tuple space as a single agent. Presumably, the way such multiple agents interact is by creating new tuple spaces, passing tuple spaces around, and having processes that communicate to more than one tuple space.

The computational model of M-Linda among these macro agents is not very different from Actors (a tuple space is like a mailbox) or CLP (a tuple space is like a logic variable[1]). M-Linda seems to have many of the "flaws" of which the Linda researchers accuse message-passing systems and CLP languages.

If the Linda people are effectively arguing that S-Linda is superior to Actors and CLP for parallel programming in the small and medium, while M-Linda gives you the good properties of Actors and CLP in the large (at the "costs" of Actors and CLP) that would indeed be an interesting and significant argument. It would, however, concede our point about uniformity.

Historical Footnote
Linda very closely resembles Bill Kornfeld's Ether language from the late 1970s [17]. The major difference being that Ether had constructs equivalent to Linda's "rd," "out," etc. but not "in."

Acknowledgments. Much of this note is derived from a network discussion which included Ehud Shapiro, Nicholas Carriero, Graem Ringwood, Ian Foster, and ourselves. We are also grateful to Jacob Levy, Vijay Saraswat, and Danny Bobrow for their comments.

Kenneth M. Kahn
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Mark S. Miller
Xanadu Operating Company
550 California Avenue
Palo Alto, CA 94306

REFERENCES
1. Kahn, K., and Miller, M.S. Language Design and Open Systems. In *The Ecology of Computation.* North Holland, 1988.
2. Kornfeld, W. Ether—A parallel problem solving system. In *IJCAI-79.* August, 1979.
3. Miller, M., Bobrow, D., Tribble, E. and Levy, J. Logical secrets. In *Concurrent Prolog: Collected Papers*, Vol. II. The MIT Press, 1987, pp. 140–161.

AUTHORS' RESPONSE
Of the comments published here, we believe Kahn and Miller's to be by far the most interesting and significant. We have responded to Kahn and Miller—our in-

[1] We think a promising direction to explore is a CLP-like language without logic variables which, instead, uses tuple spaces for communication.

tent was not so much to refute their comments as to amplify them and to explore their implications—in a brief separate paper called "Coordination Languages and their Significance" [2]. We summarize the paper's main points after addressing other comments.

We are glad that Professor Shapiro agrees with us on the topics of concurrent object-oriented and functional languages, and we appreciate the candor of his admission that "The time when this loss in performance [sustained by concurrent logic languages as against systems like Linda] is small enough to be easily compensated by the gain in expressiveness has yet to come." But, in our view, his comments add nothing significant to our comparison between Linda and concurrent logic languages. We will consider his main points in (what we take to be) descending order of significance.

Shapiro's embedding experiment is designed (presumably) to shed light on this question: *which interprocess communication operators are more expressive*? Linda communication is based on tuple spaces; in the concurrent logic languages Shapiro discusses, communication is based on streams constructed using partially-instantiated logical variables. We have explained in detail why we believe that a shared associative object memory (a tuple space) is a more flexible and expressive medium than a CLP-style (Concurrent Logic Programming) stream. Streams are merely one of a family of important distributed data structures; we need to build vectors and matrices, bags, graphs, associative pools, and (also) a variety of streams. We have argued that tuple spaces make a far simpler and more expressive basis for building these structures than do streams based on logical variables. Roughly comparable positions may explain why (for example) Ciancarini's Shared Prolog eliminates stream-based communication in favor of a blackboard model related to Linda [3], and why Kahn and Miller remark that "We think a promising direction to explore is a CLP-like language without logical variables which, instead, uses tuple spaces for-communication." (These researchers will speak for themselves, of course,—we merely present our own reaction to their work.)

Now, what does Shapiro bring to counter our claims? His argument seems to be, in essence: "contrary to what you have said, CLP communication is more expressive than Linda; if you start with CLP, you have all the resources of idiomatic CLP streams at your command, *and if you want Linda, you can also have Linda*—observe how easily we can implement Linda given CLP." But this contention as stated is transparently false. As Shapiro himself realizes, his CLP implementation of Linda is far too inefficient to be usable. (As he writes, "they [his embeddings] are inefficient since they use a naive data structure, i.e., a list, to realize a Tuple Space.") It follows, that if a programmer working in the CLP framework indeed needs the expressivity of Linda, Shapiro's embedding demonstration is irrelevant. He needs a real implementation of Linda. A *real*—that is, an acceptably efficient—implementation of Linda would (obviously) be vastly more complex than the CLP implementation Shapiro presents. For one thing, "C-Linda," as Kahn and Miller point out in their note, "relies upon some sophisticated compiler optimizations based upon a global analysis of the restricted ways in which classes of tuples are used. A good embedding should be capable of such optimization."

This point is so obvious that one might suspect Shapiro of having something else (a deeper and subtler point?) in mind. We might imagine that his Linda embedding is merely a kind of "expressivity experiment." It is not designed to realize the functionality of Linda per se, but merely to demonstrate how expressive CLP is by presenting an elegant solution to an interesting problem. But the objection, again, hinges on the fact that Shapiro's embedding represents a bad way to implement Linda, and claims regarding expressivity are presumably to be based on *good* solutions to interesting problems. The mere existence of *some* solution is uninteresting; these languages are all Turing-equivalent in any case.

One final, significant aspect of the embedding experiment: FCP is a complete programming language; Linda is a *coordination language*, dealing with creation and coordination of (black-box) computations exclusively. Shapiro has attempted to embed the *coordination aspects* of C-Linda within (the complete language) FCP; a comparable experiment would attempt to embed the *coordination aspects* of FCP within C-Linda. As we demonstrated in our article, it is trivial to express CLP-style streams in C-Linda. But *this* "embedding experiment" yields code that is efficient enough to be used. (Such structures are, in fact, used routinely in Linda programming.)

Shapiro's claims with respect to embedding remind us, in short, of what might be put forward by an enthusiastic exponent of Lego. It is true that you can build all sorts of elegant working models using Lego. But it is also true that this piece of good news has not proved sufficient, in itself, to establish Lego as a mainstay of the construction industry. Furthermore, Lego structures always wind up looking like assemblages of little boxes. That is fine if you are dealing with boxy structures. Otherwise, you might want to conduct your building experiments using a more flexible medium.

Turning to the dining philosophers example with which Shapiro begins—if we were members of the CLP community, we would find this demonstration deeply alarming. As we noted in our article, dining philosophers is neither an especially important nor an especially difficult problem. The Linda solution is trivial. In order to get a comparably concise solution in CLP, Shapiro needs to jettison Flat Concurrent Prolog (his own invention) and introduce us to the super-duper new logic language FCP(↑), which is a variant of Saraswat's FCP(↓), which is in turn a variant of FCP. Shapiro seems unaware of the fact that when a simple problem is posed, *introducing a new language in order to solve it* can hardly strengthen our confidence in this *new* language's ability to handle the unanticipated problems that *it* will be expected to tackle. This same new language becomes the basis of the CLP Linda embedding. But as we have already noted, the Linda embedding is not (so far as we are concerned) an impressive demonstration of anything, and so we are left wondering what exactly motivates the new language and whether it is the definitive CLP solution or merely one of a long series. (We are happy to concede, for what it is worth, that FCP(↑)'s dining philosophers solution is far more concise than Parlog86's.)

Finally, Shapiro's remarks on the DNA example are irrelevant. We gave Linda code for one algorithm. Shapiro gives the CLP implementation of a different algorithm. Shapiro himself notes (and surely it is obvious) that we could write a Linda program to implement this other algorithm as well. It is trivially clear that in order to compare the expressivity of two languages, we must control for the problem being solved.

We claimed, in our article, that Linda was, in general, more elegant, more expressive, and more efficient than CLP. Nothing in Shapiro's note challenges this claim. We continue to believe that a Linda-based parallel Prolog is potentially a more elegant, expressive, and efficient alternative to the CLP languages Shapiro discusses. Several groups appear to be contemplating such systems, and we look forward to their results.

Other Comments

We have no fundamental argument with Professor Kale's note. Obviously, we reject his claim that "the uniformity of tuple-space operations may be detrimental;" the approach Kale has taken in his Chare Kernel is far too complicated for our tastes. As he intimates in his note, he has provided a system in which there is a vaguely tuple-space-like communication construct and a large number of other devices as well. Although none of the others is strictly necessary (all could be implemented using the tuple-space construct), the Chare authors believe that in providing them, they have anticipated a significant set of special cases. This is the design approach that has informed a series of important programming languages from PL/1 through Ada and Common Lisp. Unfortunately, making simple things complicated is not a cost-free exercise. You burden the programmer with a more complex model to master, the program-reader with more complicated sources, and the implementer with a harder optimization problem. We choose to program in languages that provide optimized support for a few simple tools and then let us build what we need. But this is an aesthetic question; Kale is obviously entitled to his views, and we wish him luck in developing Chare. We applaud the absence of ideological fustian in his note.

Davidson's note is full of errors and misrepresentations. The claim that Linda "implements a monitor to handle the tuple space" is false. Inferences about Linda's performance or implementation drawn from Braner's work on Brenda are puzzling [cf. 1, letter by Davidson on Linda; this issue]. True, Moshe Braner has done interesting work on this Linda variant, but his system executes in a different environment from our hypercube system, shares no code with our system, is roughly an order of magnitude slower than our system [1]—and does not implement Linda in any case. In claiming that "messaging environments such as Express, Trollius, or Helios provide the equivalent of the buffering and routing capabilities of Linda," Davidson appears not to understand that the *point* of implementing a tuple space is not buffering and routing, it is a shared, associative memory. Express et al. provide no such thing. The claim "to get adequate performance custom (read expensive) hardware is needed" is false. Phrasing his comments in terms of what we must do to "convince the market" is silly, because (a) our point in this response was to address the computer science community, not the market, and (b) "the market" is rapidly becoming convinced of Linda's significance of its own accord. Well over half a dozen companies are involved in Linda work at this point.

Kahn and Miller compare two fundamentally different approaches to parallelism in the following terms: "The CLP [Concurrent Logic Programming] camp strives for uniformity within a language while the Linda camp strives for uniformity across languages." Agreed; it is our view that this distinction in approach is a deep and important one for the evolution of programming systems.

We can build a complete programming environment out of two separate, orthogonal pieces—the *computation model* and the *coordination model*. The computation model supplies tools for performing single-threaded, step-at-a-time computations—any ordinary programming language will serve the purpose. The coordination model is a glue that can bind many separate, active computations into a single composite program. A coordination language is the embodiment of a particular coordination model; a coordination language allows us to create and to manage information exchange among a collection of independent, asynchronous activities which may be separated either by space or by time. Our point in introducing the term transcends nomenclature. The issue is rather that Linda and other systems in its class are not mere extensions to base-computing languages; rather, *they are complete languages in their own right*, albeit coordination rather than computing languages; *they are independent of and orthogonal to*, not mere extensions of, *the base-computing languages with which they are joined.*

Approaching coordination as one of two orthogonal base axes that jointly define the space of programming environments clarifies the significance of systems like Linda and conduces to conceptual economy, allowing us to relate a series of superficially-separate programming problems ranging from file systems and databases to distributed and parallel applications.

We refer to the orthogonal-axis approach as the C + C scheme (computation model plus coordination model). This approach has a series of what we regard as significant advantages. It supports *portability of concept*—the machinery required for explicit parallelism is always the same, no matter what the base language (to get parallelism, we must be able to create and coordinate simultaneous execution threads). The C + C approach allows us to transform any base-computing language into a parallel language in essentially the same way—thus C.Linda, Fortran.Linda, Scheme.Linda, and so on. The C + C scheme is *granularity-neutral* so far as parallel applications go—it allows the programmer to choose an appropriate granularity for his application without having granularity decided or limited by a programming model. It supports *multi-lingual applications*—the computations that are glued together by our coordination model may be expressed in many different languages. The *isolation of orthogonal elements* it fosters tends to make natural the incorporation into programming environments of advances along either axis. Finally, the C + C model is an expressive framework for building the interesting class of applications that incorporate features of parallel, distributed, and information management systems; and the view of coordination as a "first class phenomenon" promoted by the C + C scheme simplifies our understanding of the computing environment and suggests new theoretical frameworks for modelling it more comprehensively. (These claims are explained and defended in [2].)

Kahn and Miller, in effect, oppose the C + C model; instead, they advocate the use of integrated parallel languages in which a uniform model applies to computation and also to coordination. We have given our reasons for favoring a different approach, but this disagreement cannot be resolved on objective grounds. We emphasize, that despite this disagreement, we view our own work as strongly allied to Kahn and Miller's; their work has significantly advanced our own understanding of coordination and its importance.

Nicholas Carriero
David Gelernter
Yale University
Dept. of Computer Science
New Haven, CT 06520-2158
carriero @ cs.yale.edu
gelernter @ cs.yale.edu

REFERENCES

1. Bjornson, R., et al. The implementation and performance of hypercube Linda. RR-690, Dept. Computer Science, Yale University, March 1989.
2. Carriero, N., and Gelernter, D. Coordination languages and their significance. Res. Rep. Dept. Computer Science, Yale University, June 1989.
3. Ciancarini, P. Blackboard programming in Shared Prolog. In *Proceedings of the 2d Workshop on Program Languages and Compilers for Parallelism* (Aug. 1989). To be published.

Circle #110 on Reader Service Card