Applications Experience with Linda

Nicholas Carriero and David Gelernter
Department of Computer Science
Yale University

Abstract

We describe three experiments using C-Linda to write parallel codes. The first involves assessing the similarity of DNA sequences. The results demonstrate Linda's flexibility—Linda solutions are presented that work well at two quite different levels of granularity. The second uses a prime finder to illustrate a class of algorithms that do not (easily) submit to automatic parallelizers, but can be parallelized in straightforward fashion using C-Linda. The final experiment describes the process lattice model, an "inherently" parallel application that is naturally conceived as multiple interacting processes. Taken together, the experience described here bolsters our claim that Linda can bridge the gap between the growing collection of parallel hardware and users eager to exploit parallelism.

This work is supported by the NSF under grants DCR-8601920 and DCR-8657615 and by the ONR under grant N00014-86-K-0310. We are grateful to Argonne National Labs for providing access to a Sequent Symmetry.

1 Introduction

There is a gap in the current arsenal of parallel programming systems: parallel machines but not high-level parallel languages are widely available. A "parallel language" for our purposes is a system that supports parallelism with constructs recognized by a compiler (not merely with a library of system calls); it is "widely available" if it has been implemented on machines produced by many manufacturers, and preferably on several different classes of parallel machines. "High-level" is subjective, but a high-level language should support many programming paradigms conveniently (but with reasonable efficiency). Instead, currently-available software falls mainly into three classes. Academic research has concentrated on parallelizing compilers. The manufacturers are mainly interested in proprietary (hence machine-specific and not widely available) languages or system-call libraries. Sophisticated users have in some cases developed portable parallel languages, but in most cases they are intentionally restricted in the sorts of parallelism they can handle; they tend to center on parallel DO-loops.

A good deal has been accomplished in all of these efforts, but in our view they don't add up to a satisfying whole. Restructuring compilers are fine, except for the cases in which they (*a*) don't work well enough,

(*b*) don't work at all or (*c*) are irrelevant, because the algorithm in hand is explicitly parallel. (We give examples of (*b*) and (*c*) further on.) Systems in the third category (e.g. Jordan's Force[Jor86]) are clearly useful within the domains and on the machines for which they are appropriate, but it's easy to find programming tasks that they can't handle (because they simply weren't designed to). We see little redeeming social value in the second category. The world-view in which each manufacturer promotes its private stable of locally-optimum languages, forcing users to translate their programs (in some cases even redesign their algorithms) whenever they switch machines, has been in decline since around 1960.

We have argued for some time that Linda is a good candidate for filling the gap. Linda is a high-level (by our standards) parallel language for MIMD machines and LANs; it exists in two version, a C and a Fortran dialect; it is available quite widely: we have built systems for the Encore, Sequent and Alliant shared-memory machines, the Intel iPSC hypercube, VAX/VMS LAN, Bell Labs S/Net and other machines. Several manufacturers are working independently on implementations for their own equipment; these efforts mainly involve parallel workstations. Linda is a research project, and of the systems named, only the C-based dialect on the Encore and Sequent machines has been distributed to "alien" sites (sites not involved in collaborative research with our group). But our growing (if still preliminary) experience with the system strengthens our conviction that Linda works.

This paper is devoted specifically to programming experiments. The system and its implementations have been described at length elsewhere [e.g. GB82, Gel85, CGL86, CG85, ACG86, Car87] and a recent paper specifically addresses the question of portability [BjCGL88]. Many sorts of programming experiments have been conducted using Linda. The three to be described in this paper are chosen to make specific points. The first involves a problem of significance to geneticists, DNA-sequence comparison. When new sequences are discovered, it is of interest to determine which previously-known sequences they resemble, where "resemblance" is a qualitative measure that can be approximated using string-matching-like algorithms. We discuss this experiment because it demonstrates the need for *flexibility* in a parallel language: an individual sequence-to-sequence comparison can be parallelized, or we can run many sequential comparisons simultaneously, giving us a parallel search of a large database. Each approach can be useful, and a parallel language should work in both cases.

The second and third experiments address (among other points) the issue of parallel languages versus restructuring compilers. The second involves a problem posted to the COMP.ARCH bulletin board on the ARPANET: an algorithm is presented which, it is argued, is characteristic of a class that is immune to parallelizing or vectorizing compilers. We discuss an explicitly-parallel Linda version that shows excellent performance. The third experimental system was conceived from the first as a parallel structure: an heuristic monitor for use in post-operative cardiac ICU's was designed as a lattice of increasingly-general decision processes. Here Linda is important because it is expressive: the prototype would have been far more complicated as a sequential program with a monolithic flow of control; if and when the monitor (or a similar program in another domain) grows large enough to require more computing re-

sources than one processor can offer, parallelism is on hand to provide speed as well as expressivity.

The work of systems researchers is significant only insofar as it's useful to non-systems-researchers. "Usefulness" can't be established quantitatively, and at any rate experience with Linda is still fairly preliminary. It's nonetheless our intent in these examples to argue that Linda is in fact a useful tool right now within a variety of significant domains. This argument will rest on three contentions.

1. *Linda is being used to solve "real problems".* Our first and third examples were suggested by biologists and anesthesiologists respectively. In other papers we've discussed Linda programs for matrix multiplication, the factorization step of the Dongarra Linpack benchmark [Don87] and travelling salesman [BjCGL88]; other current work involves a Linda ray-tracing program for the display of fractal images (written by Ken Musgrave working with Benoit Mandelbrot at Yale), and joint work by Robert Whiteside at Sandia-Livermore and Jerry Leichter of our group which has recently demonstrated supercomputer performance[1] using Linda on a LAN of VAX 8000-class machines[WL88].

2. *The Linda solutions to these problems are easy to understand.* We outline the workings of our example problems. The point is subjective, but readers will judge for themselves.

3. *The Linda solutions demonstrate* real *speedup.* Parallel-language performance is sometimes (see e.g. [Tha88]) described in terms of relative speedup: how much faster is the n-node version than *the same program in the same language* running on a single node? This question is interesting *only* if we know the running time of a comparable algorithm written in *a conventional, sequential language* as well. Recoding a sequential-language program in a parallel language always introduces overhead. Given an efficient system, we recoup the overhead by running on many processors, and we come out ahead in absolute terms. Given an inefficient language, we never recoup the overhead no matter how many processors we run on. We present "absolute speedup" data by comparing the performance of C-Linda programs to comparable sequential programs in C.

The data to be presented center for the most part on the C-Linda system for the Encore Multimax. These programs would port trivially to Linda systems on other parallel computers. Their performance would be (in our experience) very similar on other shared-memory machines (for example the Sequent Balance or Symmetry or the VU Tadpole), allowing for differences in absolute processor speed. When we port Linda code to *disjoint*-memory machines, the performance of non-communication-intensive programs is relatively unaffected. (We give some figures comparing Linda on the Encore and on the Intel iPSC hypercube for one example.) But of course Linda can't alter the intrinsic speed of the interconnect, thus communication-intensive programs don't do as well on hypercubes as they do on shared-memory machines. The next generation of disjoint-memory machines will show a major reduction in network delays, and Linda programs should then perform more consis-

[1] In the case of a parameter sensitivity experiment, twice the performance of a Cray 1S.

tently across architectural classes. (We will be running tests in coming months on two next-generation disjoint-memory machines, Intel's iPSC/2 and the Linda Machine[ACGK88] being built by Venkatesh Krishnaswamy and Sid Ahuja at Bell Labs). Much more data on portability and performance across architectures appears in [BjCGL88].

Because Linda has been discussed at length in the literature, we relegate a brief description to the appendix.

2 DNA Sequencing

DNA-sequencing is typical of a problem that requires flexibility in a parallel language. Parallelism can enter at either of two levels. A single sequence-to-sequence comparison can be time-consuming, and might accordingly be parallelized; when comparing against a large database of sequences, however, it may well be more efficient to parallelize not a single comparison but the entire database search—to perform many sequential comparisons simultaneously.

A parallel sequence comparison potentially involves fairly fine-grained parallelism, of the sort that is readily expressed in (for example) data-flow languages extended with I-structures [NPA86] or systolic systems and so forth. The parallel database search is quite different. An efficient way to conduct this search is to create the optimal number of search-processes given available hardware; then to pass out sequences to searchers dynamically (comparisons take varying amounts of time), and have the searchers return the results to a master process that compiles them, remembering the best result so far. Here, fine-grained techniques are irrelevant; a different kind of technique is required. Linda can handle both cases efficiently.

Figure 1 shows the performance of a Linda program running on the Encore Multimax and Sequent Symmetry, shared-memory parallel machines with NS-32332 and Intel 80386 processors respectively. This program represents a parallelized version of a single sequence-to-sequence comparison. The problem to be solved is an assessment of the degree of similarity between two DNA sequences. The assessment must be sensitive both to insertions, deletions and mutations.[2] The general method can be described as follows. Let s and t be two sequences. Construct a "similarity" matrix H, where $H[i,j]$ is the maximum similarity of all subsequences of s ending at index i as compared against all subsequences of t ending at j. $H[i,j]$ is computed by taking the maximum of three terms:

1. $H[i-1, j-1]$ plus a weight that depends on $s[i]$ and $t[j]$. Intuitively this corresponds to lengthening the match.

2. Consider all $H[i', j]$ for $i' < i$. These values modified by a suitable penalty function give similarity values consequent to the deletion of various lengths in t. Choose the maximum.

3. Similarly consider all $H[i, j']$ for $j' < j$.

When H is complete, its maximum entry is the similarity of the two sequences. For an arbitrary penalty function, terms 2 and 3 may require $O(n)$ time, leading to $O(n^3)$ time for the entire algorithm. But if the penalty functions are simple—if they are linear in the length of the deletion, say—then terms 2 and 3 can be evaluated in constant time, leading to $O(n^2)$ time overall (see [Got82]). We used an $O(n^2)$ algorithm.

[2] for example, *acgtcgt* is the same sequence as *acgtacgt*, with the second *a* deleted; *acgtacgt* is a mutated version of *acgtccgt*.

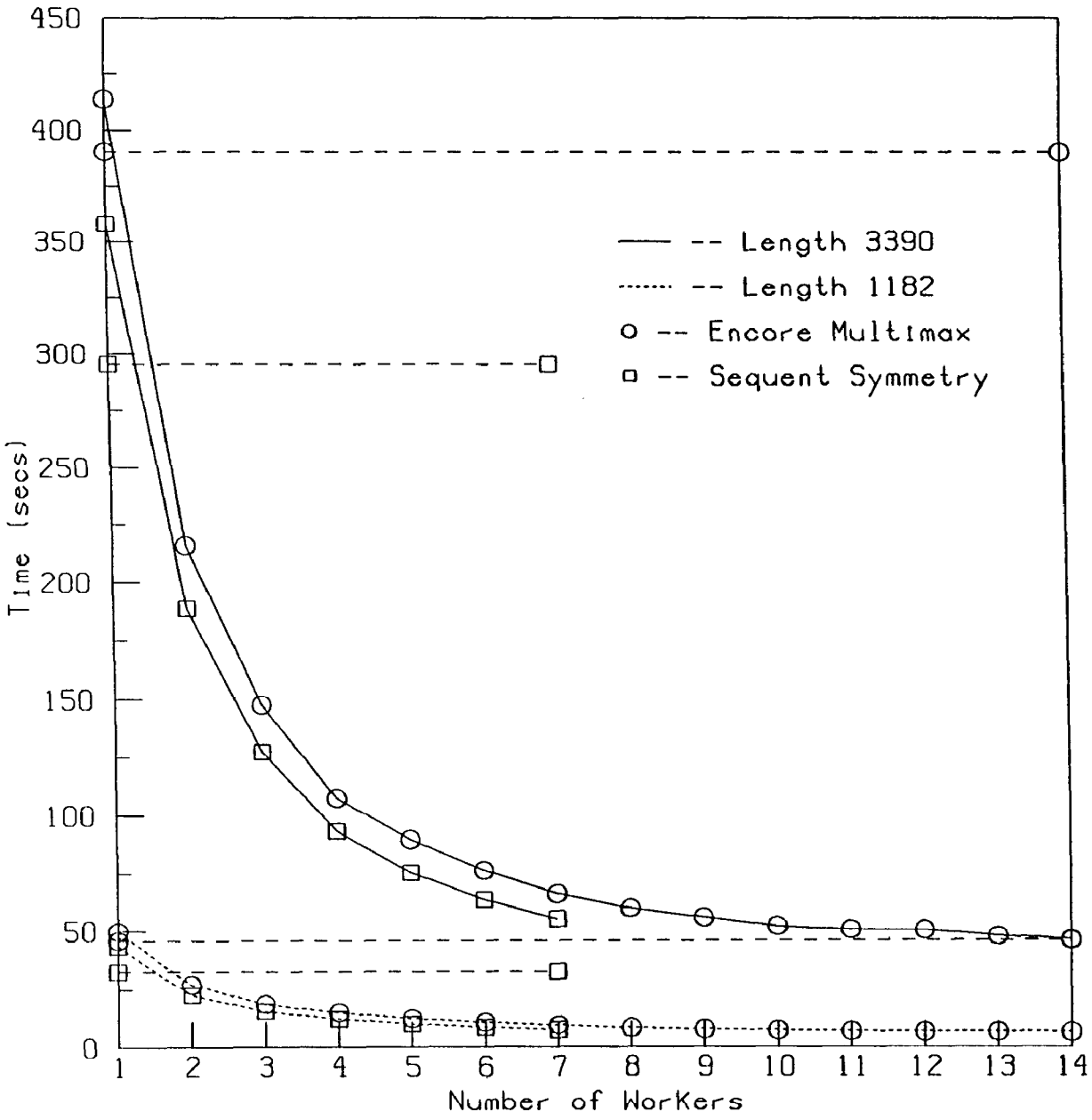

Figure 1: Parallel version of the Gotoh sequence comparison algorithm. The curve shows absolute running times (not speedup). The dashed lines represent running times for the sequential algorithm coded in C and running on one node: given two workers, Linda is faster than C in all cases, and Linda performance continues to improve as we add workers. Note that total *processes* is one more than total workers: 3 processes (a master and 2 workers) are active in the 2-worker version and so on.

The parallel Linda version works as follows. Careful study of the data dependencies leads to the observation that a counter-diagonal of H can be computed as soon as the previous counter-diagonal is complete. The same observation holds if we deal with sub-blocks rather than individual elements of H: once the upper-leftmost block is computed, the blocks directly to the right and directly beneath can be computed, and so on. We can parallelize the algorithm by defining as a task the computation of one horizontal strip of blocks of the matrix. These task will have a staggered start (first block (1,1)'s strip and then block (2,1)'s and then (3,1)'s and so on), which means that this algorithm cannot achieve ideal speedup (some processors will be idle during the staggered start phase and also during a similar phase at termination). We can control this cost by reducing the block size, but only at the cost of additional communication (the more blocks, the more data to be communicated across block boundaries). The results show that good performance is possible notwithstanding. Linda's tuple-space operations come into play when (1) the initial tasks are distributed, (2) boundary values are communicated between workers responsible for adjacent strips, and (3) each worker returns its best value to the master at termination. In each case, the requisite data is dropped into tuple space via Linda's `out` operation and retrieved using `in`—a maximally-simple use of Linda.

Figure 2 shows the performance of a different Linda program, the parallel database search routine that conducts many sequential searches simultaneously; results are shown for Linda running on the Encore Multimax and the Intel iPSC hypercube multiprocessors[3] The program is simple: worker processes are set up; the target sequence is dropped into tuple space via `out` and read by each worker using the Linda `rd` operation; the sequences found in the database are outed, and workers repeatedly grab a sequence, compare it with the target, `out` the result and repeat until the database is empty. A master process picks up these result tuples and reports the best overall. Figure 2 shows performance on a search involving a small comparison set of 256 sequences, but performance is similar when we run comparisons against an actual sequence database (the GENBANK database). To coordinate a search against a large database, the master process uses a "low watermark" approach. It `outs` an initial batch of sequence tuples, then waits until most of the initial batch has been searched-against (by tallying incoming result tuples). When the number of remaining sequence tuples falls below the low watermark, another batch is outed, and so on until completion.

3 Primes

The examples above would have posed an interesting challenge to a restructuring compiler (or equivalently, to a functional-programming system in which the compiler or run-time system takes charge of parallelization). In the second case, for example, the system would have been been required to make a reasonable decision about the distribution of the original sequence, and then

[3]These results reflect the $O(n^3)$ algorithm instead of the improved $O(n^2)$ version. Performance of the Encore and the iPSC will also be roughly comparable using the faster algorithm in searching GenBank instead of the test database. The sequences in the test database are 64 elements long; the average sequence in GenBank is on the order of 1000 elements, which makes each task compute-intensive and the difference in communication speed between the two machines insignificant.

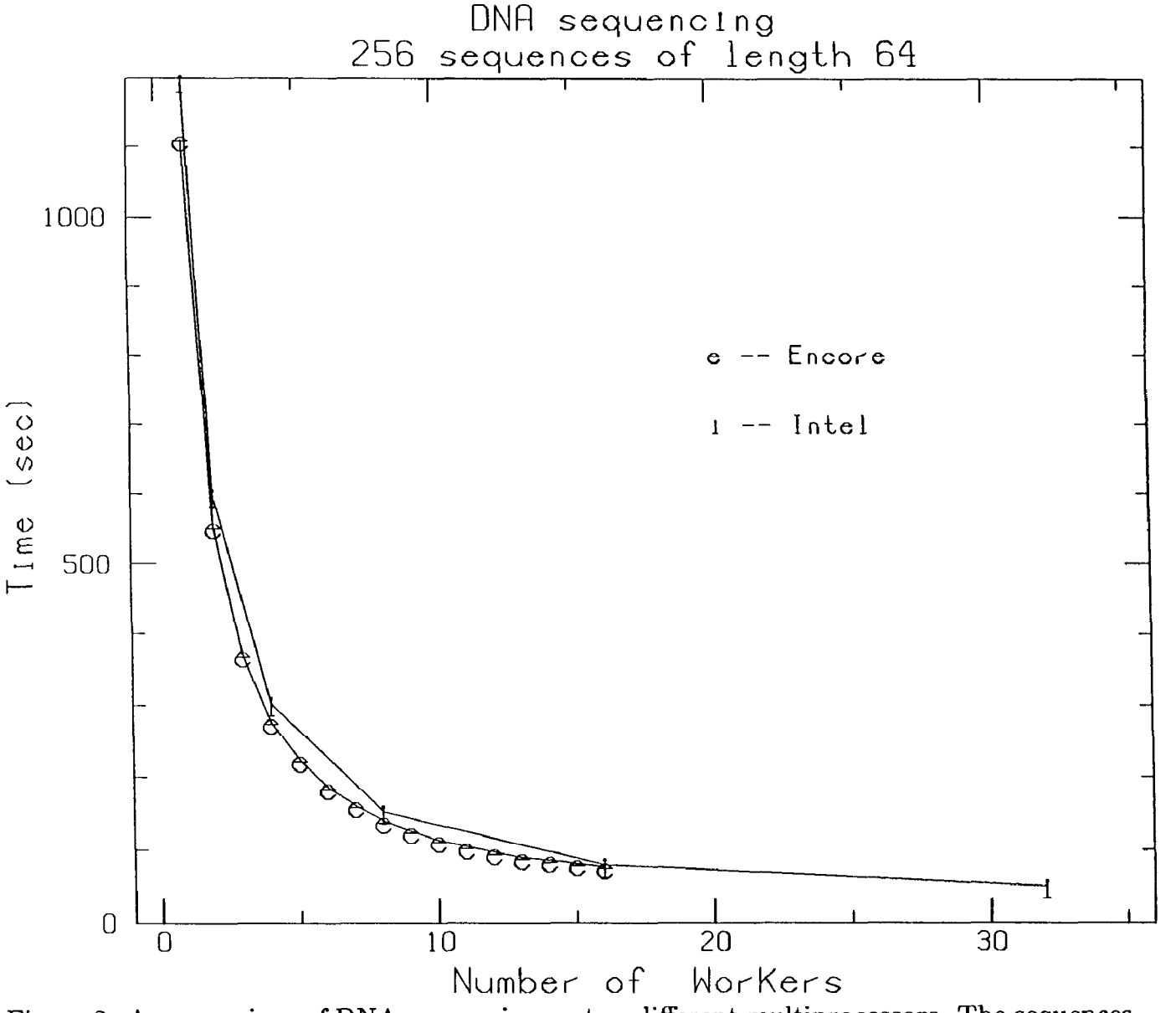

Figure 2: A comparison of DNA sequencing on two different multiprocessors. The sequences were 64 bases long, and 256 sequences were compared against the target.

to have designed and implemented a low-watermark scheme or the equivalent. More generally, it would have been required to decide whether to parallelize a single comparison or to conduct a parallel database search: the comparison algorithm itself is, of course, the same in either case. But leaving this question aside, consider a case in which parallelizing compilers are genuinely hard-pressed.

The following extract is from a note posted to an ARPANET bulletin board by Martin Fouts of NASA Ames:

> As an example of a class of algorithms which is difficult to vectorize or parallelize, let me pull out the ancient prime finder algorithm: [*figure 3*]. Although there are different algorithms for finding primes, I use this one to illustrate a class of problems which comes up frequently in my work. There exists some set from which must be drawn a subset. There exists a rule for ordering the set and another rule for determining if an element is a member of the subset [...] None of the vectorizing compilers that I have access to will attempt to vectorize this algorithm. The Alliant automatic parallelizer will not attempt to parallelize it. Most of the mechanisms I have tried to handcraft a vector or parallel variant which remains true to the previous paragraph have added sufficient extra work to the algorithm that it runs more slowly as the number of processors increase.

The Linda version (which, Fouts agrees, is faithful to the sequential algorithm, although to be fair our code is substantially longer[4]) again uses the master-worker model.

```
      IPRIME(1) = 1
      IPRIME(2) = 2
      IPRIME(3) = 3
      NPRIME = 3
      DO 50 N = 5, MAXN, 2
         DO 10 I = 3, NPRIME
            IQ = N / IPRIME(I)
            IR = N - (IPRIME(I
     +                  * IQ)
            IF (IR .EQ. 0) GO TO 40
            IF (IQ .LT. IPRIME(I))
     +         GO TO 20
 10      CONTINUE
 20      NPRIME = NPRIME + 1
         IPRIME(NPRIME) = N
         IF (NPRIME .GE. MAXP)
     +      GO TO 60
 40      CONTINUE
 50   CONTINUE
 60   CONTINUE
```

Figure 3: Prime finder

The results in figure 4 show timings for locating the first 50,000 primes in the range 1 to 10^6. A task in our code consists of checking the integers between n and $n + ChunkSize$ for primality. Workers perform

[4] "Faithful" in a particular, well-defined sense: what was important to Fouts, and captured by the Linda implementation, was the fact that some set had to be searched *in order*, and a subset selected in order. It's easy to imagine a parallel prime-finder that examines each integer independently of the rest, but such an approach is blatantly wasteful: in determining whether j is prime, we can obviously make use of the fact that we know all previous primes through the square root of j. The Linda program allows many segments of the search to be conducted in parallel, but starts the sub-searches in order; later-starting searches rely on the primes discovered by earlier ones. The approach is discussed in detail in [CG88].

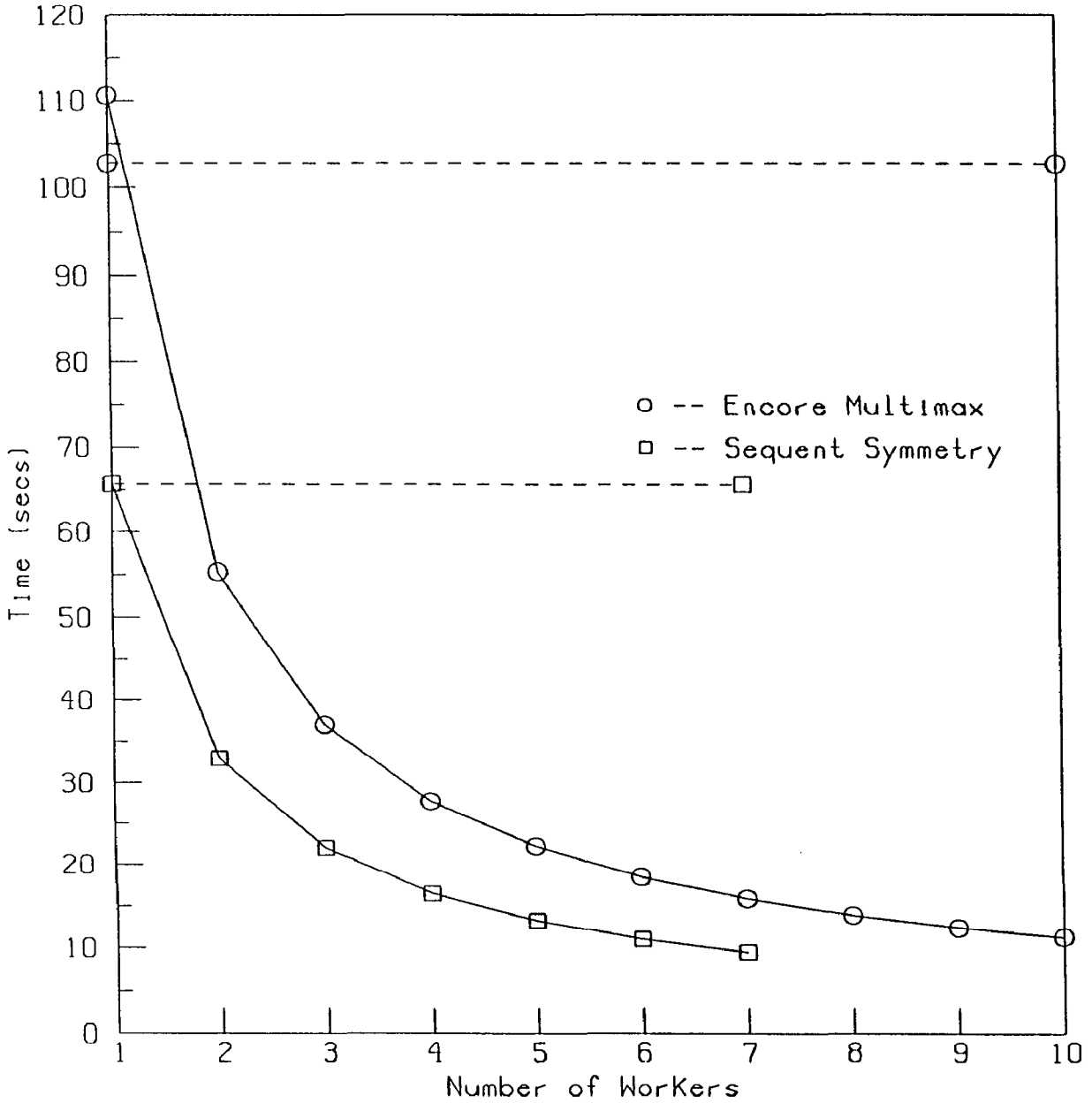

Figure 4: Parallel prime finder. Time to find the first 50000 primes in the range 1 to 100000. The *ChunkSize* was 200. Dashed line is time for sequential C code.

each task by executing code equivalent to the DO 50 loop in the figure. Tasks are distributed dynamically: a worker grabs a task-descriptor using in, then immediately generates a new task-descriptor (i.e., "check the next chunk of integers after mine") for some other worker to pick up. It proceeds to check its own chunk, reports the new primes to the master using out, and grabs another task. The master uses tuple content-addressing to pick up the results in order (in essence, the workers write a distributed (FIFO) stream of tuples, and the master reads it: this is a simple distributed data-structure technique discussed in [CGL86]). The master uses the incoming results to construct (using out) a table of primes in tuple-space (another simple distributed data structure, also dependent on tuple content-addressing). A worker consults the table (using rd) as necessary in extending its sieve.

Perhaps the problem was less difficult to solve than Fouts thought—or at any rate, was less difficult given an environment that provided suitable abstractions (efficiently implemented) to support parallelization. But Fouts wasn't alone in his initial assessment. A few days after he posted his note (and we got our Linda version running), the following comment was posted to the same bulletin board: "The algorithm presented by [Fouts] is not parallizable [*sic*]."

4 Process Lattices

Consider the following problem: given a mass of incoming data, we need a system that will perform two related but separate tasks. (1) It will act as a kind of heuristic database, ready to accept user queries either about the status of a data-stream or the likelihood of some more complex phenomenon given the current state. (2) It will act as a monitor and alarm system, posting notices when significant state-changes occur. In determining the likelihood of more complex states, the system might use quantitative tests, heuristic decision procedures (as in rule-based systems, for example), or any mix of the two. Concretely, a first prototype monitored simulated automobile-traffic flows; a large experiment now underway deals with post-operative cardiac patients. Both programs were written by Michael Factor of our group at Yale.

It's convenient to imagine such a program as a so-called *process-lattice*. Each process in the bottom rung of the hierarchy is wired directly to an external sensor, and performs initial data processing and filtering. Processes at higher levels in the lattice are responsible for more complex states. In the current system, for example, low-level nodes connect to (currently simulated) blood-loss and blood-gain devices; they and other bottom-rung nodes converse with a higher-level node that calculates total fluid volume, which communicates with a higher-level node monitoring the likelihood of hypovolemia (a particular clinically-significant state) and so on.

More precisely, we can describe the process lattice as follows. The lattice contains a collection of nodes, and each one defines a mapping from a set of input states (the states of the inferior nodes) to an output state (its own state). We posit that a node's state always reflect (or be in transition to) the value yielded by applying its own state function to the current values of its input states and its own state. It follows that, whenever some value changes, the effect of the change ripples upwards through the lattice. A node's state may be "undefined" as well, or may be "pending", which is an intermediate condition; each node's state

function determines explicitly when the local state value is defined, undefined and pending. Concretely, a value will be defined when "enough" inferior values are defined to allow the state function to be computed. Nodes with the null state function (depending on no inferiors, relying on externally-supplied values) have "undefined" state in the absence of an external data signal. Suppose, though, that some important inferior values (values that are highly weighted in the local decision procedure) are available, but others are missing; the local decision procedure might then decide that it *ought* to be defined, although it lacks sufficient data to compute a state; so it becomes "pending". A "pending" node reflects a request-for-data downwards to all undefined inferiors. They become "pending" in turn, and reflect the request downward again. Thus the effect of a data-request filters downward as data values filter upwards; as requests filter downward, they merely notify all "state undefined" nodes along the way that data is wanted. If we equip bottom-rank nodes with warning lights, a flashing warning light means "enter data". The data filter upwards, and eventually all undefined states become defined.

We supply two types of "logic probe" with the system, an "inject value" probe and a "read value" probe. We can touch any node with an "inject value" probe, thereby setting the state of the node we touch to any value we choose. In the default configuration described above, then, each node in the bottom rank has a null state function; instead, each bottom-rank node has a permanently-attached inject-value probe through which we pump new values into the system. We can read any node's current state by touching it with a "read-value" probe. If the current state of the node we touch is undefined, touching it with a read-probe changes its state to "pending", and the query propagates downward to each of the node's inferiors. Eventually new data values arrive and a query-response is computed.

It's easy to build such a program in Linda. In outline, the implementation works as follows: each lattice node is implemented by a separate process; each node stores its current state in a tuple; a node that updates its state (by removing its state tuple via `in`, and reinstalling an updated version via `out`) notifies all interested parties by sending signals along message streams. Message streams are implemented as sequences of numbered tuples; the technique is discussed in [CG88]. When node Q is informed that "node N has just updated its state", Q uses `rd` to read N's state tuple directly.

This program architecture might in the abstract be implemented on top of a message-passing or a concurrent object system; Linda isn't the only possible implementation vehicle. But Linda seems like a good choice. The salient point is that Linda allows state information to be stored in tuple space, where it is directly accessible to any interested party. Any number of concurrent processes may read any node's state directly, and simultaneously; a state is *in*accessible only when the state tuple has been (temporarily) withdrawn for updating. Ease of access to state data is important to a program architecture in which many processes may need access to this data, and new processes may dynamically and unpredictably seek access as the program runs (particularly when we attach probes). Storing each node's state in a local data structure, and shipping it out explicitly to each interested party (a message passing solution), or encapsulating state data inside a monitor, and deal-

ing with a monitor's cumbersome synchronization mechanisms to insure concurrent access to readers, exclusive access to writers, and correct delivery of "update" signals along inter-node message streams (as in a monitor-based concurrent-object language), seem like more complicated and less attractive strategies.

The ICU monitor problem was posed by anesthesiologists at the Yale Medical School; Dr. Perry Miller (director of the Medical Informatics program at Yale) and Dr. Stanley Rosenbaum are collaborators in this research. The current system (diagrammed in figure 5) is too incomplete to be clinically testable. It uses a process lattice with 38 nodes; our domain experts estimate that roughly four times as many would be required in order to cover 95% of the relevant major diagnoses. Our goals in the project extend beyond the prototype, though. The experts are pleased with the prototype's performance: a recent test involving five one-hour sets of simulation data produced "correct" and "reasonable" behavior. Development work aimed ultimately at a clinically-testable system continues.

5 Conclusions

It's clear that we can parallelize interesting applications using Linda. The transformation is by no means uniformly simple to accomplish; some thought may be involved, which is precisely why Linda is necessary and (as in the primes-generating example) compilers can't necessarily be relied on, at least for now. On the other hand, although the transformation from sequential to parallel may require some thinking, it rarely seems to require a frightening amount[CG88]. We know of nothing that should prevent any competent programmer from learning how to do it. Of course there are also cases in which no transformation is required because, as in the last example, the program structure involved is conceived in parallel from the start.

The results we've presented deal with small parallel machines only, which represents our experience to date. We can't prove, yet, that Linda will work well on large multi-computers, but we suspect that it will, and are working on kernels for larger machines now. (The Linda Machine is designed to scale upwards to roughly a thousand nodes. For the time being, a thousand powerful nodes will be plenty for our purposes.) A replicated-worker program in Linda looks the same whether there are 10 or 1000 workers (not that its performance is guaranteed to scale up, but certainly the *design* of such a program poses no unsolved problems); process-lattices involving thousands of decision nodes are also easy to imagine, and will be no harder to build than our current smaller versions.

Appendix

The Linda model is a *memory* model. Linda memory (called *tuple space*) consists of a collection of logical tuples. There are two kinds of tuples waltzing around inside it. Process tuples are under active evaluation; data tuples are passive. To build a Linda program, we ordinarily drop one process tuple into tuple space; it creates other process tuples. The process tuples (which are all executing simultaneously) exchange data by generating, reading and consuming data tuples. A process tuple that is finished executing turns into a data tuple, indistinguishable from other data tuples.

One Linda programming paradigm we rely on involves *distributed data structures* and a bunch of identical worker processes

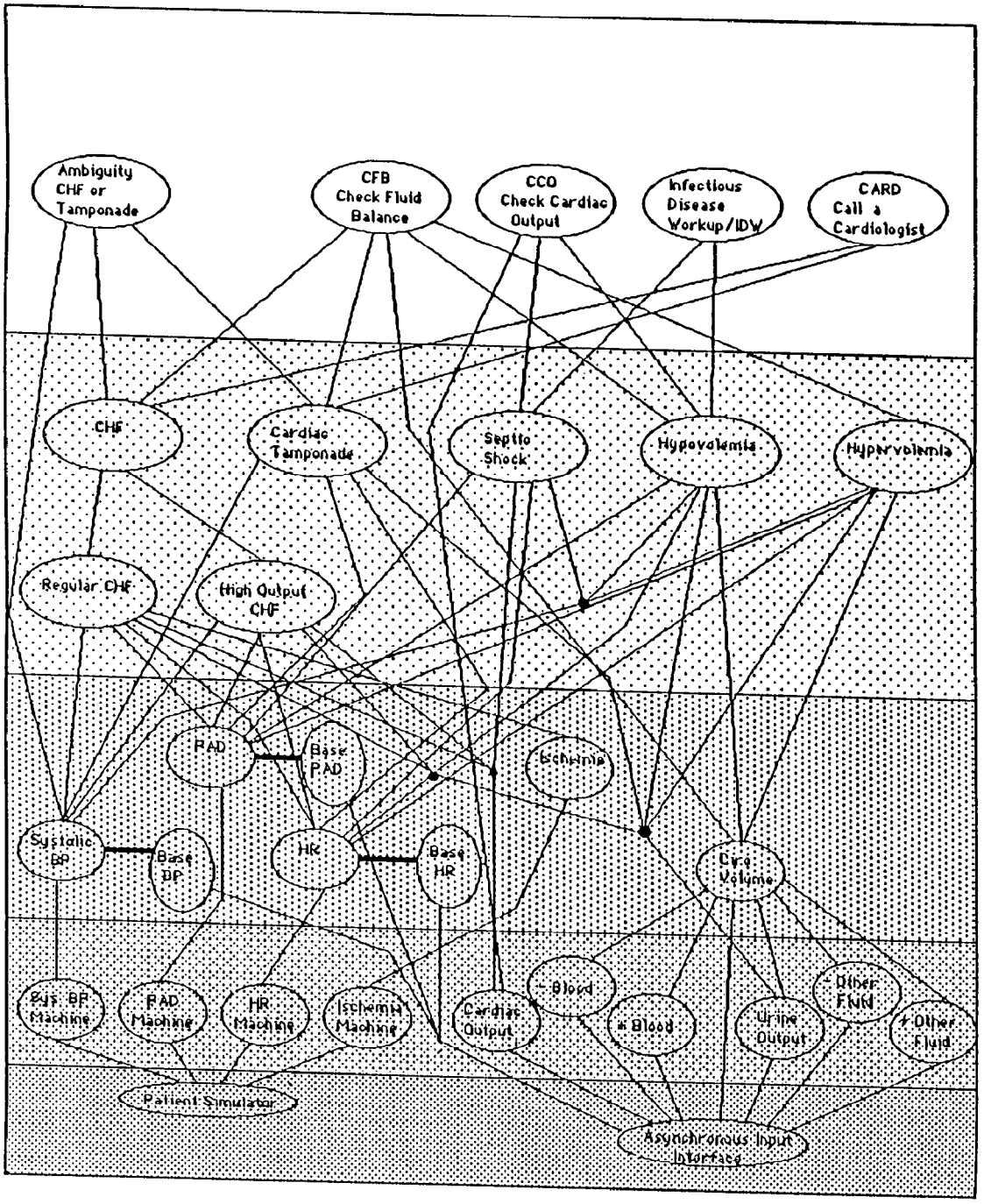

Figure 5: Process Lattice for an ICU Monitor.

(or several bunches of different kinds of processes) crawling over the data structures simultaneously. We use the term *distributed data structure* [CGL86] to refer to a data structure that is directly accessible to many processes simultaneously. Any datum sitting in a Linda tuple space meets this criterion: it is directly accessible — via the Linda operations described below — to any process that currently occupies the same tuple space. A single tuple constitutes a simple distributed data structure. We can build more complicated multi-tuple structures (arrays or queues, for example) as well.

It's reasonable to describe a parallel computer that supports Linda as an "unconnection machine". Programming models like Occam [M83] and the Connection Machine [Hil85] tend to bind concurrent processes tightly together (implicitly, through the intermediation of a parallel data structure, in the case of the Connection Machine). In Linda the opposite is true. Linda processes aspire to know as little about each other as possible. They never interact with each other directly; they deal only with tuple space. We believe that tightly-bound collections of synchronous or quasi-synchronous activities tend to force programmers to think in simultaneities. Great simplification of the potentially formidable task of parallel programming is possible, we believe, if concurrent processes are so loosely bound (so *un*connected) that each can be developed independently of the rest.

There are four basic tuple-space operations, `out`, `in`, `rd` and `eval`, and two variant forms, `inp` and `rdp`. `out(`t`)` causes tuple t to be added to TS; the executing process continues immediately. `in(`s`)` causes some tuple t that matches template s to be withdrawn from TS; the values of the actuals in t are assigned to the formals in s, and the executing process continues. If no matching t is available when `in(`s`)` executes, the executing process suspends until one is, then proceeds as before. If many matching t's are available, one is chosen arbitrarily. `rd(`s`)` is the same as `in(`s`)`, with actuals assigned to formals as before, except that the matched tuple remains in TS. Predicate versions of `in` and `rd`, `inp` and `rdp`, attempt to locate a matching tuple and return 0 if they fail; otherwise they return 1, and perform actual-to-formal assignment as described above. (If and only if it can be shown that, irrespective of relative process speeds, a matching tuple must have been added to TS before the execution of `inp` or `rdp`, and cannot have been withdrawn by any other process until the `inp` or `rdp` is complete, the predicate operations are *guaranteed* to find a matching tuple.) `eval(`t`)` is the same as `out(`t`)`, except that t is evaluated after rather than before it enters tuple space; `eval` implicitly forks a new process to perform the evaluation. `Eval` has been implemented on the Encore and Sequent but not yet on the other systems we discussed. (Where `eval` doesn't yet exist, programmers rely on the native operating system to fork processes.)

Tuple space is an associative memory. Tuples have no addresses; they are selected by `in` or `rd` on the basis of any combination of their field values. Thus the five-element tuple (A, B, C, D, E) may be referenced as "the five element tuple whose first element is A," or as "the five-element tuple whose second element is B and fifth is E" or by any other combination of element values. To read a tuple using the first description, we would write

```
rd(A, ?w, ?x, ?y, ?z)
```

(this makes A an actual parameter – it must be matched against – and `w` through `z` formals, whose values will be filled in from the

matched tuple). To read using the second description we write

```
rd(?v, B, ?x, ?y, E)
```

and so on. Associative matching is in fact more general than this: formal parameters (or "wild cards") may appear in tuples as well as match-templates, and matching is sensitive to the types as well as the values of tuple fields.

References

[ACG86] S. Ahuja, N. Carriero and D. Gelernter, "Linda and Friends," *IEEE Computer*, August 1986.

[ACGK88] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine," *IEEE Trans. on Computers*, August 1988.

[BjCGL88] R. Bjornson, N. Carriero, D. Gelernter, and J. Leichter, "Linda, the Portable Parallel," Research Report YALE/DCS/RR-520, Yale University, January 1988.

[Car87] N. Carriero, *Implementing tuple space machines.* Doctoral Diss., Yale University, 1987.

[CG85] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," in *Proc. ACM. Symp. Operating System Principles*, December 1985 and *(ACM Trans. Comp. Sys.* May 1986.

[CG88] N. Carriero and D. Gelernter, "How to Write Parallel Programs: A Guide to The Perplexed," Research Report, Yale University, May 1988.

[CGL86] N. Carriero, D. Gelernter and J. Leichter, "Distributed data structures in Linda," in *Proc. ACM Symp. Principles of Prog. Languages*, January 1986.

[Don87] J. Dongarra, "Performance of Various Computers Using Standard Linear Equations in a FORTRAN Environment," Technical Memorandum, Argonne National Laboratory, 1987.

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

[GB82] D. Gelernter and A. Bernstein, "Distributed communication via global buffer," in *Proc. ACM Symp. Principles of Distributed Computing*, (Aug. 1982):10-18.

[Got82] O. Gotoh, "An improved algorithm for matching biological sequences," *J. Mol. Biol.* 162(1982):705-708.

[H85] D. Hillis, *The Connection Machine.* MIT Press (1985).

[Jor86] H.F. Jordan, "Structuring parallel algorithms in an MIMD, shared memory environment," *Parallel Computing* 3(1986):93-110.

[May83] M.D. May, "Occam." *ACM SIGPLAN Notices*, 18-4(1983):69-79.

[NPA86] R. Nikhil, K. Pingali and Arvind, "ID Nouveau," Technical Report 265 (MIT) 1986.

[Tha88] *IEEE Software: Special issue on parallel programming*, S.S. Thakkar, *ed.*, January 1988.

[WL88] R. Whiteside and J. Leichter, "Using Linda for Supercomputing On a Local Area Network." *Supecomputing '88*, (to appear).