Linda, the Portable Parallel

Robert Bjornson, Nicholas Carriero,
David Gelernter and Jerrold Leichter

Research Report YALE/DCS/RR-520
February 1987

(revised January 1988)

This material is based on work supported by the NSF under grant # DCR-8601920.

Linda, the Portable Parallel

Robert Bjornson, Nicholas Carriero, David Gelernter and Jerrold Leichter

Yale University
Department of Computer Science
New Haven, Connecticut

Abstract. We address two questions: is it practical to provide high-level languages for explicitly-parallel programming? Is it reasonable to suggest that the same language be made available on a broad spectrum of parallel machines? We argue that the answer to both questions should be yes. We present and discuss two results: a smart optimizing pre-processor and a runtime kernel working together are a powerful basis for a complete Linda system; the Linda kernel can be implemented effectively on a broad range of parallel computers, including both shared-memory and message-passing architectures. We discuss the design and performance of Linda systems for Encore and Sequent shared-memory multiprocessors, the S/Net bus-based message-passing network and the Intel iPSC hypercube link-based network.

This material is based on work supported by the NSF under grant number DCR-8601920.

1 Introduction

We address two questions: is it practical to provide high-level languages for explicitly-parallel programming? Is it reasonable to suggest that the same language be made available on a broad spectrum of parallel machines?

To date, the predominant answer to the first question has been "maybe" — the idea has been around for some time, but relatively little implementation experience has been reported, and the advantages of a high-level environment in expressivity *and* efficiency have rarely been spelled out. The second question is usually answered *no*.

The existence of high-level languages, *portable* ones in particular, has been

taken for granted on conventional machines for decades. "I will not be forced to write assembly-level code" and "I will not have my language and programming style dictated to me by the pile of hardware I happen to have acquired" have become the defining statements of civilized programming. We believe that the same should be true on parallel machines. We present results intended to show that, at least in a preliminary way, both questions should be answered *yes*. Linda has been presented before as a communications kernel for AT&T Bell Lab's S/Net multi-computer. We present it here as a complete parallel language (a communications kernel together with a compiler) that runs on a broad spectrum of parallel computers. We will focus on implementations for the Encore Multimax and Sequent Balance shared-memory multi-computers, and the Intel iPSC hypercube and S/Net memory-disjoint multi-computers. We will refer also to operational Linda systems for the shared-memory Tadpole and for Ethernet-connected VAXes under VMS. We will discuss the performance of several simple but significant applications, including matrix multiplications, LU decomposition, parallel database search in the context of DNA sequence comparison and the travelling salesman problem.

The evidence that we will present demonstrates that a portable high-level parallel language like Linda is a reality on shared-memory machines; it does not demonstrate, but it strongly suggests, that exactly the same environment can be realized on memory-disjoint machines. Our claims with respect to memory-disjoint machines are limited by the hardware at our disposal. In particular (1) the MC-68000-based S/Net is obsolete, but parallel Linda programs nonetheless demonstrate good speedup over sequential programs running on the same processor. (The outdated S/Net is the basis for the hardware Linda Machine that is now under construction). (2) The Intel iPSC is limited by its extremely slow inter-node communication, but Linda is nonetheless a working tool in this environment, and should be a better and more flexible one on the next generation iPSC with its much-faster inter-node communication. The Encore and Sequent implementations on the other hand are effective and flexible tools on current-generation hardware, and our experience with a port to the custom-built Tadpole multiprocessor suggests that Linda is readily adaptable to other shared-memory architectures as well.

We begin by discussing the terms "high-level" and "portable" as they apply to parallel languages, and we categorize the results to be presented into two classes: the first class deals with the machine-independent Linda optimizing pre-processors, the second with the spectrum of machine-specific runtime communication kernels. Linda systems are dialects of some base language; at present C-Linda and Fortran-Linda pre-processors exist. The pre-processor, base-langugae-specific but machine-independent, is central to efficient Linda implementation and we will argue that, more generally, compiler technology has a central role to play in supporting parallel programming environments. The

runtime kernel is base-language-independent but machine-specific; our experience suggests that a portable language need not represent merely the greatest common denominator over the architectures to be supported.

In section three we briefly summarize Linda. The two following sections respectively treat the two categories, compiler and kernel design. The last section gives conclusions.

2 Defining terms and classifying our results

What does a "high-level parallel language" mean? Parallel languages differ from conventional ones in providing for the creation of parallel processes and for communication among them — either explicitly or through the medium of a parallel data structure. (It is the first case that is of interest here: we concentrate on control-level rather than on "data-level" [Hil85] parallelism.) A high-level parallel language implies two things. First, the tools it provides for handling parallelism are powerful and flexible. Second, these tools are an integral part of a programming language, recognized by a compiler. Now that parallelism is a commercial reality (more than a dozen manufacturers sell multi-computers, and many next-generation machines are under construction at research centers), it's possible to scan the field and to observe that in fact most multi-computers are supplied with low-level parallelism tools, and that in most cases these tools are supplied as system calls of which the compilers are ignorant.

What does a "portable parallel language" mean? Parallel machines span an enormous spectrum; of particular importance is the fact that ratios of computation to communication speeds differ dramatically among architectural classes. The host machine's computation-to-communication ratio is one determining influence on the way parallel programs are written. This broad spectrum of ratios is often offered, then, as an excuse for the lack of portable languages. These machines are so different, the argument goes, that it isn't possible and doesn't make sense to encompass them within one programming environment. We don't accept this excuse: n different performance levels are more easily handled, we believe, in the framework of *one* flexible, scalable language than in n separate frameworks. A program (whether parallel or not) that is ported from one machine to another very different one may run poorly, but at least it will run — and the programmer can change and tune it in the same familiar idiom he made use of on the last machine. But it's also true that a sizable and interesting class of parallel programs, the relatively coarse-grained or non-communication-intensive applications, can run well (as we discuss) on a broad spectrum of machines. And finally, performance gaps between architectural classes don't explain why two machines in the *same* class — two shared-memory multiprocessors, for example, or two hypercubes — shouldn't offer the same programming environment. But empirically they almost never do.

The domination of low-level, non-portable parallel programming tools slows the growth of knowledge about how *in general* we are to approach parallel programming. It encourages a growing body not of widely-applicable techniques, but of machine-specific hacks.

Linda consists of a small number of powerful operations that may be integrated into a conventional base language, giving a dialect that supports parallel programming. Thus C with the addition of Linda operations is a parallel programming system; it consists of a pre-processor from C-Linda into C and a kernel to support the Linda operations at runtime. Unlike (for example) Hoare's CSP, Linda was designed purely for power and simplicity, in arrant disregard for the practical problems of implementation. We reported on our first efficient Linda kernel some time ago [CG85]. In this paper we concentrate on two new results, which address the two questions raised above. In essence our results are, first, that complete high-level parallel languages (as opposed to kernel-only systems) can be implemented efficiently, and smart compilers are central to this effort; second, that even as quirky and high-level a language as Linda can be supported effectively on a wide spectrum of very different parallel machines.

Our first result is that *a smart optimizing pre-processor and a runtime kernel working together are a powerful basis for a complete Linda system.* Most multicomputer manufacturers support parallelism by offering not parallel *languages* with compilers, but rather *system calls* whose semantics are unknown to the compilers. (Linda itself was first implemented as a kernel only, with no compiler.) We have found, however, that a smart compiler offers not only the obvious advantages in cleaner, more readable, more traceable-and-debuggable source code, but major runtime performance gains as well. These gains are particularly significant in the case of Linda, which, as an abnormally "high-level" language, demands particular cleverness of its implementors. But given our experience, we see no reason why *any* parallel-programming environment, whether built around Linda or not, shouldn't offer users some kind of parallel language in preference to isolated system calls.

What is the cost of Linda's "high-levelness"? We will describe one experiment that suggests that, given an optimizing compiler, the cost of using high-level Linda versus low-level system calls is surprisingly low — clearly less, in this one case anyway, than the routine overhead associated with compiling rather than hand-coding one's programs.

Our second result addresses the portability question. *The Linda kernel can be implemented effectively on a broad range of parallel computers, including both shared-memory and message-passing architectures.*

After outlining Linda, we take up each of these two results in turn.

3 Linda

Because Linda has been discussed at length in the literature (e.g. [GB82, Gel85, CG85, CGL86, AhCG86]), we give only a brief outline here.

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 (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 an 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 on the other two systems we discuss, and so we restrict discussion primarily to the other Linda operations. (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. 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.

4 The Pre-Processors

The compiler is important to efficient Linda implementation, and we believe that compile-time analysis is in general a powerful tool in supporting parallelism. Compilers or pre-processors are central to the question of whether high-level parallel languages are a reasonable alternative to a library of system calls—compilers are required on the one hand in order to recognize the syntax of high-level languages, on the other to smoothe the way for the more complex demands these languages tend to make of runtime communication systems. How parallel-language compilers should be built and how well they perform are im-

portant questions, then, in assessing the general prospects for high-level parallel languages. There is no such thing as a "typical" high-level parallel language. Our results therefore can't be taken as showing the way to a general solution of the problem. They do show how the problem can be solved for one particularly idiosyncratic and quirky language. They also play a major part in explaining how Linda systems are constructed.

Some of the Linda pre-processor's goals should be relevant to many language environments. In any system that involves communication among disjoint address spaces, the compiler is in a position to speed things along by pre-formatting header fields, supplying information about length and type of data fields and so forth. Another general motivation involves error handling and runtime tracing. A parallel environment that relies on system calls can't alert the user at compile time to problems that have to do with the semantics of the parallel-programming operations; nor will information that can only be gathered from sources be available for runtime tracing.

These concerns are addressed in the Linda pre-processors using fairly general techniques. The pre-processors (we will refer to them collectively as *lpp*; these comments apply both to C-Linda and to Fortran-Linda) create one "proto-buffer" for every Linda statement in the source. Proto-buffers contain information about the number of fields in the tuple, their types, and the runtime location of the values they will ultimately contain. *lpp* flags errors in Linda usage; if, for example, the program it is examining includes `in` statements that no `out` will ever match, it mentions the fact. (Some of its abilities in this area are based on the tuple analysis stage, which we discuss below.) Information is accumulated that may be useful in runtime tracing. For example, the text of each Linda operation as it appeared in the source is stored in that operation's proto-buffer. In one of the several tracing modes available at runtime, the source of each Linda operation, together with the values of any actual parameters it mentions, is displayed when it is executed.

Tuple analysis is an optimization that, unlike the others we've discussed, is unique to Linda.

The generality and power of Linda's associative matching make it potentially costly at runtime. Because Linda's `in` and `rd` operations support tuple-selection based on any arbitrary combination of tuple fields, the kernel must be prepared to inspect large numbers of tuples in searching for a match. The fact that `out` allows wildcarding — the argument list to `out` may contain formals, just as the argument list to `in` may contain actuals — complicates things further. *lpp*'s goal is to optimize associative matching by supplying *exactly as much generality at runtime as a given Linda operation requires.* The Linda kernel supplies four different matching and tuple-storage routines, which perform increasingly general runtime matching. The preprocessor translates each Linda operation into an

invocation of one of the four, depending on the character of its argument list. The process by which this is accomplished is described in detail in [C87]; the following paragraphs give a general outline.

The first step is to partition all Linda calls within the program into disjoint sets, such that tuples and templates in different sets can never match. Partitioning is accomplished by using a compile-time version of the Linda matching protocol; this weaker version is the same as ordinary matching, except that non-constant actuals of the same type are assumed to match. Partitioning proceeds by choosing one call at random to seed the new partition and then repeatedly comparing all still unassigned calls against all calls in the partition, adding matches to the partition, until no more matches can be found. At this point a new seed is chosen for the next partition and the process repeats until completion.

Next, each partition is classified, based on the usage pattern for tuples and templates in the partition. This is best explained by example. Consider the following partitions:

```
A:  out("foo", i)
    in("foo", ? j)

B:  out("vector", i, j)
    in("vector", k, ? l)

C:  out("element", i, j)
    in("element", k, ? j)
    rd("element", ? k, ? j)
```

In set A, the first field is always constant; it can be removed by the compiler. Furthermore the second field within `in` statements is always a formal; since any tuple in the set will satisfy any template, no matching is required at run-time. Tuples from this set can be stored at runtime in a simple queue.

In set B, the first and third fields are analogous to the two fields in set A, but the middle field will require runtime matching. Because this field is always an actual, we can use its value as a key and store these tuples in a hash table.

The third possibility is set C, where a field exists (here the middle field) that over all `in`'s and `rd`'s is sometimes an actual and sometimes a formal. We have to do a match, but we don't always have a key. There are several ways to manage this set; we decided on a private hash table scheme. `out`s, which always have an actual to use as a key, are hashed on that key. When performing an `in`, if a key exists we follow it to the correct bucket. If no key exists, we must search the entire table. Since each "class C" partition has its own private table,

the search shouldn't be too expensive.

The last class is a catch-all for sets that don't fit into one of the above schema—these sets always include **out** statements in which formals appear, in our programming experience a rare case. We store such partitions on a list, and search the list for matches.

A significant limitation of our current *lpp* is its inability to deal with separately-compiled Linda modules. It's clear how the analysis will extend to the separate-module case, but a fair amount of additional code will be required.

Performance. It's difficult to isolate *lpp*'s contribution to Linda's runtime performance, because the compiler and new kernels to support it were developed together, and each assumes the other. But the following data point is of interest. A "primitive tuple exchange" is an out-in pair that involves a tuple with only a single small, constant field. (Thus e.g. `out("ping")` — `in("ping")`). On the NS32032-based Encore Multimax, system time for a primitive tuple exchange before we installed the compiler with its new kernel was about 700μsec; with the new combination in place, it was about 200μsec.

Of much greater interest in assessing the combined kernel-compiler system's performance are the results of a test proposed by Encore[1].

This was the proposed test: take the standard Donagarra Linpack benchmark; recode it in C; write a parallel C-Linda version based on the sequential C version, and measure its performance. (The Dongarra code is written in Fortran, but Fortran-Linda was not yet available when we performed this test.)

Dongarra's benchmark routine repeatedly solves systems of linear equations by the standard method: perform an LU factorization of the matrix, then perform a forward and backward solve. The factorization step dominates time cost — it is $O(n^3)$, while the solve is $O(n^2)$. We therefore wrote a C-Linda version of the factor routine but not of the solve routine.

The resulting Linda code worked well. It showed linear speedup through ten processors: that is, execution time could be modelled closely by a curve of the form $a/n + b$. The program used a replicated worker model involving a single master process and multiple identical worker processes. It required only two workers to finish faster than the sequential C program. In other words, the overhead of communication in Linda was recouped when 3 processors were available. Adding processors at that point led to absolute gain in runtime. (The Linda program itself is described in [C87].)

So in this case at least, the Linda kernel-compiler system indeed lends itself

[1]Encore Computers provided Scientific Computing Associates of New Haven with a machine for our use on this project, and we are duly grateful.

to the programming of a useful application that achieves real speedup. A second obvious question suggests itself: programmers expect to pay for high-level semantics in somewhat lower performance relative to low-level, high-performance alternatives; what do Linda's high-level semantics cost?

We addressed this question by writing another version of the Linpack program that was modelled algorithmically on the Linda version but used the Encore Multimax's native spinlocks instead of Linda for inter-process communication and synchronization. The results show that the Linda version ran in the worst case (small problem, few workers) approximately 10% slower than non-Linda, improving in the best case to essentially identical. In figure 1 we have graphed the performance of the Linda program against the performance of the non-Linda version. The abcissa shows number of worker processes. Note that total *processes* are one greater than total workers. The dashed line on the graph represent ideal speedup of a sequential C program, where "ideal speedup" is linear speedup in the complete absence of overhead.

These results show clearly that, in this case at least, elegance is a bargain. The performance penalty for using Linda instead of a low-level system is minimal — clearly less than (for example) the expected performance penalty in the routine use of a high-level language rather than assembly code.

5 The runtime kernels.

The nature of the problem. The Encore Multimax and Sequent Balance series are both shared-memory multiprocessors. In the current generation, a few dozen processors maximum (NS32332's or NS32032's respectively) share access to global memory. Both are Unix machines. Same processor family, basic architecture, operating systems, same-size boxes; and they agree that shared memory and locks will serve as their basis for inter-process communication in parallel programs. But the system calls with which they support locking and memory-sharing are completely different.

When we examine a wider architectural spectrum, it's not surprising that we find not simply incompatible system calls, but incompatible programming models. The S/Net's native operating system offers not shared memory with locks but communication channels that resemble distributed Unix pipes. The Intel iPSC hypercube comes equipped not with shared memory or channels but with several varieties of send-message and receive-message calls.

We've noted that parallel machines span a broad spectrum of communication efficiencies and computation-to-communication ratios. In our case, communication on the shared-memory machines is much faster than communication over

the S/Net bus, which is in turn faster than the Intel hypercube. Speed of communication is one important influence on the way parallel programs are written. On every machine, a cut-off point exists beyond which parallel programs are too communication-bound to show speedup — beyond that point, the overhead of communication overwhelms the speedup gains of parallelism. The whereabouts of the cutoff vary from machine to machine, and programmers must be roughly aware of where it lies on the machine they are using. If they aren't, they will quickly find out when they write a parallel program that doesn't speed up (or maybe slows down) as it runs on more processors.

The Linda kernel can't make these differences go away. It is inevitable that the basic Linda operations will be faster where the underlying communication system is faster. Linda programs exist that will show speedup on the Encore but not on the S/Net, and so on. (Almost anything that speeds up on a slow-communication machine like the Cube will speed up on a faster-communication machine too.)

Do these inevitable differences mean that it is unrealistic to provide the same programming environment on all of these machines? Clearly not. Linda makes the differences among machines easier to deal with by providing a uniform language that scales smoothly along the spectrum of possibilities. Linda itself is unbiased with respect to the communication-intensiveness (or equivalently, the computation granularity) of parallel programs. It can be used for coarse-grained parallel applications; but the model will work just as well for fine-grained programs that store an increasingly greater proportion of their data not in conventional local structures, but in tuple space. It's even possible to express programs in which a dependency graph of program statements is stored in tuple space, and a series of general-evaluator processes march down the graph together, executing each enabled statement they encounter. This dependency graph is simply a distributed data structure like any other.

It's also the case, of course, that a significant class of Linda applications — the relatively coarse-grained class — will run well on a broad spectrum of parallel machines. And there is no excuse for *similar* machines not supporting the same programming environment.

Implementing Linda on a broad spectrum of parallel machines. Linda kernels that define the same primitives can have radically different internal structures. These systems fall into two basic classes: shared-memory kernels and network kernels. Network kernels are themselves divided into two groups, a uniform-distribution group and a hash group.

When physically shared memory is available, then clearly that's where we'll put tuple space. This is an obvious implementation strategy, and a shared-memory multiprocessor is a good host for a Linda kernel. It's important to

point out, though, that providing hardware for physically-shared memory is by no means the *same* as providing Linda. A shared memory supported in hardware is useless to programmers unless they are given a way to use it conveniently and safely. To serve as a communication medium, it must be augmented by a signalling mechanism. If it is to be maintained coherently, we need a locking scheme. Unless programmers are willing to accept statically-assigned communication buffers (which may impose synchronization constraints at runtime), we need a mechanism to allocate and assign storage in shared space dynamically. Linda solves these problems, and (of course) adds its powerful tuple semantics.

The shared-memory kernels. On the Encore and the Sequent, the kernel attaches the necessary shared-memory space and lays out tuple-block buffers. It creates the control structures necessary to manage shared free space and to maintain the queues and hash tables on which tuples are stored, together with their associated locks. Linda operations cause a search of the appropriate storage structure, using a locking scheme that has been tuned to allow maximal simultaneous access without excessive lock-manipulation overhead. On `out`, data is copied out of user space into tuple blocks, on `in` or `rd` the reverse.

In the abstract, we might avoid copying tuples into and out of shared space by allocating all tuple data fields in a shared heap and passing pointers that are transparently dereferenced. But such an approach is hard to support in the C world, because of such difficulties as garbage-collecting the shared heap. To get some idea of how much tuple copying costs, we wrote a version of the Linda matrix multiplication program that "cheated" by storing the input matrix in shared memory and passing pointers rather than matrix-columns in tuples. We also wrote a version that was a true Linda version, because the matrices were in Tuple Space, but allowed the workers to cache the rows and columns for later use. Figure 2 compares the performance of the three versions. The differences aren't dramatic [2].

The network kernels. Supporting Linda in the absence of physically shared memory is a problem with no single obvious solution. Conjuring a shared tuple space out of collection of disjoint local memories is not an easy trick. Nonetheless, two kinds of attack suggest themselves. We can use a hash-

[2] A final interesting note pertaining to copying in shared memory multiprocessors comes from a group that has run extensive experiments on the BBN Butterfly — a shared memory machine with which we have no experience. "[A]lthough the Uniform System [a BBN-supplied parallel programming environment] provides the illusion of shared memory, attempts to use it as such do not work well. Uniform System programs that have been optimized invariably block-copy their operands into local memory, do their computation locally, and block-copy out their results... This being the case, it might be wise to optimize later-generation machines for very high bandwidth transfers of large blocks of data rather than single-word reads and writes as in the current Butterfly. We might end up with a computational model similar to that of LINDA [...], with naming and locking subsumed by the operating system and the LINDA **in**, **read** and **out** primitives implemented by very high speed block transfer hardware [Ol86, p.10]."

based solution or a uniform-distribution solution.

The *hash-based* solution calls for tuples to be stored in a distributed hash table, where different hash bins may in general be mapped to different nodes. In a *uniform distribution* scheme, tuples are broadcast by their generating nodes to all nodes within the generating node's pre-determined "out-set", and requests-for-tuples are broadcast to all nodes within an orthogonal "in-set". Each node is informed at system startup which other nodes make up its in-set and which nodes constitute its out-set. For the scheme to work, each in-set must be guaranteed to include at least one member of every out-set. That way, a tuple and a request for that tuple are bound to run into each other somewhere in the network. (Uniform distribution is described in [Gel84].)

The hash scheme is more economical: it doesn't require any broadcasting, and it doesn't require that data (that is, tuples or requests for tuples) be replicated on multiple nodes. But uniform distribution may lead to more evenly distributed network traffic patterns, it may be extensible to larger networks and, particularly when it is supported by hardware that makes broadcast cheap, it can be a highly efficient way to support distributed tuple matching. We describe a hash kernel that runs on the Intel iPSC hypercube. We then discuss the uniform-distribution scheme that runs on the S/Net, and two other uniform-distribution kernels that are designed for the VAX LAN and for the hardware Linda Machine.

Our Intel iPSC consists of 64 Intel-80286-based processor nodes linked by dedicated Ethernet channels into a binary hypercube. We've had difficulties with certain aspects of the architecture and Intel-supplied communication system, and we think these should change irrespective of Linda. But, although we're in some ways dissatisfied with it, our present Cube kernel nonetheless makes a strong basis for future work on next-generation Cubes.

The system centers on a tuple space implemented as a distributed hash table. *lcc*'s tuple analysis stage divides tuples into two categories, those that include a guaranteed search key and those that don't. (Of the ones that don't, some require no runtime matching at all, and the rest don't have a field that is guaranteed to be a search key in every case.) Guaranteed search-key tuples are hashed on the key to some storage node; other tuples are hashed on their "class number", which is assigned during tuple analysis. The hash function covers the entire network; all nodes act both as storage nodes and host nodes.

Matching on the Cube generally takes place on a node where neither the in'ing nor the out'ing process is local. Both `in` and `out`, then, require the packaging and transmission of a tuple. (In the case of `in` or `rd`, this "tuple" is simply the list of arguments to the `in` or `rd`, precisely as for `out` — we refer to such a "negative tuple" as a template). On `out`, a tuple is dispatched to

the storage node dictated by the hash scheme. It may find a matching template waiting for it; if so, it proceeds onward to the matching template's home node. If not, it is installed in a local table. Likewise for **in** and **rd**: arriving templates will either match a waiting tuple or be installed in the local table to wait hopefully for the right tuple to come along.

An **out**, then, requires a message-send. **in** and **rd** each require a message-send and a message-receive, where message-send and -receive are the communication primitives provided by Intel. Since the hashing scheme is essentially a method of randomizing tuple placement, these sends and receives are in general non-local, and may cross the entire Cube. We are beginning research into ways of reducing communication path lengths by intelligently deploying tuple hash bins; *lcc*'s tuple analysis is a strong basis for such intelligent (topology-sensitive) hashing. (In a very interesting independent project, Lucco [Lu86] has also shown that heuristics can be used to reconfigure a distributed hypercube tuple space at runtime in response to measured tuple-traffic patterns.)

We referred to difficulties with Intel's architecture: they involve both system software and hardware. The Linda kernel relies on the low-level message-exchange primitives supplied by Intel, and these have been far slower than they need to be. Architecturally, the iPSC lacks communication co-processors or front ends. The obvious consequence is that each message packet interrupts each host along its route. The resulting overhead is an unnecessary drag on performance that communication co-processors would have eliminated.

Although Cube Linda is slower than we'd like it to be, there are interesting Linda programs that show good performance on the Cube. We discuss performance issues in the next section.

Uniform-distribution network kernels. We described uniform distribution in the abstract; here are three pertinent instances of uniform-distribution schemes. Out-set(n) means "the set of nodes to which tuples generated on node n are broadcast. In-set(n) means "the set of nodes to which templates generated on n are broadcast."

1. For all n, out-set(n) = the entire network; in-set(n) = n.
2. For all n, out-set(n) = n; in-set(n) = the entire network.
3. Consider a network arranged in a square jxj grid. out-set(n) = the j nodes in n's row; in-set(n) = the j nodes in n's column.

Note that in all three cases, each node's in-set contains exactly one member of each other node's out-set.

The S/Net kernel follows the first scheme. This kernel has been described extensively elsewhere [CG85], so we omit further discussion here. The VAX-LAN

kernel uses the second uniform-distribution scheme — the inverse of the S/Net scheme. This technique is better suited to networks that (like the Ethernet and unlike the S/Net) do not support reliable broadcast. In this scheme, tuples are stored on their node of origin. Templates are broadcast, but tuples, once they've found a matching template, are shipped off to the template-generating node using a reliable point-to-point protocol. Because broadcast is unreliable, templates may not reach every node on a given broadcast — but the template generating-node rebroadcasts its template every x ticks until a matching tuple arrives. Template-holding nodes throw out templates that are more than x ticks old. Note that, in general, it's easier to manage a protocol in which the network holds an uncertain number of templates than one in which it holds an uncertain number of tuples.

Finally, the Linda Machine now under construction at AT&T Bell Labs uses the third uniformly-distributed scheme. It's configured as a grid of buses, with nodes at the bus intersections. Each node has ports to two buses. Tuples are broadcast on one and templates on the other. A matching tuple-template pair are guaranteed to meet somewhere in the network. This work is described in [AhCGK87].]

Performance. A primitive tuple exchange that takes roughly 190μsec on the Encore and 180μsec on the Sequent (both times are for the NS32032 version of these machines) requires about 1.4ms on the S/Net, and, in the fastest measurable case, about 5.3ms on the Intel cube. (Of these 5.3ms, only 1.5ms is actually Linda overhead; the rest is the iPSC's communication time)

Given these significant differences, it's interesting that it is not hard to write Linda programs that show very similar performance profiles over all of four machines.

Figure 3 gives absolute running times (*not* speedup: the number on the ordinate gives execution time for increasingly parallel versions) for the multiplication of 300x300 matrices on all four machines. The algorithm we used is essentially the one described in [CG85]: multiple identical worker processes are responsible for reading (via `rd`) the rows and columns of input matrices stored in tuple space, computing their inner products and outing the result back to tuple space. Computation tasks are assigned to workers dynamically. Workers check a floating "next-task" tuple, perform the task it indicates and update it appropriately. A single task consists of computing five rows of the result matrix.

(Note, for concreteness, that the rows of a matrix named `MAT1` can be stored in tuples this way:

```
("MAT1", 1, first-row-of-MAT1)
("MAT1", 2, second-row-of-MAT1)
```

("MAT1", 3, *third-row-of-MAT1*)

and so on; tuple fields may be composite values like vectors or **structs**. To read the *ith* row of **MAT1**, an appropriate statement is

```
rd(''MAT1'', i, ? newrow),
```

where **newrow** is a local variable of the appropriate vector type. The prepended "?" means that **newrow** is used here as a formal parameter, to which the row in question will be assigned.)

These four programs are not source-identical, but they are nearly so. They differ insofar as (1) *lpp* hasn't been ported to the S/Net, so the S/Net programmer must invoke the appropriate kernel calls directly; (2) **eval** exists on the Encore and Sequent, but on the others, programmers rely on the native operating system for process forking.

What's important in figure 3 is not, of course, the absolute running times on the various machines, but the fact that the shape of the Linda curves is so similar over the four cases. On all four machines, two Linda workers (a total of three processes, as before) were sufficient to beat a sequential version of the algorithm, and performance improved in essentially linear fashion from there.

Figure 4 shows a more limited comparison: it plots Encore, Sequent and S/Net versions of an LU decomposition routine that is similar to the Linpack test. Again, all three curves are similar in shape. The Encore's performance trails off beyond 9 workers or so: this is a small matrix (note that the scale of the ordinate is about two orders of magnitude finer than in the previous example); performance has been observed to be much better on larger problems.

Figure 5 shows a comparison of Encore, Sequent, and iPSC versions of a different kind of routine, a simple parallel database search. The problem is as follows: given a DNA sequence and a database of sequences with which to compare it, find the sequence in the database that is closest to the original, where "closest" is determined by a fairly complex string comparison algorithm that attempts to capture geneticists' understanding of relatedness across sequences. The procedure we used is simple: worker processes are set up; the target sequence is **out**ed and each worker **rd**'s it; the sequences found in the database are **out**ed, and workers repeatedly grab a sequence, compare it with the target and repeat until the database is empty. Each worker remembers its best match so far and, when the search is complete, **out**s this datum; a master process picks up these final tuples and reports the result. Figure 5 shows performance on a search involving a small comparison set of 256 synthetic 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: (1) outs an initial batch of sequence tuples (2) waits until most of the batch has been acknowledged by result tuples, whereupon it outs another batch of tuples. The process continues until all sequences in the database have been outed.

Figure 6 shows a final comparison: a Linda program developed by our colleague Henri Bal of the Vrije Universiteit in Amsterdam was executed on the Tadpole, a custom-designed shared-memory multi-processor built by Prof. Andrew Tanenbaum's group. We ran the same program (modulo a trivial alteration to accomodate the lack of **eval** in Tadpole Linda) on the Encore Multimax.

The program solves the travelling salesman problem using the parallel branch-and-bound approach described in [BTR87]. The problem state is stored in a search tree: leaf nodes represent complete tours; interior nodes represent partial tours. Interior nodes have one descendent for every possible continuation of the partial path they represent. The tree is pruned at an intermediate node when that node's partial path can't lead to a shorter tour than the shortest one known so far.

In Bal's Linda version, a master process explores the tree down to a certain level, then parcels out to worker processes the exploration of the tree below this level. Each worker repeatedly accepts an internal node and systematically generates all relevant full routes starting with the nodes's partial route. The bound (the best solution so far) is stored in a tuple which workers consult and update when they start a new task or discover an improved solution. (It would have been better in the abstract for workers to consult this tuple each time they extend a path by one hop, but the one-hope extension of a path is a computationally trivial step, and consulting the bound-tuple each time it occurs leads to excessively-frequent ining and outing, with consequent inefficiency.)

In the master-worker programs discussed above, tasks are kept in an unordered bag. In Bal's travelling salesman program, on the other hand, it's important that tasks be performed in a given order: heuristics are available to optimize the chance that a good solution will be found early in the search. Tasks are accordingly kept on the kind of distributed task queue discussed in [CGL86]. The master process numbers each task tuple by including an index field as one tuple element. Before withdrawing a new task, workers consult and update a separate index tuple; if the new value of the index is n, they ask for the nth task in the queue, by executing an

inp("task", n**, ...)** statement.

6 Conclusions

The programs whose performance we've discussed were not chosen to span or characterize the space of interesting parallel applications, and we don't claim that they do. Our examples do represent the kind of problem-solving approach that seems natural in Linda (they involve replicated workers and distributed data structures), and they solve problems that are easily grasped using algorithms that are likewise simple to explain. Although they are simple, the examples are realistic: we resisted the impulse to run recursive quicksorts or parallel fibonacci generators. But of course we are systems developers and not applications programmers, and our goal is to establish not that Linda is tried-and-true, but that it is ready to leave the sheltered systems world and be subjected on an experimental basis to the kind of demands that applications programmers make.

Linda is sometimes regarded as a relatively conservative approach to parallel programming. This isn't entirely true. As envisioned (though not yet realized), complete Linda systems do away with the standard distinctions between the program, the data and the file system; program and data live together in the same tuple space, and programs resolve to data when they're done — process tuples become data tuples. Tuple spaces are objects with independent existences. They may be stored in a file system; they may be reactivated or operated-on any number of times. They may even overflow the boundaries of their multicomputer of origin to take up residence on another machine, or on many other machines simultaneously.

Linda *is* conservative in its use of established base languages and in our insistence that it run well on a variety of real machines. We regard it as no longer legitimate to treat practical development tools for parallelism as a distant goal. Programmers now deal with real multi-computers every day.

The results we've presented deal with small parallel machines only. This 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 upward to roughly a thousand nodes. For the forseeable future, a thousand powerful nodes will be plenty for our purposes.

In discussing related work, programming languages like Concurrent Pascal [Br75] and Modula [Wir77] are a good starting point. These systems were based on strong cooperation between compiler and runtime kernel — but concurrent systems programming, not parallelism, was the problem domain in both cases. Birrell and Nelson's work on an RPC kernel [BN84] bears some important similarities to ours. In their system, interface-specification modules allow remote calls to be typechecked, and remote procedures are invoked via stubs that are

generated automatically at link time. Close cooperation between the linker, the stub-generating program and the runtime communication system brought demonstrable gains in usability and efficiency. RPC is, of course, a distributed-programming protocol; it wasn't designed for parallel applications (and in our view is unsuited to them: its synchronous character tends to suppress rather than to encourage concurrency). But efficiency was nonetheless a major goal in the RPC work, one that was evidently attained.

Many high-level parallel languages for parallel applications programming been proposed, but not many have been implemented. Occam [M83] and Ada [DoD] are prominent implemented systems; Linda's approach to language designs is fundamentally different. Multilisp [Hal86] and Concurrent Prolog [Shap86] represent a class of interesting languages that don't yet exist in efficient parallel implementations (so far as we know), but are under active study with a view towards efficient implementation. The several languages that have been developed for the Connection Machine [Hil85] represent an approach to parallelism that is, again, basically different from Linda's.

Research on portable parallel languages has been sparse. One exception involves Fortran extensions designed for numerical applications; Jordan's work on the "Force" [Jor86] is a good example. His system involves Fortran extensions implemented as macros, and it has been ported to several parallel machines, including the Denelcor HEP and the Flex/32. Parallelism in this model centers around parallel DO loops whose iterations may be performed simultaneously. This model is powerful and useful over a wide range of numerical programs. Linda is clearly more general and flexible — there are many kinds of parallel structures that don't fit neatly (or don't fit at all) into parallel DO loops. Jordan's system, on the other hand, gives maximum convenience and optimal performance within the domain for which it was designed.

Many parallel algorithms are known and more are in development. (We've concentrated on numerical examples in this paper, but applications in simulator building and graph algorithms are as interesting to us or moreso; parallel database manipulation is another area we've begun to explore.) Many parallel machines are available, and many more will be soon. But the fate of the whole effort will ultimately be decided, we believe, by the extent to which working programmers can put the algorithms and the machines together. Linda, we continue to believe, brings togetherness a step closer.

References

[AhCG86] S. Ahuja, N. Carriero and D. Gelernter, "Linda and Friends," *IEEE Computer* (Aug. 1986).

[AhCGK87] S. Ahuja, N. Carriero, D. Gelernter and V. Krishnaswamy, "The Linda Machine," Yale Univ. Dept. Comp. Sci. tech. rep. (Feb., 1987).

[BN84] A.D. Birrell and B.J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comp. Sys.* (Feb. 1984):39-59.

[Br75] P. Brinch Hansen, "The programming language Concurent Pascal. *IEEE Trans. Sft. Eng.,* SE-1,2(1975):199-206.

[BRT87] H.E. Bal, R. van Renesse and A.S. Tanenbaum, "Implementing Distributed Algorithms Using Remote Procedure Calls," in Proc. 1987 National Computer Conference, Chicago (June 1987): 499-506.

[C87] N. Carriero, *Implementing tuple space machines.* Doctoral Diss., Yale Univ., 1987.

[DoD] US. Dept. of Defense, *Reference Manual for the Ada Programming Language.* ACM AdaTEC (July 1982).

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

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

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

[Gel84] D. Gelernter, "Dynamic global name spaces on network computers," in *Proc. Int. Conf. Parallel Processing,* (Aug. 1984).

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

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

[Hal86] R. Halstead, "Multilisp: a language for concurrent symbolic computation." *ACM Trans. Prog. Lang. Sys.* (Oct, 1986).

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

[M83] M.D. May, "Occam." *ACM SIGPLAN Notices,* 18-4(1983):69-79.

[MF86] D. Mundie and D. Fischer, "Parallel processing in Ada." *IEEE Computer,* (Aug. 1986): 20-25.

[Lu86] S. Lucco, "A heuristic Linda kernel for hypercube multiprocessors," in *Proc. SIAM Conf. on Hypercube Multiprocessors,* (Sept. 1986).

[Ol86] T.J. Olson, "Finding lines with the Hough Transform on the BBN Butterfly parallel processor," Univ. Rochester, Dept. Comp. Sci. Butterfly Project Report 10, Sept. 1986.

[Shap86] E. Shapiro, "Concurrent Prolog." *IEEE Computer,* (Aug. 1986): 44-59.

[Wir77] N. Wirth, "Modula: A language for modular multiprogramming." *Softw. Pract. and Exp,* 7(1977):3-35.

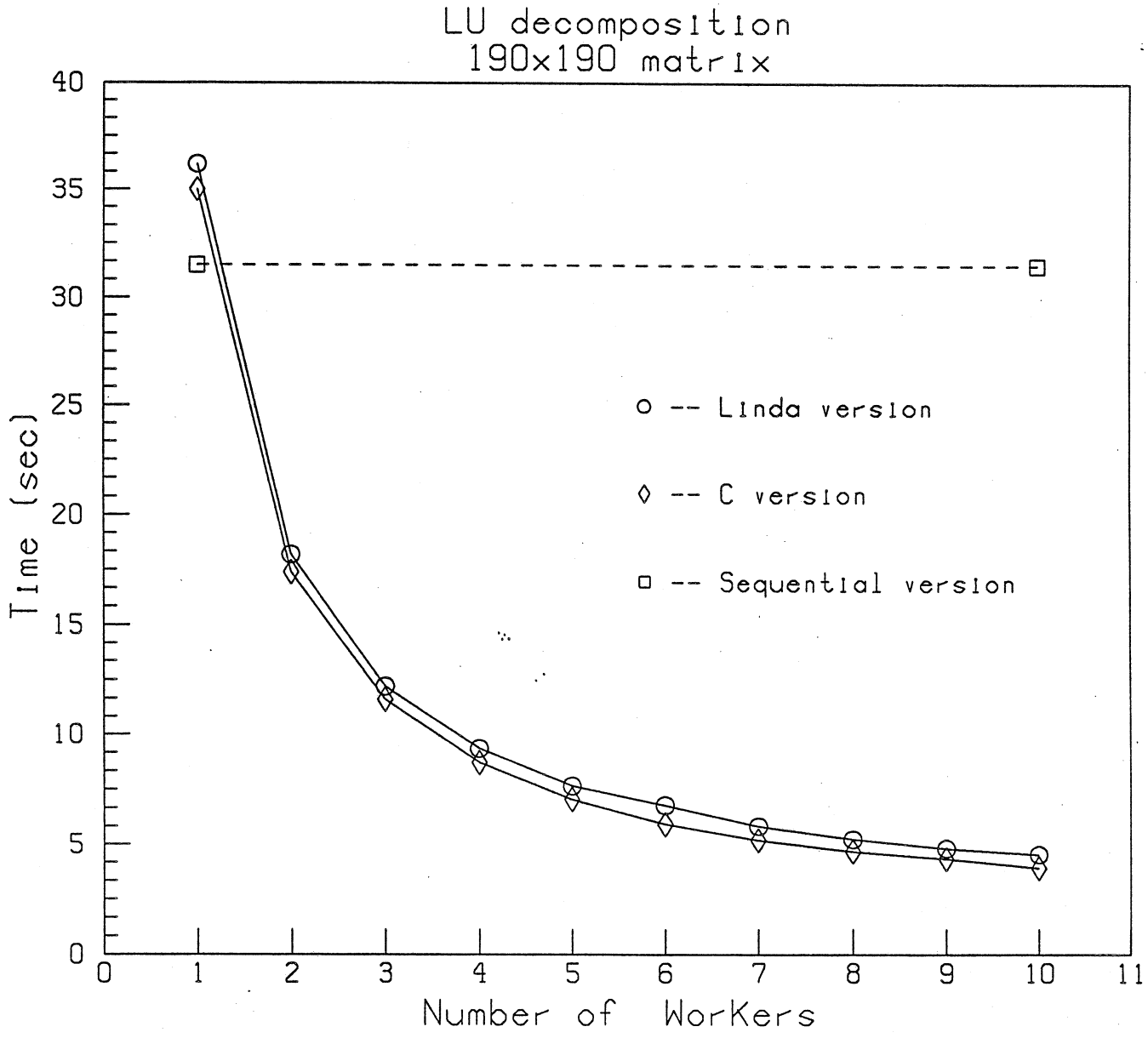

Figure 1: Comparision of Linda and non-Linda versions of LU decomposition running on the Encore Multimax.

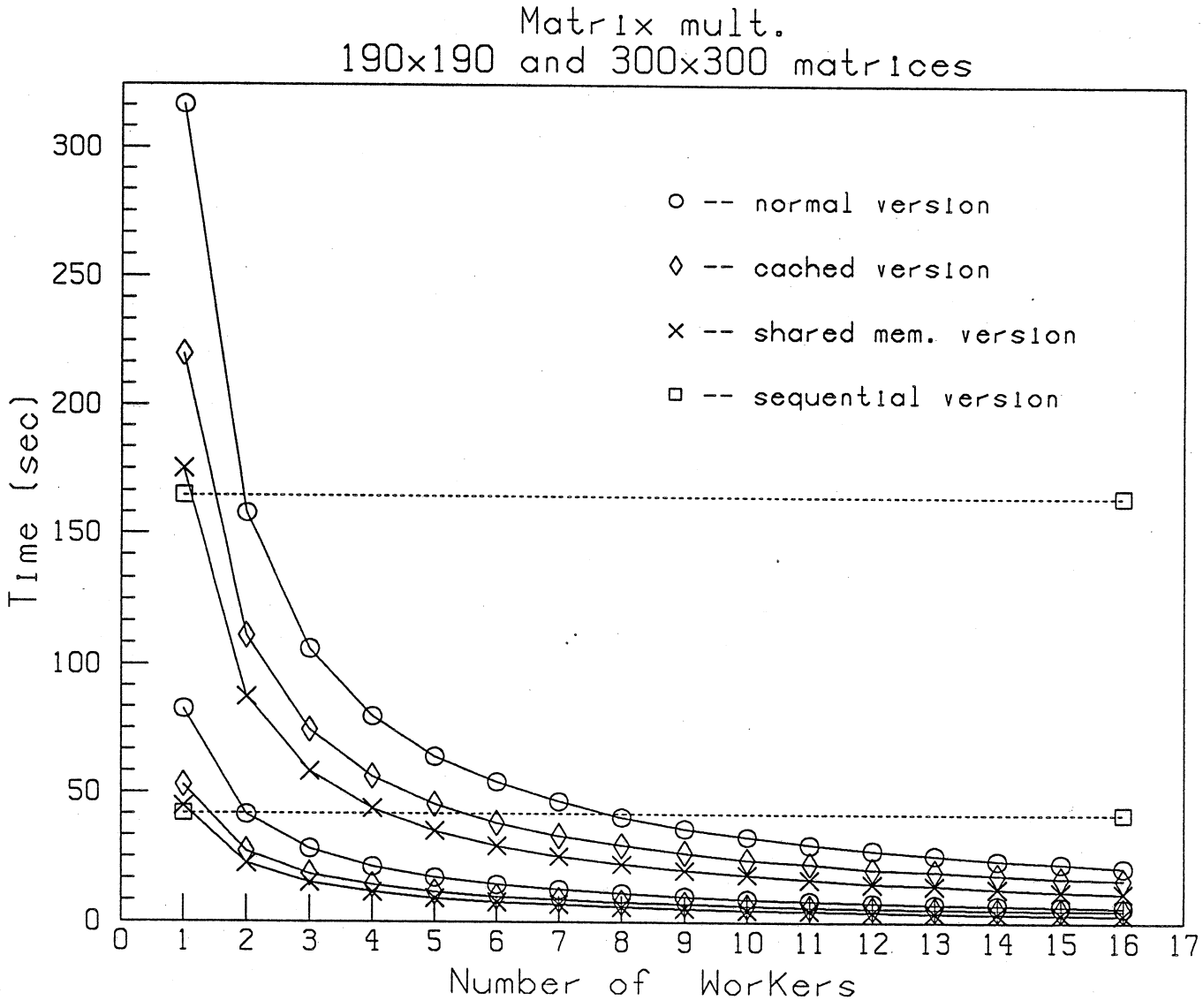

Figure 2: Comparision of different versions of matrix multiply running on the Encore Multimax.

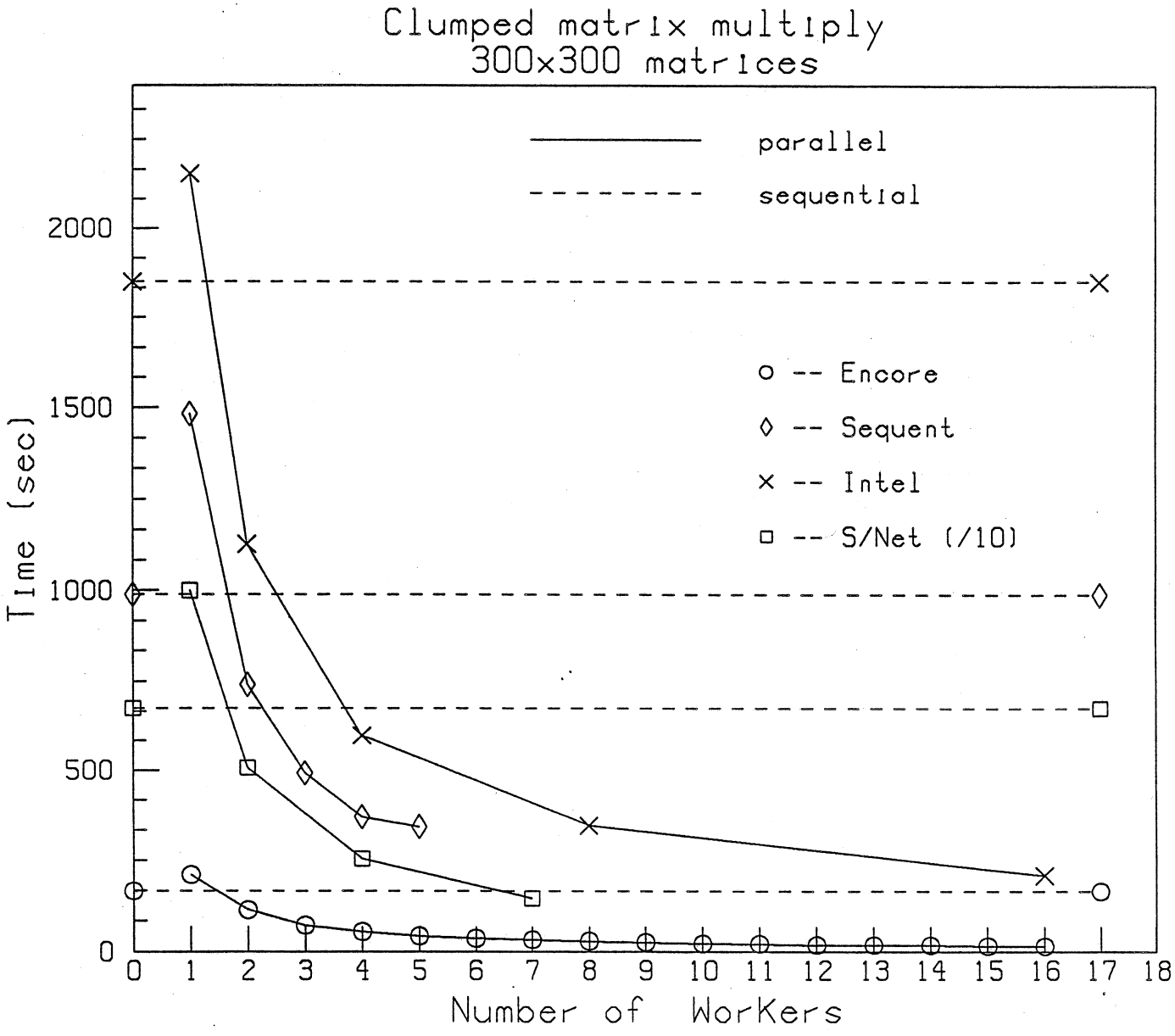

Figure 3: Comparison of the clumped matrix multiply algorithm on four different multiprocessors. The matrices are of dimension 300 x 300. The elements are floats.

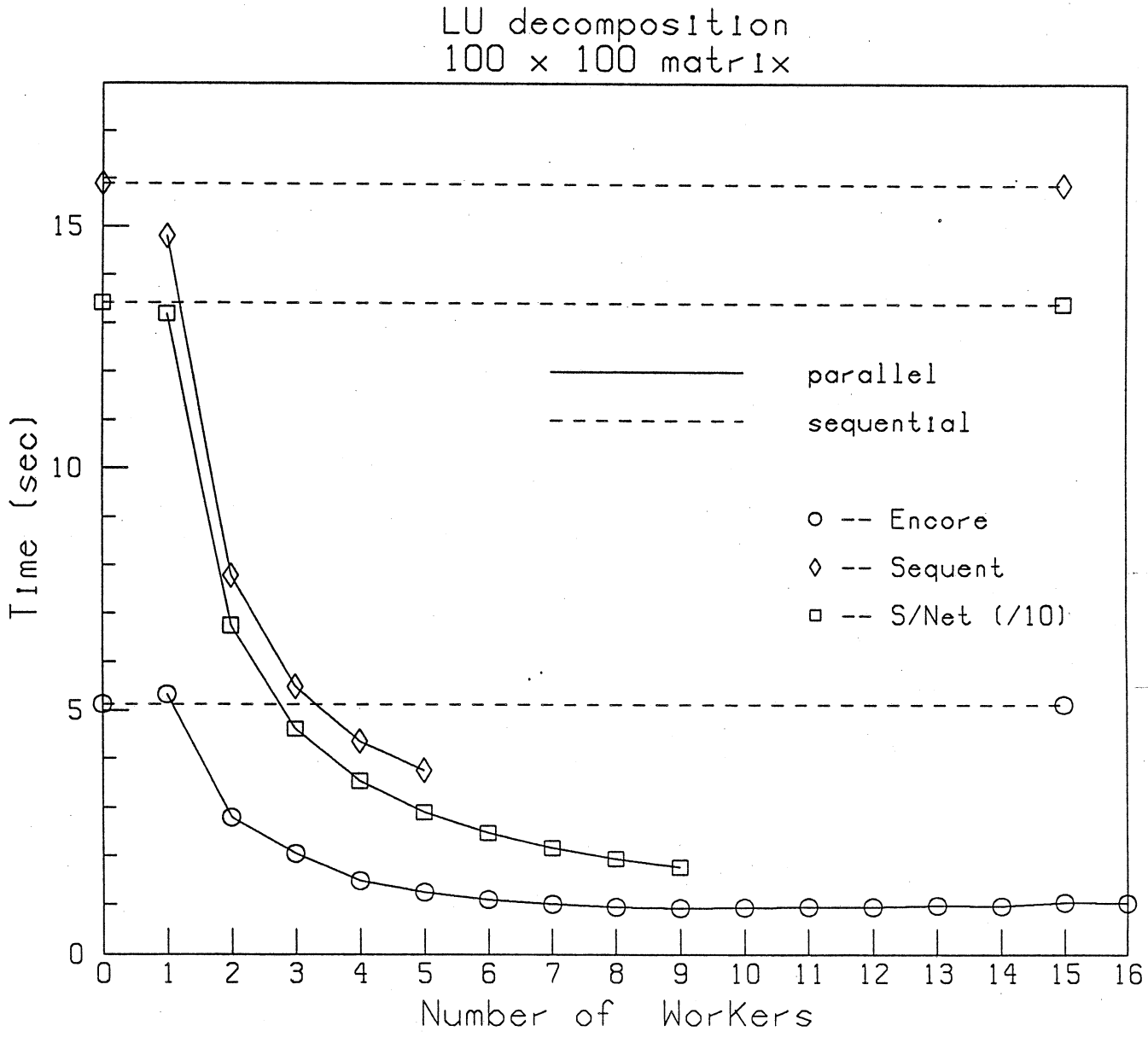

Figure 4: Comparision of LU decomposition on three different multiprocessors. The matrix is 100 x 100, and the elements are doubles.

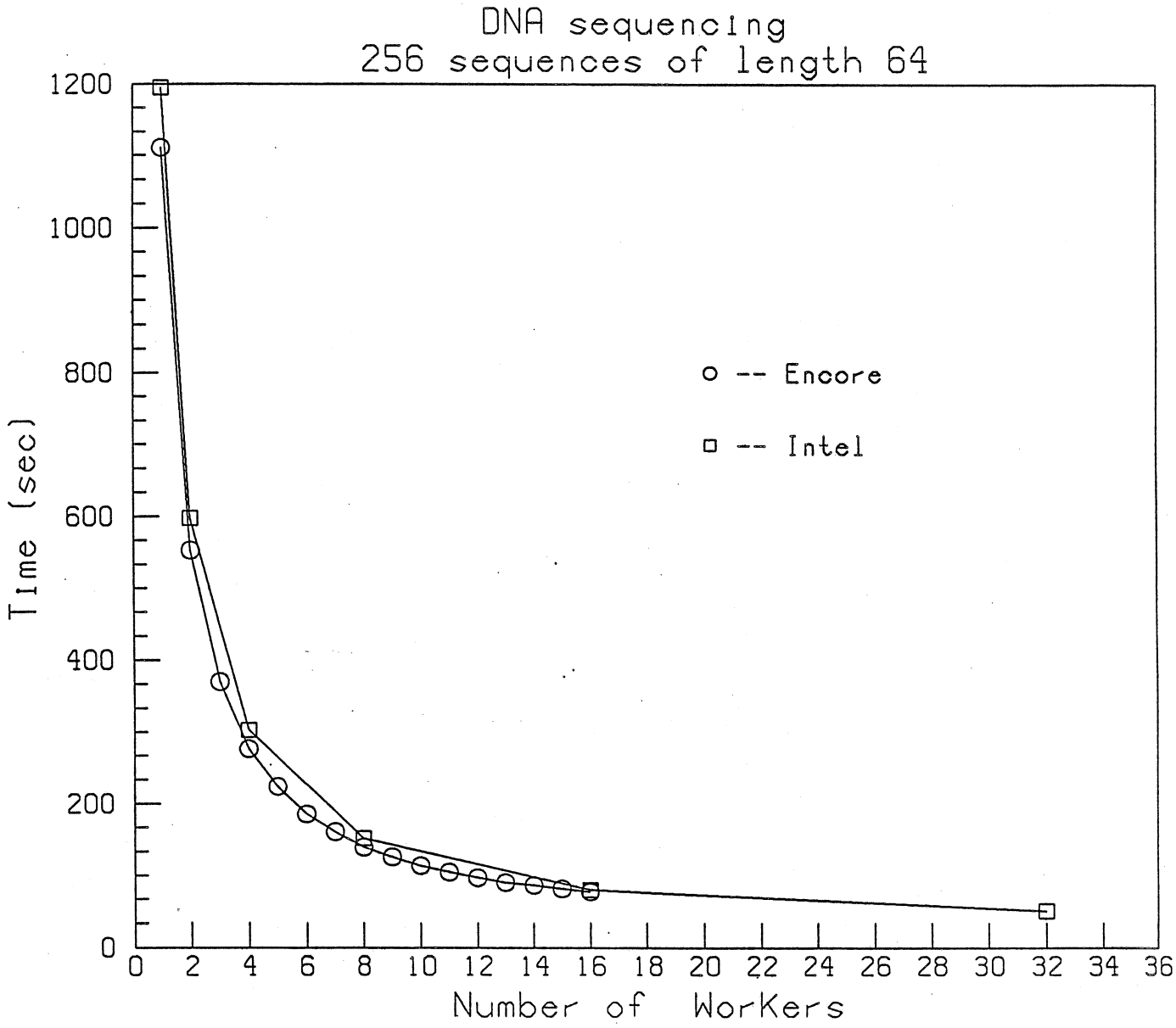

Figure 5: Comparison of DNA sequencing on two different multiprocessors. The sequences were 64 bases long, and 256 sequences were compared against the target.

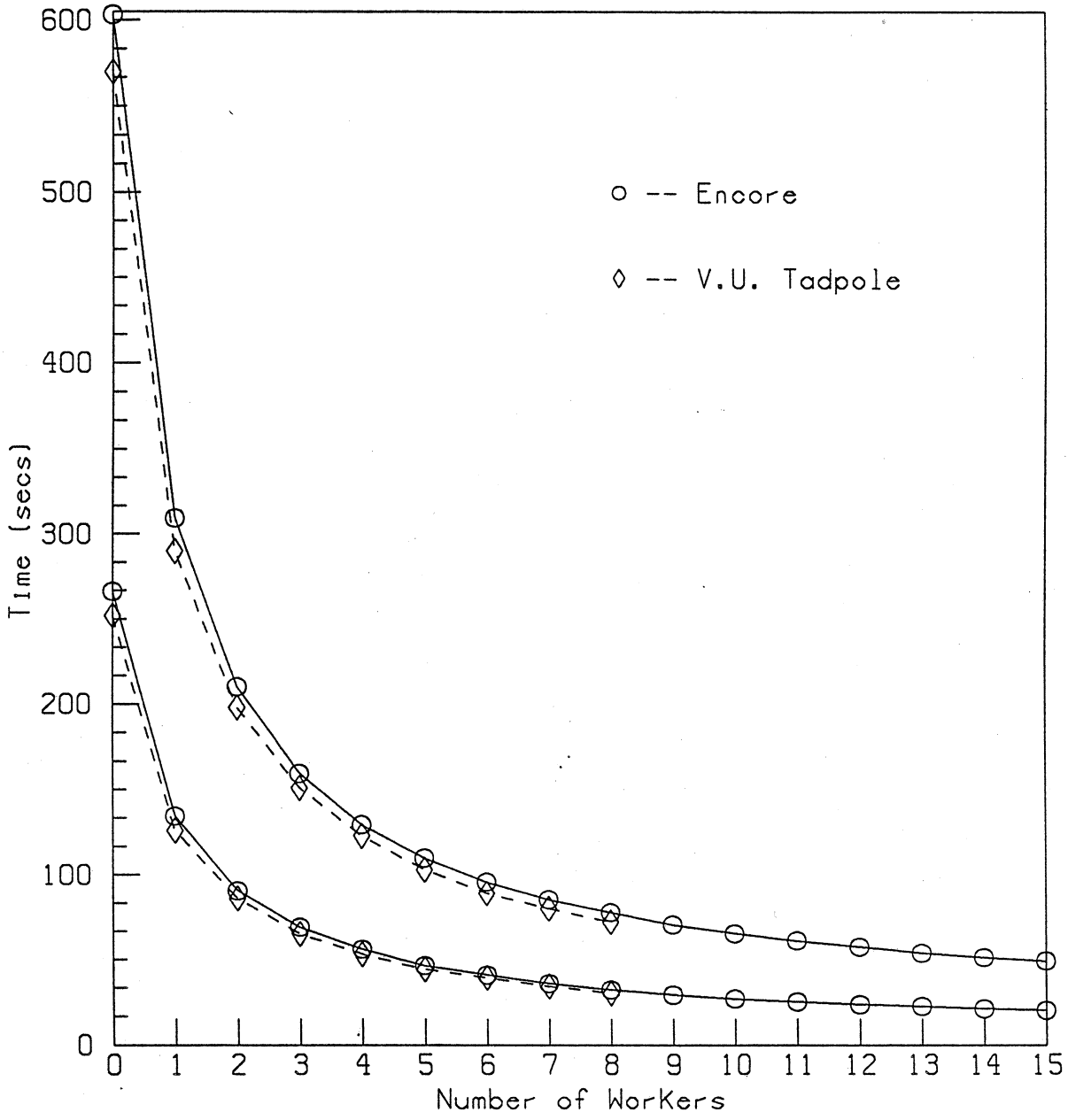

Figure 6: A comparision of Travelling Salesmen on two different multiprocessors. The program is due to Henri Bal of the Vrije Universiteit in Amsterdam. Two different benchmarks are shown; in each case there are 13 cities on the map.