Experience with Linda

Robert Bjornson*, Nicholas Carriero, David Gelernter**, Tim Mattson*, David Kaminsky** and Andrew Sherman***

**Scientific Computing Associates*
New Haven, Connecticut

***Yale University*
Department of Computer Science
New Haven, Connecticut

Abstract. Claims about Linda's effectiveness as a parallel programming tool were once based on abstract arguments and small experiments; they are now based on wide-ranging experience with production applications. We discuss the implications, and the new direction of ongoing Linda research.

1 Introduction

For most of the last decade, parallel programming has been a far more important topic for software researchers than it has for production programmers. But over the last year or so, this situation has changed dramatically. Parallelism sees active, production use at an increasing number of sites; even more significant, parallel programming is now a high-priority topic for investigation at a wide spectrum of U.S. companies—including aerospace, pharmaceutical, automobile, financial services, oil production and even insurance companies. Relatively few of these sites make use of parallelism today, but a surprisingly large number are actively researching the possibility.

The most important spurs to this surge of interest in parallelism have been (*a*) the emergence of local area networks as huge power reserves and good hosts for parallel applications, and (*b*) the considerable clarification that has occurred in the historically turbid world of parallel programming systems: a small number of solid contenders has emerged from a large pack. We'll discuss both of these factors below.

Linda[1] is a high-level coordination language (a language with operators to support process creation and inter-process communication [CG91]). In principle, Linda can be mated with any "computation language" (a conventional sequential programming language) to produce a complete parallel programming environment. In practice, the applications discussed below all use C-Linda—although in many cases, the bulk of the application consists of Fortran code, invoked from C. A prototype Fortran-Linda exists, and SCA will distribute a production version in the near future. Many other research groups have produced experimental matings of Linda and a large spectrum of other computing languages, including Scheme, Prolog, Modula-2, ML, SETL and others.

Here is the main result to be discussed in this article: a wide variety of production parallel applications can be coded in Linda and executed efficiently on shared- and distributed-memory parallel machines, and on local area networks. This "wide variety" includes applications in computational fluid dynamics, molecular dynamics, seismic simulation, financial portfolio modeling, graphics, realtime data fusion, database search, and various numerical scientific codes. All Linda programs can be ported transparently between all of these platforms. In some cases, they require post-portum tuning for good efficiency (obviously, Linda can't abolish basic hardware distinctions among the platforms on which it runs, and the distinction between fast and slow communication systems is important to parallel program performance); this kind of tuning is usually accomplished by changing a single value in the program (namely, the value that determines the granularity of a task). But Linda can and does guarantee that a program developed on any of these platforms will run on all of them, and that an identical coordination language environment is available on all of them.

Bully for Linda, but do these results have broader implications for software systems generally? Clearly they do.

Some researchers have contended that explicitly-parallel programming was simply too difficult for non-experts to master. Working programmers would not be able to harness it effectively; they would need to rely on parallelizing compilers or on "implicitly parallel" languages (functional languages, for example). We (and many others) have long believed that this contention was false. We now *know* that it's false. Some of the production Linda applications referred to above were developed with expert assistance; many were not.

Many researchers have contended that Linda was too high-level to be supported efficiently. They believed that only low-level, nose-to-the-dirt programming models like message passing (see below) would deliver good performance. This contention was plausible in the abstract, but we have now demonstrated that it is wrong. We believe that the whole field of higher-level coordination languages gains as a result. Research into these systems is no mere theoretical

[1]a registered trademark of Scientific Computing Associates (SCA).

exercise. High-level coordination models can and have been turned into efficient tools for production programming.

Many researchers continue to treat *portability* of parallel applications as a long-term research goal, not yet within reach. In fact, portability across a broad range of asynchronous platforms, including all species of commercial environments, is a reality today.

These results aren't offered in a spirit of complacency. Linda implementations continue to improve in efficiency, and considerable further improvement is desirable and possible. Linda programming tools and methods need further improvement. The Linda model itself is changing and gaining power as first-class multiple tuple spaces are more widely understood and supported. There's more work do be done. But on the other hand, a great deal has been accomplished, and further progress is jeopardized unless we can succeed in taking stock of how far we've come.

In this paper, we'll discuss Linda in the context of alternative or competing systems (section 2), performance results on a number of significant applications (3), and current research directions (4).

2 Linda in Context (Revisited)

We discussed Linda in context in a paper several years ago [CG89], but the context has changed somewhat in the meantime. We briefly take up the same topic here. Two separate, largely independent contexts have emerged: parallel programming systems under study by computer science researchers, and parallel programming systems that are used in practice by applications developers. The first bunch of systems are designed to pose problems or offer opportunities for systems research. The systems that appear in the second bunch may be arbitrarily boring in technical terms, but they must be robust and efficient enough to be valuable in real programming.

There now exists a *bona fide* research context of which Linda is a part—Linda can be understood as one member of this group, and not merely (as had generally been the case in the past) as an interloper. We discuss this particular group in the next section. In the section following, dealing with the "practical" as opposed to the "research" context, we examine the distinctions between Linda and other contenders in this class.

In Theory: High-Level Coordination Languages

Two significant classes have emerged since 1989: the general class of high-level coordination languages, and more specifically the class of Linda-like languages.

We've defined "coordination language" [Gel89, CG89b, CG90, CG91] as a language for expressing process creation and inter-process communication. Such languages may be mated with "computation languages" to produce complete programming environments. We've argued that coordination and computation ought to be treated as the two orthogonal base axes of software building, with each element supported by a compiler. (Coordination, traditionally the province of the operating system via runtime library calls, is ordinarily a classical second-class citizen in the general scheme of things.)

The "high-level" part of the definition is somewhat slippery, as usual. But two requirements are important. The models in this category must reflect a particular approach to building software, *not* a particular machine architecture. In this sense they must be top-down and not bottom-up models, and portability over a fairly wide range of machines is implicit, at least in principle. Second, high-level coordination languages models should come equipped with a *program-building methodology.* They shouldn't represent merely a set of operations; there ought to be some well-defined method for *using* these operations to develop parallel programs.

An INRIA-sponsored workshop on "Research Directions in High-Level Parallel Programming Languages" [LeM91] in June 1991 represents one of the field's first attempt to come to grips with this new area. The workshop covered Linda, Unity [CM88], Gamma [BCLM88], and a number of projects dealing with more than one of these systems. The workshop highlighted some conceptual similarities between Linda and Unity; both are top-down parallel programming models, and both are supported by complete program-development methodologies (Unity's presented in [CM88], Linda's in [CG90]). Unity's model is formal and supported by techniques for mathematical reasoning about programs; Linda's model is informal, and supported by program-building tools that guide the development of Linda applications [ACG90]. The differences in approach are clear, but there are clear similarities as well. There are analogous resemblances between Linda and Gamma.

The high-level coordination language group clearly includes a number of other systems as well; Strand [FT89], a coordination language based on the concurrent logic programming model of Parlog [R88], is one likely nominee.

Within the general high-level coordination language field, research on Linda-like languages is thriving. At a recent workshop on Linda-like languages sponsored by the Parallel Computing Center at the University of Edinburgh, papers were presented about the formal semantics of Linda, Linda embeddings in a variety of languages, and Linda implementations in a variety of settings [Wil91]. Other noteworthy recent Linda projects include work on multiple first-class tuple spaces and fine-grained programming in Linda [J91, J91b], persistent Linda [AS91], reliability ([Xu88, KW90, BS91]), Linda and Prolog [e.g. ACD90, HM91], Linda strategies for scientific computing [PS90], operating systems [e.g.

Lel90] and many others. Today, our own work at Yale is merely one piece in an increasingly productive and multi-faceted research effort.

In Practice: Linda vs. Message Passing

It's our observation that the three most widely-used approaches to building real parallel applications at the moment are message-passing systems of various sorts, loop-parallel Fortrans, and Linda. This statements needs qualification, though, in terms of the three major environments for parallelism: distributed- and shared-memory parallel computers, and local areas networks.

We'd guess that on distributed-memory parallel computers, message-passing systems are the most popular approach right now, by a fairly wide margin; on shared-memory parallel machines, loop-parallel Fortrans are most popular, by a comparable margin.

Local area networks have always been capable in principle of serving as hosts for parallel applications. But they've emerged only recently as widely-used parallel platforms—only recently has the aggregate power of interconnected workstations emerged as a formidable computing resource. Today, a modest-sized LAN of fast modern workstations easily challenges a supercomputer on certain applications. As the power of the typical "fast modern workstation" continues to increase, LANs grow ever more formidable. They are in effect the newest platform for parallelism, but by far the most widely available; which makes them (we believe) the most important platform for the immediate future. (Beyond the near term, high-speed LANs connecting parallel workstations become the most important environment for parallelism.) For parallel programming on local area networks, message-passing and Linda appear to be neck-and-neck.

In this section, we compare Linda briefly to loop-parallel Fortrans and to message passing. Message passing is the more significant competitor, as we'll explain.

Loop-parallel Fortrans are important and widely used, but they are designed for a fairly narrow range of parallel programming methods, and a fairly narrow range of platforms. They don't seem appropriate for loosely-coupled environments like local area networks; nor are they suitable for expressing any kind of coordination but *parallelism* specifically. In situations where concurrency is important for reasons *besides* running a compute-intensive application fast—to accomodate heterogeneity of machines or languages, to manage physically-dispersed resources (as in distributed operating system or databases) and so on—these systems are out of their element. Clearly they *are* powerful tools; clearly they are *not* general-purpose coordination tools, in the sense of message passing or of Linda.

Message passing, like Linda, is appropriate for fine-, medium- and coarsegrained applications and architectures. It accommodates a wide range of parallel program structure, and it can be used to build (say) a distributed operating system as well as a parallel equation solver. But message passing is the machine language of ensemble computers: in order to move bits from one computer to another over a network, you must (in fact) send a message. And as the machine language of parallelism, message passing is the problem, not the solution. Some of the fancier message-passing systems on the market today are in effect assemblers for a message-passing machine language; assembler languages represent a nice step forward, but hardly the culmination of a serious effort at programming language design.

Proponents of message-passing don't deny (in our experience, at any rate) that Linda is the higher-level and more expressive programming model. The argument here is simple: to achieve message-passing-style point-to-point communication in Linda is trivial. To achieve a Linda-style shared associative object memory (a "tuple space") in message-passing systems, on the other hand, requires that you build (in effect) a Linda emulator, which is a lot more complicated.

Linda's tuple space is important for a number of reasons. Here is a whirlwind tour. (The topic is discussed at great length elsewhere; for example, in [CG89].) The shared associative object memory is important in supporting—

(1) Dynamic tasking. Dynamic tasking is a robust, simple way to achieve good load balance and transparent program structure. Simply toss a bunch of tasks into the shared object memory (in an unordered bag, an ordered queue or any other appropriate arrangement—we can build arbitrary data structures in tuple space); idle workers withdraw them as needed.

(2) Distributed data structures. As noted, tuple space can be used to store arbitrary shared data objects and structures. Data objects in tuple space can be shared safely by concurrent processes, because they're immutable; to change an object, a process must withdraw and then re-insert it. Thus, tuple space provides universal direct access to shared data objects and structures, in shared- *or* distributed-memory environments.

(3) Uncoupled programming. Because tuple space holds *persistent objects* (a *message* on the other hand is a *transient event*), Linda supports inter-process communication across time as well as across space. Processes don't care when the information they produce will be used; they don't care when the information they consume was produced. Two processes whose lifetimes are in fact wholly disjoint may communicate via tuple space.

(4) Anonymous communication. Processes deal only with tuple space, not with each other. Thus, no process has any direct dependence on the number or identity of the other processes in the ensemble, and it's trivial to build application in which the number or job description of processes in the ensemble

changes transparently while the program runs.

(5) A dynamically-varying processor pool. The properties listed above make it easy to build Linda applications that may gain and lose processes transparently at runtime. This is a crucial property in supporting what we call "Piranha parallelism"—a network computing model in which workstation join an ongoing parallel computation when they become available, and leave when their users need them.

What message-passing proponents *do* claim (true to their heritage as machine-language defenders) is not that message passing is more expressive but that it is more *efficient* than Linda. Granted, machine languages are usually more efficient than programming languages. But in a systematic study that compares the efficiency of Linda and of message passing on distributed-memory machines [B91], the cost of using Linda versus native message-passing ranges, on realistic problems, from small to negligible. We'll discuss some of this data in the following section. The advantages of using a higher-level programming model that is marginally less efficient than a low-level alternative are so well known that it would be ludicrous to repeat them.

3 Experience with Real Applications

Our intent here isn't to describe the structure of the applications in detail. We've discussed methods for constructing Linda programs at length elsewhere [CG90]. Our goal is rather to convey some feel for the range of production applications coded using Linda, and for their performance. Our emphasis is performance on local area networks, but we will discuss other environments as well.

Numerical routines. Figure 1 shows speedups on LAN-connected Sun Sparcstations for two programs, block matrix multiply and LU factorization with partial pivoting. The matrix problem is a multiplication of two 600×600 real single-precision matrices; the factorization is an 800 row problem. In this and subsequent cases, we show speedup relative to a C (not C-Linda) program running on a single node. These programs and all subsequent ones are master-worker applications; the program structures involved in these particular cases are discussed in [CG88].

Figures 2 and 3 present comparison data for Linda versus message passing on a distributed-memory machine (the Intel iPSC/2) for matrix multiplication and LU decomposition. Matrix multiplication in this case uses the Dekel algorithm [DNS81], which was developed for dense matrix multiplication on two-dimensional mesh architectures. The problem in figure 2 is a multiplication of two 240×240 matrices. We wrote the message-passing version—which

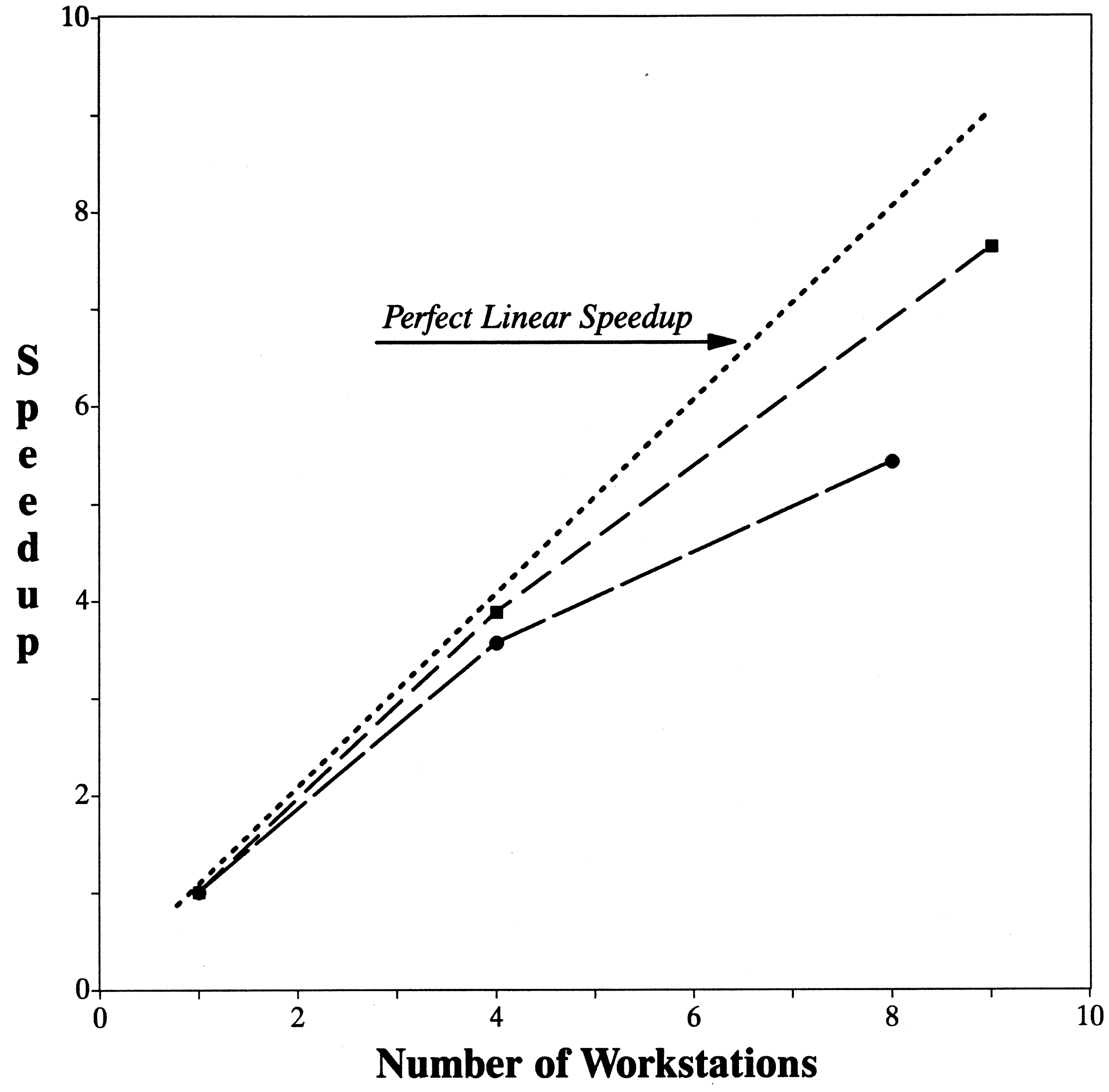


Figure 1: LAN speedups: block matrix multiply (■) and LU decomposition (●)

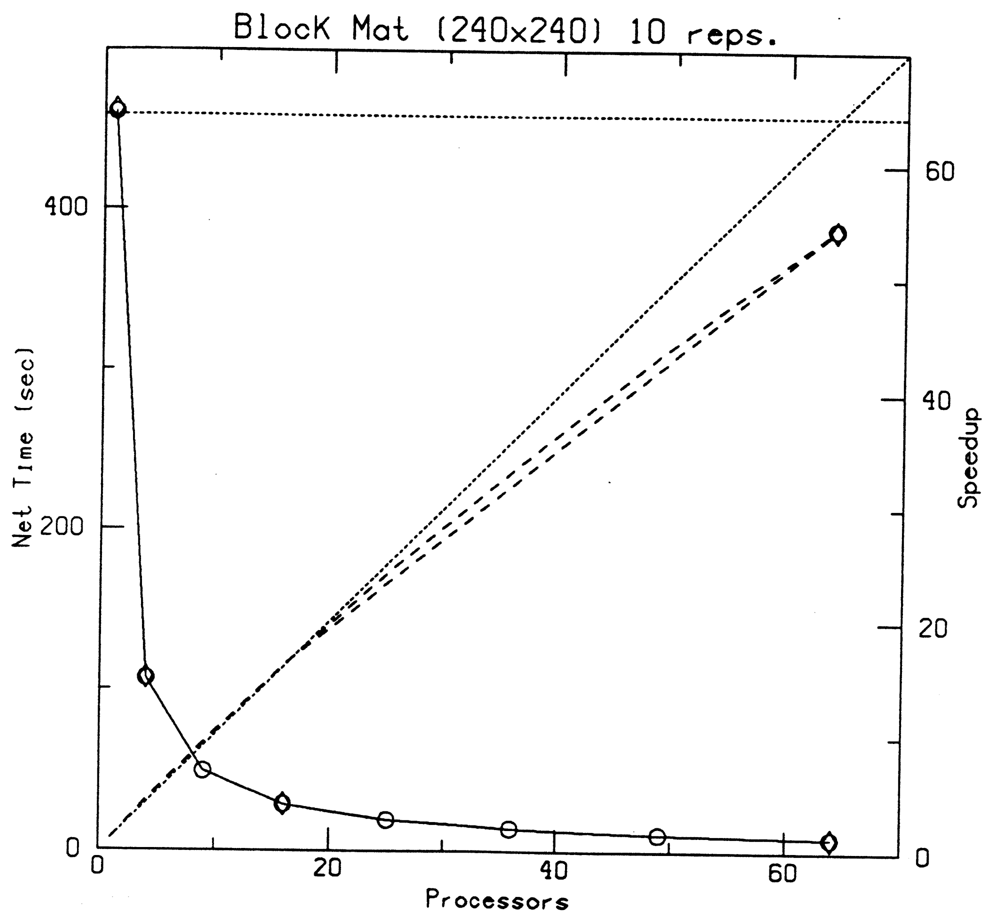


Figure 2: Comparison: Linda vs. native message passing; matrix multiplication (Dekel algorithm).

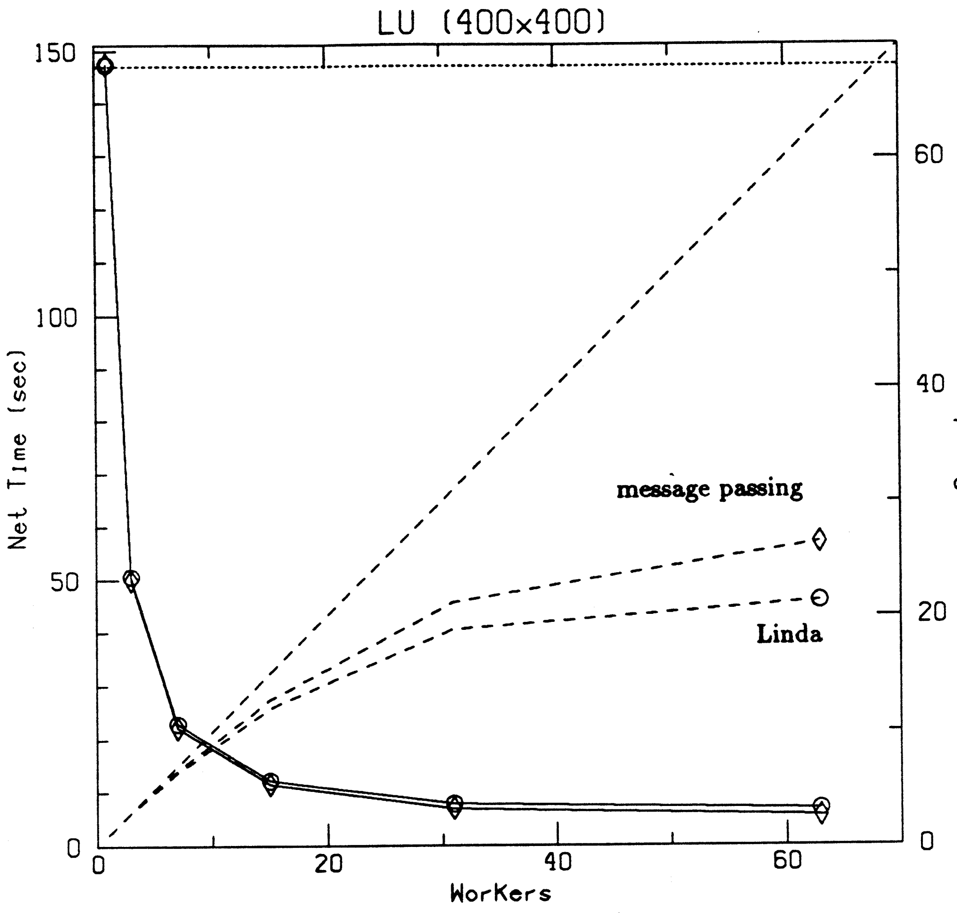


Figure 3: Comparison: Linda vs. native message passing; LU decomposition.

is fairly straightforward and unproblematic, insofar as the algorithm was developed specifically for distributed execution on meshes. Performance of Linda and of the message passing versions are essentially the same. See Bjornson's thesis [B91] for a more complete discussion. Figure 3 shows the same comparison for a 400 row LU decomposition problem. The message-passing program uses explicit broadcast (to distribute the multiplier at the start of each iteration) as well as point-to-point message sending. This is a relatively fine-grained program by the standards of the iPSC/2—on average, 6 msec of computation per Linda operation in the Linda version. Nonetheless, Linda achieves 80% of the efficiency of the message passing version in the 64 node case (63 identical workers, one master/manager process). Bjornson's thesis, again, discusses the case in detail.

Database search. Genetic sequence comparison involves a "closeness" computation between a target sequence and each sequence in some (often large) sub-set of a genetic sequence database. The computation is intended to approximate a geneticist's sense of the biological similarity between two sequences. Development of the master-worker program (sequences are taken from the database and dumped into tuple space by the master; they are removed by worker processes) is discussed in detail in [CG90]. Figure 4 shows speedup for this problem and a related problem discussed below on a network of Sparcstations.

The simple parallel database search techniques that underlie this application are the basis of the parallel expert database project described in [FG91], and will be used in a future project for parallel keyword-based text retrieval.

Figure 4 also shows performance on a network of Sparcstations of a Linda application that parallelizes the "closeness" computation referred to above. The closeness algorithm in this case is a matrix computation; all matrix elements along a given counter-diagonal may be computed simultaneously once the previous counter-diagonal is known. Thus, the computation proceeds in a wavefront across the matrix, from upper left to lower right corner—a fairly typical pattern for a variety of matrix computations. The Linda program, which divides the matrix into sub-blocks and uses a master-worker structure to compute all blocks along a counter-diagonal in parallel, is described in [CG90].

Molecular simulation. Distance geometry programs take sets of distances between the atoms in a molecule and convert them into cartesian coordinates. DGEOM is one of the best known examples of this type of program and is available from the Quantum Chemistry Program exchange.

Richard Judson of Sandia National Laboratories built a Linda version of this program. The program inputs the given distance constraints and completes the model's unspecified distances using simple geometry. The distance constraints are expressed as ranges. The program randomly selects a fixed distance within each range, and computes a reasonable set of optimized cartesian coordinates. This process is repeated many times. Since each set of fixed distances are

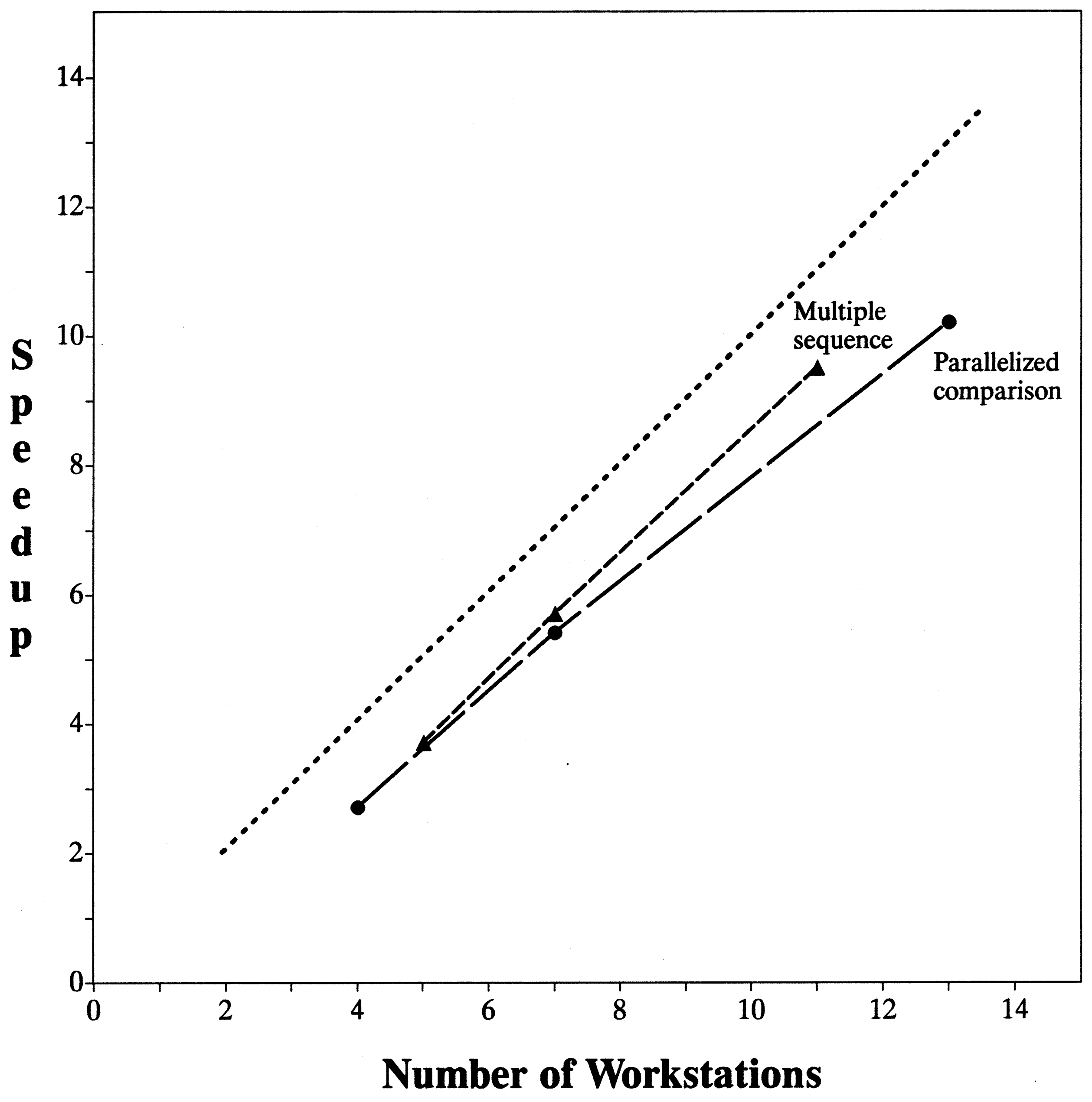


Figure 4: LAN speedups: parallel database search
multiple sequence comparisons (▲);
parallelized comparison (●)
(propagating wavefront algorithm).

independent, the parallel version of the program uses a master-worker scheme and passes each fixed distance set to a separate worker.

The standard DGEOM benchmark is a problem involving 10 cyclosporin A structures. On a Cray X-MP the sequential DGEOM program required 155.1 seconds. On a 6 node Silicon Graphics computer (with 33 Mhz CPU's) the same calculation with the C-Linda version of DGEOM took 392 seconds. Figure 5 shows speedup for a 10-structure problem on a network of DECstation 5000's, again for the cyclosporin-A molecule.

Shifman *et. al.* describe their work on the Lindafication of another molecular dynamics code in [SWSM91].

Computational fluid dynamics. FREEWAKE was developed by T. Alan Egolf at United Technologies Research Center (UTRC) for modeling the wake structure of helicopter rotor blades [E88]. The Linda version was developed by Kevin Dowd and Harry Dolan of UTRC. The inner kernel is a triply-nested loop; at the core of the loop, the spatial displacements of each relevant element are calculated, based on the position and strengths of all other elements. These calculations are mutually independent, and so the kernel is highly parallelizable. (On the other hand, its inter-process communication requirements aren't trivial; on each iteration, the master must collect results from everyone and then re-distribute them to all workers.)

The speedup data (figure 6 *a*, *b* and *c*) are particularly interesting, because we ran the same program on a shared-memory machine, a network of DECstation 5000's, and a 128-node Intel iPSC/860—these platforms define the future of parallel computing. They represent the environments on which the field will center: small shared-memory machines, large distributed-memory supercomputers, and local area networks. (Note that, although the code is highly parallel,

Portfolio optimization. Professor John Mulvey of Princeton has used Linda in parallelizing complex financial models that involve stochastic optimization for portfolio management. Figure 7 shows speedup for a "typical" problem on a network of Silicon Graphics Personal Iris workstations.

Shallow water equations. SHALLOW [WP86] is an explicit finite difference solution of the shallow water equations. These equations, important in climate modeling, consist of three coupled differential equations which depend on time and two spatial dimensions.

A Yale research group (including Martin Schultz and Ashish Deshpand) developed a Linda version of the program. The algorithm was based on domain decomposition and, as is frequently the case with domain decomposition, used Linda as a flexible message passing system.

In the course of the project, a number of different domain decompositions were studied. The optimal decomposition was produced by dividing the spatial

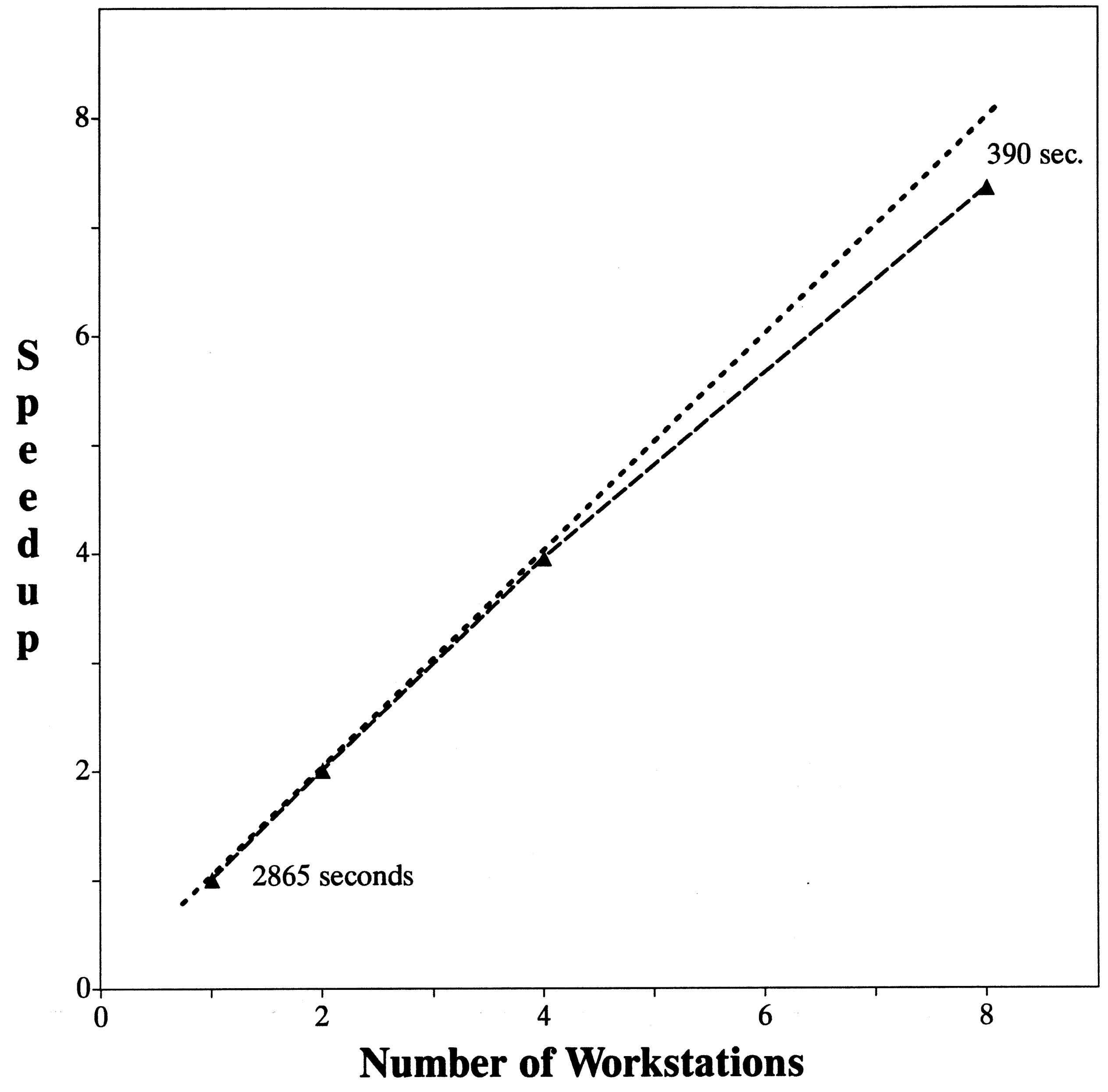


Figure 5: LAN speedups: DGEOM

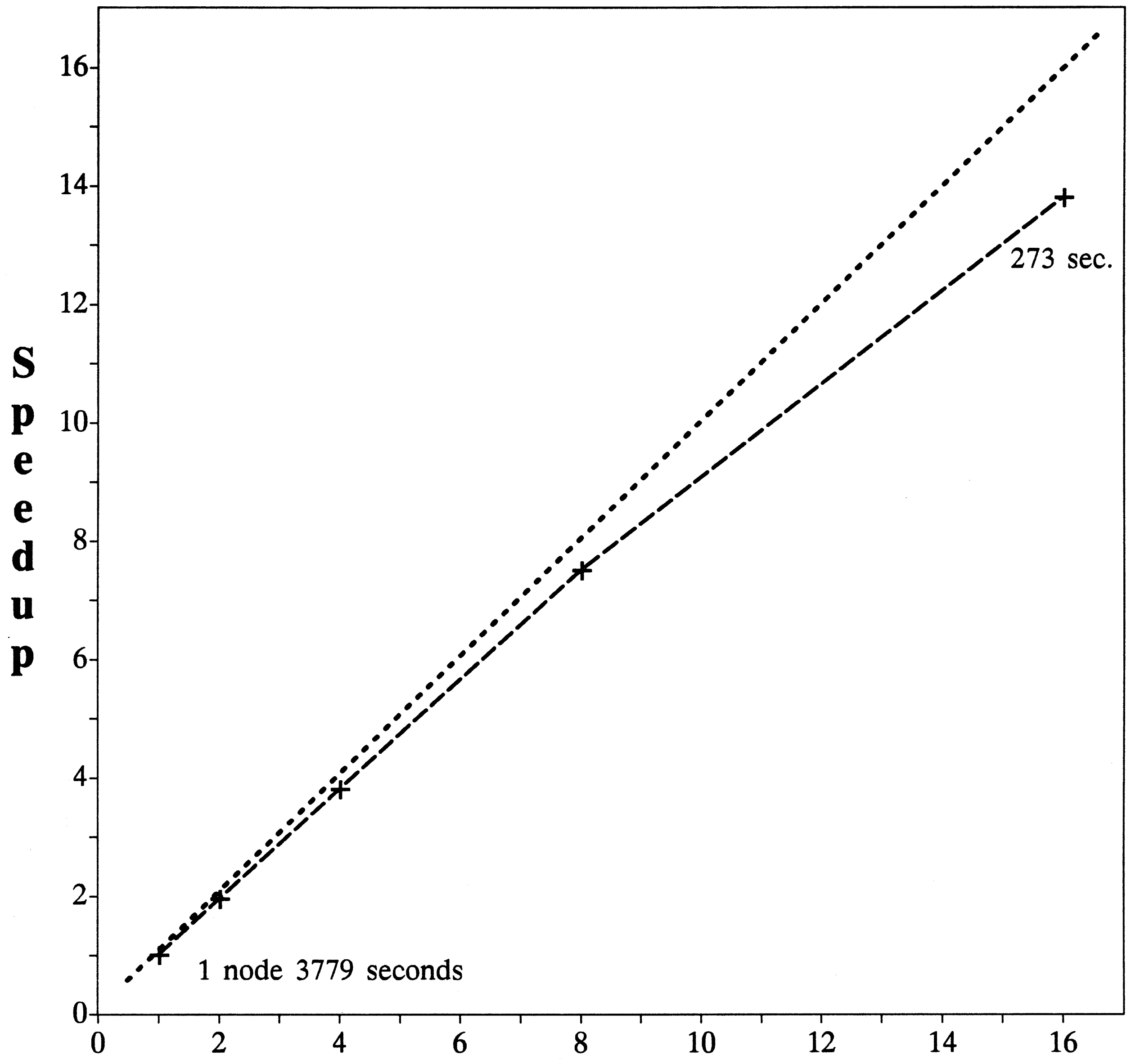


Figure 6a: shared–memory multiprocessor: FREEWAKE (1k element problem size)

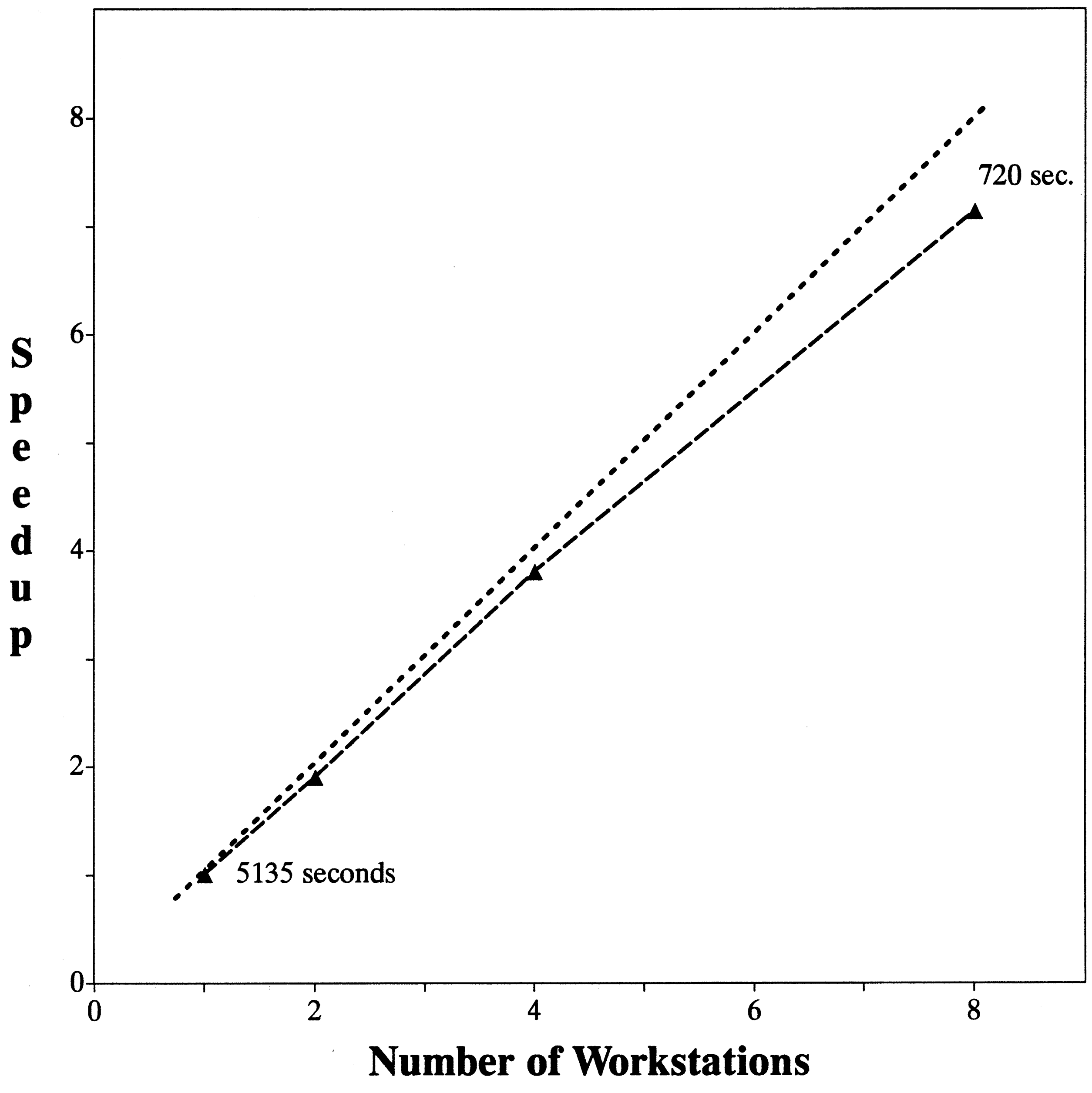


Figure 6b: LAN speedups: FREEWAKE (8k element problem size)

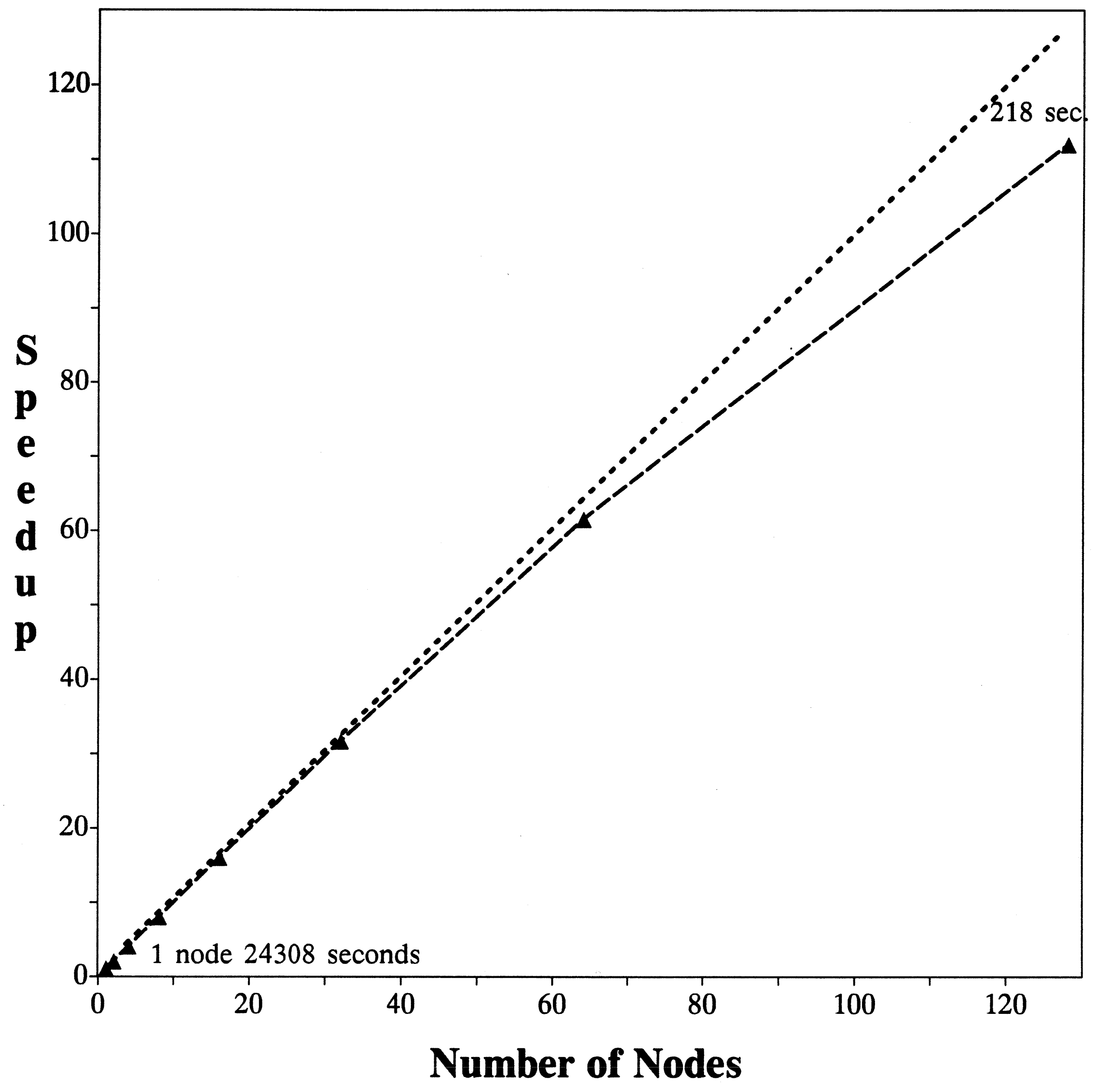


Figure 6c: distributed–memory multiprocessor: FREEWAKE (8k element problem size)

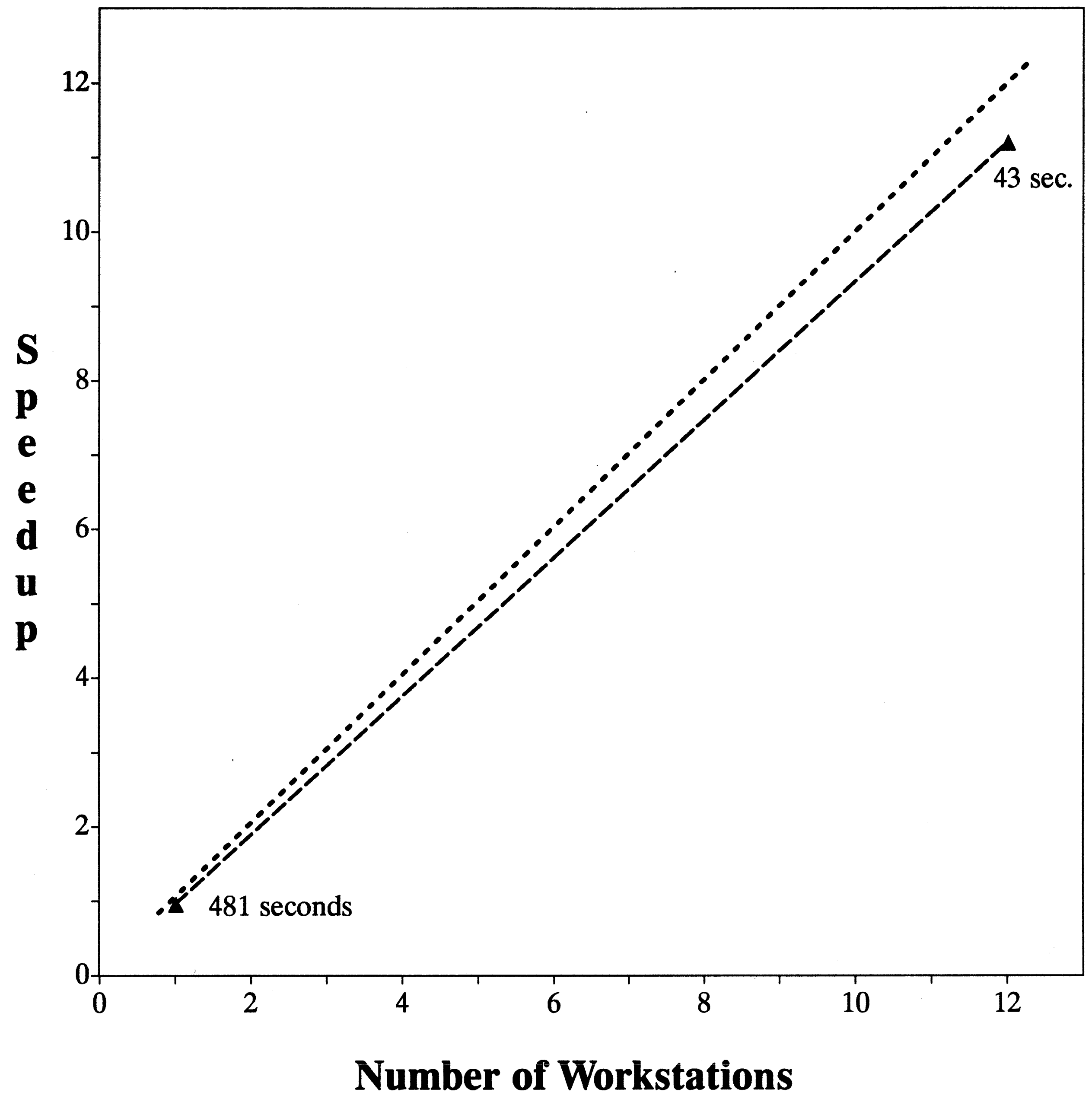


Figure 7: LAN speedups: stochastic optimization, portfolio analysis

dimensions into strips with one strip per processor. The algorithm is inherently message passing, so the program was reimplemented with Intel's native message passing environment (NX3.2) for purposes of comparison.

The result of the comparisons between Linda and message passing met the expectations of experienced Linda users—the Linda version of the program performed about as efficiently as the message passing program. For example, for a 512x512 data set with 200 time steps on a 64 node iPSC/2, the code took 130.8 seconds with Linda and 126.5 seconds with NX3.2—a performance difference of about 3%.

Ray-tracing. "Rayshade" is a ray-tracing program developed by Craig Kolb as an undergraduate at Princeton University and a research assistant to Benoit Mandelbrot at Yale University; the program is typical of the ray-tracing applications that are in widespread use for rendering color images. Kolb Lindafied the program using a standard master-worker model. A task is the computation of a single scan-line; task descriptions are dumped into tuple space; workers withdraw them, perform the specified computation, dump the result into tuple space and repeat until the computation is complete (see [BJS91] for a more complete discussion of parallel Rayshade). The results in figure 8 show a better than 30-times speedup on a network of 40 Sun Sparcstations.

Computer artist Ken Musgrave of Yale has Lindafied his imaging programs and used them to produce a series of striking pictures, including images displayed at a Spring, 1990 show at the Guggenheim Museum in New York; see [MM89].

(Related to these ray-tracing applications is INTEGRA, a code that does seismic modeling based on sophisticated ray tracing algorithms. The theory and the sequential program are the work of Dr. Victor Pereyra of Weidlinger Associates. He produced a Linda version of the program with the help of Mathew Koshy, also of Weidlinger associates. In preliminary tests, the Linda version, like many ray-tracing applications, displays nearly ideal speedup.)

Others. What we've presented is by no means a complete survey of extant applications. Linda has been used in realtime data fusion [F90], radar cross section codes, genetic linkage mapping, FFT [B91], neural network simulators, circuit simulators and many other applications; the list continues to grow.

4 New Directions

We discuss two projects (the Linda Program Builder and the Information Landscape) briefly, and a third ("Piranha") in more detail.

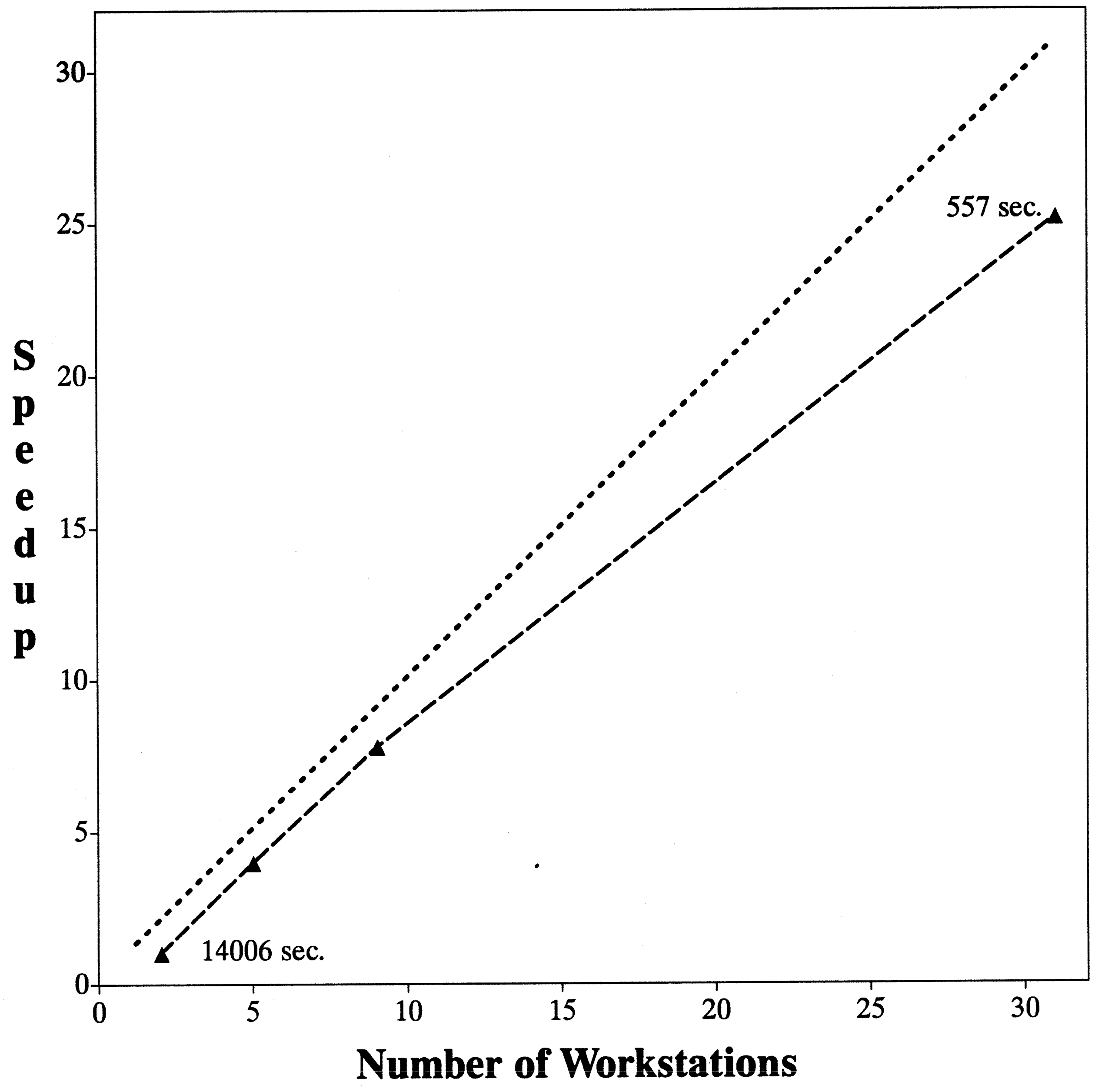


Figure 8: LAN speedups: Kolb's "Rayshade"

The Linda Program Builder

The Linda Program Builder (LPB) [ACG90], which is the work of Shakil Ahmed of Yale, is a structured editor that builds Linda programs under the programmer's guidance. The user specifies templates (say, a master-worker program, in which workers are created in some specified way, tasks are stored in a specified data structure, and so on). The LPB saves keystrokes and memory-work, and prevents errors, by laying down a significant part of an application's coordination framework. But its potential significance goes well beyond these administrative functions.

The LPB "understands" an application it has built at a deeper level than what a compiler can achieve. The Linda pre-compiler parses and analyzes operations that create and manipulate tuples; but the LPB knows *why* those operations are being carried out—it "understands" that a certain data structure exists to apportion tasks to workers, another to feed results back to the master, a third to hold intermediate results that the worker processes will need—even though the three data structures may be similar or identical. At a slightly lower level, it "understands" how basic Linda operations are used to create data structure; it sees these operations not in isolation, but as elements in a bigger picture.

The first kind of information is potentially valuable in program visualization. Our Tuplescope visualizer [BC90] presents an evolving image of tuple space and its contents as an application executes; the LPB's superior knowledge will make it possible to organize this display in terms of the programmer's own conception of his program's structure.

The second kind of knowledge is valuable to the compiler. The LPB can tell the compiler what a series of Linda operations is intended to achieve; the compiler may be able to accomplish a semantically-identical effect in a more efficient way. Instead of carrying out a series of Linda operations as discrete events, fusing the whole sequence into a single operation at runtime may result in more efficient code. For example: to remove an element from the head of a multi-sink queue of tuples, a process will execute three Linda operations. First, it will remove the head-index tuple; second, it will put this tuple back, after incrementing the field that designates the current head element in the queue; finally, it will remove the queue tuple whose index field matches the original value in the head-index. The LPB can inform the compiler: I put these three Linda operations into the program in order to remove an element from a multi-sink queue. Instead of carrying out the three operations separately, the compiler can fuse them together and execute them more efficiently, while preserving the semantics of the original.

Now, the obvious question is: if the programmer thinks "I'm removing an element from a queue," and the compiler is told "the program is removing an element from a queue"—why do we need the Linda operations themselves?

Wouldn't it be a good idea simply to add the higher-level "multi-sink queue" structure directly to the language?

In fact it would be a rotten idea, for several important reasons.

First, the Linda operations, which are powerful but simple, are a concise way to *define*—to specify the semantics of—an arbitrarily rich set of distributed data structures and their associated operations. We can't allow the basic Linda operations to be superseded by higher-level structures because these basic operations are a *lingua franca* of asynchronous parallelism: they allow us to build and to understand *new* structures when the existing very-high-level ones don't quite work. And all working programmers understand that, *whatever* your built-in, very-high-level operations may be, they won't always do what you want; you will sometimes be forced to build new ones of your own.

The second reason why the basic Linda operations *must* play a role, even in an LPB world of very-high-level tuple space structures, has to do with language design. As we learn more about programming methods and idioms, there's a natural tendency to capture this new understanding in very-high-level languages. But this tendency is best resisted; the LPB allows us to achieve the same effect in a better way—to have our cake and eat it too.

Languages in which very-high-level operators have been incorporated suffer from two problems. First, they tend either to be *narrow* in applicability or unacceptably *complex* in the range of operators they provide, or both. Very-high-level operations are in most cases intrinsically specialized in what they can accomplish. (A cogent example is Ada's *accept* statement, which provides a very-high-level communication service that is well matched to a particular species of client-server transaction, but completely inappropriate to many other purposes—for example, to the communication needs of most parallel applications [GP90].) Second, a language is by nature a fairly static object. Revising and re-implementing a language is an operation not lightly undertaken, to say the least. Very-high-level operations, on the other hand, tend to reflect ongoing advances in programming methodology, which may be continual in a young field like parallel programming; and because they are often quite specialized, they may be fairly site-specific—the right ones to have depend on the kind of applications you build.

In short, it would be nice to be able revise a language whenever your understanding of programming methodology advances, to customize it to the needs of your particular site, and to learn (and *support*) only those parts of the language you really need—*without* losing compatibility with the rest of the language's user community. These ends are evidently impossible within the programming language framework. But the LPB readily accomodates them. The LPB framework can be evolved, customized, sub-setted or refined easily and at whim: the software product that emerges is *still Linda*, no matter what the vagaries of your particular LPB. And Linda itself is sufficiently high-level to be portable and transparent: the LPB framework is useful, but by no means essential. This language-hierarchy model—a portable, high-level and powerful language, with a specialized very-high-level program builder superimposed—seems to us to be a promising approach to a wide variety of programming language questions. (We will be pursuing it beyond, not merely within, the coordination language framework. For example, we are investigating an object-based Linda based on an LPB generating C-Linda code: class and object structure is provided by the program builder, not by the computing language, which remains plain old C.)

Information Landscapes

This project centers on the use of multiple, persistent tuple spaces in building distributed databases and information systems. The goal is to provide a "walk in database"—browsers within a database partition temporarily become part of it; they may converse with other browsers and leave daemons behind to represent them. The one element of the project that is relevant to a discussion of systems issues is the interface, which provides a graphics-based interpreter for Linda. The interpreter allows users to read or remove tuples by mousing on them, and to insert tuples directly from an editor via appropriate mouse manipulations.

Piranha

At most sites, LAN-connected workstations are often idle; it's long been regarded as natural and desirable to allow these idle workstations to pitch in and help out with ongoing parallel computations, returning to their owners at a moment's notice when they are needed. These observations go back at least to the Worm project at Xerox PARC in the 1970's. Fast workstations have made this recycling idea particularly important: mere *wasted* power in a typical modern LAN (let alone *total* LAN power in the aggregate) often amounts to supercomputer-equivalents on appropriate problems. The argument in favor of recyling has shifted accordingly. At many sites, it's no longer "is it worth taking a shot at recyling idle nodes?" Instead, it's "how can we possibly justify *not* taking a shot at recycling? We're supposed to take all this bought-and-paid-for computing power and simply *dump it* unceremoniously *down the drain*? Get *outta* here!" (All right, maybe they don't always use these words precisely, but you get the idea.)

Linda and Node Recycling are a particularly close fit. Linda supports *uncoupled, anonymous* communication. Processes in a Linda ensemble make no assumptions about their fellow processes: who they are, where they are, what they're doing, when they lived (or will live). Idle-node recyling requires, for its part, that nodes be free to join or to leave an ongoing computation as the spirit moves. Recyling demands a programming model that can accomodate dynamic process ensembles; Linda's anonymous, uncoupled communication model provides exactly that.

In our Piranha model, a program is structured as a cloud of tasks (either an ordered collection or an unordered bag); computational "piranhas" attack the task cloud. The more piranhas attack, the faster the cloud is consumed and the program completes. The part of The Piranhas is played by workstations on the network: at any time they may join an ongoing feeding frenzy (grab and execute a task from an existing task collection), or withdraw from a feeding frenzy (stop consuming tasks, despite the fact that there are still tasks remaining and the computation is incomplete). In practice, workstations join when they become idle; they leave when their users need them, or when a more attractive task collection appears. The meaning of *idle* reflects the owner's wishes; the current default on our system defines it conservatively as 10 minutes of keyboard idle time, a 1-minute load average below 0.7 and a 10-minute load average below 0.3. The user may customize the definition in a variety of ways. "Their owners need them" means that a key has been struck, the mouse has moved, or a special seize-node command has been issued (usually via modem). The meaning of a "more attractive" task collection depends on the system's distributed scheduling algorithm: on what basis, in other words, available Piranhas should apportion themselves among contending jobs. The system itself is compatible with a wide variety of strategies, and experiments are underway.

A Piranha program is a specialized form of Linda program (the LPB supports the construction of Piranha programs). The coordination framework of a Piranha program consists of three functions, named *feeder*, *piranha* and *retreat*. Where a Linda program uses the `eval` operation to create processes explicitly, a Piranha program does no explicit process creation. Instead, the system creates a varying pool of worker processes automatically, each executing the *piranha* function. Initially, one *piranha* process is created for each workstation that is idle and willing to participate in the computation. When new workstations join, new instances of *piranha* are created automatically. When the user needs his workstation back, the *retreat* function is invoked by the system; *retreat* makes a note in tuple space of any unfinished business. The *feeder* serves the role of master—initializing, coordinating and finalizing the computation as need be.

Our prototype system seems to be delivering on the abstract promise of the model. Figure 9 shows the execution histogram of a program developed by researchers in the Electrical Engineering department at Yale. In the histogram, each workstation is represented by a single horizontal line; when the line is white, the corresponding node is attending to its user; when the line is black, it is participating in an ongoing Piranha computation.

The application involves the biomagnetic imaging problem. Srinivas Rao, a graduate student in the Electrical Engineering department, supplies the following description:

> Given a set of magnetic field measurements from a discrete array of sensors with a known geometry, find the positions and magnitudes of N test dipoles such that the squared error between the magnetic field produced by these test dipoles and the given magnetic field measurements is minimized. This amounts to minimizing an objective function (the squared error) in a 3-N dimensional search space... We decided to do a thorough investigation of the energy surface, focusing specifically on the local minima: where they are located, and their width and depth (relative to the global minima).

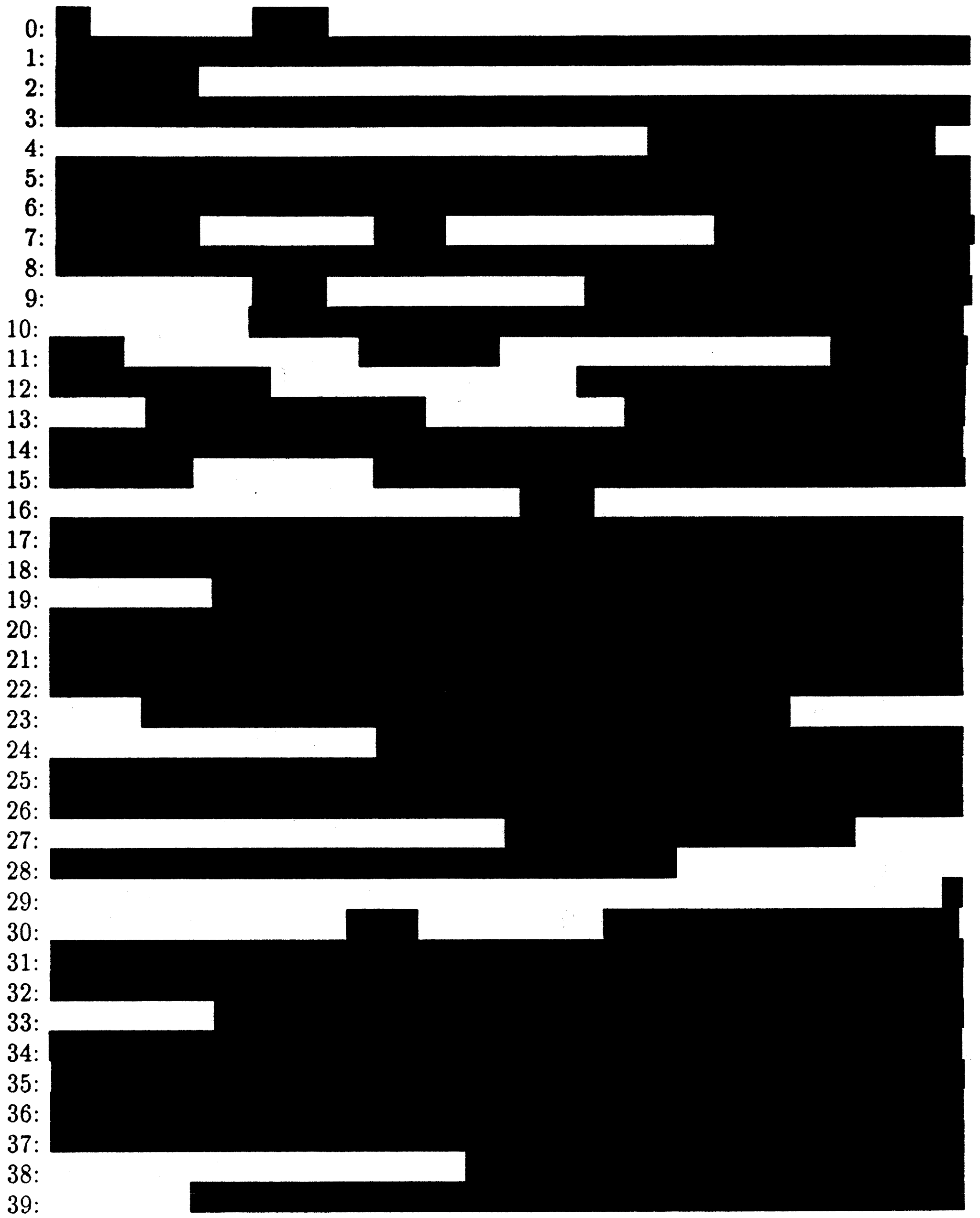
The histogram shows that, within an elapsed wall-clock interval of about 50 minutes, the Linda program running under Piranha delivered more than *25 hours* of computing time.

Figure 10 shows another example. The program involves an application in theoretical physics. Martin White, a graduate student in the Physics Department, describes it as follows:

> For the last 20 years there has existed in physics and astronomy a conflict between theory and experiment known as the solar neutrino problem... A most promising approach to the problem assumes that neutrinos have properties beyond those required by the current standard model of particle physics... We assume a set of parameters (mass differences, mixing angles and magnetic moments) for the neutrinos and solve numerically the quantum mechanical equations for the propagation of the neutrinos from their point of origin, through the Sun, to the Earth. We use a two-flavor model (including the anti-particles), since this is expected to capture the essential physics that may be operating such as resonant flavor changing or the Mikheyev-Smirnov-Wolfenstein effect which predicts that in certain circumstances the amount one flavor 'rotates' into another in the interior of the Sun could be enhanced...

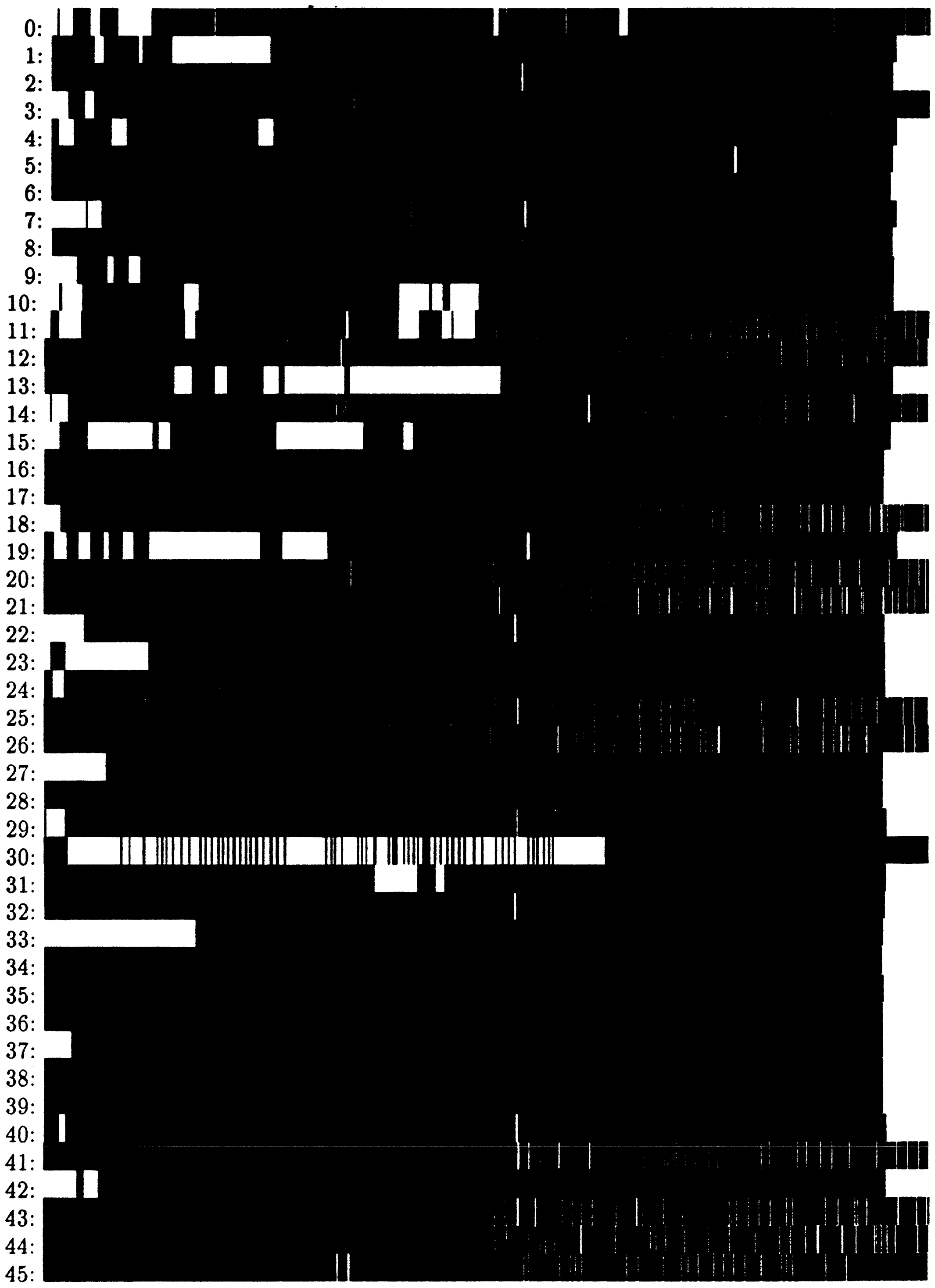
During an elapsed time of about 13 hours, the Linda program running under Piranha delivered almost 559 hours, more than 23 days, of aggregate computing time.

These applications have run many times under Piranha—as have others, including the ray-tracing, linkage mapping and "Freewake" codes discussed above.



run time = 3108 secs
run time = 0.86 hours
node time = 25.79 hours
burst = 50
ave number of nodes = 29.88
ave burst len = 1857.20
full nodes = 19

Figure 9: Piranha: biomagnetic imaging problem.



run time = 46731 secs
run time = 12.98 hours
node time = 558.50 hours
burst = 996 ave number of nodes = 43.03 ave burst len = 2018.68 full nodes = 9

Figure 10: Piranha: solar neutrino problem.

5 Conclusions

We doubt whether any single approach to parallel programming will dominate the field any time soon. We believe that a fairly small number of alternatives will hold sway over the next several years; we don't know the precise composition of the list, but we do know that Linda will be on it. New work will change Linda, making it more suitable for fine-grained parallelism, integrating multiple, first-class, persistent tuple spaces, adding a strategy for network reliability borrowed from one of the several groups now attacking this problem. The system will change; but not unrecognizably. We are betting that building on experience will continue to be a prudent and profitable approach to parallel programming.

References

[ACD90] V. Ambriola, P. Ciancarini and M. Danelutto, "Design and distributed implementation of the parallel logic language Shared Prolog," in *Proc. 2nd ACM SIGPLAN Symp. Princ. and Pract. Parallel Programming* (Seattle: 1990):40-49.

[ACG90] S. Ahmed, N. Carriero and D. Gelernter, "The Linda Program Builder." *Proc. Third Workshop Languages and Compilers for Parallelism* (Irvine, 1990).

[AS91] B.G. Anderson and D. Shasha, "Persistent Linda: Linda + Transactions + Query Processing," in [LeM91], 129-142.

[B91] Robert D. Bjornson, *Linda on Distributed Memory Multiprocessors.* Yale Univ. Dept Comp. Sci. PhD. Diss. (1991).

[BC90] P. Bercovitz and N. Carriero, "TupleScope: A graphical monitor and debuffer for Linda-based parallel programs." Yale Univ. Dept. CS RR-782 (April 1990).

[BCLM88] Jean-Pierre Banâtre, Anne Coutant and Daniel Le Métayer, "Parallel machines for multiset transformation and their programming style." *Informationstechnik* 30(2, 1988):99-109.

[BKS91] R. Bjornson, C. Kolb and A. Sherman, "Ray Tracing with Network Linda." *SIAM News,* 24,1(Jan. 1991).

[BS91] David E. Bakken and Richard D. Schlichting, "Tolerating Failures in the Bag-of-Tasks Programming Paradigm," in *Proc. of the 21st Int. Symp. Fault-Tolerant Computing,* Montreal, Canada (June 1991): 248-255.

[CG88] N. Carriero and D. Gelernter, "Applications experience with Linda," in *Proc. ACM Symp. Parallel Programming*," July 1988:173-187.

[CG89] N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, April 1989(32,4):444-458.

[CG89b] *Technical correspondence* re "Linda in context." *Comm. ACM*, Oct. 1989(32.10):1244-1258.

[CG90] *How to Write Parallel Programs: A First Course.* (Cambridge: MIT Press, 1990).

[CG91] N. Carriero and D. Gelernter, "Coordination languages and their significance." *Comm. ACM* (to appear).

[CM88] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation.* Addison-Wesley (Reading, Mass: 1988).

[DBC91] H. Dolan, R. Bjornson and L. Cagan, "A Parallel CFD Study on Unix Networks and Multiprocessors."

[DNS81] E. Dekel, D. Nassimi and S. Sahni, "Parallel matrix and graph algorithms." *SIAM J. Computation* 10(1981):657-673.

[E88] Egolf, T. A., "Helicopter Free Wake Prediction of Complex Wake Structures Under Blade-Vortex Interaction Operating Conditions," Proceedings of the 44th Annual Forum of the American Helicopter Society, June 16-18, 1988.

[F90] M. Factor, *The Process Trellis Software Architecture for Parallel, Real-Time Monitors.* Yale Univ. Dept. Comp. Sci. PhD. Dissertation (July 1990)

[FG91] S. Fertig and D. Gelernter, "A Software Architecture for Acquiring Knowledge from Cases," in *Proc. of the International Joint Conference on Artificial Intelligence*, August 1991.

[FT89] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Prentice-Hall (Englewood Cliffs, NJ: 1989).

[Gel89] D. Gelernter, "Beyond parallelism to coordination," in J.Dongarra, *ed., Proc. Fourth SIAM Conf. on Parallel Processing for Scientific Computing* (Dec. 1989.)

[GP90] D. Gelernter and J. Philbin, "Ada Lite: Motivation, informal description and examples." Yale Univ. Dept CS TR (Jan. 1990).

[HM91] S. Hiroyuki and S. Masaaki, "Communication in Linda/Q: Datatypes and Unification," in *Proc. Int. Conf. Parallel Processing* (Aug. 1991).

[J91] S. Jagannathan, "Customization of first-class tuple spaces in a higher order language," in *Proc. PARLE '91* (Eindhoven, June 1991): 254-276.

[J91b] S. Jagannathan, "Expressing fine-grained parallelism using distributed data structures," in [LeM91], 83-100.

[Lel90] . Leler, "Linda meets Unix." *IEEE Computer* 23,2(Feb. 1990):43-55.

[KW90] Srikanth Kambhatla and Jonathan Walpole, "Recovery with limited replay: Fault-tolerant processes in Linda." Oregon Grad. Inst. Dept. CSE TR CS/E 90-019 (Sept. 1990).

[LeM91] Daniel LeMetayer, *ed.. Research Directions in High-Level Parallel Languages* (Mont Saint-Michel: IRISA-INRIA, June 1991): Springer Verlag (forthcoming).

[MM89] F.K. Musgrave and B.B. Mandelbrot, "Natura ex machina", *IEEE Computer Graphics and Applications*, 9,1(Jan. 1989):4-7.

[M+91] Perry L. Miller, Prakash Nadkarni, Joel E. Gelernter, Nicholas Carriero, Andrew J. Pakstis and Kenneth K. Kidd, "Parallelizing genetic linkage analysis: A case study for applying parallel computation in molecular biology." *Comp. Biomedical Res.* 24(1991):234-248.

[R88] G.A. Ringwood, "Parlog86 and the dining logicians," *Comm. ACM* 31,1 (Jan. 1988):10-25.

[PS90] Richard Peskin and Edward Segall, "Linda Strategies for Scientific Computing Environments", CAIP-TR-137, Computer Aids for Industrial Productivity Center, Rutgers University, Piscatway, NJ, 1990.

[SWSM91] M.A. Shifman, A. Windemuth, K. Shulten and P.L. Miller, "Molecular Dynamics Simulation on a Network of Workstations Using a Machine-Independent Parallel Programming Language." Technical Report, Yale Univ. Center for Medical Informatics (1991).

[W91] Greg Wilson, ed., *Linda-Like Systems and Their Implementations* (Edinburgh: June 1991) (forthcoming).

[WP86] Warren M. Washington, Claire L. Parkinson. Mill Valley, Calif. University Science Books; New York Oxford University Press, 1986

[Xu88] A.S. Xu, "A Fault Tolerant Network Kernel for Linda." MIT LCS Master's Thesis (1988).