Linda and Message Passing: What have we learned?

Nicholas Carriero and David Gelernter

YALEU/DCS/RR-984
August 1993

Linda and Message Passing: What have we learned?

Nicholas Carriero and David Gelernter*
Yale University
Department of Computer Science
New Haven, Connecticut

August 11, 1993

Abstract

We weigh the relative merits of two popular coordination systems: Linda, a high-level coordination language, and PVM, a message passing coordination library. In the past, such a comparison might have been merely academic. Today, however, both systems see significant use in parallel programming, especially in LAN settings. Our comparison here will build on this foundation of practical experience. It will reflect practical considerations like expressivity, performance and flexibility. We conclude that the relationship is analogous to one that holds between low- and high-level computation languages: both types of system have a role to play, but for a generously broad sweep of applications, the expressivity and flexibility gains of the high-level approach easily offset what are often surprisingly modest performance losses relative to the low-level approach.

*This work funded in part by the ONR (N00014-93-1-0573).

1 Introduction

Although the development of software tools for parallel programming continues to be an active and evolving area, the sub-field centering on the use of local area networks as platforms for parallel applications has achieved a certain stability—probably because LANs are today the most important platforms by far for parallelism in commercial settings. Three of the most widely used coordination models in the local area network arena appear to be PVM from ORNL, Parasoft's Express, and Scientific Computing Associates's Linda. (A *coordination model* defines a system for building parallel programs—defines the "extra ingredients" that must be added to a sequential language like C or Fortran to achieve a system that can be used for parallel programming.) These three are often mentioned together. Vendors of parallel machines and workstation clusters, for instance, have made a point of mentioning the availablity of these systems in recent product announcements—IBM's SP1 announcement is one example. PVM and Express are based on the message-passing coordination model; Linda uses a form of virtual shared memory called a "tuple space."

Neither message passing nor Linda is restricted to local area network use: both models can and have been implemented on the full spectrum of shared-memory multiprocessors, distributed-memory multiprocessors and networks. We single out networks only because a greater degree of programming environment stability seems to have been achieved there.

One way to assess the relative merits of programming systems (in terms either of expressivity or performance) is to compare them head-to-head on the same problems. In [CG89] we pursued this exercise in the context of several systems that were popular in academic circles at the time. Since [CG89], an interesting development: parallelism is now a reality, in daily use for production computing. So we can now ask not merely what's popular in academia, but (far more interesting) what's popular with software developers—which systems see actual use.

After briefly introducing Linda, we take up expressivity (how expressive is Linda vs. message passing?) by examining some simple programming examples taken from the PVM programming manual, recoding them in Linda and comparing. We amplify by discussing the use of Linda on a real application of current interest.

We then examine the performance issue (how efficient is Linda vs. message passing?) by discussing an experiment carried out by another group in which the performance of Linda and PVM are compared directly on the same application; we add some comments on similar experiments performed elsewhere.

We reach the following conclusions: (1) Linda is in general more expressive than message passing, and (2) the performance of Linda and of message passing programs is generally comparable. A point we take up in conclusion is that (3) Linda appears to offer a smoother transition than message passing to adaptive parallelism, which we believe will become a dominant paradigm in the near future.

Linda has disadvantages too. These are in line with the disadvantages of optimized high-level systems (which make things easy and *in most cases* efficient) versus low-level systems (which are inconvenient, but potentially unbeatable with respect to efficiency). The same trade-offs hold when we compare C or Fortran to assembler code. We believe that Linda is more expressive than message passing essentially always: this follows from the fact that message passing operations are trivially expressible in Linda. Linda is usually as efficient message passing, but there are exceptions. The exceptions are those applications that rely on point-to-point communication that is *irregular*. In an application relying on "regular" point-to-point communication, each process sends a stream of messages to a fixed or slowly-evolving set of recipients. Linda detects patterns like this, and supports such applications as efficiently as a direct message-passing system would. (In practice, most message-passing-style applications seem to fall into this category.) But when message destinations are *unpredictable*—when, for example, processes generate messages and send them to randomly-chosen recipients—then there is no pattern for Linda to detect, and Linda's realization of such a program is less efficient than a direct message-passing version would be.

The current releases of C-Linda and Fortran-Linda from Scientific Computing Associates form our basis for discussion. Although we will describe ongoing research in a few cases, our main goal is to discuss the Linda environment as it is commercially distributed—the environment that sees production use today.

The authors express their particular thanks to Al Geist of Oak Ridge National Laboratory, for his comments on our comparison of Linda and PVM, and to Clemens Cap and Volker Strumpen of the

University of Zurich, for their comments on our discussion of their study comparing the performance of Linda, PVM and Parform.

2 Linda

2.1 Coordination Languages

Linda is a "coordination language." The "coordination" part of that term means that Linda is designed for gluing separate serial computations into ensembles. Our contention has been that parallel applications, distributed systems, certain kinds of heterogeneous systems, "time-coordinated" as opposed to "space-coordinated" computations (in which processes run at different times rather than in different places) and "Turingware" ensembles (in which people and processes are intermixed) are all instances of the same software species, and should be supported by a single general-purpose coordination model. The field has generally preferred to create special programming models for parallel applications, for distributed systems and so on. We've argued that this strategy is insufficiently flexible and an unnecessary violation of conceptual economy. The study of software ensembles is, we believe, the emerging centerpoint of systems research, but the lack of general-purpose coordination languages impedes progress in discovering and understanding general principles.

The "language" in "coordination language" distinguishes a system like Linda from a coordination *library*—for example, a message passing library like PVM [GBD+93]. A language is supported by a compiler: in the case of C-Linda, for example, by a pre-compiler that translates C-Linda into plain C with library calls, plus some data files that are used at linktime to produce custom-generated interface routines that invoke appropriate runtime support routines for each type of Linda operation in the source program [CG92]. A programmer who uses a coordination library is responsible for choosing the right runtime operation for every communication operation in his program, and often for hand-assembling messages out of their component data values. In the case of a coordination language, the optimizing compiler chooses an appropriate realization for each communication operation, supports a uniform, integrated syntax, supports debugging and generally provides the range of services for which compilers have made themselves

well-liked.

2.2 The Linda Model

The Linda coordination model is based on a form of shared memory that is tailored specifically to the needs of software ensembles. Since the earliest shared-memory multiprocessors, it has been recognized that writing and reading data in a shared memory is a convenient way for processes in an ensemble to communicate. But what kind of shared memory is right for the purpose? It has often naively been assumed that there *is* only one kind—that "shared memory" necessarily means shared address space, a linear array of bytes. But a conventional address space of bytes is a poor fit to the special needs of inter-process coordination. The byte-level access it supports is wrong for inter-process communication: processes don't need to execute code out of shared memory, and they don't send bytes to each other—they send bunches of data objects (using "object" generically—a value or an aggregate of values). Its synchronization characteristics are also wrong. Because bytes in a shared address space can be overwritten at will, some form of added locking mechanism has to superimposed to insure safety in the presence of multiple readers and writers.

There are in principle, though, many sorts of memory, and the Linda model is based on a sort called "tuple space" that is designed specifically to accommodate inter-process coordination. A tuple space stores not bytes but "tuples." A "tuple" is (not to put too fine a point on it) a tuple: an ordered aggregate of data objects. A tuple space provides three basic access operations instead of the two (read and write) that are provided by conventional address spaces, and these operations have built-in synchronization: the `out` operation generates a tuple and adds it to memory; the `in` operation looks for some "matching" tuple and removes it, blocking if necessary until one is available; the `rd` operation is like `in`, but copies rather than removes the matched tuple. The tuples in a tuple space are immutable. "Matching" works in the style of a relational database. Tuple space is an associative memory: `in` or `rd` statements specify a "matching template" or anti-tuple which may include either values or typed place-holders or both. A tuple matches an anti-tuple exactly when, for all k, the *kth* element of the tuple and the anti-tuple are identical (if the anti-tuple's *kth* element is a value) or type-consonant (if the anti-tuple's *kth* element

is a typed place-holder).

Abolishing physical addresses and using tuples rather than bytes as the storage unit (tuples are usually much larger than bytes) collectively have radical implications for implementation. A byte-level address space requires hardware support if it is to be realized efficiently. Linda can be realized efficiently in software. A tuple space can be provided, in effect, wherever separate computers are wired together. The architecture of a distributed-memory parallel machine or a local area network precludes communication via conventional shared memory, but Linda-style shared memory can and has been efficiently implemented in both settings.

(An address space structured on larger storage units—say, pages—can be realized without hardware support. But where Linda allows larger-than-byte units that are *semantically meaningful*—that have meaning to the programmer—to be stored in memory and moved around the communication system, a page-level shared memory deals in arbitrary fixed-size blocks, which in the case of any given communication event are essentially guaranteed to include either more or less stuff than the programmer actually needs.)

To complete the outline of the model, Linda provides a process-creation mechanism integrated with the tuple space abstraction: the `eval` operation generates and places in tuple space an unevaluated tuple. Each field of the unevaluated tuple is specified by some expression (which may of course be a constant); those expressions are evaluated concurrently within tuple space. When they have all been fully evaluated, the unevaluated tuple turns into an ordinary tuple which can be read or removed using the standard operations.

The model doesn't assume a single tuple space; multiple tuple spaces are useful for many reasons [Gel89]. The most sophisticated version of multiple first-class tuple spaces currently appears in Jagannathan's Scheme-Linda [Jag91]. Our implementations have recently acquired a more limited multiple tuple space capability.

3 Expressivity

3.1 A simple head-to-head

Linda has often been described as a highly expressive coordination model—one that lends itself, in other words, to clear and concise programs. Thus, for example, "the Linda formalism uses an extremely powerful, but simple, primitive to give unparalleled expressiveness and flexibility" ([BA90], p. 114). It's impossible to prove such an assertion, but one way to support it is to examine representative programming examples offered by developers of competing systems, recode them in Linda, and compare.

Of PVM and Express, PVM appears to be gaining ground more rapidly, which is testament to the facts both that it is a well-designed and expertly implemented system and that it happens to be free. (Parasoft has recently announced that the Express system will support PVM programs.) Comparing Linda to programming examples in a recent PVM manual should, accordingly, be a useful exercise.

The examples in the PVM manual are chosen to illustrate "message-passing style" applications—applications in which communication relies on message sending, not on writing and reading shared data objects. Linda supports the creation and manipulation of shared objects and data structures in a way that a message-passing system, given that it provides no shared memory of any sort, obviously can't. But of course the shared-memory model trivially encompasses message-passing as well. So let's compare these approaches on message-passing turf, and examine the PVM examples.

Figures 1–3 shows a PVM program that does the following, in general terms. (1) A master process creates a collection of identical worker processes. (2) It hands each worker an array of input data. (3) Each worker performs an operation on the input, sends the result to the next worker in a virtual ring, receives a result from the previous worker in the ring, adds its result to the result it has just received, and forwards the sum to the master process.

Figures 4 and 5 show a Linda program with identical functionality, written so as to follow the PVM approach as closely as possible. Why is it so much shorter? Notice that

(1) in the PVM program, a message must be assembled explicitly out of constituent values, and explicitly disassembled: thus operations

like

```
pvm_pkint(&nproc, 1, 1),
pvm_upkint( &who, 1, 1 )
```

and so forth. Because Linda is a language and not a library, this sort of assembly and disassembly is handled automatically by the system. The programmer specifies the desired tuple of data objects, or an anti-tuple for matching; message assembly and disassembly are the system's job. Further, Linda's simpler format makes coding errors less likely, and the fact that the compile-time system handles the details of message construction makes it possible for the the system to detect and report some kinds of errors (for example, type mismatches between the sender's and the receiver's view of the data) at compile-time.

(2) In PVM, messages must be sent to specific recipient processes. Processes are identified by "task id"; a process learns its task id by executing `mytid = pvm_mytid()`. In order to send a message to any other process, it must know that process's task id too; and in order to communicate with the next process and the previous one in a virtual ring, worker processes must have some uniform way of establishing which task id will designate the "next" process in the ring and which will designate the "previous" process. The necessary coordination is accomplished by having the master broadcast to each worker an array holding the task id's of every worker (this array was returned by the `pvm_spawn(...)` operation that created the workers); each worker searches for its own task id within this array, and calls the task id following its own the "next task" in the ring, and the task id preceding its own the "previous task."

In Linda, none of this mechanism is necessary. Linda tuples aren't sent to recipients, they are simply deposited in shared memory. When the master process in the Linda version creates a worker using the `eval("slave", worker(i))` statement, the worker knows who it is from the start—the index `i` that distinguishes workers has been passed to the newly created process via the `worker()` function. The fifth worker in a 10-process ring communicates with the next worker in the ring by generating the tuple `("sum", 6, n)`, where `n` is the value to be transmitted. Worker number 6 uses associative matching to grab this tuple; by using the operation `in("sum", 6, ? psum)`, it designates for removal a tuple whose first element is the string `"sum"`, and whose second is the integer 6. The last element in the matched

tuple is assigned to the local variable `psum`.

The Linda example followed the PVM approach as closely as possible. If it had been written in "native Linda style," it would have been slightly simpler still: workers wouldn't have executed an `out` to send their results to the master—the `worker()` function would simply have returned a result, whereupon the unevaluated tuple created by the master's `eval` would have collapsed into an ordinary data tuple, which would have been `in`'ed by the master directly. In PVM messages, an array-valued field must be followed by another field giving the length of the array, and the Linda example does the same; but Linda supports transmission and receipt of array lengths within the same field as the array itself.

Figures 6, 7 and 8 show PVM and Linda versions of another simple program. This application creates a ring of identical processes, and passes a single token round the ring. The explanations for the conciseness and clarity of the Linda version are the same as before, but this example makes the point even more dramatically.

What conclusions should be drawn? Not by any means that Linda is guaranteed to produce solutions that are clearer and more concise than competing systems across the board. Rather that, in a range of programming examples chosen not to show off Linda but to illustrate the features of other systems, the subjective claim that Linda is highly expressive receives concrete support. (We'll provide some information on the performance of Linda vs. PVM and other systems in the next section.)

In comparing Linda and PVM, we need to keep sight of the fact that these systems are radically different in design and goals. Our conclusion is by no means that (with respect to expressivity) PVM must be judged a failure and Linda a success. PVM was designed with specific, pragmatic goals in mind (to provide a well-designed, portable message passing service), and it appears to have achieved these goals very successfully. The Linda project's aims are different: to define and explore a novel coordination paradigm, and the idea of a general-purpose coordination language. In pursuing these goals, the realization of an efficient and portable Linda implementation was merely a necessary precondition.

Thus, within the parallel-programming domain, Linda supports all three of what we have argued are the basic paradigms of asynchronous parallelism—"specialist," "agenda," and "result" parallelism [CG90].

Message passing systems are well-suited only to the first of these; but in addition to programs of the sort we have discussed in this section, Linda supports others that rely heavily on distributed data structures stored in tuple space (generally speaking, these are "agenda parallel" applications), and in principle the elegant fine-grained applications that emerge from the result-parallel style. The qualification is essential because, in practice, communication and process management are too expensive on most current platforms to make this sort of program efficient. But Jagannathan and Philbin's work [JP92] suggests that this aspect of Linda's expressivity will pay off handsomely within highly-optimized environments that provide cheap communication and process management.

Linda is also well-suited to the needs of adaptive parallelism, as we'll discuss below.

Beyond parallelism, Linda tends to suggest new approaches to coordination in general. The discussion in [CG90] of Linda in the context of an appointment calendar and meeting-maker application presents many of these issues; and the ensembles that fall under the heading of "Turingware" [Gel91], some with a groupware flavor, are the topic of a current dissertation project at Yale.

Research questions posed by the Linda model itself have inspired a variety of projects at other institutions, dealing (among other topics) with Linda in the context of a variety of computing languages, with object-oriented Lindas and with formal characteristics of the model and of Linda applications. There is a substantial literature on these topics. Recently, evidence has appeared that the tuple space model is influencing other coordination paradigms as well (see for example Agha [AC93] or Liskov [Lis92]).

3.2 Expressivity on real applications

Applications exist using C-Linda and Fortran-Linda in a wide variety of domains. Many of these have been reported only informally or not at all—they are production applications developed routinely by users of the commercial system. But applications in a number of areas have been described in the literature: ray tracing [MM91, BKS91], financial analytics [NB92, CCZ93, Cag93], realtime data fusion [FGK+91], seismic applications [BS92], probabilistic fatigue analysis [SLSC93] and the Level-3 BLAS [GS92], among others.

In this section we take up a few simple examples, designed particularly to show "idiomatic" rather than (as in the previous section) "message passing style" Linda. *Idiomatic* Linda makes use of shared objects and distributed data structures.

Shared Objects: Assigning tasks... X-PLOR[Brü92] is a widely-used application developed by Axel Brünger of the Yale Molecular Biophysics and Biochemistry Department. We parallelized a subroutine of X-PLOR that is invoked by the main code at each of a series of time steps. The subroutine is a good target for parallelization because it represents a major part of X-PLOR's computational cost.

This subroutine "(`ENBRD`") is structured as an n-body-style computation over a collection of atoms. A doubly nested loop computes displacement, forces and energies resulting from atomic interactions. The outer loop runs over the whole list of atoms that comprise the molecule under study; the inner loop runs over only those atoms that interact with the current "outer loop" atom. Scalars are used to collect aggregate energy values, and three vectors (indexed by atom id) are used to collect the aggregate force on each atom. The conformation of the molecule—the position of each atom—evolves as the computation proceeds, but atomic positions are updated not continuously but only at the close of each iteration. It follows that, for a given invocation of the routine, the interaction computations based on each atom are independent of all the rest.

We parallelized the routine using the "owner computes" approach that has in the past been associated with synchronous, data parallel languages. (This approach proves to be just as valuable using a general-purpose coordination language like Linda as in the context of special-purpose languages targeted at synchronous data parallelism [CG93].) The general method is as follows. Each process stores a complete description of the molecule under study. Each process "owns" certain atoms in the molecule, and performs interaction computations for the atoms it owns.

Here's what makes the problem interesting: all atoms are not created equal. Computations based on some atoms are more time consuming than others, because some atoms interact with more of the surrounding molecule than others do. So it clearly will *not* be acceptable to perform a uniform, static partitioning of atoms over processes. More interesting still: the conformation of the molecule changes as

the program runs. Hence, the workload on each atom changes. An atom that is relatively "easy" at one point in the computation may become hard later on. In short, the initial partitioning must reflect the character of the input data, *and* that initial partitioning must *itself* evolve as the application runs. Clearly, some sort of dynamic load balancing will be required; but load balancing must be carried out in a sensible way. The conformation of the molecule may change slowly. Load-rebalancing should be carried out only when the conformation has changed enough to make the overhead of rebalancing worth the trouble.

Linda's tuple space provides abstractions that make it easy to accomplish these goals.

In order to accomplish the dynamic partitioning of atoms, processes consult a tuple. They grab the tuple using `in`. The first element of the tuple is an string serving as an identifier; the second element holds some atom's id value:

```
IN('brd filt atom id', ? NEWID)
```

(Syntactic conventions reflect the fact that this code is in Fortran-Linda.) When this operation completes, some atom's id value will have been assigned to a local variable called `NEWID` belonging to this process.

The process now "owns" a series of atoms beginning with that id. (The number of atoms it owns—the "chunk size"—is adjustable. A chunk size of 1 conduces to the finest-granularity load balance, but by handing out atoms in chunks instead of one-at-a-time, we can still achieve a good balance and at the same time hold down the overhead of the load balancing operation itself.) The tuple is now replaced in tuple space, with its atomic-id value duly incremented by the chunk size:

```
OUT('brd filt atom id', NEWID+CHUNK)
```

The process now goes to work computing interactions based on the atoms it has acquired. When it's finished, it grabs the atom-assignment tuple again and acquires another chunk. A process that has been assigned (by the luck of the draw) an easy-to-compute chunk will grab a second chunk while unluckier fellow processes are still hard at work on their first, hard-to-compute chunks.

The shared counter tuple makes a simple but important point: when you want a shared variable, it's nice to have a shared variable. (Not that it's *essential*—of course this same application could be programmed using a message passing system. The point hinges rather on aptness and convenience, on closeness of fit between the programming model and the programmer's way of thinking.) Linda applications often use shared objects or data structures to carry out dynamic task distribution. X-PLOR is a particularly simple example; the owner-computes character of this code makes it possible to assign a task merely by (in effect) pointing to it. Other codes use distributed data structures whose elements are task descriptors. Sometimes an unordered bag is the right structure; sometimes tasks must be started in order and some sort of ordered structure is called for, for example a stream of tuples. These and many other cases are discussed in [CG90].

At the end of the iteration, processes circulate a table in which the results of the iteration are accumulated. This table-circulation operation is in essence identical to the circulating token example discussed above. Processes use `in` to grab the table from the preceding process:

```
      IN('brd filt merge data', WID, ?DXL:, ?DYL:, ?DZL:,
     +        ?LEVDW, ?LELEC, ?LEVDWV, ?LELECV, ?MINTIM,
     +        ?MAXTIM)
```

and `out` to pass it on.

```
      OUT('brd filt merge data', NEXT, DX:NATOM, DY:NATOM,
     +        DZ:NATOM, EVDW, ELEC, EVDWV, ELECV, MINTIM, MAXTIM)
```

This operation could of course (as the previous example makes clear) be accomplished using message passing. But it can be carried out more straightforwardly in Linda; more important, it can be carried out *using the same operations that also support shared variables.* Sometimes shared variables are right and sometimes message passing is right; Linda easily supports both, using one model and one set of simple operations. We regard this as, in most cases, clearly preferable to a set of operations that support *only* message passing, or (worse) a coordination system that includes two entirely separate coordination mechanisms.

The table represented by the "merge data" tuple serves as the basis for adaptive rebalancing. Each process keeps track of the amount

of time each iteration requires. The circulating table records the minimum and maximum times required for the iteration previous to the just-completed one. When those times differ by more than a predetermined percentage, the next iteration starts out with a rebalancing: each process goes back to the atom-assignment tuple and draws a fresh load of chunks.

Although our main focus in this paper is Linda and message passing, this example makes a point worth considering with respect to the relationship between both Linda and message passing on the one hand and parallel languages based on data-partitioning on the other. These languages, particularly High Performance Fortran, have emerged as the main software focus of those high-performance computing efforts that focus on massively parallel processors ("big iron") versus networked clusters. Data-partitioning operations are unquestionably valuable and convenient in some contexts. In others, they are inappropriate. The routine discussed here would *seem* to be a perfect fit to data-partitioning languages; after all, it uses data-partitioning to control parallel execution. But note that the data-partitioning required was easily accomplished in Linda. More important, Linda provided the flexibility that was essential to achieve the "multi-way adaptive partitioning" upon which the application depends. Mere data-partitioning alone is clearly *not sufficient*; the flexibility provided by systems like Linda or message passing is essential.

Although parallel X-PLOR is an ongoing project in the early stages, preliminary results suggest that the code described here performs well [CG93].

...and storing data We take up one more simple example of the way real Linda programs use shared objects. One major use for such objects is in task distribution; another is in storage of state information that characterizes and ongoing application. "LINKMAP" [Ott91] is another owner-computes application discussed in [CG93]. The program helps determine the location of a gene responsible for some genetic trait, given inheritance information pertaining to this trait and others of known location.

Our intent here isn't to describe the algorithm or parallelization method, but merely to draw attention to another typical Linda programming pattern. Information under development by many parallel processes must be adjusted relative to information characterizing one

end-point of the genetic interval in question. This information is stored in a tuple (using `out`), and consulted by interested parties using the tuple read operation, `rd`. Thus the operation

```
out("like table", thisped, like);
```

creates a shared object. The value bound to the variable `like` constitutes the actual datum; the first field is an identification string, and the second one allows processes to locate the right entry in a table. The operation

```
rd("like table", thisped, ? *like_p);
```

copies the value of `like` into the corresponding local variable. Note that both the first and second fields are used for matching ("`thisped`" means "this pedigree").

Again, more complex examples exist (see [CG90]). But this simple one captures the basic point: shared data structures are a useful programming technique, and Linda applications rely on them routinely.

4 Linda Performance

A great deal of Linda performance data has been published. Each of the "real application" reports cited above includes performance data, and much other data has appeared besides. These reports show that Linda is an efficient tool for the particular applications under discussion. But in the current context, performance *comparisons* are particularly interesting, particularly comparisons between Linda and message-passing systems. In this section we review two such studies.

Deshpande and Schultz [DS92] compare the performance of Linda and the native Intel message-passing library on distributed-memory Intel multiprocessors, and of Linda and PVM on workstation LANs, on the shallow water equations—an application they describe as "representative of the types of problems that researchers in several disciplines are attempting to solve." On 64 nodes of the multiprocessor, the Linda version's performance is within roughly 10% of the native low-level version. On the network, Linda and PVM are closely comparable.

We focus here on another study, conducted by Clemens Cap and Volker Strumpen of the University of Zurich in 1992. Having discussed

PVM and Linda with respect to expressivity, this case study allows us to cite some data, reported neither by the Linda nor the PVM groups, in which the performance of the two systems is compared on the same substantial problem. And this case study focusses, again, on a *message-passing style* program. Again, in other words, we are attempting to meet some prominent competing systems on their own turf.

4.1 The Cap and Strumpen study

In [CS92], Cap and Strumpen investigate network parallelism in the context of a PDE problem (heat conduction). The focus of their report is Parform, a system of their own design. The Parform project addresses an important aspect of local area networks in the role of parallel-machine-of-the-masses: the project centers on the realization of sensor-data-driven dynamic adaptability. These ideas were the inspiration for some of work reported in the X-PLOR example. Furthermore, Cap and Strumpen conveniently chose to present Parform performance data along with data on Linda and PVM.

Cap and Strumpen's paper describes the Parform system as "a new and optimized design, aiming primarily at high performance for special applications" [p.2]. Accordingly their intent is, among other things, to show that the special features of Parform make it particularly well-suited to the sort of "special applications" in which they are interested. They attempt to explain why (as one would expect) Parform, the special-purpose system, is more expressive for this sort of problem than Linda, the general-purpose alternative.

Cap and Strumpen's interests center on experimenting with a variety of strategies for load balancing, ranging from a simple fixed-sized static partitioning of the work, to static but variable partitionings, to a dynamically reconfigured partitioning.

4.2 The performance data

Cap and Strumpen's second concern was performance[1]:

[1]Cap and Strumpen re-examined this issue as a result of exchanges with members of our group. As we will see, they hold a decidedly different view now. A new version of their report has been issued[CS93].

> The main bottleneck of [Linda] in a distributed environment is the concept of tuple space, especially the necessary scanning operation to find tuples of certain formats.[page 4]

and later

> The poor performance of the [Linda] implementation POSYBL [POSYBL is one public-domain implementation of a portion of the Linda model] can thoroughly be explained by the overhead of tuple space management. With 10 to 20 processors the speedup essentially remains constant. Although work on high speed tuple space implementations is in progress, tuple space management is a principle bottleneck of the [Linda] approach, rather suited for shared memory architectures than for distributed systems.[page 12]

These views are paradigmatic of a widely held assessment: (1) tuple matching must be expensive, and (2) in a distributed environment, managing tuple space must be a bottleneck. Both assertions are in general false. Cap and Strumpen's own data provide an excellent illustration of the fact. As their statement indicates, their initial "Linda" data was gathered using a Linda variant called POSYBL [Sch91]. This system is completely *unoptimized.* Cap and Strumpen's initial data set amply reflects the consequences of using such a naive system (table 1; see [CS92] for a complete description of the experiment).

To explain why optimizations are particularly important in this case, we must first describe the program in a bit more detail. The problem being solved, a heat conduction PDE on a grid, was parallelized by decomposing the grid into strips. Boundary values must be communicated between adjacent strips. Mapping strips to processes yields a chain-shaped logical process structure. The chain results in a stable communication pattern that is captured by runtime heuristics designed to detect these patterns. Communication within the system is reconfigured in order to support them efficiently. In the event, actual message traffic is virtually the same as if the program had been written using a message-passing paradigm.

We arranged with Cap and Strumpen to retest using SCA's Linda. They did, and reported the results to the "comp.parallel" bulletin

Heat Conduction Timings (POSYBL)

Processors	POSYBL	PVM	Parform
1	1370.6	1370.6	1370.6
2	737.2	648.0	654.8
4	442.6	328.0	332.3
6	339.3	219.0	221.7
8	284.6	168.4	170.2
10	260.2	143.6	137.4
12	244.7	116.6	116.0
14	242.7	100.1	103.5
16	239.5	90.0	89.0
18	242.6	97.5	80.9
20	241.6	85.8	73.5

Table 1: Cap and Strumpen's original data

board on the internet. We reproduce the table for Linda, PVM and Parform (static, homogeneous partitioning) in table 2.

We leave the summary to Cap and Strumpen:

> This result verifies, that SCA Linda's runtime system can handle regular problems like a PDE solver very well. Certainly, a heap of work went into the optimizers of this system.

5 Adaptive Parallelism and Piranha

"Adaptive parallelism" refers to parallel computations on a dynamically changing set of processors: processors may join or withdraw from the computation as it proceeds.

Any parallel application might in principle gain if it is built as an adaptive program. In any computing environment (including a dedicated multiprocessor), such a program is capable of taking advantage of new resources as they become available, and of gracefully accommodating diminished resources without aborting. An adaptive parallel program might grow or shrink within a single multiprocessor, or encompass processors both within a multiprocessor and in a LAN.

Heat Conduction Timings (SCA-Linda)

Processors	SCA-Linda	PVM	Parform
1	1370.6	1370.6	1370.6
2	662.2	648.0	654.8
4	342.6	328.0	332.3
6	235.5	219.0	221.7
8	175.8	168.4	170.2
10	144.3	143.6	137.4
12	122.1	116.6	116.0
14	104.5	100.1	103.5
16	92.8	90.0	89.0
18	84.5	97.5	80.9
20	76.0	85.8	73.5
22	71.5	68.5	67.5
24	66.5	63.6	62.5
26	63.1	60.5	58.6
28	58.5	56.7	55.8
30	55.1	53.5	53.0
32	54.0	54.0	51.0
34	52.4	54.0	50.8
36	51.4	52.0	48.5
38	51.3	54.0	48.4

Table 2: Cap and Strumpen's revised data, using Scientific's Linda system

Workstation networks are the most important setting for adaptive parallelism at the moment. Workstations at most sites tend to be idle for significant fractions of the day, and those idle cycles may constitute a powerful computing resource in the aggregate. Ongoing trends make "aggregate LAN waste" an even more attractive target for recycling: desktop machines continue to grow in power; better interconnects will make communication cheaper, and in doing so expand the universe of parallel applications capable of running well in these environments. For these reasons and others, we believe that adaptive parallelism is assured of playing an increasingly important role in parallel systems and applications development over the next few years.

Several *ad hoc* systems have been designed to solve specific computational tasks adaptively—for example testing primality or computing travelling salesman tours [LM90]. Other approaches to adaptive parallelism have tended to center on what we call the "process model," in which an application is structure in terms of a set of *processes* that are dynamically remapped among free processors: when a processor withdraws, its processes are migrated somewhere else. Such an approach was discussed as long ago as the "MuNet" project and the early stages of Actors research [Agh86]; a variant of this approach formed the basis of the "Amber" adaptive parallelism system [CAL+89].

Piranha, in contrast, is an adaptive version of master-worker parallelism (see [CG90]). Programmers specify in effect a single general purpose "worker function." They do not create processes and their applications do not rely on any particular number of active processes. When a processor becomes available, a new process executing the "piranha" function is created there; when a processor withdraws, a special "retreat" function (which must be provided by the programmer) is invoked, and the local piranha process is destroyed. Thus, there are no "create process" operations in the user's program, and the number of participating *processes* (and not merely processors) varies dynamically.

This approach to adaptive parallelism has a number of advantages. Processes need never be moved around. The approach supports strong heterogeneity: *task descriptors* and not *tasks* are the basic movable, re-mappable unit in the computation. Task descriptors are stored in tuple space, and tuple space is implemented by the underlying Linda system in a heterogeneous way, allowing nodes of different types to share access to a single tuple space. A process image representing a

task in mid-computation can't realistically be moved to a machine of a different type, but a task descriptor can be. In the Piranha system, a task begun by a Sun node can be picked up and completed by an IBM machine. (Industrial harmony in action.)

Our approach has the disadvantage that applications must spell out explicitly what steps to take when a process is forced to vacate a node. The process won't simply be frozen and moved; the programmer must supply a special exception handler (the "retreat" function) that deals with the situation.

In practice, a variety of applications has executed successfully under piranha, running the gamut from simple cases that are readily piranhified to complicated ones whose inter-task dependencies require rather complex "retreat" functions. The system has been available for several years in a succession of research versions at Yale; applications from the departments of physics, electrical engineering, mathematics, human genetics and computer science, among others, have executed successfully on waste cycles in the computer science department's LAN. The close relationship between the Piranha model and the underlying Linda system should be clear.

Scientific Computing Associates's latest Linda release is the first to incorporate Piranha support in a commercial product. At Yale, Piranha has been ported to the CM-5 distributed-memory multiprocessor, and research on this and related topics continues.

We conclude by affirming one of the few unambiguous lessons of the history of programming languages: no one model will "win". Our goal is not to suggest that message passing is made obsolete by Linda. On the other hand, efficient high-level languages have historically faced challenges establishing themselves, but once established they have been widely embraced. Our goal is to suggest the relationship of Linda to message passing reflects this general pattern. That is, the relationship between Linda and message passing is similar, say, to that between efficient high level languages and assembler code. Assembly code is not obsolete, and assembly coding remains an important art. However, counter intuitive as it may appear, there exists a broad range of computations that can be solved not only more easily but with comparable efficiency using high-level languages. So it is with Linda and coordination.

References

[AC93] G. Agha and C. Callsen. Actorspaces: An open distributed programming paradigm. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, San Diego, May 1993. To Appear.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice-Hall, Hertfordshire, U.K., 1990.

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

[Brü92] A. T. Brünger. *X-PLOR, Version 3.1, A system for Crystallography and NMR.* Yale University Press, New Haven, 1992.

[BS92] J. L. Black and C. B. Su. Networked parallel seismic computing. In *24th Annual Offshore Technology Conference*, pages 169–176, Houston, TX, 1992. Paper Number OTC 6825.

[Cag93] L. D. Cagan. Investment analytics on networked workstations. *High Performance Computing Review*, May/June 1993.

[CAL+89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 147–158. ACM SIGOPS, ACM Press, December 1989.

[CCZ93] L. D. Cagan, N. J. Carriero, and S. A. Zenios. A computer network approach to pricing mortgage-backed securities. *Financial Analyst's Journal*, pages 55–62, March/April 1993.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, Apr. 1989.

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A first course.* MIT Press, Cambridge, 1990.

```
#include "pvm3.h"
#define SLAVENAME "slave1"

main()
{
  int mytid; /* my task id */
  int tids[32]; /* slave task ids */
  int n, nproc, i, who, msgtype;
  float data[100], result[32];

  /* enroll in pvm */
  mytid = pvm_mytid();

  /* start up slave tasks */
  puts("How many slave programs (1-32)?");
  scanf("%d", &nproc);

  pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);

  /* Begin User Program */
  n = 100;
  /* initialize_data( data, n ); */
  for( i=0 ; i<n ; i++ ) data[i] = 1;

  /* Broadcast initial data to slave tasks */
  pvm_initsend(PvmDataRaw);
  pvm_pkint(&nproc, 1, 1);
  pvm_pkint(tids, nproc, 1);
  pvm_pkint(&n, 1, 1);
  pvm_pkfloat(data, n, 1);
  pvm_mcast(tids, nproc, 0);

  /* Wait for results from slaves */
  msgtype = 5;
  for( i=0 ; i<nproc ; i++ ){
    pvm_recv( -1, msgtype );
    pvm_upkint( &who, 1, 1 );
    pvm_upkfloat( &result[who], 1, 1 );
    printf("I got %f from %d\n",result[who],who);
  }
  /* Program Finished exit PVM before stopping */
  pvm_exit();
}
```

Figure 1: PVM: master creates and sends data to a ring of workers, workers send data to the the next and previous workers in the ring, then return a result to the master. (Part 1 of 3)

```
#include <stdio.h>
#include "pvm3.h"

main()
{
  int mytid; /* my task id */
  int tids[32]; /* task ids   */
  int n, me, i, nproc, master, msgtype;
  float data[100], result;
  float work();

  /* enroll in pvm */
  mytid = pvm_mytid();

  /* Receive data from master */
  msgtype = 0;
  pvm_recv( -1, msgtype );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&n, 1, 1);
  pvm_upkfloat(data, n, 1);

  /* Determine which slave I am (0 -- nproc-1) */
  for( i=0; i<nproc ; i++ )
    if( mytid == tids[i] ){ me = i; break; }

  /* Do calculations with data */
  result = work( me, n, data, tids, nproc );

  /* Send result to master */
  pvm_initsend( PvmDataDefault );
  pvm_pkint( &me, 1, 1 );
  pvm_pkfloat( &result, 1, 1 );
  msgtype = 5;
  master = pvm_parent();
  pvm_send( master, msgtype );

  /* Program finished. Exit PVM before stopping */
  pvm_exit();
}
```

Figure 2: PVM: part 2 of 3

```
float
work(me, n, data, tids, nproc )
    int me, n, *tids, nproc;
    float *data;
{
  int i, dest;
  float psum = 0.0;
  float sum = 0.0;

  for( i=0 ; i<n ; i++ ) sum += me * data[i];

  /* illustrate node-to-node communication */
  pvm_initsend( PvmDataRaw );
  pvm_pkfloat( &sum, 1, 1 );
  dest = me+1;
  if( dest == nproc ) dest = 0;
  pvm_send( tids[dest], 22 );
  pvm_recv( -1, 22 );
  pvm_upkfloat( &psum, 1, 1 );

  return( sum+psum );
}
```

Figure 3: PVM version, completed.

```
float
work(id, n, data, nproc)
     int        id, n, nproc;
     float      *data;
{
  int   i, dest;
  float psum = 0.0;
  float sum = 0.0;

  for (i = 0; i < n; ++i) sum += id*data[i];

  /* illustrate node-to-node communication */
  dest = id+1;
  if( dest == nproc ) dest = 0;
  out("sum", dest, sum);
  in("sum", id, ? psum);

  return (sum+psum);
}

worker(id)
     int        id;
{
  int   n, nproc;
  float data[100];

  rd("init data", ? nproc, ? n, ? data);
  out("result", id, work(id, n, data, nproc));
}
```

Figure 4: Linda version, same problem: part 1 of 2.

```
real_main()
{
  int   i, j, n, nproc;
  float data[100], result;

  /* Start up workers. */
  puts("How many slave programs?");
  scanf("%d", &nproc);
  for (i = 0; i < nproc; ++i) eval("slave", worker(i));

  /* Begin User Program */
  n = 100;
  /* initialize_data( data, n ); */
  for (i = 0; i < n; ++i) data[i] = 1;

  out("init data", nproc, n, data);

  /* Collect results. */
  for (i = 0; i < nproc; ++i) {
    in("result", ?j, ? result);
    printf("I got %f from %d\n", result, j);
  }
}
```

Figure 5: Linda: the rest of the solution.

```
#define NPROC 4

#include <sys/types.h>
#include "pvm3.h"

main()
{
  int mytid; /* my task id */
  int tids[NPROC]; /* array of task id */
  int me; /* my process number */
  int i;

  /* enroll in pvm */
  mytid = pvm_mytid();

  /* find out if I am parent or child */
  tids[0] = pvm_parent();
  if( tids[0] < 0 ) { /* then I am the parent */
    tids[0] = mytid;
    me = 0;
    /* start up copies of myself */
    pvm_spawn("spmd", (char**)0, 0, "", NPROC-1, &tids[1]);

    /* multicast tids array to children */
    pvm_initsend( PvmDataDefault );
    pvm_pkint(tids, NPROC, 1);
    pvm_mcast(&tids[1], NPROC-1, 0);
  }
  else { /* I am a child */
    /* receive tids array */
    pvm_recv(tids[0], 0);
    pvm_upkint(tids, NPROC, 1);
    /* loop maps 'mytid' to 'me', a value in [0-(NPROC-1)] */
    for( i=1; i<NPROC ; i++ ) if( mytid == tids[i] ){ me = i; break; }
  }

  printf("me = %d mytid = %d\n",me,mytid);
  dowork( me, tids, NPROC );

  /* program finished exit pvm */
  pvm_exit();
  exit(1);
}
```

Figure 6: PVM: a ring of identical processes, with a single token circulated once around the ring; part 1 of 2

```
dowork( me, tids, nproc )
     int me;
     int *tids;
     int nproc;
{
  int token;
  int dest;
  int count  = 1;
  int stride = 1;
  int msgtag = 4;

  if( me == 0 ) {
    token = tids[0];
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &token, count, stride );
    pvm_send( tids[me+1], msgtag );
    pvm_recv( tids[nproc-1], msgtag );
    printf("token ring done\n");
  }
  else {
    pvm_recv( tids[me-1], msgtag );
    pvm_upkint( &token, count, stride );
    pvm_initsend( PvmDataDefault );
    pvm_pkint( &token, count, stride );
    dest = (me == nproc-1)? tids[0] : tids[me+1] ;
    pvm_send( dest, msgtag );
  }
}
```

Figure 7: PVM version, completed

```
#define NPROC 4

ring(id)
{
  if (id) in("token", id);
  out("token", (id+1)%NPROC);
  if (!id) in("token", id);
}

real_main()
{
  int i;

  for (i= 1; i < NPROC; ++i) eval("ring node", ring(i));

  ring(0);
}
```

Figure 8: Linda: same problem.