Efficient Parallel Programming with Linda*

ASHISH DESHPANDE AND MARTIN SCHULTZ

Department of Computer Science, Yale University, New Haven, CT 06520

ABSTRACT

Linda is a coordination language invented by David Gelernter at Yale University, which when combined with a computation language (like C) yields a high-level parallel programming language for MIMD machines. Linda is based on a virtual shared associative memory containing objects called tuples. Skeptics have long claimed that Linda programs could not be efficient on distributed memory architectures. In this paper, we address this claim by discussing C-Linda's performance in solving a particular scientific computing problem, the shallow water equations, and make comparisons with alternatives available on various shared and distributed memory parallel machines. © 1993 by John Wiley & Sons, Inc.

1 INTRODUCTION

Linda is a set of objects called tuples and a small number of powerful operations on those objects, which can be combined with a host language (like C or Fortran) to yield a high-level dialect for parallel programming on MIMD machines and local area networks (LANs). Linda has been discussed extensively in the literature and we shall assume some knowledge of Linda. We refer the reader to reports by Carriero and Gelernter [1–3] for further details. For the sake of completeness, we include a brief discussion of Linda in the Appendix.

It has long been the contention of many researchers that the overhead associated with management of the shared tuple space in Linda is too high to permit an efficient implementation on distributed memory machines even for relatively compute-intensive problems. To study this contention, we will present performance data for C-Linda on a variety of machines, in particular, on distributed memory machines. We also compare the performance to that obtained using traditional message passing systems.

We have taken a specific scientific computing problem (the shallow water equations, a model of atmospheric flow), which is relatively compute-intensive but is still representative of the types of problems that researchers in several disciplines are attempting to solve. Its communication structure is based only on nearest neighbor communication, making it ideally suited to message passing on hypercubes. Our message passing implementations of this problem have this optimal communication structure explicitly encoded in them and are able to take advantage of it. In contrast, the Linda program has no communication structure explicitly encoded in it to take advantage of the regularity of the problem. Nevertheless, as we shall see in subsequent sections, Linda appears to hold its own.

We will also investigate the performance and viability of a cluster of workstations connected together by a LAN as an environment for concurrent computing. Networks of workstations can be an

* This work is supported in part by ONR Grant No. N0014-91-J-1576, NSF/CER Grant No. DCR 8521451-5, and Sandia National Laboratories.

Received May 1992
Accepted December 1992

© 1993 by John Wiley & Sons, Inc.

Scientific Programming, Vol. 1, pp. 177–183 (1992)
CCC 1058-9244/93/020177-07

economical, practical, and powerful resource for parallel computing. They are already available to a large number of users. They offer considerable potential for fault tolerance. If a single workstation goes down, the remaining nodes can usually continue to function normally. This is often not true of tightly coupled multiprocessors where a failure in a single processing element results in the entire machine going down and the computation being suspended. Workstations also offer a user-friendly environment with familiar tools to aid in program development.

We have implemented and evaluated the performance of the Linda program on a variety of machines. We will present results for shared memory machines (Sequent Symmetry and the Encore Multimax), distributed memory machines (iPSC/2 and iPSC/860 hypercubes), and for a network of Sparcstations connected by an ethernet. The same Linda program was executed on all these machines and its performance was evaluated and compared to that of implementations using alternative methods available on all machines.

In our experience, the Linda program has generally been easier and more convenient to write than the native versions for each machine. Although this is a subjective issue, it is certainly true that the portability of the Linda version (as evidenced by our ability to run it efficiently on widely differing machines without having to change the code in any way) makes it easier and more convenient than writing a separate code for each machine under consideration. Obviously, there is a price to pay for this ease of use but we shall see that the loss in efficiency is relatively small on all the machines that we have considered.

2 THE SHALLOW WATER MODEL

The shallow water equations are a simplification of the primitive equations of atmospheric motion. They represent a simple and computationally efficient approximation to more accurate but more complicated models representing real atmospheric flow. However, they still involve most of the parallel algorithmic and programming issues exhibited by the more complex models. More generally, the parallel issues we consider are applicable to most explicit time marching schemes for time dependent (systems of) partial differential equations.

In Cartesian coordinates, the equations are of the form

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + \frac{\partial \phi}{\partial x} - fv = 0$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + \frac{\partial \phi}{\partial y} + fu = 0$$

$$\frac{\partial \phi}{\partial t} + \frac{\partial(\phi u)}{\partial x} + \frac{\partial(\phi v)}{\partial y} = 0$$

where u, v are the velocity components in the x- and y-directions, $\phi = gh$ is the free surface geopotential, g is the acceleration due to gravity, f is the Coriolis parameter, and h is the height of the fluid.

The numerical solution of these equations is usually required in a rectangular region $0 < x < L$ and $0 < y < D$ for $t > 0$. The boundary conditions are periodic in the x-direction

$$\mathbf{w}(x, y, t) = \mathbf{w}(x + L, y, t)$$
where $\mathbf{w} = (u, v, \phi)^T$ is the solution,

rigid wall in the y-direction

$$v(x, 0, t) = v(x, D, t) = 0,$$

and the initial condition is

$$\mathbf{w}(x, y, 0) = F(x, y).$$

We use the finite difference approximations described by Robert Sadourney [4], which have been used in previous studies of the same problem. We solve these equations numerically using simple, finite difference approximations:

$$\frac{\partial f}{\partial x} = \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x}$$

which can be derived from the Taylor series expansion for a function. This is a second-order approximation that results in an explicit time marching algorithm for solving the equations. Thus, given the values of u, v, and ϕ at t and $t + \Delta t$, we can calculate their values at $t + 2\Delta t$ at all grid points simultaneously. Washington and Parkinson [5] provide a detailed description of the algorithms.

3 LINDA IMPLEMENTATION

The Linda implementation of our problem relies on the production and consumption of tuples for flow control. As always, we would like to minimize

the amount of time spent by each process in communication. In the case of Linda programs, this means minimizing the number of accesses to tuple space and at the same time limiting the amount of data exchanged between the processes by minimizing the size of the tuples exchanged.

Tuples reside in the shared main memory on shared memory machines and are distributed across the local memories of the individual processors in a distributed memory machine.* There is a communication cost associated with performing a Linda operation on a tuple. This cost depends on the latency associated with the underlying communication medium as well as the size of the tuple.

If the latency is large, we would like to minimize the total number of accesses to tuple space even at the cost of communicating more bytes of data. This translates into dividing the grid into strips and allocating one strip to each processor. Each processor then needs to communicate two edges to its neighboring processors in every time step.

However, if the latency is small, we would like to minimize the total number of bytes communicated even though this may require more Linda operations. This translates into dividing the grid into tiles and allocating one tile to each processor. Each processor would then need to communicate data along the edges of the tile to eight neighboring processors.

Thus, in our implementation, each processor is given a portion of the domain (either a strip or a tile depending on the characteristics of the underlying communication medium). It is responsible for calculating velocity and pressure values for grid points in its domain. In each time step, the processor obtains the necessary boundary data from tuple space where it has been stored by neighboring processors in the previous time step. After doing the necessary computations, it outputs the boundary data to tuple space for neighboring processors to use in the next time step. The communication times for the various versions range from under 5% (for the shared memory versions) to about 30% (32 node Sparc network) of the total execution time.

It turns out that latency becomes the dominant factor only on a network of workstations connected together on an ethernet. Hence, all results presented are for tiles except for the network of workstations.

* In practice, the storage of tuples is optimized using a technique called distributed hashing. The system tries to store all tuples matching a particular template on the same node, which is determined by the hashing function and called the rendezvous node. Bjornson [6] provides more details.

4 RESULTS

The results of our experiments are presented in terms of the actual execution times of the parallel versions on various machines. In order to facilitate comparisons, we also present the speedup obtained for the distributed memory versions. We calculate speedup based on a sequential time, obtained from the execution time of an efficient C (not C-Linda) program for the same problem on the different machines and not from the execution time of the parallel C-Linda program on one processor. Memory restrictions do not permit us to solve large enough grids on a single processor. Hence, the sequential times used for speedup calculations are extrapolated from those for smaller grids. (The extrapolation was done by measuring the execution time for the sequential program for a number of grid sizes [from 25 × 25 to 256 × 256] and extrapolating from all the data.)

Our emphasis will be on distributed memory machines and LANs although we will present performance data for shared memory environments as well.

4.1 Shared Memory Machines

In Table 1 we show the execution times for the program using Linda (Version 2.4) on a 20-node Sequent Symmetry with 80386 processors and 80 MB of memory and an 18-node Encore Multimax with NS32332 processors and 64 MB of memory. The Linda program performs extremely well with

Table 1. Shared Memory Execution Time in Seconds—512 × 512, 200 Time Steps

Processors	Sequent Symmetry	Encore Multimax
1	14417.9	15338.0
2	7058.0	7566.1
4	3566.2	3807.2
6	2412.8	2584.5
8	1799.4	1934.9
10	1471.9	1577.9
12	1227.4	1320.5
14	1069.9	1137.9
16	944.5	999.1
18	872.4	917.8

Table 2. Hypercube Execution Time in Seconds–512 × 512, 200 Time Steps

Processors	iPSC/2 Linda	NX/2	iPSC/860 Linda	NX/860
1[a]	6841.4	6841.4	1060.6	1060.6
4	—	—	280.1	276.2
8	864.6	857.1	144.2	141.1
16	437.2	432.0	69.1	66.9
32	227.4	222.9	37.2	35.0
64	116.7	112.8	19.8	17.7
128	—	—	11.8	9.8

[a] Extrapolated.

efficiencies generally in excess of 90%. This performance is to be expected because Linda is essentially a shared memory model and tuple space and its associated operations can be implemented very efficiently on shared memory machines. However, the same Linda program can be recompiled and executed on distributed memory machines as we shall see in the following sections.

4.2 Distributed Memory Machines

The shallow water algorithm maps naturally to message passing between nearest neighbor processors. Hence, in essence, our implementation uses Linda as a high-level message passing system among worker processes. However, the Linda programmer does not have to worry about figuring out explicit destination processors for the messages: Linda's tuple matching process takes care of that for us. In addition, the Linda program can be developed and debugged in a more user-friendly environment such as a Sparcstation with better utilities for software development. As a result the Linda program is easier to code. As we have indicated earlier, there is a price to pay (in terms of performance degradation) for this ease of use.

In Table 2, we show the execution times for the Linda (Version 2.4) program on the iPSC/2 and the iPSC/860. We also show the times for the same problem using the native message passing primitives on both machines. Clearly, the Linda program demonstrates excellent efficiency when compared to the native message passing version. The difference in execution time between the two is about 10% for the 64-processor case. The difference is due to the overhead associated with Linda's management of tuple space in the distributed memory environment.

4.3 LANs

Networks of workstations connected by an ethernet are commonplace and have been widely available as an inexpensive, potential resource for parallelism. However, until recently, this potential computing power has not been exploited due to lack of appropriate parallel programming tools. Workstations are becoming increasingly powerful and affordable while the speed and bandwidth of LANs promise to grow in the near future. Finally, a number of parallel programming environments are becoming available for these machines. Hence, in our opinion, groups of workstations connected by high-speed LANs have emerged as a powerful, practical, and affordable resource for exploiting parallelism and Linda is emerging as a powerful programming environment for writing efficient and portable parallel programs for such LANs.

We investigated the performance of Network Linda (Version 2.4.6) and PVM (Version 2.3.2*) on a LAN of SUN Sparcstations. PVM (which stands for Parallel Virtual Machine) was developed at Oak Ridge National Labs, the University of Tennessee and at Emory University [7–9]. It is a direct, message passing system for networks of computers.

In our experiments, we used PVM and Linda programs that used the same computation module, in order to eliminate possible differences in compilation. The two programs differed only in the communication routines. The data presented are the average of several runs in each case. We also tried to ensure that the network was free of contention from other unrelated processes.

In Figure 1 we show the speedup obtained using Linda and PVM on a network of up to 28 SUN Sparcstations. The actual execution times are shown in the first two columns of Table 3. The measurements were made using the CPU time spent executing the program as a measure of execution time. As the graph shows, both the Linda and PVM programs demonstrated excellent and practically identical performance on the network.

This was a surprising and unexpected result, because our experience with both the Linda and PVM versions suggested that both took somewhat

* We tried Version 2.4.1 but the timing data was very inconsistent, and we were unable to get any meaningful results. In several cases, the execution time using Version 2.4.1 was greater than that using Version 2.3.2. We have notified the authors and an investigation is currently in progress.

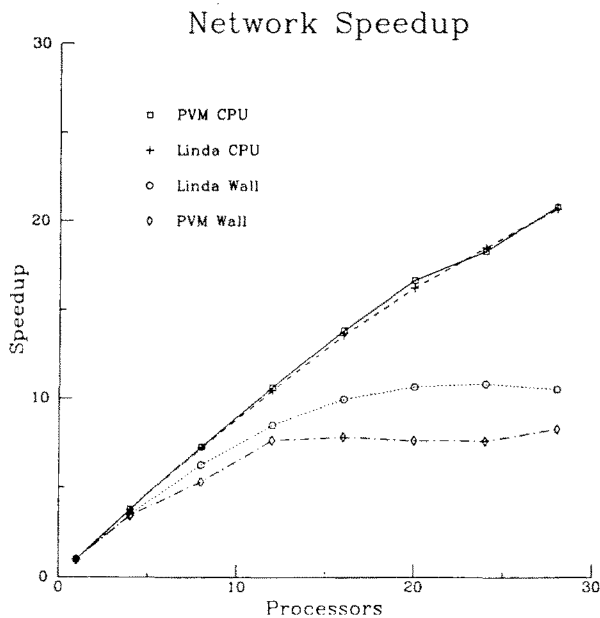


FIGURE 1 Speedup over a LAN. 512 × 512, 200 time steps.

Table 3. Network Execution Time in Seconds—512 × 512, 200 Time Steps

Processors	Linda CPU	PVM CPU	Linda Wall	PVM Wall
1[a]	3330.9	3330.9	3351.8	3351.8
4	885.6	884.1	966.2	976.0
8	458.3	454.8	531.6	627.4
12	318.5	313.1	393.2	436.5
16	245.1	240.2	336.4	426.2
20	204.9	199.7	314.4	437.3
24	180.0	182.1	309.1	437.5
28	161.2	160.2	318.6	403.7

[a] Extrapolated.

longer in practice than the CPU timings suggest. Moreover, the CPU data implies that adding more and more workstations will continue to yield improved performance. However, the network is a shared resource and adding workstations (thereby increasing the amount of communication) should eventually result in a significant degradation in efficiency due to collisions of transmissions from various stations.* This drop in network efficiency should show up in our performance data in the form of increased communication time and reduced efficiency.

Unfortunately, measuring CPU time neglects a significant amount of the overhead associated with communicating across the network. CPU time is the actual time spent by the system to execute the particular process being timed. This method of timing, in the case of both Linda and PVM, totally ignores the time spent by the system in managing traffic across the network. In the case of PVM, the CPU time further neglects the overhead associated with the PVM system's management of message transmits and receives, which is done by means of an additional process (called the PVM daemon) running on each node. This overhead is included in the CPU time for a Linda program because Linda bundles the tuple management code into the same process as the executing program.

In order to account for the extra overhead, we measured the performance in terms of the actual elapsed time between the start and completion of the process. This is the real performance that a user would see when executing a program on the network and is, hence, a better measure of performance for the LAN.

These execution times are shown in the last two columns of Table 3 for both Linda and PVM. The speedup obtained using these wall clock times is shown in Figure 1. Both systems suffer considerable degradation in performance in going from CPU to wall clock times. The data show the expected reduction in efficiency due to network collisions as the number of workstations is increased. The large reduction in speedup for both Linda and PVM is clearly due to taking all the overhead into account when measuring time. However, Linda clearly outperforms PVM in this regard. Apparently, having the various PVM processes communicate via an intermediate PVM daemon incurs significantly greater than that associated with Linda's management of tuple space. (A planned enhancement to PVM will attempt to reduce these effects.) Hence, a user trying to execute an application in parallel on a network of workstations is likely to see better performance with Linda than with PVM.

We note that the PVM system is primarily targeted at large-grain applications with several relatively independent components, each of which can run on an architecture best suited to its re-

* This is on account of the communication protocol used on the ethernet that essentially permits stations to transmit packets of data as soon as they are ready to do so unless the channel is busy. Hence, as the load on the network increases, the performance degrades rapidly.

quirements and communicate using the machine independent mechanisms provided by PVM. However, the authors of PVM do claim good performance for traditional parallel applications [8, 9].

5 CONCLUSIONS

A number of computer scientists have contended that Linda cannot possibly be implemented efficiently on distributed memory machines because there is simply too much overhead. They believe that Linda will not be able to compete with message passing on such machines even for solving compute-intensive problems.

We have used Linda to solve a real problem, similar to those that researchers in several fields are attempting to solve. The problem is relatively compute-intensive but, on hypercubes, has a communication structure based entirely on nearest neighbor communication, thus making it ideally suited to the message passing approach. This communication structure is explicitly encoded in the message passing solution that is able to fully exploit this regularity in the problem to produce a near-optimal solution as evidenced by the excellent speedup and efficiencies observed.

In contrast, the Linda solution has no explicit way to exploit this regularity in the communication structure of the algorithm. In fact, the Linda program was not even developed on a distributed memory machine. In spite of this, Linda performs fairly well and appears to hold its own.

Further, we ported the Linda program to a variety of architectures. It continues to demonstrate excellent performance in all these environments. In Figure 2 we detail the speedup obtained using Linda on several different machines. We see excellent performance on the tightly coupled shared memory and distributed memory machines. Further, Linda also performs creditably on a network of workstations where we see good efficiencies up to about 12–16 workstations, that is, until we saturate the network. Current LANs have limited communication bandwidth. But, as the bandwidth and speed of networks continue to increase, we believe that the Linda system will be able to efficiently support increasingly larger networks. We believe that Linda will continue to be a practical and powerful environment for exploiting this vast potential resource for parallelism.

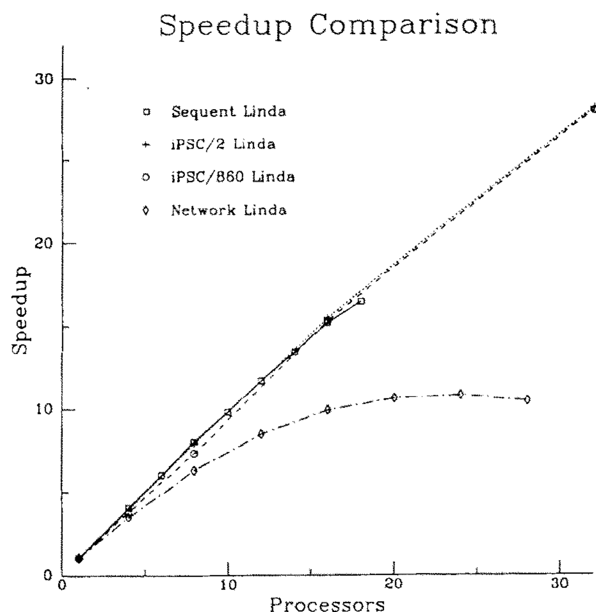


FIGURE 2 Speedup comparison across machines using Linda.

REFERENCES

[1] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the perplexed," *ACM Comput. Surv.*, vol. 21, pp. 323–357, 1989.

[2] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, pp. 444–458, 1989.

[3] D. Gelernter, "Generative communication in Linda," *ACM Trans. Programming Languages and Systems*, vol. 1, pp. 80–112, 1985.

[4] R. Sadourney, "The dynamics of the finite-difference models of the shallow water equations," *J. Atmospheric Sci.*, vol. 32, pp. 680–689, 1975.

[5] W. M. Washington and C. L. Parkinson, *An Introduction to Three-Dimensional Climate Modeling*. Mill Valley, CA: University Science Books, and New York: Oxford University Press, 1986, 422 pp.

[6] R. Bjornson, "Experience with Linda on the iPSC/2," Technical Report DCS/RR-520, Yale University, March 1989.

[7] A. Beguelin, J. J. Dongarra, A. Geist, B. Manchek, and V. Sunderam, "A user's guide to PVM parallel virtual machine," Technical Report TM-1126, ORNL, July 1991.

[8] G. A. Geist and V. S. Sunderam, "Network based concurrent computing on the PVM system," *Concurrency: Practice and Experience* (in press).

[9] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, pp. 315–339, 1990.

APPENDIX 1

Linda is a high-level coordination language, which may be coupled with any computation language like C or Fortran to provide a high-level dialect for parallel programming on MIMD machines. The Linda model is based on a shared associative memory known as tuple space, which consists of a collection of tuples. A tuple is just a collection of fields, each of which has a specific type. The types are drawn from those available in the host language. There are two kinds of tuples: active process tuples and passive data tuples. Process tuples execute concurrently and communicate with each other by reading, writing, and consuming data tuples from tuple space. When a process finishes executing, it turns into an ordinary data tuple. All communication is achieved by using four simple operations on tuple space. Individual processes are not aware of and do not care about how other processes do what they do. This uncoupled style makes it easier for programmers to write parallel programs because they do not have to worry about low-level details such as message destinations and explicit synchronization. Tuples can only be modified after extracting them from tuple space, thus providing an implicit locking mechanism.

There are four basic tuple space operations, **out**, **eval**, **in**, and **rd**. **out** (**t**) causes tuple **t** to be added to tuple space; the executing process continues immediately. **eval** (**t**) is the same as **out** (**t**) except that **t** is evaluated after rather than before it enters tuple space: **eval** implicitly forks a new process to perform the evaluation. **in** (**s**) causes some tuple **t** that matches the template **s** to be withdrawn from tuple space; the values of the actual fields in **t** are assigned to the formals in **s**, and the executing process continues. If no matching **t** is available, the process is suspended until one becomes available. If many **t**s are available, one is chosen arbitrarily. **rd** (**s**) is the same as **in** (**s**) except that the matching tuple **t** remains in tuple space.

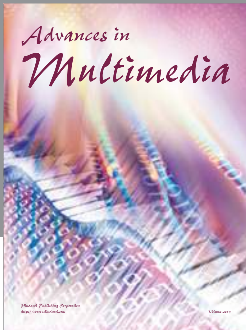
Tuples have no addresses; they are selected by **in** or **rd** on the basis of any combination of their field values. Thus the five-element tuple (A, B, C, D, E) may be referenced as "the five-element tuple whose first element is A," or as "the five-element tuple whose second element is B and fifth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

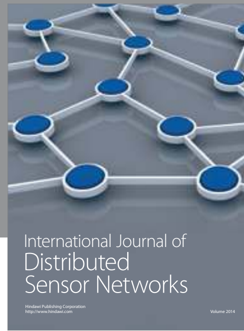
```
rd(A, ?w, ?x, ?y, ?z)
```

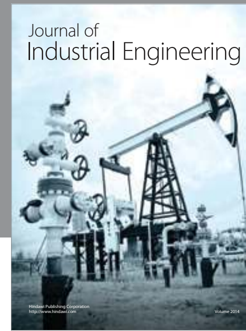
(this makes A an actual parameter—it must be matched against—and w through z formals, whose values will be filled in from the matched tuple). To read using the second description, we write

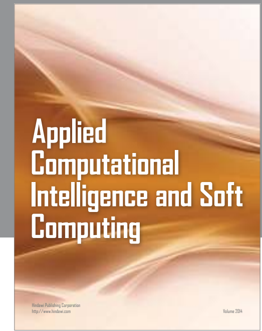
```
rd(?v, B, ?x, ?y, E)
```

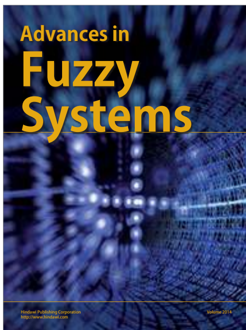
and so on. Associative matching is in fact more general than this: formal parameters (or "wild cards") may appear in tuples as well as match-templates, and matching is sensitive to the types as well as the values of tuple fields.











Hindawi

Submit your manuscripts at
http://www.hindawi.com