ELSEVIER

Parallel Computing 20 (1994) 633–655

PARALLEL COMPUTING

The Linda®† alternative to message-passing systems

Nicholas J. Carriero [a], David Gelernter [a], Timothy G. Mattson [b,‡], Andrew H. Sherman [c,*]

[a] *Department of Computer Science, Yale University, New Haven, CT 06520, USA*
[b] *Intel Supercomputer Systems Division, 14924 N.W. Greenbrier Parkway, Beaverton, OR 97006, USA*
[c] *Scientific Computing Associates, Inc. One Century Tower, 265 Church Street, New Haven, CT 06510-7010, USA*

(Received 17 May 1993; revised 25 November 1993)

Abstract

The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming. The principal argument against such an approach for portable software has always been that efficient implementations could not scale to massively-parallel, distributed memory machines. Now, however, there is growing evidence that it *is* possible to develop efficient and portable implementations of virtual shared memory models on scalable architectures. In this paper we discuss one particular example: Linda. After presenting an introduction to the Linda model, we focus on the expressiveness of the model, on techniques required to build efficient implementations, and on observed performance both on workstation networks and distributed-memory parallel machines. Finally, we conclude by briefly discussing the range of applications developed with Linda and Linda's suitability for the sorts of heterogeneous, dynamically-changing computational environments that are of growing significance.

Key words: Message passing; LINDA; Virtual shared memory; Evaluation; Parallel programming paradigm

1. Introduction

Most of the papers in this special issue deal with message-passing libraries. Message passing is a coordination model that arises directly from the architecture

* *Corresponding author.*

† Linda is a registered trademark of Scientific Computing Associates, Inc.

‡ Dr. Mattson participated in this work prior to joining Intel, while he was affiliated with both Yale University and Scientific Computing Associates, Inc.

SSDI 0167-8191(93)E0102-2

of networks and distributed-memory multiprocessors. Communication in such systems takes place via clumps of data, or 'messages,' sent from one address space to another – reflecting the fact that sending bits over a wire is the physical communication mode in those environments. An alternative model, coordination by means of a virtual shared memory, comes about in a very different way – it arises naturally from a familiar paradigm for writing parallel programs, namely, multiple processes interacting by means of shared data.

The use of distributed data structures in a logically-shared memory is a natural, readily-understood approach to parallel programming. After all, it represents a minimal extension of the underlying basis for sequential programming, and it has been used widely on shared-memory parallel hardware. The principal argument against such an approach for portable software has been that efficient implementations could not scale to massively-parallel, distributed memory machines. As a result, paradigms like message-passing were developed to cater to non-shared-memory architectures.

Now, however, there is growing evidence that it *is* possible to develop efficient and portable implementations of virtual shared memory models on scalable architectures. The example we discuss in this paper is Linda, specifically the commercial C- and Fortran-based systems available from Scientific Computing Associates, Inc. With Linda, programmers can develop programs that use a shared-memory model, are portable, and achieve high performance over a wide range of machines and networks, independent of whether the hardware itself provides any support for shared memory (either real or virtual).

In this paper we address a number of issues regarding Linda and contrast it with message passing. After presenting an introduction to the Linda model, we will focus mainly on expressivity of the model, on performance issues, and on flexibility and usability. Because of the great amount of interest in cluster or farm computing, particularly on heterogeneous collections of powerful RISC workstations, we frame much of our discussion in terms of a network setting. However, Linda is widely used on other architectures as well, so we also include some basic discussion of performance in such settings. Finally, in our concluding remarks, we briefly address two additional important issues that go to the heart of Linda's long term viability:

(1) Linda's impact in practice (that is, the range of applications developed with Linda); and
(2) Linda's adaptability (that is, its suitability for the sorts of heterogeneous, dynamically-changing computational environments that are of growing significance).

2. The Linda model

We begin our discussion by providing some basic background on Linda. Here and throughout this paper, we will consider only the commercial releases of C-Linda [24] and Fortran-Linda [25] from Scientific Computing Associates, Inc.

(SCIENTIFIC). In part, this restriction is due to some significant differences in syntax and semantics between SCIENTIFIC's language-level systems and other library-based implementations of Linda-like systems with which we are familiar. More importantly, though, the performance of any Linda implementation depends heavily on the levels of optimization achieved at compile-time and at run-time, and we have direct knowledge of the details only for SCIENTIFIC's systems.

As a language extension, Linda comprises a small number of powerful operations that may be integrated into a conventional base programming language, yielding a dialect that supports parallel programming. Thus, for example, C and Fortran with the addition of the Linda operations become the parallel programming languages C-Linda and Fortran-Linda. SCIENTIFIC's implementations translate from the Linda parallel language (C-Linda or Fortran-Linda) into the corresponding base language (C or Fortran), automatically generate required auxiliary routines, and incorporate optimized kernel libraries to support the Linda operations at run-time. Portability comes from the consistency of the language processing between systems, while efficiency comes from the use of native C and Fortran compilers for the actual generation of object code, and from hardware-specific implementations of the kernels. Commercial versions of Linda now run well on a broad range of parallel computers, from shared-memory multiprocessors, to distributed-memory machines such as hypercubes, to networks of workstations. Since Linda has been discussed at length in the literature (e.g. [2,3,11,13]), we provide a relatively brief description here.

Linda-based languages and their associated run-time systems provide support for the Linda programming model, which is a *memory* model based on a virtual, associative, logically-shared memory called *tuple space*. Tuple space contains a collection of ordered sequences of data called *tuples*. Each field of a tuple contains actual data in the form of one of the valid types (including aggregates like arrays, structures, and, in the case of Fortran-Linda, common blocks) of the base programming language. For example, the following are valid tuples in Fortran-Linda: [1]

```
('comment string', 1, 12, 4.99)
('logical data', .FALSE.)
('array data', [1 3 5 7])
```

Tuples may be created and inserted into tuple space in two different ways: serially or in parallel. Serial creation is accomplished using Linda's `out` operation. For example, assuming that `a` is a `REAL` one-dimensional array and that `f( )` is a `REAL`-valued Fortran function subprogram, then the following `out` operations both produce (different) 4-field tuples:

```
out('north', i, j, a)
```

[1] When necessary for clarity, we use the notation '[...]' to delimit aggregate tuple fields. Such fields always differ from sequences of single-valued fields. We also note that use of a string as the first field is purely stylistic, though it may sometimes permit better discrimination of tuple classes during optimization and thus lead to better performance.

or

```
out('table entry', i, j, f(i,j))
```

When the `out` operation is used, all tuple fields (which appear as arguments to `out`) are completely evaluated serially by the process containing the `out`. Following evaluation, the resulting tuple is installed in tuple space as a passive data tuple, and the process continues with the next statement.

Linda's `eval` operation is used for parallel creation of tuples, as in

```
eval('table entry', i, f(i))
```

In principle, all arguments to `eval` are evaluated in parallel, in separate processes, while the original process continues immediately. For efficiency, however, most implementations actually evaluate fields serially, as with `out`, except for those fields containing procedure invocations. As in this example, the `eval` operation may cause the creation of new independent processes (here to evaluate `f(i)`), which run in parallel and accomplish work by creating, using, and consuming tuples. After all the arguments to `eval` have been completely evaluated, [2] the resulting tuple is installed in tuple space, just as with `out`. Once a tuple has been inserted into tuple space, the manner of its creation is irrelevant; identical tuples created by different Linda operations are indistinguishable.

Tuple space is an associative memory. Tuples have no addresses; they are selected for retrieval 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. Linda provides two basic operations to retrieve data from tuples in tuple space: `in` and `rd`. `in`(*s*) causes some tuple *t* that associatively matches the *template s* to be withdrawn from tuple space. `rd`(*s*) is identical to `in`(*s*), except that the matching tuple *t* remains in tuple space.

A template is a sequence of typed fields that may be either actual values (just as in tuples), or formal place-holders. Roughly, a tuple *t* matches the template *s* if both have the same number of fields, the types of the fields match pairwise, and each actual value in *s* matches the value in the corresponding field of *t*. If a suitable *t* is found, the values of the actuals in *t* are bound to the corresponding formals in *s*, and the invoking process continues. If no matching *t* is available when `in`(*s*) or `rd`(*s*) executes, the invoking process suspends until one becomes available, after which time it proceeds as before. If many matching *t*'s are available, one is chosen arbitrarily.

To read the five-element tuple (*A, B, C, D, E*) using the first description given above, one would write `rd`(*A*, `?w, ?x, ?y, ?z`). In this case *A* is an actual parameter to be matched against, and `w` through `z` are formals whose values will be filled in from the matched tuple. To read the same tuple using the second description given above, one would write `rd(?v,` *B*, `?x, ?y,` *E*).

[2] A procedure evaluates to its return value, or to INTEGER 0, in the case of Fortran subroutines.

3. Evaluating Linda

As noted in the Introduction, we want to discuss Linda's effectiveness as a programming model and to contrast its performance with that of message passing in a number of settings. One way to evaluate a programming model is to examine its expressiveness – what can be expressed, and how easily and concisely. We will address both of these topics: first, by examining Linda's ability to support a number of different parallel programming paradigms, and second, by looking at how it can be applied to a particular class of computations implemented frequently on scalable parallel architectures. Expressiveness by itself, of course, is not especially valuable if it comes with a substantial loss of efficiency. In order to address the common misconception that efficient implementations of virtual shared memories are unachievable, we next discuss some of the optimization techniques used to implement Linda systems efficiently. Finally, to confirm our assertion that Linda can be both expressive and efficient, we present a number of performance results both for networks and for arguably-scalable parallel architectures.

3.1 Support for parallel programming paradigms

Linda provides excellent support for a wide variety of approaches to parallel programming. One particular paradigm that has been used frequently with Linda is known as the *Master / Worker Model* [13]. Typically, this entails the use of distributed data structures and a group of worker processes (not necessarily identical) that examine and/or modify the data structures in parallel under the general supervision of a master process (which may, itself, do work as well). A great strength of the Linda model is its explicit support for distributed data structures, i.e. data structures that are uniformly and directly accessible to many processes simultaneously. Any tuple sitting in Linda tuple space meets this criterion: it is directly accessible – via the Linda operations described above – to any process using that tuple space. Thus, a single tuple constitutes a simple distributed data structure, but it is easy and often useful to build more complicated multi-tuple structures (arrays, queues, or tables, for example) as well. By comparison, message passing systems deal solely with transient data (messages) that exist for only a limited time: between assembly by the sender and disassembly by the receiver. Moreover, the messages are accessible only to two processes and at specific times: the sender before transmission and the receiver after transmission.

Another feature of the Linda model is its intentionally loose coupling among processes. Some other models implicitly or explicitly bind processes tightly together. Taken to an extreme, this gives data parallel or SIMD models in which all processes perform identical operations in lock step. Even message passing assumes that there is significant underlying synchronization between message senders and receivers. In contrast, Linda processes can be designed to know exactly as much about one another as is appropriate for the programming situation at hand. Since processes interact only through the intermediation of data stored in tuple space, programmers need not think in terms of any particular logical process architecture,

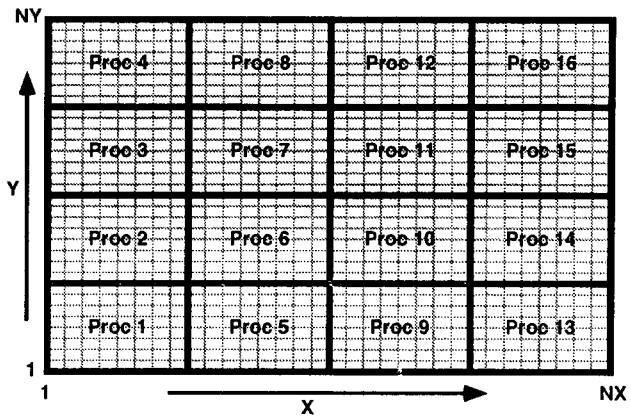

Fig. 1. Domain decomposition for rectangular region.

nor even in terms of simultaneously-executing processes. This simplifies greatly the potentially formidable task of parallel programming, since each individual process can be developed more-or-less independently of the others.

Linda's loose interprocess coupling has other advantages as well. Tuple space can be viewed as a long term data memory – once installed, tuples remain in tuple space until they are explicitly removed by some process. Thus, processes can interact through *time* as well as space (or machine location), since the producer and consumer of a tuple need never coexist simultaneously. A natural application of this idea arises when parallel computations produce output data that must later be used as inputs by completely independent visualization programs or other postprocessors. This sort of interaction is easy to express in Linda, but may well be extremely difficult to express using paradigms like message passing for which the 'data' (i.e. messages) have no long term existence.

The Linda model is extremely flexible and can support both static and dynamic load balancing strategies. Static strategies generally arise from fixed decompositions of large data structures into pieces that are assigned to and managed by the individual processes. For example, an array containing the values of some variable in every cell of a computational grid can be decomposed into non-overlapping pieces corresponding to subgrids, with each process taking responsibility for all computations involving one piece (see Fig. 1). If a process should require data from another process's piece, it then has to acquire that data from the owning process using some appropriate data sharing protocol. The efficiency of such an arrangement depends on a suitable decomposition in which relatively few data transfers are required between processes. In general, any repetitive computation can only be well balanced statically if the pieces owned by the different processes require roughly equal amounts of computation.

Implementing a static strategy in Linda is straightforward: each process stores its piece of the data locally, and data tuples containing subdomain boundary data are created when data sharing is required. This is efficient because most of the

data is in local (private) storage, and tuple space and Linda operations are used only for necessary data sharing. Since efficient Linda implementations exist for a wide variety of machines, software developers can easily build software that achieves high performance while still retaining portability.

There are many situations, however, where static load balancing strategies fail because it is impossible to create an even division of labor based on *a priori* analysis. Therefore, it is important that Linda can efficiently support dynamic load balancing strategies as well as static ones. One technique for doing so involves viewing tuple space as a 'bag' of tasks to be performed, with individual tuples holding the inputs for a single task. Processes can acquire one of these 'task tuples,' perform the required work, and create a new tuple containing the results. Load balance occurs almost automatically, even with heterogeneous processors, since processes that complete tasks quickly can complete several tasks in the time taken by other processes to complete just one. The key to the efficiency of this approach is that there need be no *a priori* assignment of tasks to processes; the Linda operations implicitly support the notion that processes can acquire task tuples exactly as rapidly as they are ready for them. Moreover, a simple extension of this idea, using an ordered task queue, rather than a bag, can yield good performance even in the presence of variably-sized tasks, for which it is important to perform larger tasks first.

To illustrate the simplicity and elegance of a Linda approach to dynamic load balancing, we examine a Fortran-Linda program fragment that implements a dynamically-balanced approach to the phase behavior computations arising in compositional petroleum reservoir simulation. In the program fragment shown in Fig. 2, it is assumed that each worker process is responsible for one subdomain of the computational mesh (as in Fig. 1, for example), and, as is often the case in practice, that the cost of solving the single-cell nonlinear phase behavior equations (in the routine `pvtcalc`) is at once both large and extremely variable, not only from grid cell to grid cell within a time step, but also from time step to time step for any given cell. To achieve dynamic load balance, each worker begins by creating a large number of task tuples containing the worker's logical task number, a local cell number, and the input data for that cell. After creating all of its task tuples, each worker enters a loop in which it grabs a task, carries out the necessary computation, and deposits a result tuple in tuple space. Finally, each worker retrieves the results for its own cells. Of course, it is possible to refine this approach in a variety of ways to improve performance, but the basic idea remains the same. For a full discussion of this particular application, see [27].

3.2 Expressiveness for a typical application

Linda has often been described as a highly expressive coordination model – one that lends itself to clear and concise programs. It's impossible to 'prove' such an assertion, of course, but one way to support it is to examine representative programming examples offered by developers of other systems, recode them in Linda, and compare. For example, Carriero and Gelernter [14,15] have done this

```
c
c  task tuple creation loop
c
      if (my_last_cell .eq. total_cells) out('cells left', total_cells)
      do i = my_first_cell, my_last_cell
        out('pvt data', my_proc_num, i, p(i), t(i), ... )
      enddo
c
c  grab tasks and do the computation
c
      do i = 1, total_cells
        in('cells left', ?count)
        out('cells left', count-1)
        if (count .le. 0) go to 10
        in('pvt data', ?proc_num, ?j, ?pressure, ?temp, ... )
        call pvtcalc( pressure, temp, ..., satl, satv, ... )
        out('pvt results', proc_num, j, satl, satv, ... )
      enddo
 10   continue
c
c  result collection loop
c
      do i = my_first_cell, my_last_cell
        in('pvt results', my_proc_num, ?j, ?satl, ?satv, ... )
        sl(j) = satl
        sv(j) = satv
      enddo
```

Fig. 2. Simple program for dynamic load balancing.

for a number of popular systems. In this section, we take a somewhat different approach in which we examine the advantages of using Linda to express a type of computation often implemented on scalable parallel architectures using message-passing systems.

The particular computation we consider here is the use of a simple explicit method for finite difference solution of a two-dimensional time-dependent parabolic partial differential equation on a rectangular domain. A standard approach to parallel implementation of such a method is based on the idea of *domain decomposition*: the computational domain is divided approximately evenly among a group of processors, each member of which is responsible for advancing the solution in its own subdomain. There are a number of ways to do this, and we focus on the use of subdomains that are two-dimensional tiles (see Fig. 1). Because of the local nature of finite difference approximations, the computations in each subdomain are independent of the others, except at the subdomain edges, where data must be exchanged with the processors responsible for neighboring subdomains. In general, the computation within a subdomain dominates the cost of data exchange across the subdomain boundaries, though that is certainly dependent on the size of the mesh, the number and kind of processors, and the medium through which data is exchanged.

The description of the computational process makes it sound like a natural application for message passing, and, indeed, there have been numerous implementations based on various message-passing paradigms. Deshpande and Schultz [17] discuss problems like this, comparing message-passing and Linda implementa-

tions developed in C and run on a variety of platforms. Here, however, we want to examine the ways in which Linda's expressivity can ease the implementation in Fortran of this type of finite difference computation.

Figs. 3a and 3b show fragments of a Fortran-Linda program that takes a specified number (*nts*) of time steps of an explicit Euler-type scheme to advance the solution $u(x, t)$ of the heat equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

with suitable boundary and initial conditions on a rectangular $nx \times ny$ computational mesh. For a timestep dt and a uniform mesh spacings dx and dy (in the x and y directions, respectively), the basic computational equation (ignoring the boundary conditions) is given by:

$$\begin{aligned} u(x, y, t+dt) = u(x, y, t) \\ &+ \frac{dt}{dx^2}\{u(x+dx, y, t) + u(x-dx, y, t) - 2u(x, y, t)\} \\ &+ \frac{dt}{dy^2}\{u(x, y+dy, t) + u(x, y-dy, t) - 2u(x, y, t)\} \end{aligned}$$

While this method is only useful in practice for restricted values of dt, dx, and dy, the programming issues involved in implementing it are similar to those arising from more advanced methods.

Our Fortran-Linda implementation makes use of a single master process and a number of identical worker processes. The master process handles overall setup and process management, invokes the workers, and participates in the computation by handling one of the mesh subdomains. Each worker process is responsible for a single subdomain, cooperating with other processes to exchange data along subdomain boundaries. For convenience in presentation, we have omitted most of the variable declarations and some of the less important code. We assume that variables whose names are entirely in capital letters (e.g. NXMAX) are specified in Fortran PARAMETER statements.

The master process fragment (Fig. 3(a)) begins by placing some global common-block data into tuple space for access by all processes using the statement:

```
out('parms common', /parms/)
```

Notice that Fortran-Linda permits the use of a common-block name to refer to the entire common block in Linda operations, essentially viewing the common block as a special datatype. This sort of syntax extension is possible with a language-level tool like Linda, but is impossible in systems based only on the use of communication libraries.

Next, the master creates the worker processes using Linda's `eval` operation. The arguments of the `eval` operation in this case include a reference to the `worker` subroutine, with arguments, thus causing the workers to begin execution at

the start of that subroutine. It is significant here that there is no artificial 'process id' required to set up the worker processes – each of the workers determines what it does and how it interacts with other processes based solely on the four arguments specifying the boundaries of its subdomain. In most message-passing systems, each process has some sort of id or handle, and it is essential that each worker know the ids of each neighboring process.

The Linda approach here has an important advantage in that the master can assign subdomains to the workers in any way it chooses, so long as the corners of adjacent subdomains match up properly. There is no requirement that regular subdivision be used, or that each worker even be able to identify its neighboring processes; each worker only needs to be able to describe the data it needs from

ⓐ

```
      subroutine real_main()
c
      common /parms/ cflx, cfly, nts
c
      dimension u(NXMAX,NYMAX)
c
      ⋮ (compute initial data, determine nxloc, nyloc, dt, dx, dy, nts, etc.)
c
c  set up compute processes
c
      cflx = dt / dx**2
      cfly = dt / dy**2
      out('parms common', /parms/)
      np = 0
      do ix = 1, nx, nxloc
        ixmin = ix
        ixmax = min(ix + nxloc - 1, nx)
        do iy = 1, ny, nyloc
          iymin = iy
          iymax = min(iy + nyloc - 1, ny)
          np = np + 1
          if (ixmax.lt.nx .or. iymax.lt.ny) then
            eval('worker', worker(ixmin, ixmax, iymin, iymax))
          endif
          out('initial data', ixmin, iymin, u(ixmin:ixmax, iymin:iymax))
        enddo
      enddo
c
c  do computation locally as well
c
      call worker(ixmin, ixmax, iymin, iymax)
c
c  collect results
c
      do i=1,np
        in('result id', ?ixmin, ?ixmax, ?iymin, ?iymax)
        in('result', ixmin, iymin, ?u(ixmin:ixmax,iymin:iymax))
      enddo
c
      ⋮
c
      return
      end
```

Fig. 3(a). Linda program fragment (master routine).

(b)

```
      subroutine worker(ixmin, ixmax, iymin, iymax)
      common /parms/ cflx, cfly, nts
      dimension uloc(NXLOCAL+2, NYLOCAL+2, 2)
c
      nxloc = ixmax - ixmin + 1
      nyloc = iymax - iymin + 1
c
      rd('parms common', ?/parms/)
      in('initial data', ixmin, iymin, ?uloc(2:nxloc+1, 2:nyloc+1, 1))
c
      ... Set edges of uloc to boundary values as appropriate
c
      iz = 1
      do it = 1, nts
        call step(ixmin, ixmax, iymin, iymax, NXLOCAL+2, nxloc, nyloc,
     1            iz, uloc(1, 1, iz), uloc(1, 1, 3-iz))
        iz = 3 - iz
      enddo
c
      out('result id', ixmin, ixmax, iymin, iymax)
      out('result', ixmin, iymin, uloc(2:nxloc+1, 2:nyloc+1, iz))
c
      return
      end

      subroutine step(ixmin, ixmax, iymin, iymax, nrows, nxloc, nyloc,
     1                iz, u1, u2)
c
      common /parms/ cflx, cfly, nts
      dimension u1(nrows, *), u2(nrows, *)
c
c  exchange boundary data
c
      if (ixmin.ne.1) out('west', iz, ixmin, iymin, u1(2, 2:nyloc+1))
      if (ixmax.ne.nx) out('east', iz, ixmax, iymin, u1(nxloc+1, 2:nyloc+1)
      if (iymax.ne.ny) out('north', iz, ixmin, iymax, u1(2:nxloc+1, nyloc+1))
      if (iymin.ne.1) out('south', iz, ixmin, iymin, u1(2:nxloc+1, 2))
      if (ixmin.ne.1) in('east', iz, ixmin-1, iymin, ?u1(1, 2:nyloc+1))
      if (ixmax.ne.nx) in('west', iz, ixmax+1, iymin, ?u1(nxloc+2, 2:nyloc+1))
      if (iymin.ne.1) in('north', iz, ixmin, iymin-1, ?u1(2:nxloc+1, 1))
      if (iymax.ne.ny) in('south', iz, ixmin, iymax+1, ?u1(2:nxloc+1, nyloc+2))
c
c  update solution
c
      do ix = 2, nxloc + 1
        do iy = 2, nyloc + 1
          u2(ix, iy) = u1(ix, iy)  +
     1                 cflx * (u1(ix+1, iy) + u1(ix-1, iy) - 2.*u1(ix, iy))  +
     2                 cfly * (u1(ix, iy+1) + u1(ix, iy-1) - 2.*u1(ix, iy))
        enddo
      enddo
c
      return
      end
```

Fig. 3(b). Linda program fragment (worker routines).

neighboring subdomains. In most message-passing systems, a more complicated approach must be taken (unless the system has special library calls applicable to exactly this sort of mesh problem). For example, the master might tell each worker

who its neighbors are (by sending the process ids after all processes are created). Alternatively, each worker might compute the logical process numbers of its neighbors (based on a regular subdivision of the mesh), and then use some sort of mapping function to find out the process ids. In either case, the programmer would have to deal with details of the parallel system having nothing whatever to do with the application.

One other more subtle point in the process creation loop bears comment. The Linda operation that sets up each process also passes arguments to the subprogram executed by that process, just as if the subprogram were invoked in an ordinary manner by a local calling routine. This can reduce the need for explicit data communication, of course, but it also means that the worker routine invoked remotely by means of `eval` can be identical to the worker routine invoked locally by the master. (In fact, it can exist in the same file as the master routine.) In message-passing systems, like PVM for example, the process creation mechanism actually invokes an entirely separate worker *program*, complete with some sort of main routine. This can make it more difficult to share code, and there will almost certainly be some code redundancy due to infrastructure repeated in the two programs.

Following the creation of each worker, the master `outs` the initial data for that worker's subdomain. Once again, we see repeated the theme of problem-related, not system-related, data identification. In this case, the initial data is identified not by the id of the receiving process, but by the indices of the mesh point at the lower left corner of the subdomain. Apart from the naturalness of this sort of data identification, there is the additional advantage that the data is 'self-describing' in a way that enables a programmer to identify the data in the context of the problem merely by examining it (possibly using Tuplescope™, SCIENTIFIC's visual debugger for Linda), without having to unpack and decipher it.

Moreover, the handling of the initial data illustrates two other ways in which the fact that Linda is based on language-level processing pays off with enhanced expressiveness and conciseness. First, there is no need to gather into a single contiguous space the diverse data to be placed in tuple space; Linda handles this automatically by generating suitable copying routines at compile and link times, based on the use of the most efficient, machine-dependent, copy operations available and targeting suitable machine-independent data formats (like XDR) if appropriate. This leads to a significant reduction in code size, means that the programmer is freed from the need to allocate and manage temporary space used solely for message buffers, and facilitates software portability.

The `out` operation for the initial data also illustrates another Fortran-Linda extension of basic Fortran 77 syntax: the ability to use Fortran 90 array index syntax inside of Linda operations. This permits very concise specification of a scattered subarray of the initial data matrix in a way that is impossible with library-based tools.

After it has created all the workers, the master process itself calls the worker subroutine to carry out computations on the upper-rightmost subdomain. When all computation is complete, the master collects the results in a loop over all the

subdomains. Notice that the collection loop can accept the results in random order, rather than by process number or in some specific subdomain order.

We now turn to the worker process illustrated in Fig. 3(b). The basic design for the worker process involves the use of two scratch arrays alternately to contain the old and new solution values. The first order of business for each worker is to copy the common block data and the initial solution data from tuple space. In this case, a `rd` operation is used for the common block, since the same tuple must be consulted by all workers. An `in` operation is used to retrieve the initial data and place it into one of the scratch arrays.

Following some initialization for those workers on the boundaries of the global mesh, each worker enters a loop over the number of timesteps to be computed. That loop invokes the computational routine, `step`, alternating the roles of the two scratch arrays as input and output arrays.

The `step` routine is quite simple, comprising a data exchange (communication) section and a local computation section. The data exchange section uses Linda `out` and `in` operations to place boundary data in tuple space and to retrieve the boundary data from neighboring subdomains. Notice once again that the data is self-describing, using index parameters natural in the context of the application. This contrasts with the need in message-passing systems to package each piece of the subdomain boundary into a message targeted for a specific processor. The computation section is, of course, quite straightforward, since it is identical to what would be included in any program for this application.

We haven't included here a message-passing variant of our program fragment. However, message-passing implementations by others of similar methods (e.g. [17]) are more complicated and require more lines of code to deal with parallel issues irrelevant to the underlying sequential computation. More importantly, the message-passing implementations tend to be less natural, in the sense that they make substantial explicit use of system/architecture information (such as process ids and the like) in addition to problem-specific information. Together these observations lead one to expect that the most frequent route to a parallel program – parallelization of an existing sequential program – should be much simpler with Linda than with message passing. This, in turn, should lead to significant cost savings due to reduced development time and increased software portability and reliability.

In comparing Linda and message-passing systems, we need to keep sight of the fact that these systems have radically different designs and goals. Message-passing systems like PVM have successfully achieved a specific, pragmatic goal: support for a well-designed, portable message passing service. Linda, on the other hand, is designed to play a more ambitious role – that of a high-level coordination language that is at the same time general, portable and efficient. Thus, within the parallel-programming domain, Linda is designed to support all of the basic paradigms of asynchronous parallelism, whereas message passing systems are well-suited to only some. In addition to programs of the type discussed in this section, Linda (as a language) supports many others, including those that rely heavily on distributed data structures (stored in tuple space), or that emerge from a result-driven dataflow paradigm. In practice, of course, physical communication

and process management may be too expensive on many current platforms or networks to fully exploit this expressiveness, but the capability is important because it means that Linda can serve as a unifying environment for today's parallel applications and the ones that will be available in the future within highly-optimized environments providing cheap communication and process management (cf. [20]).

3.3 Implementation efficiency

Linda is available on a wide range of parallel machines, including shared-memory multiprocessors, distributed-memory parallel computers, and heterogeneous collections of networked workstations. Each Linda implementation involves three basic components: a language-dependent precompiler, a link-time optimizer, and a machine-dependent run-time library, which fit together to provide efficiency in each of these environments.

As we noted earlier, the precompiler processes C-Linda or Fortran-Linda source code to produce pure C or Fortran modules in which the tuple space operations are replaced by calls to functions which will, in turn, invoke routines in the run-time library. (These intermediate functions are generated automatically during optimization at pre-link time.) The pure C or Fortran modules are then compiled using native compilers. In the course of this processing, the precompiler collects information about tuple space usage which is saved in a 'Linda object file' along with the base language (C or Fortran) object code.

At link-time, the Linda prelinker analyzes all the tuple-space accesses used in the complete program and 'fills in' the bodies of the intermediate functions mentioned above. In essence, these functions are quite simple: they create and fill suitable data structures for the tuple data and invoke an appropriate pre-existing library routine. Once the intermediate functions have been compiled by the Linda system using native compilers, the standard system linker is used to produce the final executable.

The prelinker is the real key to achieving run-time efficiency with Linda. One of its roles is to select the proper run-time library routines for the implementations of the tuple space operations. In SCIENTIFIC's systems, the run-time library is actually implemented as a polylibrary – that is, as a collection of families of run-time routines which can be used to implement different kinds of tuple space operations. While the Linda associative-matching protocol is very general, link-time analysis of data collected at compile time makes it possible to select the most appropriate member of the family applicable to each operation, thereby maximizing run-time efficiency while maintaining exactly the minimal required amount of generality.

In addition, the prelinker can use its complete knowledge of every tuple space access in the program to accomplish a good deal of 'proto-matching' at link time. A significant reduction in the cost of run-time searching can be achieved merely by exploiting the simple observation that an n-field template with a given type signature can only match an n-field tuple with matching types. However, it is

possible to do even more, for example, by throwing out consistently-used constant fields (in other words, 'pre-matching' them), establishing whether or not run-time value matching is needed and, if it is, determining whether there is a search key (for a hash table, for example). For more details, see [12].

The goal of the compile- and link-time processing of a Linda program is to convert a program containing very general Linda statements into an executable that will run as efficiently as if the program had been written with much less general process interaction constructs. In the case of programs designed for a message-passing process-interaction model (such as the example from the last section), this means that the resulting Linda program should run as fast (or nearly so) as a well-written 'natural' message-passing program. In addition to the optimizations already discussed, key to this efficiency are the run-time library routines.

Linda run-time libraries are targeted to specific host platforms. On shared-memory multiprocessors, tuple space is (naturally) mapped to physically-shared memory, and library routines use efficient shared-memory operations. In this paper, we want to focus mainly on Linda's use on distributed-memory machines and workstation networks, since those are the principal domains in which message passing is used. While several techniques have been used to implement tuple spaces in such environments, the most successful were developed by Bjornson and are described in [1].

In all distributed-memory Linda implementations, each processor acts both as a computational server (computing fields from `eval`s) and as a tuple space server, responsible for managing a particular disjoint section of tuple space. This distributes both the computational load and the burden of handling Linda operations across all participating processors, and it allows tuple space to be accessed simultaneously by many processes. As described above, the compile- and link-time systems divide tuples and templates into classes based on the number of fields, the type signature, etc. At run time, each class is mapped to some participating processing node, which provides storage for all tuples and templates in that class – in principle, each tuple or template is sent, upon generation, to the node associated with its class, and that node serves as a rendezvous point. Tuples wait there for matching templates, and *vice versa*. When a match occurs, the matched tuple is bundled off to the template's node of origin, and in that way a tuple generated by an `out` is delivered to a process that has executed a matching `in` or `rd`.

This basic scheme leads to the use of a three-message protocol for data movement at run-time, as shown in Fig. 4. When an `out` occurs, the computed tuple is sent to the responsible rendezvous node (Fig. 4(a)). For an `in` or `rd`, the node performing the operation sends the template to the rendezvous node and blocks awaiting a response. The rendezvous node sends the matched tuple back as soon as one is available (Fig. 4(b)).

A number of optimizations are used to make this basic scheme work well. First, some classes (those implemented as hash tables, for which there are search keys) may themselves be distributed over several nodes. Second, some tuple fields (in particular any large ones that play no role in tuple matching) aren't sent to the rendezvous node, but remain on their nodes of origin until needed by a recipient

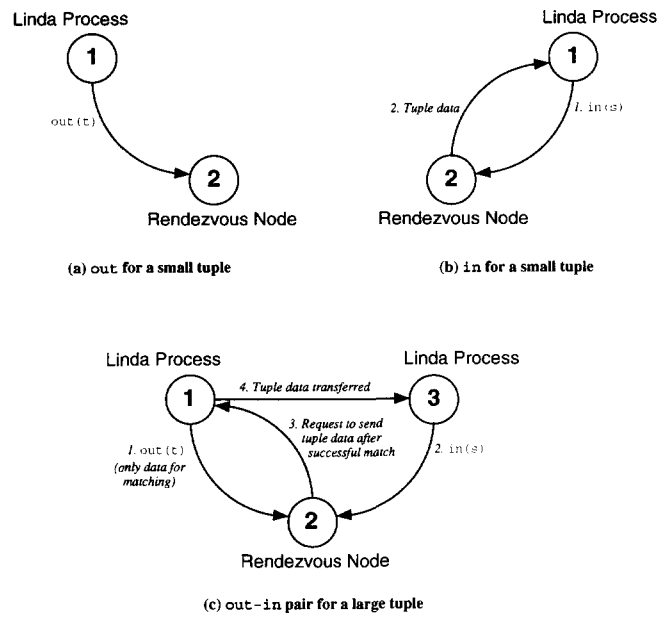

Fig. 4. Tuple processing in distributed-memory Linda implementations.

process (see Fig. 4(c)); this often substantially improves performance, since the large fields frequently consume most of the actual communication time. Third, broadcast operations may be used automatically by the run-time system to pre-distribute tuples accessed by `rd` operations (on the assumption that many nodes will access such tuples). Finally, the class-to-rendezvous-node mapping may change dynamically at run-time in response to observed tuple traffic. For example, if one process consistently requests tuples from a given class (or subclass, in the case of classes implemented using hash tables), the rendezvous node for that class is reassigned to the recipient process's node. This last optimization has an important consequence: Linda applications that behave like message-passing programs at run-time – those which involve very stable and predictable communication patterns – will *perform* like message-passing programs at run-time.

To conclude this discussion, we return briefly to the Fortran-Linda application of the last section. The real key to achieving high performance for that program is to make certain that the exchange of boundary data performs exactly like the message sending and receiving that it really is. Compile- and link-time analysis will lead Linda to use a distributed hash table implemention for the tuple classes used in the `out` and `in` operations of `step`. The hash keys will be derived from combinations of the non-constant fields that have actual values in both tuples and templates; in the cases at hand, these are the integer indices that are the second through fourth fields. At run time, the hash-table bins will be distributed initially at random, leading to a three-message communication protocol similar to that shown in Fig. 4. This is clearly less efficient than pure message passing, which

would involve only a single message. However, very rapidly (after three repetitions in current implementations), the run-time system will observe a static traffic pattern (since each hash key instance corresponds to data intended for a unique subdomain and, therefore, node). When this happens, those hash keys are removed from the standard hash table processing, and the corresponding rendezvous node is forced to the destination node. This leads to a single-message protocol. The dominant cost of satisfying the `in` requests then becomes the local memory-to-memory copy operations required to place the data into the proper output locations. This cost should be no worse for Linda than for message-passing systems (which must unpack the message buffers).

3.4 Performance results

We turn now to some performance evaluation of Linda, and a comparison between it and various message-passing systems. We emphasize that, although we provide quantitative results, one should draw only qualitative conclusions, since all of the systems discussed are undergoing constant improvement, and actual timings may depend somewhat on local hardware or software system configuration issues. Moreover, as we noted earlier, the Linda performance results described here represent only the performance of SCIENTIFIC's commercial Linda systems; other Linda-like systems may not use similar optimizations, so their performance may differ substantially.

In an absolute sense, Linda performance really depends on two separate issues: the effectiveness of the tuple classification strategies in reducing or eliminating expensive searches in tuple space, and the efficiency of the machine-dependent implementations of data transfer. We focus here mainly on the latter issue, though we note that Bjornson [1] has performed extensive studies quantifying the amount of searching. He concluded that over a number of different applications and problem sizes, Linda's compile- and link-time analysis and optimization were so effective that the cost of 'searching' was insignificant – in most cases, in fact, the first tuple examined (as a candidate match for a template) was the correct one. That tells us that, at least for SCIENTIFIC's systems, the quality of Linda performance will depend primarily on the degree to which the run-time system can avoid extra messages (relative to message-passing systems) and exploit the underlying low-level communication system.

To begin we will address performance on networks, comparing SCIENTIFIC's Network Linda System to PVM, a high-quality representative of portable message-passing systems. For a variety of reasons, including the issue of system evolution mentioned above, it is particularly difficult to assess quantitatively the relative performance of these two systems. For example, there are a number of algorithms (such as those involving dynamic load balancing, to name one class) that are easy to implement efficiently in Linda, but that may be difficult to implement in PVM. Moreover, PVM itself incorporates at least two different message-passing implementations (`pvm_send|pvm_recv, pvm_vsnd|pvm_vrcv`) that have differing performance characteristics, and it is unclear which one to use for fair comparisons.

Table 1
PVM/Linda communication time comparisons (ping-pong test)

Message size (bytes)	PVM time (msec)	Network Linda time (msec)
100	5.7	7.9
1,000	9.2	10.9
10,000	42.5	53.6
100,000	356.3	389.1
1,000,000	3,479.3	3,711.5

Noting all this, and bearing in mind that our goal here is to understand performance for algorithms to which Linda and PVM are equally applicable, we will present only results from two rather limited examples. Our comparisons used Version 2.5 of the Network Linda System and Version 2.4.0 and 2.4.1 of PVM, since those are the most widely distributed at the time of this writing. Somewhat newer versions now available might offer some quantitative improvements, but the qualitative conclusions would most likely stay the same. [3]

Our first example is a two-node 'ping-pong' program designed to study communication costs in Linda and PVM on a dedicated ethernet network. The ping-pong program contains two processes which pass a single message 'token' back and forth 100 times (that is, there are 200 actual messages). The fact that only two nodes are involved potentially hides significant issues related to scalability – for example, the program uses the `pvm_vsnd|pvm_vrcv` operations in PVM which may not scale to large networks due to Unix limitations on socket usage. However, the results do provide an indication of the underlying communication overheads in the two systems, independent of such hard-to-predict issues as network contention and the like. (We should note that the more scalable `pvm_send|pvm_recv` paradigm in PVM suffered a factor of two performance degradation over the `pvm_vsnd|pvm_vrcv` paradigm.)

The results in Table 1, originally reported by Douglas, Mattson, and Schultz [18], show the average time in milliseconds for a single roundtrip as a function of message size. They were obtained using Sun SPARCstation 1 workstations on an isolated ethernet network. The programs were written in C using version 2.5 of the Network Linda System and version 2.4.1 of PVM. In a sense, these numbers represent a worst case for the two systems, since there is no computation to dilute any of the communication cost. (Real programs that make sense to run on networks typically spend only a small amount of time communicating, so even a large increase in the pure communication time represents only a small portion of total wallclock execution time.) However, the results shown in Table 1 show that the Linda and PVM times are within 10–20% in most cases. [4]

[3] The current version of the Network Linda System is Version 2.5.2, and the current version of PVM is Version 3.1. Dr. A. Geist (private communication) has indicated that performance in the new version of PVM should be comparable to that reported here.

Table 2
PVM/Linda wallclock time comparisons (Cap & Strumpen data)

Number of processors	PVM wallclock time (sec)	Linda wallclock time (sec)
1	1,370.6	1,370.6
2	648.0	662.2
4	328.0	342.6
8	168.4	175.8
16	90.0	92.8
32	54.0	54.0
38	54.0	51.3

A more useful comparison between Linda and PVM on networks can be obtained by looking at the total wallclock execution time of specific applications. The particular application we'll examine is the solution of a heat conduction equation on a two-dimensional grid using strip-based domain decomposition – a computation similar to our earlier tile-based example, but using a somewhat simpler set of subdomains. Since the results represent work done by others, we will not focus on the details, but will instead present comparisons that indicate the roughly-equal performance of Linda and PVM. That is, after all, exactly what one would hope for – a high-level paradigm (Linda) performing to the same standard as a well-implemented message-passing system (PVM). The program was written by Cap and Strumpen [10], from whom we have obtained the data reported in Table 2. Their algorithmic approach was based on a data parallel scheme designed specifically to cater to some degree of heterogeneity in the networked workstations (in their case a mixture of various Sun SPARCstations). The reported results are wallclock times using C with PVM 2.4.0 and Network C-Linda version 2.5.0. [5] We see that the combination of Linda optimizations described above (particularly the run-time reassignment of rendezvous nodes) is capable of achieving Linda performance comparable to that of message passing systems.

The Cap and Strumpen example is not unique in the PDE area. Comparative results for the Shallow Water Equations (also using strip-based decomposition)

[4] For small message sizes, the difference seems to be due to some data management overhead and, to a lesser extent, Linda's need to send a few extra messages involving the rendezvous node. For large message sizes, the difference appears related to Linda's use of UDP and PVM's use of TCP. We note, however, that Douglas, Mattson, and Schultz [18] report that PVM performance deteriorates badly for large message sizes in other tests run on a four-node network.

[5] The times for 1–16 processors were obtained on a homogeneous SPARCstation 1 network. The 32-processor network included 23 SPARCstation 1 workstations, 8 SPARCstation 1+ workstations and one SPARCstation 2. The 38-processor network added 3 additional SPARCstation 2 workstations, two SPARCserver 490s, and a SPARCserver 390. Cap and Strumpen also compared PVM and Linda to their own special-purpose system PARFORM. Essentially, all three showed equal performance (within approximately 5%). The results using SCIENTIFIC's Network Linda System were obtained with the assistance of Mr. David Kaminsky and his colleagues at Yale University.

Table 3
EUI/Linda SP1 wallclock time comparisons (single token ring tests for 10 processors)

Token size (bytes)	Linda results		EUI results	
	Time (msec)	Throughput (Mb)	Time (msec)	Throughput (Mb)
1	1.58	0.00063	0.31	0.0032
4096	2.48	1.65	1.28	3.21
8192	3.51	2.34	1.92	4.28
16384	5.57	2.94	3.13	5.23
65536	14.69	4.46	10.18	6.44
262144	46.11	5.69	39.05	6.71
1048576	172.35	6.08	149.33	7.02

have been reported by Deshpande and Schultz [17], and they, too, found roughly similar performance for Linda and PVM versions on local area networks.

We and others have also looked at Linda performance on more scalable architectures, and we examine here some communication tests from recent work on the IBM 9076 SP1 computer, a new distributed memory machine that includes a scalable communication switch. SCIENTIFIC has developed a machine-specific version of Linda for the SP1 that is specifically designed to achieve high throughput for large tuples, and we report some preliminary results for two ring-communication programs in Tables 3 and 4. In the first program, a single tuple of varying size is passed around a logical ring of p processors; while, in the second, p such tokens are passed around a p-processor ring. The Linda version, using Linda's `out` and `in` operations, was compared against a version using IBM's native EUI message-passing commands. Both versions were run on an SP1 under IBM's POE (Parallel Operating Environment) system. In each case, the tables report both the time (in milliseconds) per single token transfer (including both `out` and `in` for Linda) and the corresponding throughput (in megabytes per second). Note that in the multi-token test, the *aggregate* throughput is p times the reported number.

The SP1 results indicate that Linda performance is qualitatively similar to that of IBM's native message-passing environment, particularly insofar as throughput for reasonably large data sizes is concerned. (For many real applications, for

Table 4
EUI/Linda SP1 wallclock time comparisons (multi-token ring tests for 10 processors)

Token size (bytes)	Linda results		EUI results	
	Time (msec)	Throughput (Mb)	Time (msec)	Throughput (Mb)
1	2.43	0.00041	1.22	0.00082
4096	6.00	0.68	7.41	0.55
8192	8.88	0.92	8.24	0.99
16384	14.54	1.13	12.84	1.28
65536	34.95	1.88	32.98	1.99
262144	104.08	2.52	115.43	2.27
1048576	396.64	2.64	454.66	2.31

Table 5
Hypercube native/Linda wallclock time comparisons (Deshpande and Schultz data)

Number of processors	iPSC/2 wallclock times (sec)		iPSC/860 wallclock times (sec)	
	Linda	NX/2	Linda	NX/860
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

example, in fields like seismic processing, typical data sizes are in excess of a megabyte, so small-data-size latency is relatively unimportant.) We note that Linda's latency for small data sizes is significantly larger than that with EUI, due to the particular design of the current Linda system on the SP1. Further research and development, now under way, is expected to reduce the latency difference.

Turning finally to application performance on distributed memory machines, we will look at two sets of results. First, the work of Deshpande and Schultz discussed above also examined performance for the Shallow Water Equations in other settings, including the use of tile-based decompositions on distributed memory machines from Intel. It is interesting to note that they observed that Linda achieved better than 90% of message-passing performance (using native Intel message-passing libraries), even though a relatively small problem was used (a 512×512 grid on up to 64 processors). Their results for 200 time steps on a 512×512 grid are summarized in Table 5.

Our last performance results are for two-dimensional FFTs on the Intel machines, and are based on work reported by Segall [26]. Segall developed codes using both C-Linda and native Intel message-passing libraries, and studied the question of what percent of native-code performance was achieved by the Linda codes. He found that the C-Linda version 'asymptotically approached the performance of the [optimized native] version as the matrix dimension [problem size] increased. It came to within a few percent for matrix sizes that are commonly encountered in practice.' Segall went on to note that the ratio of computation speed to communication speed made Linda performance relatively better on the iPSC/2 than on the iPSC/860, but Linda performance was well above 90% of native (optimized) message-passing performance for matrices of size 1024, even on 64 processors of the iPSC/860.

4. Concluding remarks

In this paper we have discussed the Linda model for parallel computing and compared it to message-passing models, both qualitatively and quantitatively. What we have seen is that Linda is a higher-level, more expressive and general approach, and that it is able to achieve run-time performance that is quite similar to that of

message passing. It is also significant that the Linda model has been very successful for real computations. While we do not have space here to provide details, we can say that Linda systems have been used widely for a diverse set of applications (cf. [3,11]), specifically including, for example, financial analytics [8,9,23], petroleum applications [5,6,27,28], electronic device design [7], and ray tracing [4,22]. In the near future, we expect to see the release of commercial-quality applications in these fields and others like computational chemistry and electronic chip design.

As important as current performance and usage may be, we think it is at least as important to focus on the future. For Linda it is clear that the future will include support for highly adaptive network computation in which processors enter and leave computations dynamically based on individual priority scheduling and for the development of hierarchical, heterogeneous systems (involving tuple spaces shared between independent parallel applications, for example). Already SCIENTIFIC's released Network Linda System supports the 'piranha model' in which programs make use of an adaptive master-worker paradigm for parallel computing (cf. [13,19,21]) that responds dynamically to workstation availability constraints. Under development is support for 'open tuple spaces' ([16]) that can be shared among different programs. Open tuple spaces will be used to build coherent applications on *very* heterogeneous mixtures of machines (hypercubes and workstations, or different SIMD machines, for instance) and for machines which are themselves heterogeneous (the Convex META, for example). They will also support multidisciplinary applications (such as combinations of PDE solvers with visualization programs) by providing a persistent shared memory resource (much like a file system, but with Linda semantics and much higher performance).

5. References

[1] R. Bjornson, Linda on distributed memory multiprocessors, Ph.D. Dissertation, Department of Computer Science, Yale University, 1993.

[2] R. Bjornson, N. Carriero and D. Gelernter, The implementation and performance of hypercube Linda, Research Report, Department of Computer Science, Yale University, 1989.

[3] R. Bjornson, N. Carriero, D. Gelernter, D. Kaminsky, T. Mattson and A. Sherman, Experience with Linda, Research Report, Department of Computer Science, Yale University, August, 1991.

[4] R. Bjornson, C. Kolb and A. Sherman, Ray tracing with network Linda, *SIAM News* (1991).

[5] J.L. Black and C.B. Su, Networked parallel seismic computing. Paper Number OTC 6825, *24th Annual Offshore Technology Conf.* Houston, TX (1992) 169–176.

[6] J.L. Black, C.B. Su and W.S. Bauske, Networked parallel 3-D depth migration, *SEG 61st Annual Internat. Meeting and Exposition*, Houston, TX (1991) 353–356.

[7] L. Cagan and A. Sherman, Linda on networks, *IEEE Spectrum* (1993).

[8] L.D. Cagan, How to make the most of financial codes: Investment analytics on networked workstations, *High Performance Comput. Rev.* 1(5) (May/June, 1993) 16–24.

[9] L.D. Cagan, N. Carriero and S. Zenios, Pricing mortgage-backed securities with network Linda, *Financial Analysts J.*: to appear (1993).

[10] C. Cap and V. Strumpen, Efficient data parallel computing in distributed workstation environments, Institut fur Informatik, University of Zurich, Zurich, Switzerland, January 1993.

[11] N. Carriero and D. Gelernter, Applications experience with Linda, *Proc. ACM Symp. Parallel Programming* (1988).

[12] N. Carriero and D. Gelernter, A foundation for advanced compile time analysis of Linda programs. In: *Languages and Compilers for Parallel Computing*, U. Banerjee, et al., eds. (Springer, Berlin, 1992) 389–404.
[13] N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course*. (MIT Press, Cambridge, MA, 1990).
[14] N. Carriero and D. Gelernter, Linda and message passing: what have we learned, Research Report, Department of Computer Science, Yale University, August 1993.
[15] N. Carriero and D. Gelernter, Linda in context, *Comm. ACM* 32(4) (1989) 444–458.
[16] N. Carriero, D. Gelernter and T.G. Mattson, Linda for heterogeneous networks, *Proc. First Annual Workshop on Heterogeneous Processing* (1992).
[17] A. Deshpande and M.H. Schultz, Efficient parallel programming with Linda, *Supercomputing '92*, Minneapolis, MN (1992) 238–244.
[18] C.C. Douglas, T.G. Mattson and M.H. Schultz, Parallel programming systems for workstation clusters, Research Report YALEU/DCS/TR-975, Yale University, Department of Computer Science, August 1993.
[19] D. Gelernter and D. Kaminsky, Supercomputing out of recycled garbage: Preliminary experience with Piranha, *Sixth ACM Internat. Conf. on Supercomputing*, Washington, DC (1992).
[20] S. Jagannathan, Optimizing analysis for first-class tuple spaces, *Languages and Compilers for Parallel Computing II* (1991).
[21] D. Kaminsky, The Piranha system for network computing. Research Report, Dept. of Computer Science, Yale University, 1991.
[22] F.K. Musgrave and B.B. Mandelbrot, The art of fractal landscapes, *IBM J. Res. Develop.* 35(4) (July, 1991) 535–540.
[23] I. Nelken and R. Bjornson, Fast Lady: Financial software must produce results quickly to keep up with the market, *RISK* (April 1992).
[24] Scientific Computing Associates Inc., *C-Linda User's Guide & Reference Manual* (New Haven, CT, 1993).
[25] Scientific Computing Associates Inc., *Fortran-Linda Reference Manual* (New Haven, CT, 1992).
[26] E.J. Segall, Tuple space operations: Multiple-key search, on-line matching, and wait-free synchronization. Ph.D., Rutgers University, 1993.
[27] A.H. Sherman, A hybrid approach to parallel compositional reservoir simulation. Paper Number OTC 6829, *24th Annual Offshore Technology Conf.* Houston, TX (1992) 191–198.
[28] A.H. Sherman, Parallel petroleum software on workstation clusters. *Supercomputing Europe '93*, Utrecht, The Netherlands (1993).