The Efficiency of Linda for General Purpose Scientific Programming

TIMOTHY G. MATTSON

Intel Corporation, Supercomputer Systems Division, Beaverton, OR 97006

ABSTRACT

Linda (Linda is a registered trademark of Scientific Computing Associates, Inc.) is a programming language for coordinating the execution and interaction of processes. When combined with a language for computation (such as C or Fortran), the resulting hybrid language can be used to write portable programs for parallel and distributed multiple instruction multiple data (MIMD) computers. The Linda programming model is based on operations that read, write, and erase a virtual shared memory. It is easy to use, and lets the programmer code in a very expressive, uncoupled programming style. These benefits, however, are of little value unless Linda programs execute efficiently. The goal of this article is to demonstrate that Linda programs are efficient making Linda an effective general purpose tool for programming MIMD parallel computers. Two arguments for Linda's efficiency are given; the first is based on Linda's implementation and the second on a range of case studies spanning a complete set of parallel algorithm classes. © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

The most cost effective hardware for supercomputing is based on multiprocessor computers. This includes everything from clusters of workstations to massively parallel supercomputers. When software availability is factored in, however, the advantages of parallel computing markedly decrease because so few application programs run on multiprocessor computers.

With the need so clear, why hasn't more application software migrated to multiprocessor computers? Primarily because parallel computing is hard. Parallel programmers must painstakingly decompose their problems into relatively indepen-

dent tasks and distribute them about the computer's multiple processors. This type of programming is error prone and difficult. The only way application programmers will use parallel computers is if they only have to parallelize their software once; not once for each multiprocessor architecture. What is needed are tools that let the programmer write portable programs for multiprocessor systems.

Numerous portable parallel programming environments exist [1, 2] including inherently parallel languages such as PCN [3], coordination libraries such as PVM [4], P4[5] or TCGMSG [6], and coordination languages such as Linda [7]. While comparing these environments is an important undertaking [8, 9], this article avoids comparisons and discusses a single system—the Linda programming environment.

Linda is an example of a coordination language [7] (i.e., a language that is restricted in scope to the execution and interaction of processes). In

Received June 1993
Revised December 1993

© 1994 by John Wiley & Sons, Inc.
Scientific Programming, Vol. 3, pp. 61–71 (1994)
CCC 1058-9244/94/010061-11

Linda, coordination takes place through six operations that manipulate an associative, virtual shared memory. This coordination mechanism is both simple and general making it possible to map Linda onto any multiple instruction multiple data (MIMD) computer system and to merge it with most if not all traditional languages for computation.

Linda was developed at Yale University [10, 11] with implementations available in the public domain [12, 13] and from commercial vendors [14]. Linda is easy to use and very flexible. Parallel computers, however, are used for their speed. Therefore, in order to be useful Linda must be efficient as well as programmer friendly.

The goal of this article is to address the issue of Linda's efficiency for general purpose scientific computing. This article is not concerned with selecting the best programming environment. Such a debate is as useless as arguing whether C is better than Fortran—a debate doomed to frustration as so much of the matter comes down to personal preference. Rather, the point of this article is to address the efficiency of Linda and justify the statement that “Linda is an effective tool for general purpose, MIMD parallel computing.”

The case for Linda's efficiency is divided into two parts. The first asserts that “Linda is efficient because it was implemented to be efficient.” This statement is supported by describing the implementation strategies outlined in two dissertations from the Linda group at Yale University [10, 11]. These dissertations show how Linda systems can be implemented to optimize Linda's operations and produce efficient programs. The basic idea is to map Linda's operations onto low level data structures (such as queues and hash tables) that are supported by efficient runtime libraries and result in programs that approach the performance of the underlying communication layer.

The second argument is more pragmatic asserting that “Linda programs are efficient because they have been found to be efficient.” At this point, the scope of the discussion narrows to the Linda technology provided by Scientific Computing Associates, Inc.—the primary commercial providers of Linda. This is an argument by existence. Anecdotal arguments are dangerous, however, because they can be manipulated to reach any conclusion by judicious selection of case studies. To avoid this trap, a complete range of case studies are considered where completeness is established by using the algorithm classes presented

by Fox [15]. By showing that Linda performs well for each of these classes, the claim that Linda is efficient for general purpose MIMD programming is supported.

Although the primary focus of this article is runtime efficiency, this is not always the most important issue to the programmer. In most cases, the major concern is the programmer's efficiency over the full software development cycle. This is briefly discussed with regard to Linda where the role played by distributed data structures substantially enhances programmer efficiency.

2 THE LINDA MODEL

Linda is a small programming language consisting of six operations to manipulate objects within a shared memory. These operations can be added to any language to create a dialect for programming parallel and distributed computers. In this article, the C-Linda system from Yale University [10, 11] and Scientific Computing Associates, Inc. [14] is briefly described. A more complete discussion can be found in Carriero and Gelernter's book [7].

The objects manipulated by Linda's operations are called tuples. As the name implies, tuples contain a number of fields that hold actual or formal values. For example, the following are Linda tuples:

```
(1, 3.4, "hello world")
("I think", 3.1415, "therefore i am", 2)
```

Tuples are collected into a single logical space called *tuple space*. Tuple space provides a separate memory region shared among any number of processes and available only to Linda's operations. As the following paragraphs show, process creation, communication, distributed data structures, and all other parallel processing support are provided through tuple space.

Tuple space is manipulated with four basic operations: *out*, *eval*, *in*, and *rd*. *Out* and *eval* create new tuples from their arguments. *Out* creates a passive tuple in which each field is evaluated before placement in tuple space. *Eval* creates an active tuple by conceptually invoking a separate process to evaluate each field (whether a new process is actually created for each field is implementation dependent). When all fields have been evaluated, the active tuple reverts to a regular (passive) tuple.

In and *rd* extract information from tuple space. They differ in that *in* removes the tuple from tuple space, whereas *rd* leaves the tuple behind for a later *in* or *rd*. In both cases, the arguments to *in* or *rd* define a template consisting of values and place holders (or formal values) that is matched against the tuples in tuple space. The template matches a tuple when the types of corresponding fields agree and corresponding values are identical. Once the match occurs, the tuple is accessed and any assignments indicated by place holders take place.

For example, we can create a simple tuple with the command,

```
out(17, "phoo", 4.5)
```

Any of the following Linda commands can read this tuple (for this discussion, the variables *i*, *str*, and *dbl* have types *int*, *char **, and *double*, respectively):

```
rd(17, ?str, ?dbl)
in(17, ?str, 4.5)
rd(?i, "phoo", ?dbl)
```

The question marks indicate place holders, i.e., fields to be assigned following a match. An active tuple would be created by the command,

```
eval("barr", 7, doit(7))
```

where *doit()* is a user provided function. This function can return any scalar type, but for the sake of this discussion, assume it returns an *int*. Once *doit()* has terminated, the active tuple becomes a regular tuple and the result can be accessed with *rd* or *in*:

```
in("barr", 7, ?i)
```

Both *in* and *rd* block if no tuple is available to match the template. Nonblocking forms exist for *in* and *rd* called *inp* and *rdp*. These functions return 1 (0) depending on whether a tuple exists (does not exist) that matches the indicated template. If the tuple exists the indicated operation is carried out as well.

These six operations provide a sufficiently complete set of operations for expressing algorithms for MIMD computers. They provide process creation (*eval*), process communication (*in/rd* paired with *out/eval*), and synchronization (blocking characteristics of *in* and *rd*).

3 EFFICIENCY

For a concept that receives so much attention, the precise meaning of the term *efficiency* is surprisingly vague. Qualitatively, it is generally agreed that efficiency measures how effectively the resources of the multiprocessor system are utilized. The problems arise when the term is made more quantitative.

Quantitative definitions of efficiency take the form:

$$\varepsilon = \frac{t_{\text{ref}}}{P t_{\text{par}}}$$

where *P* is the number of nodes, *t_{ref}* is some sequential reference time, and *t_{par}* the parallel time. The most rigorous definition of efficiency sets *t_{ref}* to the execution time for the best sequential algorithm corresponding to the parallel algorithm under study. When analyzing parallel programs, "best" sequential algorithms are not always available, and it is common to use the runtime for the parallel program on a single node as the reference time. This can inflate the efficiency because managing the parallel computation always (even when executing on one node) incurs some overhead.

The above definition combines algorithmic efficiency with the inherent efficiency of the parallel programming tools. To isolate these two effects, a separate programming environment efficiency can be defined:

$$\varepsilon' = \frac{t_{\text{par}}^0}{t_{\text{par}}}$$

In this case, the reference time, *t_{par}⁰*, is the runtime for the parallel program coded directly with the low level resources used to implement the parallel programming tool. For example, on Intel iPSC systems, the Linda runtime environment is implemented using the iPSC native message passing system (the NX/2 message passing library). Therefore, the parallel reference time would refer to the same algorithm used for the Linda program but coded directly with NX/2 calls.

Network parallel computing presents a final twist to the definitions of efficiency. In this case, it is impractical to implement scientific applications using the same network protocols that underlie the parallel programming environment. In the case of Linda, a useful reference system for the network environment would be any good, low level mes-

sage passing system. This is less than ideal but it does capture the chief elements in the comparison; namely management of tuple space versus directly specified point to point message passing.

4 LINDA IMPLEMENTATIONS FOR RUNTIME EFFICIENCY

Linda presents the programmer with a shared memory environment regardless of the underlying hardware architecture. This places great demands on the implementor of Linda systems to provide compilers, linkers, and runtime libraries that are capable of delivering high runtime efficiencies.

Two separate issues must be addressed by the Linda implementor. First, the Linda system must minimize runtime searching of tuple space as templates and tuples are matched. Essentially, the compiler and linker must work together to do as much matching as possible as the program is built. Second, the runtime management of tuple space (in the absence of a physical shared memory) must be evenly distributed among the nodes of the multiprocessor computer and lead to a minimum of extra messages.

Effective solutions to both of these problems can be found in two dissertations from David Gelernter's group at Yale University. The first (by Nicholas Carriero [10]) described the compiler optimizations required to minimize runtime matching between templates and tuples. The second dissertation (by Robert Bjornson [11] expanded on the work of Carriero [10] and addressed the problem of mapping tuple space support onto distributed memory computers.

First, consider the optimizations required to minimize the searching of tuple space at runtime. This is achieved through a local analysis of tuple space utilization as each Linda module is compiled and a global analysis when all of the modules are linked into an executable. This analysis partitions the program's Linda operations into distinct sets. The effect of this partitioning is two-fold:

1. Templates produced by operations within a particular set need only check tuples in the same set.
2. Each set can be represented individually by a data structure tailored to the usage observed in the set.

The end result is a mapping of tuple space into a collection of conventional data structures that reduce template/tuple matching to simple data structure manipulations.

For example, consider the following C-Linda operations

```
out("phoo", i, float_array);
/* Statement 1 */
out("barr");
/* Statement 2 */
in("phoo", 5, ?float_array);
/* Statement 3 */
in("barr");
/* Statement 4 */
```

The Linda compiler is able to divide these statements into two distinct partitions. One partition (statements 1 and 3) is the set of three field tuples with type signature *(string, int, float array)* and with the first field being a constant of value "phoo." The other partition (statements 2 and 4) is the set of one field tuples with type signature *(string)* and with the single field being a constant of value "barr."

As the program is linked into a final executable, the tuple space analysis based on type signature and constant values is applied globally, so all possible templates and tuples are partitioned into specific sets. Next, the second phase of the optimization finds low level data structures tailored to the tuple access patterns within each set. For the above example, the first set can be represented as a hash table. This is apparent by noting that templates within this set always use the second field as a literal to match against (i.e., a key) and the third field to hold a value (i.e., a place holder). The second set doesn't contain keys or place holders and its functionality at runtime can be provided by a counting semaphore.

This sort of analysis is carried out for each partition resulting in low level, efficient data structures representing the possible relations between templates and tuples. Only when consistent template patterns do not exist will the Linda environment need to search tuple space to match a template. Furthermore, when a tuple space search is required, only the single relevant partition must be searched, so global searches of tuple space are never required in practice.

The success of this optimization was demonstrated in a series of experiments [11] on a number of programs (LU decomposition, 2D FFT,

matrix multiply, and others). These studies found the average number of tuples touched while attempting to match a template was less than two (including the tuple ultimately matched). Although matching costs are not completely eliminated, they are reduced to a small and frequently insignificant portion of the program's overall effort.

Compiler analysis to minimize runtime matching only solves half of the efficiency problem. The other half requires a runtime system that efficiently maps Linda's virtual shared memory onto the physical memory system. Two cases are encountered with MIMD computer systems: one for shared memory computers and the other for distributed memory systems. Shared memory implementations of Linda [10] are the most straightforward because they have access to a physical shared memory. Therefore, the semantics of tuple space can be strictly enforced using locks of some type.

Linda implementations for distributed memory systems [11], however, require much more effort. The idea is to evenly spread the tuple space partitions among all of the nodes, thereby minimizing the chances that any single node will become a performance bottleneck. Although in most cases, an entire partition is mapped to a single node, some tuple sets can be broken up into pieces and spread among all of the nodes. Regardless of the low level details, tuple space partitions are deterministically mapped onto the nodes of the distributed memory system, so each node knows exactly where to go to access or store a tuple. The last step is to map the Linda operations onto an underlying message passing layer.

The details of a particular Linda implementation's use of message passing is beyond the scope of this article and would potentially compromise proprietary information—after all multiple vendors provide commercially supported Linda systems. Although specific implementations cannot be discussed, certain key points from the distributed memory Linda dissertation [11] can be presented.

As discussed earlier, the mapping of the tuple space partitions is deterministic, so every node knows where to put or access any tuple. The node responsible for a tuple space partition is called the *rendezvous node* for that partition. When a tuple is created, it is sent to the rendezvous node. If that tuple includes a large aggregate object (e.g., a large array), the rendezvous node is notified that

the tuple exists, but the data remains on the node that generated the data (the source node). When a different node (the destination node) requests a tuple, the rendezvous node is notified and the data is sent directly from the source node to the final destination—completely bypassing the rendezvous node.

Counting the messages generated for each combination of source, rendezvous, and destination nodes is beyond the scope of this article. It is useful, however, to consider a single worst case scenario. This occurs when the source, rendezvous, and destination nodes are all different for a large tuple. In this case, the Linda system generates the following messages:

1. A small message from the source node to the rendezvous node when the tuple is created
2. A small message from the destination node to the rendezvous node containing the template
3. A message from the rendezvous to the source node telling it where to send the actual tuple
4. A potentially large message from the source to the destination node containing the tuple itself.

This adds up to three short messages and one long message. Because the first message can be overlapped with computation, the observed penalty is usually two additional messages (destination-to-rendezvous, rendezvous-to-source). For large tuples where message transmission occurs directly between the source and destination nodes, the two extra messages appear at runtime as a small additional latency. However, for algorithms where many short messages are sent, this latency penalty can be substantial and seriously degrade the efficiency of Linda.

An important optimization discussed by Bjornson [11] can sometimes eliminate these extra messages. When a consistent communication pattern is detected, the Linda runtime system can remap the rendezvous node to the destination node. This, for example, would arise in any static domain decomposition problem where consistent communication patterns dominate the program's behavior. The result of this remapping is to eliminate some of the extra messages.

Some subset of these ideas plays a role in the implementation of any effective Linda system. The most complete use of these optimizations,

however, can be found in the Linda technology marketed by Scientific Computing Associates, Inc. (SCIENTIFIC). In fact, SCIENTIFIC has developed additional optimizations [14] that push Linda's efficiency beyond that possible solely with the techniques described by Carriero [10] and Bjornson [11].

5 "EXISTENCE PROOF" FOR LINDA'S EFFICIENCY

In the previous section, it was shown that Linda can be implemented to reduce the overhead associated with tuple space management and yield programs that execute efficiently. This argument by construction goes a long way towards establishing the case for Linda's efficiency.

In this section, claims about Linda's efficiency are supported with an existence proof, i.e., "I assert that Linda is efficient because cases exist where Linda programs executed with great efficiency." The claim is *not* that Linda will lead to efficient programs for every algorithm. This argument only requires that for a sufficiently broad range of applications, Linda programs exist that display high efficiencies.

The phrase "sufficiently broad range of applications" needs to be made more precise to give this argument any validity. What needs to be done is to establish a suitably complete taxonomy of parallel algorithm classes and demonstrate a representative example for each class.

One way to classify parallel algorithms [15] is in terms of the regularity of the underlying data structures (space) and the synchronization required as these data elements are updated (time). Based on this classification scheme, four general classes of parallel algorithms exist:

1. Synchronous: Tightly coupled manipulation of identical data elements. Regular in space and time.
2. Loosely synchronous: Tightly coupled as with the synchronous case, but the data elements are not identical. Irregular in space, regular in time.
3. Asynchronous: Irregular in time and usually (although not always) irregular in space.
4. Embarrassingly parallel: Independent execution of uncoupled tasks.

Although it is possible to derive even finer classifications [15], the parallel algorithm classes

given above capture the key elements of a parallel algorithm and provide a useful scheme for analyzing parallel programs.

The above set of algorithm classes is complete because it includes all combinations of spatial regularity/irregularity with temporal regularity/irregularity. This claim is important because these classes will be used to "argue" that Linda is an efficient tool for general MIMD programming by giving examples of an efficient Linda program for each class. The adjective "general" can only be established if the classification scheme is complete.

In the following subsections, case studies from the Linda literature will be described for each of the four algorithm classes. Rather than exhaustively list Linda programs for each algorithm class, the only case studies included in this article are those for which the efficiency of the Linda programming environment can be quantitatively assessed. In addition, these case studies utilize either real applications or computational kernels based on real applications (not trivial, "toy" problems). This greatly reduces the number of available case studies, because most parallel applications projects do not provide the information required to quantitatively evaluate the programming environment's efficiency.

5.1 Synchronous

Synchronous algorithms are those in which regular data elements are updated at regular time intervals. These "data parallel" problems are very common. Although ideal for single instruction multiple data (SIMD) computers, these algorithms also effectively map onto MIMD systems.

To evaluate Linda's efficiency, the ideal case study would provide data for a program coded with Linda and various message passing systems. This is exactly what was done by Deshpande and Schultz [16] using a program to solve the shallow water equations on workstation clusters, iPSC hypercubes, and shared memory computers. They used a simple, explicit finite difference algorithm for the project. First, the problem's domain was divided into rectangular regions with one region mapped onto each processor. Each processor then updated the interior of its region, swapped boundary data with its neighbors, and then updated its own boundaries.

This algorithm is ideally suited for message passing systems. These systems let a programmer hand tune a program to take advantage of faster

Table 1. Programming Environment Efficiency (ϵ') for Linda on iPSC Systems

Nodes	iPSC/2 ϵ'	iPSC/860 ϵ'
8	.99	.98
16	.99	.97
32	.98	.95
64	.97	.90
128		.83

communication between nearest neighbor nodes and to minimize communication contention. Linda does not support such optimizations: both the mapping of tuple space onto the nodes and where evals spawn processes are not under control of the Linda programmer. Therefore, it was expected that this problem would emphasize any weaknesses of Linda relative to message passing systems.

The programming environments used [16] were NX/2, PVM 2.4.1, and Linda. In each case, the programs were identical except for the calls to the programming environments themselves. For the sake of this discussion, the most relevant results [16] are for the Linda and fully optimized NX/2 versions of the program for iPSC hypercubes. Because the version of Linda for the iPSC is implemented on top of NX/2, these results can be used to compute programming environment efficiencies. These efficiencies are given in Table 1 for a calculation with 200 time steps on a 512×512 spatial grid.

From Table 1, Linda is seen to be efficient for this synchronous problem with programming environment efficiencies ranging from .83 to .99. Another trend apparent from Table 1 is that Linda's performance is lessened on the iPSC/860 relative to the iPSC/2. This occurs because the iPSC/860 has a faster compute node relative to the interprocess communication, so the iPSC/860 is more sensitive to communication overhead than the iPSC/2.

An important parameter controlling the efficiency of the Linda program is the granularity. This is actually a general phenomena for parallel algorithms with MIMD systems requiring somewhat coarser granularities than SIMD computers. In table 1, the impact of granularity appears as an efficiency dropoff as the granularity becomes finer (i.e., more nodes added for a fixed problem size). Although this effect is important, granularity is usually a quantity the programmer can change by increasing the amount of data mapped to each node.

As an additional example of the efficiency of Linda for synchronous problems, consider the explicit finite difference program for solving a heat conduction PDE developed by CAP and Strumpen [17] and Carriero et al. [18]. Their parallel algorithm was basically the same as that used for the shallow water case study [16]. This case study is useful to consider in this article, because the data from Carriero et al. [18] support an estimation of Linda programming environment efficiencies for workstation clusters.

In Table 2, elapsed times are given for PVM 2.4.1 and Linda 2.5.0 versions of the finite difference, PDE solver [18]. From these elapsed times, programming environment efficiencies were computed and placed in Table 2. These data show that Linda performs well on workstation clusters for this synchronous problems. Essentially, the overhead incurred by Linda's management of tuple space never exceeded 4.6%.

How does Linda achieve such high performance in these two case studies? Note that both of these algorithms are iterative with regular communication patterns. Therefore, the remapping optimization discussed earlier can be used causing the Linda runtime system to map the rendezvous nodes onto the destination node for each of the boundary tuples. Hence, no extra messages are required to manage tuple space causing the Linda and the message passing versions of the programs to pass essentially the same number of messages.

In addition to the two cases discussed in this section, other synchronous problems have been discussed in the Linda literature [19] with a range of granularities. These include LU decompositions, matrix multiplication [20], and general studies about the expression of data parallel algorithms with Linda [21].

Table 2. Programming Environment Efficiency (ϵ') for Linda on Workstation Networks

Number of Nodes	PVM 2.4.0	Linda 2.5.0	Programming Environment Efficiency
1	1,370.6	1370.6	1.0
2	648.0	622.2	0.98
4	328.0	342.6	0.96
8	168.0	175.8	0.96
16	90.0	92.8	0.97
32	54.0	54.0	1.0

5.2 Loosely Synchronous

A loosely synchronous algorithm is one that synchronously updates data elements that differ from one node to another (i.e., regular in time, irregular in space). A good example of this class occurs within certain molecular dynamics programs.

Molecular dynamics programs compute trajectories for each atom in a molecule by solving the classical equations of motion for the atoms moving under the influence of a force field. The bulk of the program's runtime is consumed by computing the value of the force field at each time step. The force field contains a number of terms representing the various forces atoms exert on each other. Of these forces, the most computationally expensive by far are those due to the long range, electrostatic forces. To compute these forces, each atom interacts with every other atom making this a classic N-body problem.

An important research code for molecular dynamics is MD from Klaus Schulten's group at the Beckman Institute [22]. MD does not approximate the nonbonded force (most codes use cut-offs) so the nonbonded computation is a true N-body problem with computational complexity of $O(N^2)$. The MD program was parallelized [23] using Linda. The parallelization focused exclusively on the nonbonded term that consists of a double sum over the atoms in a molecule. To better visualize the parallel algorithm, consider the pairwise interactions organized into a symmetric matrix. The parallel algorithm distributed complete rows of the lower triangular portion of the interaction matrix to parallelize the outermost loop in the N-body double sum. The rows of this lower triangular matrix are of varying sizes. The overall program, however, regularly stepped through time, so that algorithm is loosely synchronous.

The parallel version of MD was benchmarked with a 3,634-atom system on workstation clusters and a shared memory multiprocessor computer (a Sequent Symmetry symmetric multiprocessor). In both cases, the efficiencies were high and met expectations. The data in Shifman et al. [23] do not support computation of programming environment efficiencies—evaluation of Linda was not the goal of the project. The data for the shared memory system, however, do support an estimate of a theoretical efficiency limit, thereby providing a way to quantify the efficiency of the Linda programming environment.

The key fact that permits a theoretical efficiency estimate is the statement that on the Sequent Symmetry, 5% of the sequential program's

time was consumed by routines that were not parallelized [23]. Based on this figure and assuming the nonbonded force computations were perfectly parallel, the maximum allowable efficiency on 15 nodes of the Sequent is 57%. The measured efficiency for the same case was 54%. Therefore, the Linda program programming environment was efficient enough to let the program execute at 91% of the theoretical limit. The observed 9% overhead is probably an overestimate because the parallel part of the code was most likely not perfectly parallel (as was assumed when computing the theoretical limit).

5.3 Asynchronous

Asynchronous algorithms—other than the embarrassingly parallel subset—are the most rare. Its not that there is a lack of problems that would benefit from asynchronous algorithms; its just that the programs are so difficult to construct that they aren't commonly parallelized.

Linda is ideal for asynchronous algorithms. Linda communicates through persistent objects rather than transient events. This lets tasks interact intimately without having to be coupled in space or time.

An example of this class of programs is the Process-Trellis system [24]. The process trellis is a general software architecture for monitoring a collection of asynchronous processes. It has been used in a number of projects, the most extensive of which was to model a cardiac intensive care unit.

A number of approaches are possible for such an asynchronous, real time application. In the naive approach, one would execute an *eval* for each monitored process. This approach would scale poorly with respect to the number of monitored processes. Furthermore, it leaves too many scheduling decisions to the operating system, which could compromise enforcement of real time constraints.

Factor et al. [24] selected a number of processing nodes and created one Linda process per node. Each of these processes then swept through the monitor nodes assigned to them. By selecting a number of processing nodes such that the longest sweep time is less than the real time constraint, the system provided real time tracking of the asynchronous monitor processes. Working with model systems of 104 monitors on 10 nodes of a Sequent computer (a shared memory, symmetric multiprocessor system), they achieved speedups of 1.99 on two nodes and 9.29 on 10 nodes.

An even more interesting asynchronous application is the implementation of Linda for workstation networks [25]. The coordination of tuple space operations uses tuple space. Efficient Network-Linda programs have been mentioned in this article; programs that by necessity include an asynchronous component. Hence, efficiency for the asynchronous class can be inferred from successes with Network-Linda for the other classes.

5.4 Embarrassingly Parallel

Technically, the class of embarrassingly parallel algorithms is a subset of the asynchronous case for which the tasks are completely autonomous. Parallelization is trivial for this class, because little or no intertask coordination is required.

The most common (although by no means universal) way to construct these programs with Linda is to use a *master/worker* algorithm. In this algorithm, a master process creates a number of identical workers and a set of tasks. The workers grab a task, execute the indicated work, return the results to tuple space, and then grab the next task. The master either waits to pick up the results or transforms into a worker. This program structure is easy to code and leads to programs that are automatically load balancing [7].

An example of an embarrassingly parallel code is the vibrational analysis portion of MOPAC [26], a semi-empirical quantum chemistry program. In this code, a large matrix of second derivatives is constructed and diagonalized. The most time consuming part of the program is the construction of the second derivative matrix. Each element of this matrix is independent and calculated by a time consuming semi-empirical molecular orbital calculation.

The Linda version of MOPAC [27] was benchmarked on both workstation networks and on an iPSC/860. For the purpose of this discussion, the iPSC/860 results are the most interesting as they provide an opportunity to compare a Linda version of the code with a version using the iPSC/860 native communication environment (NX/2). The programming environment efficiencies ranged from .99 to 1.0 for calculations running on from 1 to 16 nodes—an expected result given the coarse granularity of the application.

The category of embarrassingly parallel codes is not surprisingly the category with the largest numbers of applications and is reviewed by Bjornson et al. [19]. Although most of these case studies do not include the data required to compute programming environment efficiencies, they do

confirm that high efficiency of Linda by delivering nearly perfect linear speedups (where the speedups were usually computed correctly, i.e., relative to a sequential program).

6 PROGRAMMER EFFICIENCY

No topic in parallel computing is as subjective or as controversial as “programmer efficiency”; hence, it is de-emphasized in most discussions of programming environments for parallel computing. Programmer efficiency, however, is perhaps the most important issue to consider when evaluating programming environments.

Ultimately, programmer efficiency is based on an individual’s comfort with a particular style of programming making concrete comparisons of dubious merit. Some claims concerning Linda, however, can be firmly established. First, Linda is very easy to learn. With only six operations, programmers quickly master Linda and write effective programs. Second, Linda programs are easy to debug. Linda coordinates the execution of distinct processes through objects in a globally accessible memory rather than through transient events. This object-based interaction is easy to visualize leading to effective program visualization tools that capture the state of a parallel program at any moment in terms of a simple picture [28]. This provides a tremendous tool for debugging and analyzing parallel programs. Finally, Linda provides a great deal of algorithmic flexibility for the parallel programmer. Objects can be placed in or removed from tuple space in numerous ways letting the programmer build programs around standard constructs (such as messages or queues) or general distributed data structures. In essence, Linda’s generality lets the programmer’s efforts be guided by the needs of the algorithm not the needs of the programming environment.

Most parallel programmers use message passing, so it is important to address Linda’s utility for message passing algorithms. In Linda, the programmer is not restricted to message control based solely on source and destination information. Linda lets the programmer write algorithms that use communication in which the sender and receiver do not know each others identity—a programming mechanism known as anonymous communication. This provides an important extension to message passing and is used in many algorithms such as master/worker codes and other dynamic load balancing algorithms. Essentially,

Linda can use anonymous communication to “out message pass” message passing.

7 CONCLUSIONS

The goal of this article was to demonstrate the utility of Linda for general purpose, scientific programming of MIMD parallel computers. Past reviews of the Linda programming model [18, 19] have done a good job of showing that Linda is expressive enough to meet the needs of scientific programming. Unfortunately, these reviews have not adequately addressed the issue of Linda's efficiency leaving many scientific programmers in doubt about the utility of Linda for their applications.

This article has addressed those doubts by showing that Linda programs execute with efficiencies that are competitive with those provided by lower level programming environments. This efficiency argument was divided into two complementary parts. The first was based on a description of Linda's implementation and the steps taken to replace Linda's high level operations with efficient low level, machine specific constructs.

The second argument was an argument by existence—namely, “Linda is known to be efficient because it has been widely used and found to be efficient.” To make the argument credible, algorithm classes were defined that encompass most if not all parallel programs. Using this complete set of algorithm classes, generality of Linda's efficiency was demonstrated by providing representative examples for each class.

When this project began, the goal was to provide an “existence proof” of Linda's efficiency. With data available from the literature, it has not been possible to achieve this goal. Application programming projects only rarely include the experiments required to evaluate programming environment efficiencies. This is not a criticism of these projects as research on programming environment efficiencies is the concern of programming environment implementors—not users. Therefore, the data available for use in this article only permit a demonstration-by-existence of Linda's efficiency rather than a proof.

It is useful to consider what would be required to transform this demonstration-by-existence into a proof-by-existence. If nothing else, this might help application programmers understand how to extend their projects to help implementors more effectively evaluate their systems. Basically, ϵ' values must be determined for each algorithm class

and for a range of node counts. This would demonstrate the effect of scalability on the efficiency of Linda. Furthermore, these measurements must be done for each category of MIMD architecture: shared memory, network cluster, and tightly coupled distributed memory systems.

Even though the goal of an existence proof was not fully satisfied, the demonstration given here does support the central claim of this article—that Linda programs can provide high efficiency for a full range of scientific programming problems. The stronger statement—that Linda will always provide the desired efficiency—has not been made.

It is important for the Linda programmer to understand when Linda will not deliver good performance. Consider the set of applications discussed in this article. Notice that they all share a couple traits:

1. Communication is a small fraction of the total computational effort so any deficiencies in the programming environment play a minor role in the program's performance.
2. The communication is bandwidth limited (rather than latency bound) and preferably involves regular communication patterns.

The first point applies in some degree to any successful parallel algorithm. It is the second point that deserves more careful examination.

Linda programs must contend with two sources of overhead not present with lower level, message passing environments. The first is due to the matching of a tuple and a template at runtime. As shown in Section 4 and confirmed experimentally by Bjornson [11], this effect is minimal. The second source of additional overhead is the most important. Depending on the mapping of tuple space and the origin of tuple requests, additional messages may need to be generated as the Linda runtime system manages tuple space. This effect appears to the programmer as an additional latency associated with Linda communication. Therefore, any algorithm that is sensitive to communication latency will run slower with Linda than with an explicit message passing scheme. This effect is important, but it should not be overemphasized. Most good candidates for parallel computing are not latency bound. Furthermore, a programmer can frequently restructure an algorithm to emphasize bandwidth and de-emphasize latency. If this is not possible, however, a latency bound algorithm might benefit from the use of a lower level programming model than Linda.

The fact that problems exist that Linda cannot handle efficiently is not a fatal flaw for Linda. What Linda provides is an effective programming model that usually delivers most of the available efficiency. Just as users of high level programming languages sometimes use assembly language coding, it is fully expected that Linda programmers will occasionally need to use low level message passing constructs.

This suggests a more rational approach to software development for parallel computers. In this approach, a portable high level tool provides the first pass at a parallel program. After the program is up and running, then and only then, the programmer optimizes the program as needed with low level constructs. The end result would be a high level of code reuse and more responsible software engineering for parallel and distributed computers.

REFERENCES

1. L. Turcotte, "A survey of software environments for exploiting networked computing resources," Technical Report No. MSM-EIRS-ERC-93-2, Mississippi State, 1993.
2. D. Y. Cheng, "A survey of parallel programming languages and tools," NASA Ames Research Center Technical Report RND-93-005, 1993.
3. I. Foster, R. Olson, and S. Tuecke, "Productive parallel programming: The PCN approach," *Sci. Prog.*, vol. 1, pp. 51-56, 1992.
4. V. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency Pract. Exp.*, vol. 2, pp. 315-339, 1990.
5. R. Butler and E. Lusk, "User's guide to the p4 programming system," Technical Report ANL-92/17, Argonne National Laboratory, 1992.
6. R. J. Harrison, "Portable tools and applications for parallel computers," *Int. J. Quantum Chem.*, vol. 40, pp. 847-863, 1991.
7. N. Carriero and D. Gelernter, *How to Write Parallel Programs: A First Course*. Boston: MIT press, 1991.
8. C. C. Douglas, T. G. Mattson, and M. H. Schultz, "Parallel programming systems for workstation clusters," Yale University Computer Science Department, Technical Report, YALEU/DCS/TR-975, 1993.
9. T. G. Mattson, *Proceedings of Hawaii Systems Sciences Conference*. Los Alamitos, CA: IEEE Computer Press, 1994.
10. N. Carriero, "Implementing tuple space machines," Yale Computer Science Dissertation, New Haven, CT, 1987.
11. R. Bjornson, "Linda on distributed memory multiprocessor systems," Yale Computer Science Dissertation, New Haven, CT, 1993.
12. G. Schoinas, "Issues on the implementation of programming systems for distributed applications," University of Create. Technical Report, 1992.
13. R. M. Butler and E. L. Lusk, "p4-Linda: A portable implementation of Linda," Argonne National Laboratory Technical Report, 1993.
14. Scientific Computing Associates, Inc., *C-Linda User's Guide and Reference Manual*. New Haven, CT: Scientific Computing Assoc., Inc., 1992.
15. G. C. Fox, *Supercomputing '91*. Albuquerque, NM. Los Alamitos, CA: IEEE Computer Press, 1991.
16. A. Deshpande and M. H. Schultz, *Proceedings of Supercomputing '92*. Minneapolis, MN. Los Alamitos, CA: IEEE Computer Press, 1992, pp. 238-244.
17. C. Cap and V. Strumpfen, "Efficient data parallel computing in distributed workstation environments," Institut fur Informatik, University of Zurich, Zurich, Switzerland, January 1993.
18. N. Carriero, Gelernter, T. G. Mattson, and A. H. Sherman, "The Linda Alternative to Message-Passing Systems," *Parallel Comput.* (in press).
19. R. Bjornson, N. Carriero, D. Gelernter, T. G. Mattson, D. Kaminsky, and A. Sherman, "Experience with Linda," Yale University Computer Science Department, Technical Report RR-866, August, 1991.
20. B. Ghosh and M. H. Schultz, *Proceedings of the Scalable High Performance Computing Conference*. Williamsburg, VA. Los Alamitos, CA: IEEE Computer Press, April, 1992.
21. N. Carriero and D. Gelernter, *Languages and Compilers for Parallel Computing*. Berlin: Springer-Verlag (to appear).
22. A. Windemuth and K. Schulten, "Molecular dynamics simulation on the connection machine," *Mol. Simulation*, vol. 5, pp. 353-361, 1991.
23. M. A. Shifman, A. Windemuth, K. Schulten, and P. L. Miller, "Molecular dynamics simulation on a network of workstations using a machine-independent parallel programming language," *Computers Biomed. Res.*, vol. 25, p. 168, 1992.
24. M. Factor, D. Gelernter, C. Kolb, P. Miller, and D. Sittig, "Real time data fusion in the intensive care unit," *Computer*, vol. 23, p. 45, 1991.
25. M. Arango, D. Berndt, N. Carriero, D. Gelernter, and D. Gilmore, "Adventures with network Linda," *Supercomput. Rev.*, vol. 3, pp. 42-46, 1990.
26. K. Baldrige, "MOPAC on the iPSC/860," submitted, 1993.
27. T. G. Mattson, *Proceedings of the RCI North American Annual Member Executive Conference*. Washington DC, 1992.
28. P. Bercovitz and N. Carriero, "TupleScope: a graphical monitor and debugger for Linda based parallel programs," Yale University Computer Science Department, Technical Report, RR-782, 1990.