Using Linda for Supercomputing On a Local Area Network

Robert A. Whiteside
Scientific Computing Division, Sandia National Laboratories, Livermore, California, 94551.
whiteside@sandia-2.arpa
Jerrold S. Leichter
Department of Computer Science, Yale University P.O. Box 2158 Yale Station New Haven, CT 06510.
leichter-jerry@cs.yale.edu

Abstract

A distributed parallel processing system based on the LINDA programming constructs has been implemented on a local area network of computers. This system allows a single application program to utilize many machines on the network simultaneously. Several applications have been implemented on the network at Sandia National Laboratories and have achieved performances considerably faster than that of a Cray-1S. Several collections of machines have been used including up to eleven DEC VAXes, three Sun/3 workstations, and a PC.

1 Introduction.

Many organizations have extensive networks of time-shared computers. Although these machines may be quite heavily loaded during the day with various interactive computations, it is not uncommon for them to sit idle overnight. The reason for this idle time is that any one of these machines is too slow to be useful for many interesting large scientific and engineering applications. These applications are allocated to supercomputers on the network instead. Collectively, however, the smaller machines can represent a considerable computational resource. Sandia National Laboratories (SNL), for instance, has an NSU equivalent of twenty-five Cray-1S's in such machines. (NSU is a metric used by the U. S. Department of Energy to measure computer performance.) Of course, many of these machines are dedicated to specific tasks and are not attached to the network. Nonetheless, the collective computational power of the machines on the SNL network is substantial. In this paper, we demonstrate that distributed parallel processing can permit some applications to achieve supercomputer performance from such a local area network.

The parallel applications described here utilize a VAX VMS Ethernet implementation of the Linda [1,2,3] programming constructs for communication and synchronization. The current implementation is restricted to a single physical Ethernet. That is, machines on the other side of a gateway cannot be fully integrated into the parallel system. However, some simple extensions permit a subset of the Linda operations to take place over DECnet (giving access to distant VAX/VMS systems) and TCP/IP (giving access to SUN workstations and some PC's).

In the following section we describe Linda and its VMS Ethernet implementation. Section 3 describes and gives performance results for three applications that have been implemented on our network: a charged particle transport program, a parameter sensitivity analysis, and a finite element equation solver. Finally, section 4 provides some concluding remarks.

2 The Linda language

The Linda language was first defined by Gelernter in [1]. As it has evolved, however [2,3], Linda is not a complete language. Rather, it is a set of objects and operations on those objects that are intended to be *injected* into an existing language, thus producing a new language intended for distributed programming.

In the following sections we present a simple, language-independent description of Linda.

2.1 Fundamental objects

Linda is based on two fundamental objects: *tuples* and *tuple spaces*.

2.1.1 Tuples

Tuples are collections of *fields*. Fields have fixed types associated with them; the types are drawn from the underlying language. A field can be a *formal* or an *actual*. A formal field is a place-holder — it has a type, but no value. An actual field carries a value drawn from the set of possible values allowed for that type by the underlying language.

For example, suppose that the underlying language has types **int** (integer) and **float** (floating-point number). Then

$$\langle 1_{\mathbf{int}}, 1.5_{\mathbf{float}}, 2_{\mathbf{int}} \rangle$$

is a tuple with three fields: Two integers, 1 and 2, and a floating point value, 1.5. It is essential to distinguish it from the tuple:

$$\langle 1_{\mathbf{int}}, 1.5_{\mathbf{float}}, 2_{\mathbf{float}} \rangle$$

which differs in the type of its third field.

The previous two tuples contain only actual fields. The tuple

$$\langle 1_{\mathbf{int}}, \square_{\mathbf{float}} \rangle$$

contains an integer actual, and a float formal.

The definition of the Linda language places no *a priori* restrictions on what types are allowed. *Any* types from the underlying language are allowed, including records, arrays, and pointers. Similarly, tuples may have any number of fields.

2.1.2 Tuple space

Tuples live in tuple space which is simply a collection of tuples. It may contain any number of copies of the same tuple; it is a *bag*, not a set.

Tuple space is the fundamental medium of communication in Linda. Linda processes communicate through tuple space. All Linda communication is a three-party operation — Sender interacts with tuple space, tuple space interacts with Receiver — rather than a two-party operation as in traditional models.

Tuple space is a *global, shared* object — all Linda processes that are part of the same Linda program have access to the same (logical) tuple space. In a strong sense, it is access to this shared object that *defines* what processes constitute a single Linda program.

2.2 The operators

The **out** operator inserts a tuple into tuple space; for example, if f is a variable of type **float** with the value 1.5, and i is an **int** with the value 2, then

$$\mathbf{out}(1, f, i)$$

would insert into tuple space the tuple we saw earlier,

$$\langle 1_{\mathbf{int}}, 1.5_{\mathbf{float}}, 2_{\mathbf{int}} \rangle$$

The **in** operator extracts tuples from tuple space. It finds tuples that *match* its arguments, in a sense we will describe shortly. However, equal tuples match. The tuple of the previous paragraph could be extracted by the operation:

$$\mathbf{in}(1, 1.5, 2)$$

Formal fields are created by adding a question mark prefix. What should follow a question mark is language-dependent; the intent is that it should be "anything that can be on the left hand side of an assignment statement."

When a formal is used in an **in**, any actual in the tuple will match. The operation

$$\mathbf{in}(1, ?f, 2)$$

might extract the same tuple as in the previous example without a formal. In addition to removing the tuple, it would assign 1.5 to f. Note that the type of f is significant — for this match to be possible, f must have type **float**. If f has any other type, even double precision floating point, the match will not succeed.

After an **in** operation, the tuple matched is removed from tuple space. The **rd** operator is similar to **in**, but leaves the matched tuple in tuple space unchanged. It is used for its side effects — bindings and synchronization.

The "?" prefix may be used with **out** as well. The tuple

$$\langle 1_{\mathbf{int}}, \square_{\mathbf{float}} \rangle$$

could be produced by **out**(1,?f). An "**out**" of this form is unusual, and its utility is described in the next section on "tuple matching".

2.3 Tuple matching

The **in** and **rd** operators are defined in terms of *matching*. Call the tuple defined by the fields in an **in** or **rd** a *template*. A template $\mathcal{M}$ matchs a tuple $\mathcal{T}$ in tuple space if:

- $\mathcal{M}$ and $\mathcal{T}$ have the same number of fields;
- Corresponding fields have the same types;
- Each pair of corresponding fields $F_{\mathcal{M}}$ and $F_{\mathcal{T}}$ match as follows:
  - If both $F_{\mathcal{M}}$ and $F_{\mathcal{T}}$ are actuals, they match if and only if their respective values are equal, where equality is defined by the base language for objects of this type;
  - If $F_{\mathcal{M}}$ is a formal and $F_{\mathcal{T}}$ is an actual, they match; the value of $F_{\mathcal{T}}$ may eventually be assigned to some variable. No assignment takes place unless *all* the fields match, however.
  - If $F_{\mathcal{M}}$ is an actual and $F_{\mathcal{T}}$ is a formal, they match unconditionally. The value of $F_{\mathcal{M}}$ is discarded.
  - If both $F_{\mathcal{M}}$ and $F_{\mathcal{T}}$ are formals, they never match.

If no matching tuple can be found in tuple space, **in** and **rd** block, and the process waits for a tuple to appear. If there is more than one matching tuple, **in** and **rd** choose one non-deterministically.

2.4 Linda implementations

The first implementation of Linda, due to Nicholas Carriero, ran on an experimental machine developed at Bell Labs, the S/Net. Carriero has also produced implementations for the Encore and Sequent shared memory multiprocessors [4]. Robert Bjornson at Yale produced an implementation for the Intel iPSC, and another hypercube implementation is reported in [5]. All of these implementations inject Linda into the C programming language. Carriero [6] reports progress on a "Linda in FORTRAN" implementation. Another current project [7] is developing hardware that directly supports the Linda operations.

The particular incarnation of Linda used in the experiments reported in this paper is known as VAX LINDA-C. As the name indicates, it is an embedding of Linda into VAX C. VAX LINDA-C extends the C type system to make it more appropriate for use with the (type-driven) Linda tuple-matching operations. In addition, the extensions allow the programmer to provide the LINDA-C compiler with information it needs to determine the sizes of objects to be placed in tuples. It is possible, for example, to **out** an array whose size is determined at run-time. A full description of the language appears in [8].

VAX LINDA-C was designed to support parallel Linda programs at two levels of coupling simultaneously: multiple processes running on a single "node", and multiple nodes communicating across an Ethernet. A "node" may contain more than one processor, but only a single shared memory. A great deal of flexibility is required to make effective use of hardware ranging from shared memory multiprocessors to independent computers communicating over an Ethernet. We believe Linda can provide that flexibility, although the set of machines available to us does not include any multi-processor VAXes. Nonetheless, the ability to run multiple processes on a single node is useful. For example, the implementation of TACO3D discussed in Section 3.3 was developed for a hypercube and assumes that the number of worker nodes is a power of two. We were able to run the program on three VAXes, for instance, by starting multiple processes on one of them. This is also useful for programs structured as a master with multiple workers. While the workers in such a program are busy, the master is often idle. Letting the master share its node with a worker avoids wasting a processor.

The current implementation of LINDA-C is limited to a single physical Ethernet. Communication among the nodes uses a private Ethernet protocol, so no gateways can intervene. LINDA-C was designed that way since the designer (Leichter) felt that the performance using a direct Ethernet connection would be about as slow as anyone would find useful. However, it has become clear from our experiments that this assessment was wrong: some applications can utilize a large number of machines, even with very high communications costs. The network we used for these experiments has several physical Ethernets interconnected *via* gateways. We have therefore used DECnet task-to-task communication to extend a subset of Linda operations to more remote VMS machines and TCP/IP tools to access UNIX machines. A high-priority item for future work is to merge these extensions, modified to support all the Linda operations, back into the base system.

The tuple space in LINDA-C is distributed — each of the VAXes participating may hold some subset of the tuples. This is desirable for performance reasons, since each machine may be able to satisfy an **in** or **rd** from its local set of tuples without accessing the network. However, this distribution of tuples also means that a loss of any machine in the network can cause part of the tuple space to be lost, and perhaps to cause the entire computation to fail. Thus, this possible performance/robustness tradeoff is a topic for further investigations.

2.5 Specialized communication using Linda

The Linda operators and tuple space provide a very general and powerful communication facility. Most programs need only a fraction of this potential power, though exactly *which* fraction varies from program to program. The codes we discuss were all developed using much simpler communications regimes. One, TACO3D (Section 3.3) was originally developed for an Intel iPSC Hypercube [9]. Simulating the iPSC's message-passing functions takes a couple of lines of Linda code, and each hypercube node is implemented by a separate Linda process. Messages are implemented as tuples whose fields correspond to the parameters of the iPSC **send** routine: the name of the recipient (the node number within the cube), the message type (an integer), the number of bytes in the message, and the message itself. One difficulty arose from the need to provide explicitly for the sequencing of messages, since tuple space communication is not guaranteed to be order-preserving.

Note that even when simulating message passing, the power of the Linda communication model is helpful. We may *think* of messages as being sent from one particular process on a particular node to another, but in fact, how the processes are assigned to nodes is completely invisible to the program.

2.6 An example

To provide a flavor of what VAX LINDA-C programs look like, we include a brief example. Figure 1 is a matrix multiplication worker for square matrices. It waits for the arrival of tuples of the form:

$$\langle \Lambda_{REQUEST}, rownum_{\mathbf{int}}, dimension_{\mathbf{int}} \rangle$$

The first field in the tuple contains "$\Lambda_{REQUEST}$". This denotes a null value of type *REQUEST* and serves to identify the tuple as input to the matrix multiplier. The second and third fields define a row number r within a square product matrix with side *dimension*. The worker proceeds to **in** row r of the left multiplier matrix and **rd** the columns of the right multiplicand. The left multiplier is available in tuple space by rows, and the multiplicand is available by columns. The rows and columns do not need markers since they contain vectors of distinct types, *ROW* and *COLUMN*. The worker calculates inner products and builds a row of the result matrix, which it finally **out**'s, marking it with a null value of type *RESULT*. Notice that the worker **in**'s the row vector — no other worker will need it. However, it

```
/*
 * Matrix multiply worker process.
 */
#include "matrix.h"

LindaMain()
{       int r, c, dim;
        $MAKE_ARRAY(row,ROW,rowData,double,MAXDIM);
        $MAKE_ARRAY(col,COL,colData,double,MAXDIM);
        $MAKE_ARRAY(result,ROW,resultData,double,MAXDIM);

        for (;;)
        {       in(REQUEST,?r,?dim);
                $SET_DIM(row,dim);
                $SET_DIM(col,dim);
                $SET_DIM(result,dim);
                in(r,?row);

                for (c = 0; c < dim; c++)
                {       rd(c,?col);
                        resultData[c] =
                                innerProd(rowData,colData,dim);
                }
                out(RESULT,r,result);
        }
}

static double
innerProd(rdata,cdata,dim)
{       /* Compute inner product */         }
```

Figure 1: Matrix multiplication worker

rd's the columns, since they are needed by other workers computing other rows. The right multiplicand is a *distributed data structure* [10].

Figure 2 illustrates code that makes use of the matrix multiply worker. Figure 3 is the include file referred to in the other two figures.

3 Applications and results.

It is generally true in parallel processing that the performance is best when a large amount of computation is performed for each communication or synchronization operation. Because of the high overhead involved in network operations, this effect is particularly severe in computations on local area networks as compared to more tightly coupled multiprocessors. Thus, applications that run best in this environment will have a large granularity, dividing into large, autonomous pieces.

So far, three applications have been implemented in this environment. Two of these divide into large, independent pieces in a straightforward manner, and give quite good performance results. In the third application, a finite element equation solver, the parallel computations have a much smaller granularity, and the resulting poorer performance shows a limitation of the current implementation.

3.1 The BOHR program.

The BOHR computer program[11] is used at Sandia for studying early events in radiation damage of semiconductors. These computations examine the interactions between a high-energy incoming projectile and a target system. For example, one computation studied an argon projectile scattering from a silicon target. This Monte Carlo technique uses a few random numbers to initialize the projectile direction and velocity. It then follows the trajectory by numerically integrating the classical equations of motion until the collision is completed. Results of the collision are then assessed. Electrons in the target, for instance, may have been ejected, excited, or transferred to the projectile. The recoil energy of the target is computed. After this, another trajectory is set up and followed. One run of BOHR consists of following thousands of trajectories for which various possible outcome results are averaged, counted, and tabulated.

The parallel implementation of Bohr is straightforward, since each trajectory is completely independent of

```
#include "matrix.h"

LindaMain()
{       double  A[DIM][DIM], B[DIM][DIM], C[DIM][DIM];
        int     r, c;
        $ARRAY(rowTemp,ROW,A[0]);
        $MAKE_ARRAY(col,COL,colData,double,DIM);

        /* Drop the rows of A and the columns of B into tuple space, along
         * with a "request" tuple for each result row. C stores arrays
         * row-wise, so out the rows of the A matrix in place. But the
         * non-contiguous elements of B are copied into COL.
         */
        for (r = 0; r < DIM; r++)
        {       out(REQUEST,r,DIM);
                rowTemp.data = A[r]; out(r,rowTemp);
                for (c = 0; c < DIM; c++)
                        colData[c] = B[c][r];
                out(r,col);
        }
        /* Extract the results; re-use rowTemp          */
        for (r = 0; r < DIM; r++)
        {       rowTemp.data = C[r];
                in(RESULT,r,?rowTemp);
        }
        /* Clean up all the junk                        */
        for (r = 0; r < DIM; r++)
                in(r,?COL);
}
```

Figure 2: Matrix multiplication master

```
/*
 * Matrix multiplier standard include file
 */
#ttcontext matrix
#include "LINDA_LIBRARY:lindadefs.h"
#define MAXDIM 100
#define NULL 0

$ARRAY_TYPE(ROW,double);
$ARRAY_TYPE(COL,double);

newtype void REQUEST;
newtype void RESULT;
```

Figure 3: Matrix multiplication include file

the next. We employ a single master process and many worker processes—one on each machine participating in the computation. Each slave process repeatedly follows one trajectory, and sends the results back to the master process. We used the technique proposed by Percus and Kalos[12] to provide each slave processor with its own independent sequence of random numbers. The master process receives and tabulates trajectory results until the required number has been received, and writes the output file.

One useful aspect of this implementation is its robustness in the face of inevitable difficulties with machines on the network. If a machine running a slave process crashes, or runs very slowly because it is heavily loaded, the computation as a whole still proceeds with only a degradation in performance. Although the master process receives no results from that one slave, eventually enough trajectories are received from other machines to complete the job. Load-balancing occurs quite naturally and easily: each machine computes as fast as it can, and the job completes when the master has received and processed the required number of trajectories.

We have run the network version of BOHR on several configurations of machines on our network, and performance results are summarized in Table 1. Although a

Configuration	*Time (min)*	*Cray Eq*
Cray-1S	7.4	1.0
VAX 8700	152	.05
SNL-L[a]	42	.18
SNL[b]	15	.49

[a]The SNL-L set consisted of: 2 VAX 8700's, 1 VAX 8650, 3 VAX 780's, and 3 MicroVAX II's
[b]The SNL set consisted of 4 VAX 8700's and 7 VAX 8650's

Table 1: Summary of performance results for the BOHR application. The "*Cray Eq*" column gives the fraction of Cray-1S performance delivered by the configuration.

production run for the program often requires over 2000 trajectories, in generating these results we stopped the run at only 200. Production runs would be at least ten times longer than those in our timing runs.

Comparison of the times in Table 1 for the Cray-1S and the VAX 8700 shows that the BOHR program vectorizes rather well on the Cray. The Cray-1S runs BOHR about 20 times faster than the 8700, and our experience is that for strictly scalar computations, the Cray-1S runs only about five times faster than the VAX 8700. Thus, BOHR is rather well suited to the Cray.

The configuration labeled "SNL-L" in Table 1 consists of a set of machines on our network located at Sandia National Laboratories in Livermore (SNLL). There were nine machines in all, with three 8000 class VAXes, three MicroVAXes and three 780's. An accurate estimate of the "ideal" performance of this collection of machines would be difficult. However, a rough estimate can be obtained by considering the three 8000 class machines as equivalent, and the six smaller machines collectively as about an 8700, giving a total of four VAX 8700 equivalents. The Cray therefore should run about 5 times faster than the SNL-L set. Instead, from Table 1, we see that the Cray is 5.7 times faster than the SNL-L set, indicating some degradation from the ideal performance in the SNL-L set.

The SNL set consisted of eleven VAX 8700, 8650, and 8550 machines in Livermore, CA and Albuquerque, NM. The two sites are networked together *via* a 56 Kbit link. Since a single VAX 8700 requires 152 minutes to perform the computation, this ensemble of eleven similar machines should take about 14 minutes. Table 1 shows the time required to be 15 minutes, once again indicating some small source of performance degradation which we have not yet characterized. It can also be seen that the "SNL" set runs at about half the speed of the Cray-1S.

Although performance results for this are not reported in Table 1, we were able to run this application on a (more) heterogeneous collection of machines. In one such set, the parallel BOHR program utilized five VAX (VMS) systems, three Sun/3 (Unix) workstations, and an Intel 310 (XENIX) personal computer.

3.2 Rocket plume sensitivity analysis.

CHARM is a computer program for modeling high-altitude rocket plumes. The model has a number of parameters, and a study underway at Sandia seeks to determine the sensitivity of plume simulation results to the values of these parameters. A sophisticated technique is used to select parameter values for scanning the space, but the computationally intensive portion of the study reduces to running CHARM many times with different input values. Each CHARM run requires from 15 minutes to 2 hours of execution time on a VAX 8700, and a full sensitivity analysis may require hundreds of such runs.

The parallel implementation is once again straightforward. Three types of processes are employed: one master, one post-processor, and a slave process on each worker VAX. Each slave participating in the computation posts a Linda tuple indicating that it is idle, then waits for a response from the master. The master process waits for such "Idle VAX" tuples to appear, and sends each such process the index of the next simulation run to be performed. When the run is completed, the slave posts a message to the post-processor indicating that the output from the CHARM run is available for analysis. It then requests more work from the master.

This application, too, can be made robust in the face of difficulties with a remote slave processor. The master assigns simulation runs to slaves until all have been assigned. At this point, although all of the simulation runs have been assigned to slaves, not all of them have completed. When another slave process requests more work, it is assigned one of these not yet completed simulations. Thus, if one slave process never completes its task, either because of a machine failure or because the machine is heavily loaded, it will simply never report back to the master. Its task, however, will eventually be re-assigned to another slave process and the job as a whole will still complete.

The timing data for the parallel implementation are presented in Table 2. These are for a sensitivity analysis that required 99 runs of the CHARM program. The analysis required about 1500 minutes on the VAX, or about 15 minutes per simulation. CHARM runs mostly in single precision on the VAX, and vectorizes very poorly on the Cray. Thus, the Cray-1S runs only a factor of 4.4 faster than a VAX 8700 for this application, and a collection of fourteen 8000-class machines runs over twice as fast as the Cray. Thus, for this application at least, the network delivers true supercomputer performance.

There are five different types of VAXes in our collection of fourteen "8000-class" machines, and each type has different performance characteristics. Because of this heterogeneity, the computation of parallel efficiency, or some relative speedup is difficult. Nonetheless, based upon our estimates of the performances of each machine in the ensemble, we estimate that in the ideal case the job should have completed in 121 minutes, rather than

Configuration	*Time (min)*	*Cray Eq*
Cray-1S	343	1.0
VAX 8700	1516	.23
14x VAX 8000	143	2.40

Table 2: Timing results for CHARM. The "*Cray Eq*" column gives the fraction of Cray-1S performance delivered by the configuration.

the 143 minutes reported in Table 2. This loss of performance may be due to communication overheads on the network. However, we did not have exclusive access to the machines during the timing runs, and the loss of performance may be simply due to other jobs competing with CHARM on some of the machines. Since the performance degradation is small, we have not performed the experiments necessary to pinpoint its source.

3.3 Finite element thermal analysis.

A number of applications have been developed both at Sandia and at Yale for the Intel iPSC hypercube[9]. The system calls used for communication and synchronization on the hypercube[13] (**send**, **recv**, *etc*) can be implemented on our network with appropriate Linda constructs. We have done this, and hope in this way to run a number of programs developed for the hypercube on our network. The hypercube programs were developed, of course, with some assumptions about the relative speeds of computation and communication operations. These assumptions are invalid for our network: the computation is faster and the communication slower. However, if we use only tens of processors, rather than the hundreds that the hypercube algorithms targeted, it is possible that acceptable performance will be achieved for some of these algorithms. Furthermore, this hypercube work represents a source of parallel programs that can be run on the network with little development effort, and that will stress the communication network. By studying the performance of these we will understand more clearly the limitations of this type of network parallel processing.

The first of these parallel hypercube applications that we have studied on the network is an equation solver for TACO3D, a finite element thermal analysis computer program[14] in use at Sandia. The iterative algorithm employed is a modified version of an SOR solver. The matrix and vectors are distributed in a block-wise fashion between the participating machines. Within a block, each processor performs the usual SOR computations, but the inter-block computations are modified to permit parallelization. A constraint on the current implementation, due to its origins as a hypercube program, is that the number processors participating must be a power of two.

There are two sources of performance degradation in the parallel equation solver as the number of participating processors in increased. The first arises from communication costs which increase with the number of processors. The second is that the number of iterations required to achieve convergence also increases with the number of processors. We modified the usual SOR procedure to permit parallelization of the computation. However, these modifications slow the rate of convergence when multiple processors are used. For instance, our sample problem required 19 iterations for convergence when a single procesor was used, but required 21 iterations when using two processors.

Configuration		*Times in seconds (speedup)*		
8700's	780's	Total	# Iter	Time/Iter
1	0	32.5 (1.0)	19	1.71 (1.0)
2	0	24.9 (1.3)	21	1.19 (1.4)
0	1	174.9 (1.0)	19	9.21 (1.0)
0	2	106.1 (1.6)	21	5.05 (1.8)
1	3	69.7 (2.5)	24	2.90 (3.2)

Table 3: Execution times for the TACO3D equation solver.

Timing results for solution of a sample TACO3D matrix are presented in Table 3. The matrix is of size 1000 with a bandwidth of 258. As shown in Table 3, we ran the solver on several combinations of Vaxes, including one and two 8700's, and one and two 780's. Unfortunately, we had access to neither four 8700's nor four 780's for these runs, but we did run the problem on a set of three 780's and one 8700. In the current implementation, the work is divided into pieces of equal sizes and no load-balancing is done, so we expect that the performance of this last set will be very similar to that for four 780's.

For each set of machines, we report several values. The column labeled "Total" gives the total execution time in seconds required to solve the matrix, and in parenthesis the speedup is given. The *speedup* s_n of an n processor parallel program is given by:

$$s_n = \frac{t_{serial}}{t_n}$$

where t_{serial} is the execution time for the serial version of the program, and t_n is the execution time for the parallel program running on n processors. In this particular program the serial version of the solver is identical to the parallel version running on only one processor, so $t_{serial} = t_1$. Thus we see that two 8700's runs the solver 1.3 times faster than only one. Although the execution times for the 780's are of course larger, the speedup results for these machines are better. This is because the time spent doing the numeric computations in these slower machines increases by a factor of over five, while the increase in communication overhead is much smaller. Thus, our estimated speedup for four VAX 780 computers on this application is 2.5.

As described earlier, in addition to the communication overhead there is an algorithmic source of perfor-

mance degradation in the parallel solver. This can be seen in the column in Table 3 labeled "# Iter" showing the number of iterations necessary to produce the solution, which increases as the number of processors increases. The final column labeled "Time/Iter" isolates the communication overhead by presenting the execution time for one iteration, and the corresponding speedup values in parenthesis.

4 Conclusion and summary.

We have successfully implemented several application programs on a local area network of VMS VAX computers using Linda. Two of the applications, BOHR and CHARM, decouple nicely into parallel pieces, and the network of VAXes runs these at a substantial fraction of the speed a Cray-1S supercomputer. For BOHR, which vectorizes on the Cray, this was achieved with a collection of eleven 8000 class VAXes. For CHARM, which vectorizes poorly on the Cray, only three of these VAXes provides over half of the performance of the Cray-1S.

The TACO3D equation solver provides a more stressing test of the performance of the Linda implementation. The parallel processes in this application are much more tightly coupled than in the other examples, and there is frequent interprocess communication. The resulting performance on the network is certainly worse than for the other two applications, but we find even these results encouraging. While TACO3D running on four machines is not twice as fast as on only two, it does run faster, and we have not reached the point where adding more machines actually slows the computations down. Thus, we may find the network Linda implementation useful for even fairly tightly coupled computations. Even if *efficiency* of the use of the computers is poor in some relative sense, the absolute performance may nonetheless be acceptable, and we can make productive use of a computer resource that might otherwise be unused.

5 Acknowledgements.

Work at Sandia was supported by the United States Department of Energy. Jerrold Leichter's work is supported by a grant from the Digital Equipment Corporation's Graduate Engineering Education Program, and by National Science Foundation grants CCR-8601920 and CCR-8657615, and ONR N00014-86-K-0310. We are grateful to Ray Cline for providing the parallel random number generator for the BOHR application.

References

[1] D. Gelernter. *Generative Communication in Linda.* Technical Report YALEU/DCS/RR-294, Yale University Department of Computer Science, November 1983.

[2] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 26–34, August 1986.

[3] R. Bjornson, N. Carriero, D. Gelernter, and J. Leichter. *Linda, the Portable Parallel.* Technical Report YALEU/DCS/TR-615, Yale University Department of Computer Science, February 1987.

[4] N. J. Carriero. *Implementation of Tuple Space Machines.* Technical Report YALEU/DCS/RR-567, Yale University Department of Computer Science, December 1987.

[5] S. Lucco. A heuristic Linda kernel for hypercube multiprocessors. In *Proceedings of the SIAM Conference on Hypercube Multiprocessors*, September 1986.

[6] N. J. Carriero. Private communication.

[7] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Trans. on Computers*, to appear.

[8] J. S. Leichter. *The VAX Linda-C User's Guide.* Technical Report YALEU/DCS/TR-520, Yale University Department of Computer Science, March 1988.

[9] R. Asbury, S. Frison, and T. Roth. Concurrent computers ideal for inherently parallel problems. *Computer Design*, 24(11):99–107, September 1985.

[10] N. Carriero, D. Gelernter, and J. Leichter. *Distributed Data Structures in Linda.* Technical Report YALEU/DCS/TR-438, Yale University Department of Computer Science, November 1985.

[11] R. E. Olson and A. Salop. Charge-transfer and impact-ionization cross sections for fully and partially stripped positive ions colliding with atomic hydrogen. *Phys. Rev. A*, 16(2):531–541, August 1977.

[12] O. E. Percus and M. H. Kalos. *Random Number Generators for Ultracomputers.* Ultracomputer note 114, New York University, Division of Computer Science, 251 Mercer Street, New York, NY 10012, February 1987.

[13] *iPSC Programmer's Reference Guide.* Intel Scientific Computers, Beaverton, OR, 1987.

[14] W. E. Mason. TACO3D—*A Three-Dimensional Finite Element Heat Transfer Code.* Technical Report SAND83-8212, Sandia National Laboratories, Livermore, CA, 94550, April 1983.