From Two Packets to One:

Increasing the Performance of Linda-LAN

by

Jason L. Christian

Project Report submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Approved:

James D. Arthur, Chairman

Osman Balci

Calvin J. Ribbens

February 3, 1997

Blacksburg, Virginia

Keywords: Linda, Linda-LAN, Parallel Processing, Network, Interprocess Communication

From Two Packets to One:

Increasing the Performance of Linda-LAN

by

Jason L. Christian

Dr. James D. Arthur, Chairman

Computer Science

(ABSTRACT)

Although networked-based computational environments such as Linda-LAN provide parallel processing capability, performance is still a major concern. One critical factor that can hinder the performance of any parallel processing environment is the high cost of interprocess communication. The goal of this project is to increase the performance of C-Linda programs executed in the Linda-LAN environment. With the current two packet message passing scheme, a receiving process listens for a header packet to arrive on a socket connection. Once the header packet has been received, the process then reads the data packet that follows. Reading the data packet may cause the process to block since the data may not have arrived. By modifying the two packet scheme to use only a single packet, the potential for blocking is removed since the header and data packets are guaranteed to arrive at the same time. Hence, the time spent waiting for the data is nullified, thus producing an increase in performance.

Acknowledgments

I would like to thank Dr. James D. Arthur for his guidance, insight, and extreme patience during this project.

I would also like to thank Chuck Schumann and Joe Chase for their allowing me to "pick their brain" about different issues concerning this work. The discussions and insight provided have been most helpful.

I am very grateful to my parents and brother for their unending love and support in all of my endeavors.

My deepest thanks go to my wife Deborah and my daughter Catherine. Their love and support have sustained and encouraged me throughout my effort to complete this project.

Table of Contents

List of Figures

1. Introduction

1.1 Background and Motivation

Because of the increased speed and performance of today's networks and microprocessors, networked workstations have become an attractive medium for parallel computation. By using the extensive hardware and software base that already exists in many organizations, parallel processing is achievable at a low cost [AMZA96]. Currently a number of different models exist that provide a basis for implementing and controlling network parallel processing environments. One such model is Linda-LAN [CLIN92].

Linda-LAN is a distributed parallel processing control framework based on the Linda paradigm [GELE88]. Linda-LAN utilizes the multiple processors provided by a network of workstations to create a low cost parallel processing environment. Specifically, Linda-LAN offers

- a coordination language that allows for the creation and coordination of multiple processes through the use of a shared memory called Tuple Space,
- simplified programming by allowing the programmer to develop the different processes of execution independently of each other,
- portability (Linda implementations exist on several different platforms), and
- the exploitation of idle network CPU cycles [CLIN92].

Although network parallel processing environments such as Linda-LAN provide the parallel processing capability, performance is still a main concern. One important factor that can hinder the performance of any parallel processing environment is interprocess communications. Processes on parallel machines communicate using the machine's local hardware by way of a shared global memory, point-to-point connections, or a combination of both. Processes implemented on workstations, however, communicate using networks and network protocols [STON92]. Because of the high latency and low bandwidth of networked systems, network parallel processing environments often exhibit a slower computational speed when compared to more conventional parallel architectures. To achieve acceptable performance levels, special attention must be given to the interprocess communications for network parallel processing environments.

1.2 Problem Statement and Proposed Solution

The goal of this project is to increase the performance of C-Linda programs executed in the Linda-LAN environment. More specifically, the current two packet message passing scheme is to be modified to use a single packet. With the current message passing scheme, a receiving process listens for a header packet to arrive on a socket connection. Once the header packet has been received, the process attempts to read the associated data packet that follows. Reading the data packet may cause the process to block since the data may not have arrived. By modifying the message passing scheme to use a single packet, the potential for blocking is removed because the data packet is guaranteed to have arrived. The time spent waiting for data is nullified, thus producing an increased performance level.

To achieve the goal, a copy of the Linda-LAN software is ported to the DEC 5000 workstation through minor modifications to the make files and source code. Following the port, the source code is examined to determine those modules that implement the message passing scheme. The message passing scheme is then modified to use a single packet. The modifications are tested against the original Linda-LAN environment by executing two benchmark programs in each environment. The results are then compiled and analyzed, to assess the extent to which the performance is enhanced.

1.3 Plan of Report

Chapter 2 outlines the background information, focusing on the functionality of Linda-LAN. The Linda paradigm is described, followed by an overview of the Linda-LAN environment and its components. Chapter 3 provides a discussion of the current message passing scheme and operations. The proposed modifications to the message passing scheme are then presented with an explanation of the expected results. Chapter 4 presents the experiment design. Each of the benchmark programs are briefly discussed followed by an explanation of the testing procedures. Chapter 5 presents the results and discussion concerning the benchmark testing. Chapter 6 provides a summary of our work and some concluding remarks.

2. Background

This chapter presents an overview of Linda. In particular, Section 2.1 discusses the Linda paradigm; Section 2.2 provides an overview of the Linda-LAN environment.

2.1 The Linda Paradigm

Linda is a parallel processing model whose operations emphasize process creation and coordination (communication and synchronization of processes). The principal focus is not with what is being computed but with how to exploit process creation and coordination to produce a coherent program. Linda's orthogonality, portability, ease of use, and efficiency have enabled implementations on several different computer platforms (including the Intel iPSC-2 hypercube, VAX/VMS, Sequent, AT&T Bell Lab's S/Net as well as Sun, DEC, Apple Mac II, and Commodore AMIGA 3000UX workstations) using many different programming languages (including C, FORTRAN, Modula-2, and Lisp) [CLIN94,GELE88].

2.1.1 Tuple Space

Linda uses a "tuple space" (TS) for process creation and coordination. TS is an "associative memory" repository for tuples, where a tuple is an ordered sequence of typed fields. Tuples can either be data tuples that are used for process coordination [e.g., <"Hi Limit",84.5> - a two-tuple consisting of a string and a real number] or "live tuples" that are used for process creation [e.g., <"pi",pi()> - a two-tuple consisting of a string and a function that returns the value for pi]. Live tuples contain one or more fields that are evaluated after it is placed in TS. This evaluation occurs independently and in parallel with the process that placed it in TS and other live tuples currently being evaluated. When the evaluation is complete, the live tuple transitions to a data tuple, containing the result of the evaluation [GELE88].

2.1.2 Template and Tuple Matching

Since TS is considered to be an associative memory, tuples do not have an "address". Retrieving a tuple from TS is achieved by matching tuple elements with those found in a requesting template. The template must contain the same number of elements, the same type of elements in the same order, and possibly the same values as the elements in the tuple to be matched. For example, suppose the tuple <"xyz",4,9.9> is in TS. The tuple can be referenced by requesting a tuple where the first element is a string, the second element is an integer, and the third element is a real number. It can also be referenced by requesting a tuple where the first element has the value 'xyz', the second element is an integer, and the third element is a real number. In other words, the tuple can be referenced by using any combination of its ordered element values and/or types.

To read the tuple <"xyz",4,9.9>, the operation *rd("xyz",?i,?r)* is used, assuming that *i* is an integer and *r* is a real number. *"xyz"* is considered an *actual parameter* while *i* and *r* are considered *formal parameters*. For this example, the element types, the element order, and the value of the first element are used to match the tuple. After matching, the formal parameters are assigned their corresponding tuple values: *i* is assigned the value 4 and *r* is assigned the value 9.9.

The tuple can also be read using the operation *rd(?s,?i,?r)*, assuming that *s* is a string, *i* is an integer, and *r* is a real number. In this example, the element types and their order are used to match the tuple. After a match is achieved, the formal parameters of the template are assigned their corresponding tuple values in the data tuple: *s* is assigned "xyz", *i* is assigned the value 4, and *r* is assigned the value 9.9 [GELE88].

2.1.3 Tuple Space Operations

Linda utilizes four primary operations and two variants for manipulating TS: *out, in*, *rd*, *eval*, *inp* and *rdp*.

out(t) causes the tuple *t* to be placed in TS. The elements of *t* are evaluated before they are placed in TS. For example, given the operation *out("pi",pi())*, the function *pi()* is first evaluated and the resulting tuple (e.g. <"pi",3.141593>) is placed into TS.

in(x) and *rd(x)* cause a tuple *t* in TS to be matched to the template *x*. The actuals in *t* are assigned to the formals in *x*. If more than one matching tuple is available, a tuple is arbitrarily chosen. If no matching tuple is found, the process that invoked the *in(x)* or *rd(x)* is suspended until a tuple is available. The *in(x)* operation also causes the matched tuple *t* to be removed from TS, whereas the *rd(x)* does not.

eval(t) causes the tuple *t* to be placed in TS. The elements of *t* are not evaluated until after the tuple is placed into TS. For example, given the operation eval*("pi",pi())*, the function *pi()* is not evaluated until after the tuple is placed into TS. Thus a process is created to evaluate *pi()*. Once that process is complete, the live tuple becomes a data tuple (e.g. <"pi",3.141593>).

The variant forms of *in(x)* and *rd(x)*, *inp(x)* and *rdp(x)*, perform the same operation as their counterparts except if a matching tuple is not available in TS, a 0 is immediately returned (the requesting process is not suspended). If a matching tuple is found, a 1 is returned and the actual-to-formal assignments are performed [GELE88].

2.2 Linda-LAN

Linda-LAN is a distributed parallel processing paradigm based on the Linda paradigm. It utilizes the multiple processors provided by a network of workstations to create a low cost parallel processing environment. Linda-LAN is primarily software based with a network being the communication medium. Its current implementation supports the C-Linda language [CLIN92].

Linda-LAN is divided into two subsystems: control and data. This separation permits a simpler view of the Linda environment and allows independent modification of the subsystems during research efforts. The subsystems and their components are interconnected using BSD sockets via the TCP/IP network protocol.

Figure 2.1 illustrates the Linda-LAN conceptual topology and Figure 2.2 illustrates a possible physical topology [CLIN94].

2.2.1 Control Subsystem

The control subsystem is responsible for controlling the Linda-LAN environment. It consists of the Linda-LAN Manager (L-Man) and the Linda-LAN Communications Managers (L-Com).

The L-Man is the system administrator of the Linda-LAN environment. On startup, L-Man instantiates L-Com processes on all participating workstations. When a Linda-LAN program is compiled and scheduled for execution, the L-Man is responsible for distributing the executable code to the participating workstations and for starting the data subsystem. The L-Coms assist in these operations. During program execution, the L-Man is also responsible for receiving and processing messages from the L-Coms and the Eval TS Manager (part of the data subsystem). After the Linda-LAN program has completed, L-Man terminates execution of the data subsystem. When the Linda-LAN environment itself is terminated, the L-Man terminates L-Com processes and then itself [CLIN94,ROBI94].

The L-Com is responsible for controlling the workstation on which it is active. When the Linda-LAN environment is started, the L-Man starts the L-Coms on all participating workstations. When a Linda-LAN program is executed, the L-Com starts the L-Kernel and executes commands from the L-Man to distribute the executable code. During program execution, the L-Com forwards messages from the L-Kernel to the L-Man.

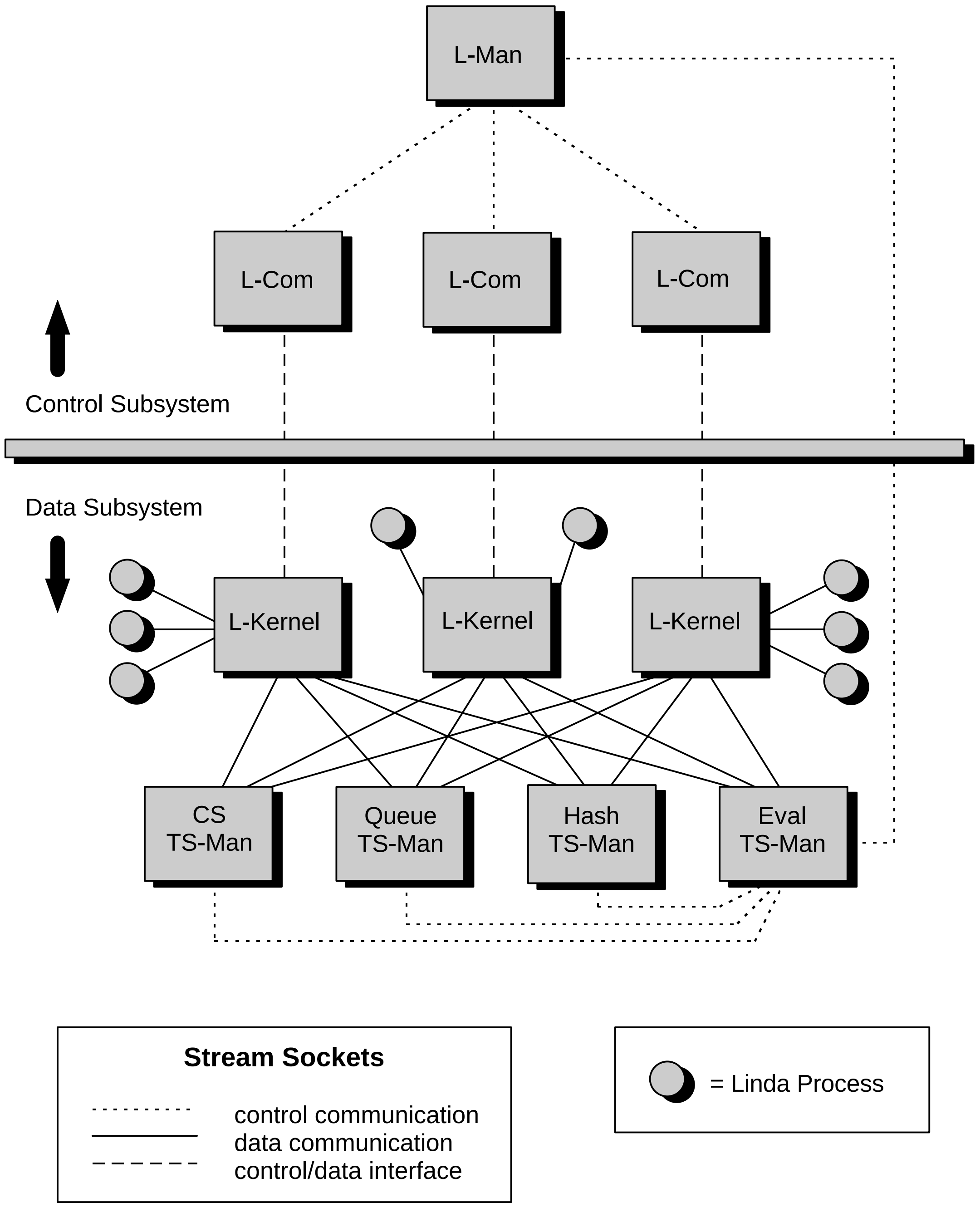
Likewise, messages received from the L-Man are either processed locally or passed on to the L-Kernel. Once the Linda-LAN program has terminated, the L-Com terminates the L-Kernel. The L-Com remains active until terminated by the L-Man.

2.2.2 Data Subsystem

The data subsystem is responsible for processing TS requests and for the communication between the Linda processes. It consists of TS, the TS managers, the Linda Kernels (L-Kernel), and the Linda processes. The data subsystem is primarily active just before and during the execution of a Linda-LAN program.

TS is a globally shared memory that is used to share data (tuple data) among the Linda processes. It is divided into four parts; each part is structured according to the tuple categories it is intended to process. Each TS part is managed by a different manager: CS TS-Man (counting semaphore), Queue TS-Man, Hash TS-Man, and Eval TS-Man. The TS managers are responsible for placing tuple data into their respective TS, receiving templates for matching the tuple data, and sending the tuple data back to the requesting Linda process. When the data subsystem is started, the Eval TS-Man is started first. It then starts the other TS managers. When the data subsystem is terminated, the Eval TS-Man is responsible for terminating the TS managers it created, and then itself [CLIN94].

The L-Kernel is the "middle man" between the TS managers and the Linda processes. Its main purpose is to eliminate the need for the Linda processes to directly connect to the TS managers, thus reducing the connection management burden on the managers. The L-Kernel receives tuple data and template data from the Linda processes and forwards them to the appropriate TS manager. Likewise, it receives the response tuple data from the TS managers and forwards it on to the appropriate Linda process. The L-Kernel also receives messages from the L-Man via the L-Com during the distribution of the Linda program [CLIN94, ROBI94].



Source: Robinson, Patrick. "Distributed Linda: Design, Development, Characterization of the Data Subsystem." MS thesis, Virginia Polytechnic Institute and State University, 1994.

Figure 2.1 Conceptual Topology of Split Tuple Space Linda-LAN

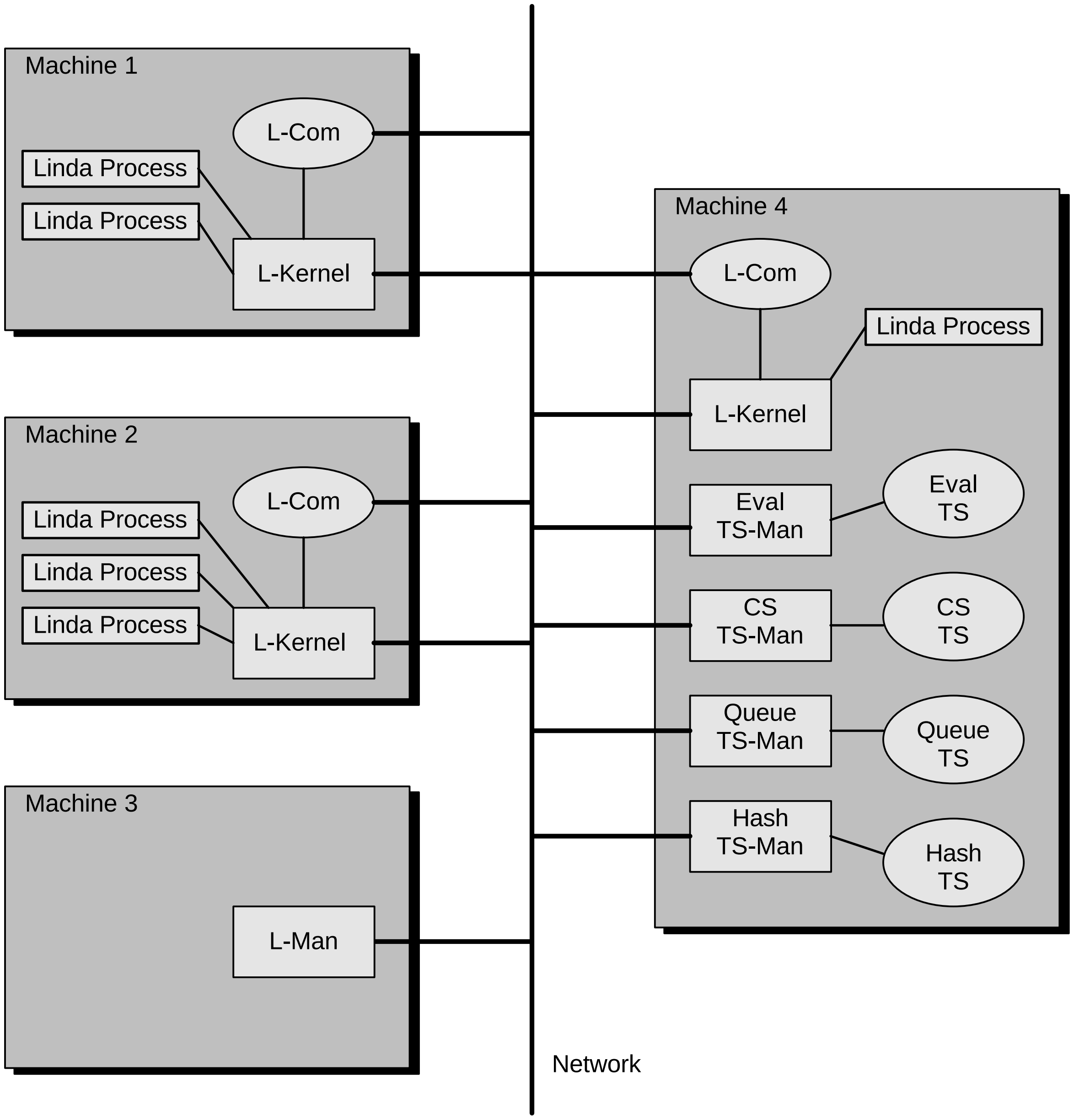


Figure 2.2 Example of Physical Topology of Split Tuple Space Linda-LAN

3. Solution Approach

The current Linda-LAN implementation uses a two packet message passing scheme consisting of a header packet and a data packet. After a process receives a header packet, it attempts to read the associated data packet. If the data packet has not yet arrived, the process will block. By modifying the message passing scheme to use a single packet, the header and data packet will arrive at the same time, thus removing the potential for the receiving process to block while attempting to read the data packet.

This chapter presents the two message passing schemes being investigated in this project. Section 3.1 presents the current message passing scheme implemented in the Linda-LAN environment. Section 3.2 presents the proposed modifications to the message passing scheme and its benefits.

3.1 Current Message Passing Implementation

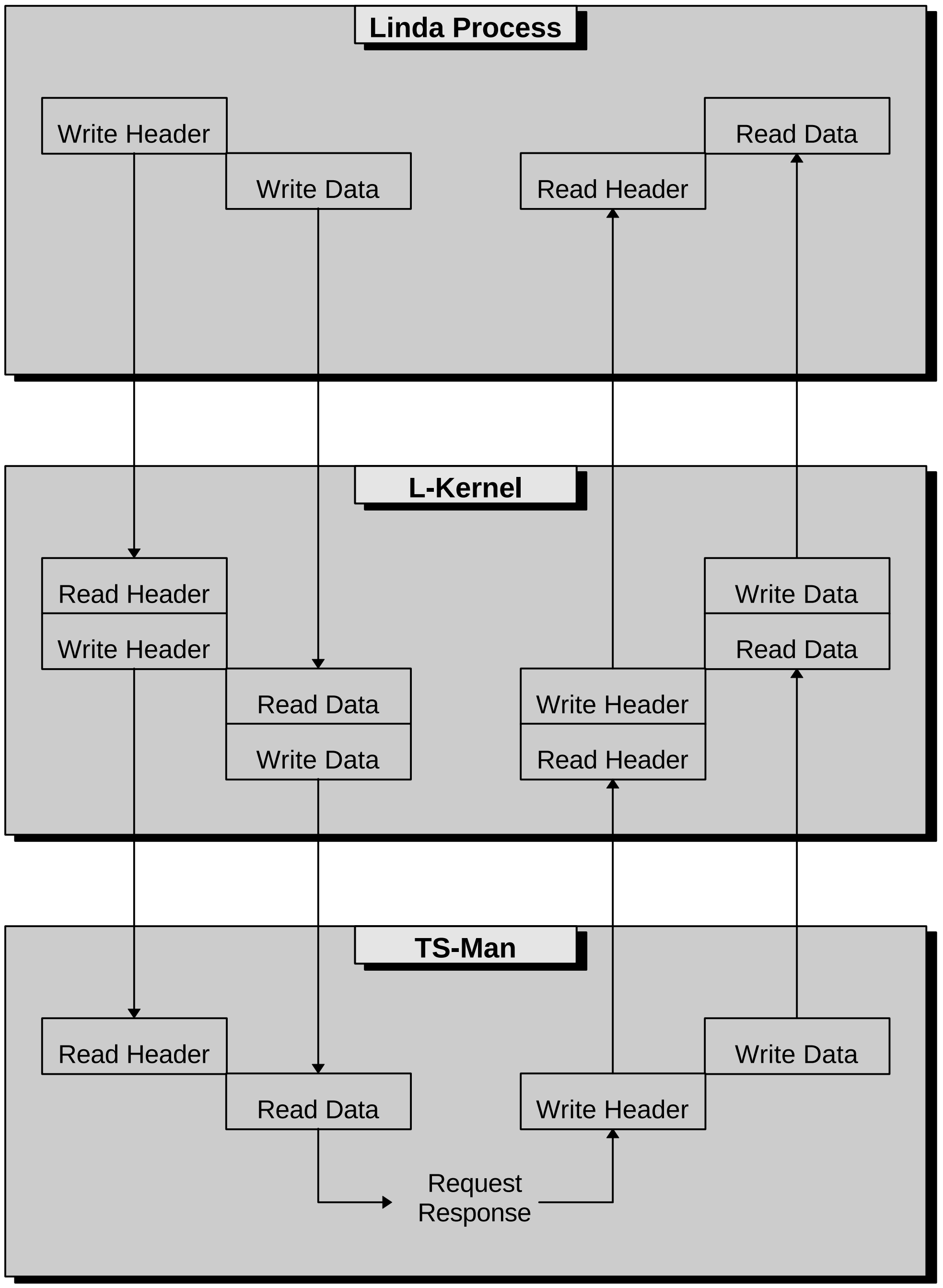
The current message passing implementation uses a two packet scheme: a fixed size header packet and a variable size data packet. Figure 3.1 illustrates how the two packet message is passed through the data subsystem.

The fixed size header is sent first. Since it is a fixed size, the receiving process always knows how much data to read. Once the header has been read, the receiving process determines the size of the data packet from the information in the header. The data packet is then read and the message can be processed. The two packet scheme allows the receiving process to know exactly how much data is to be read without having to perform any special data handling techniques (such as looking for markers within the data and complex buffering schemes). It also allows the receiving process to begin processing the header packet while the data packet (which may be quite large) is being sent [ROBI94].

The TS managers and the L-Kernels process the header and data packets as a single message, as if only one packet is received. For example, when an L-Kernel receives a header packet, it will receive the data packet and forward it on to the appropriate TS manager before receiving any other packets from the connected Linda processes. Likewise, when a TS manager receives a header packet, it will receive the data packet and process it before receiving any other packets from the connected L-Kernels. Since the L-Kernels and TS managers process the header packet and data packet before continuing, they may end up blocking if the data packet has not yet arrived. Delays will be introduced because the blocking process is unable to receive other requests that may be available for processing [CLIN94].

3.2 New Message Passing Implementation

The revised implementation uses a single, variable size packet for message passing. The packet contains the header information and data information as in the two packet scheme, but both the header and data information are passed as a single packet. Figure 3.2 illustrates how a single packet message is passed through the data subsystem.



Source: Robinson, Patrick. "Distributed Linda: Design, Development, Characterization of the Data Subsystem." MS thesis, Virginia Polytechnic Institute and State University, 1994.

Figure 3.1 Data Path For Two Packet Messaging Scheme

First, a packet is created containing the header and data. The packet is then sent to the receiving process. Since the data is buffered by the socket subsystem, it can still be treated as two packets (a header followed by the data) by the receiving process. The receiving process reads the fixed sized header, determines the size of the remaining data from the header information, and then reads the rest of the packet. The message is then processed.

Because the receiving process can still treat the header and data as two separate packets, the Linda-LAN code for reading the data does not need to be modified. The only modifications necessary are to the code that sends the data.

Several advantages are gained by changing from a two packet scheme to a single packet scheme.

- The problem of header and data packet interleaving is eliminated. Since only one packet is being sent, there is nothing to be interleaved.
- An improvement in performance is expected. When the receiving process performs a socket read, there may be no data available. If this happens, the process will block and the operating system scheduler determines when the process can again continue. With the two packet scheme, the receiving process is triggered when the header packet arrives. The process must perform two socket reads, one for the header packet that has already arrived and one for the data packet. Since the data packet is not guaranteed to have arrived, the receiving process has the potential to block. With the single packet scheme, the receiving process still performs two socket reads, but the data packet is

guaranteed to have arrived. Since there is no potential for blocking, a decrease in the possible delay is expected, thus providing an increase in performance.

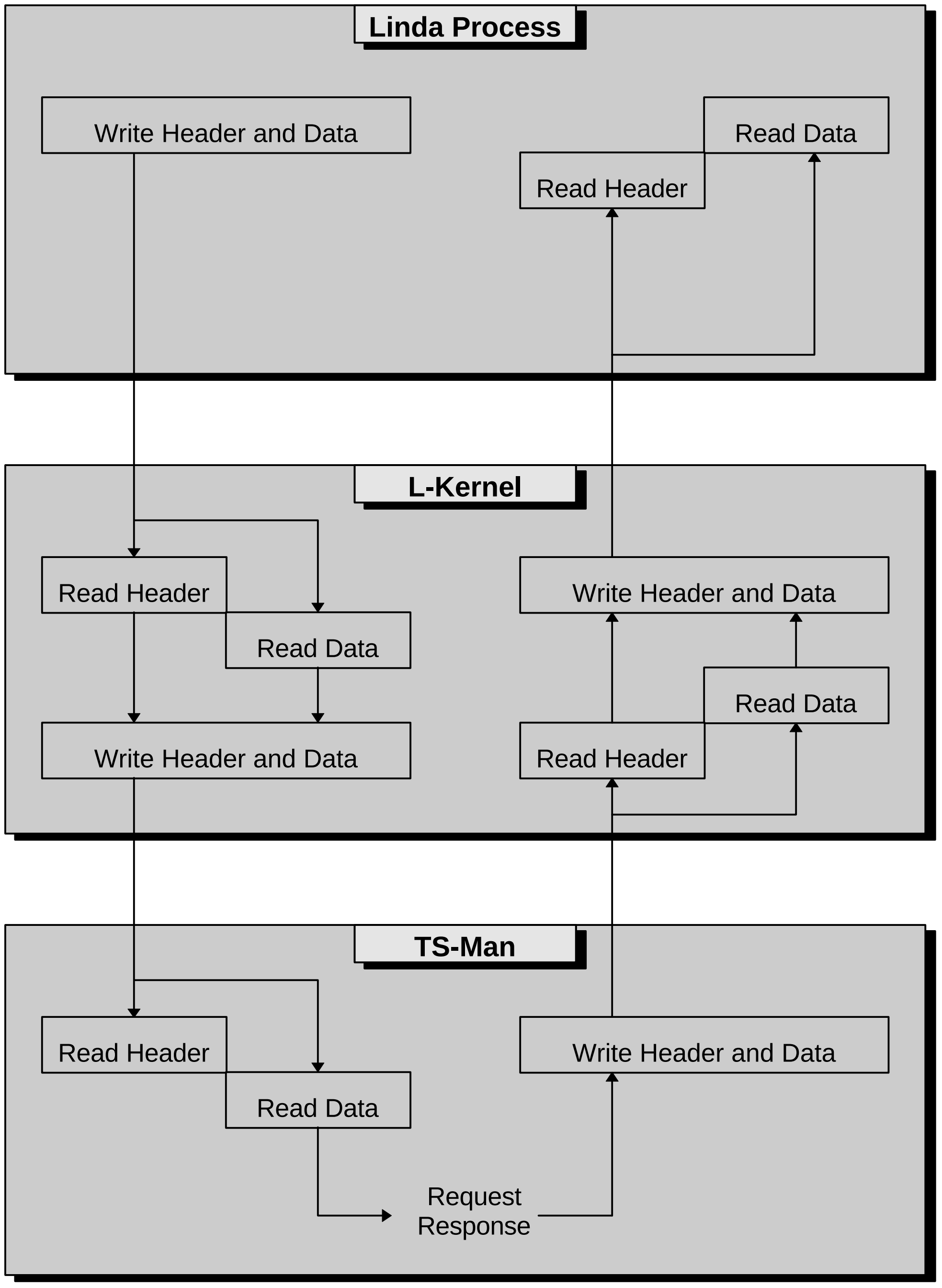


Figure 3.2 Data Path For Single Packet Messaging Scheme

4. Experiment Design

The modifications to the message passing scheme are tested using two simple test programs. The first program, jmatrix.cl, is computationally intensive while the second, jblock.cl, is input/output intensive. By showing that the message passing modifications produce an increase in performance for these two programs, it can be inferred that many other applications which contain similar characteristics will also exhibit an increase in performance. A brief overview of each program is presented in Sections 4.1 and 4.2 followed by an explanation of the testing procedures presented in Section 4.3.

4.1 Jmatrix.cl Benchmark

The computationally intensive program is jmatrix.cl. The program performs matrix multiplication on two symmetric matrices by creating concurrent processes to perform the row/column multiplications. Even though the program is computationally intensive, an increase in performance is still expected. The modifications to the message passing scheme do not directly affect the speed of the computations, but do affect the speed at which the data used in the computations is made available, through the input/output operations.

First, workers (Linda processes) are created that perform the multiplication of a row in the first matrix by a column in the second matrix.

```
for ( x = 1; x <= num_workers; x++ )
   eval(worker(x));
```

Next, each row in the first matrix and each column in the second matrix are placed into TS along with their corresponding row and column indices. Since symmetric matrices are being multiplied, the matrices are represented in the main program using an array. The ith row and/or column is found by simply indexing the array at the ith position and counting off the number of elements in the dimension. For example, if two symmetric matrices of dimension three are multiplied, the matrix array could contain the values <2, 4, 6, 8, 10>. To determine the second row and the second column in each of the matrices, all that is necessary is to index the array at the second position, thus giving the elements <4, 6, 8>.

```
for ( x = 0; x < (2*dimension); x++ )
   val[x] = x + 2;

for ( x = 0; x < dimension; x++ ) {
   out(ROW,x,&val[x]:dimension);
   out(COLUMN,x,&val[x]:dimension);
   }
```

After placing the rows and columns into TS, the indices of the row/column pairs to be multiplied are placed into TS. The indices are retrieved by the workers to determine what multiplications need to be performed.

```
for ( row = 0; row < dimension; row++ )
   for ( column = 0; column < dimension; column++ )
      out(TASK,row,column);
```

The dimension of the matrices is then placed into TS. Placing the dimension into TS allows the workers to gain access to the matrices' dimensions and also enables the worker processes to start. As the multiplication results are made available by the worker processes, they are then removed from TS by the main program.

```
/* out dimension which starts the workers */

out(DIM,dimension);

/* retrieve results from the workers */

num_results = dimension * dimension;

for ( x = 0; x < num_results; x++ )
   in(RESULT,?row,?column,?result);
```

During execution, each worker checks TS to see if any row/column index pairs exist. If none are found, the worker terminates. If a pair is found, it is removed from TS, the corresponding row data and column data are retrieved from TS, and the multiplication is performed. After the result is placed into TS, the worker again checks TS for another row/column index pair.

```
while ( !xit ) {
   /* see if there are still columns and rows to be
      multiplied */

   result = inp(TASK,?row_index,?col_index);

if ( result == FAIL )
    xit = TRUE;
else {
   /* get row and column data to be multiplied */

   rd(ROW,row_index,?row:dimension);
   rd(COLUMN,col_index,?col:dimension);

   /* do multiplication */

   result = 0;

   for ( x = 0; x < dimension; x++ )
      result += row[x] * col[x];
   /* out result */

   out(RESULT,row_index,col_index,result);
   }
}
```

For testing, a series of matrix multiplications are performed. The first set consists of varying the matrix dimension from ten to fifty in increments of ten with the number of workers held constant at five. The second set consists of varying the number of workers from one to five with the matrix dimension held constant at ten. In each case, the execution time for the matrix multiplication is measured (the measurement does not include the creating of the workers and placing the matrix data into TS).

For the first test set, as the size of the matrices increases, the performance level is expected to increase. The increase is due to the increase in the number of input/output operations. For the second test set, the increase in performance is expected to remain relatively constant. Since the matrices' dimensions do not increase, the number of input/output operations remains constant.

4.2 Jblock.cl Benchmark

The input/output intensive program is jblock.cl. It's purpose is simply to place a block of data into TS and then to remove it. The modifications to the message passing scheme directly affect the input/output operations, therefore an increase in performance is expected.

```
while ( size <= (total_blocks*step) ) {
   sprintf(buff,"size - %d",size);
   timer_split(buff);
   out("block",oblock:size);
   in("block",?iblock:size);

   size += step;
   }
```

For testing, ten thousand iterations are performed with an initial block size of thirty-two bytes. For each iteration, the size of the data block is increased by thirty-two bytes and the amount of time necessary for placing the block into TS and then removing it is measured. Since the modifications to the message passing scheme affect the input/output operations by a constant factor (reducing two packets to a single packet), the performance is expected to increase by a constant factor.

4.3 Testing Procedures

The modifications to the message passing scheme are tested on single and dual DEC 5000 workstations (see Figure 4.1 and 4.2 for the testing topologies). A test run consists of executing a test program using the two packet scheme and then executing the same program using the single packet scheme. Interleaving using the old message passing scheme and then the new message passing scheme helps to ensure that each program is executed under similar system load conditions. Each test run is performed ten times to give a good sampling of the results. The results are then averaged over the ten runs and the results are plotted for comparison. For the jblock.cl results, a regression analysis is used.

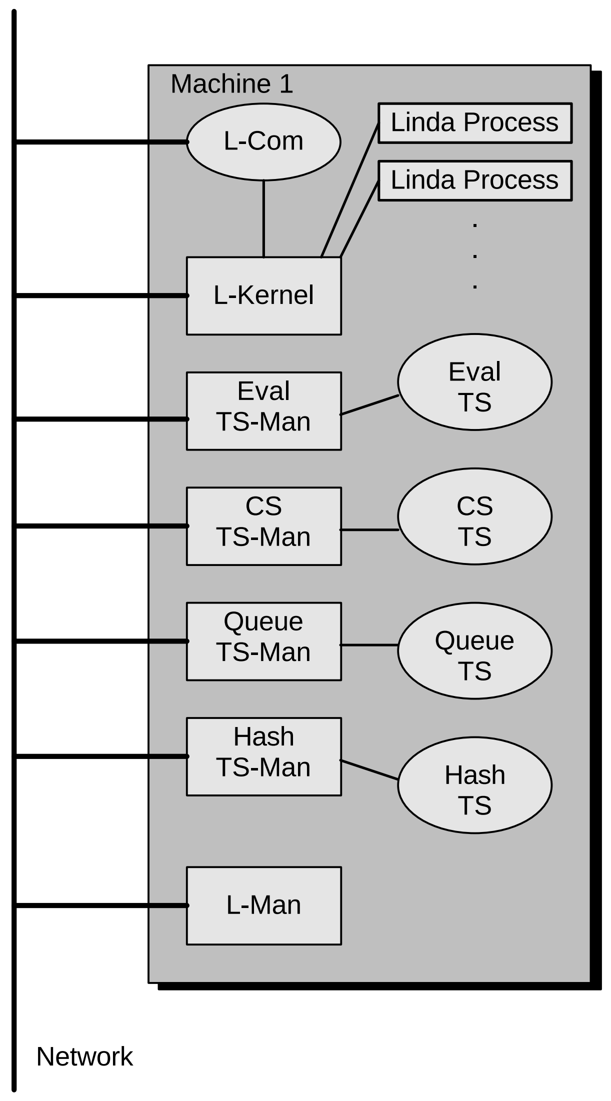


Figure 4.1 Physical Topology for Single Workstation Testing

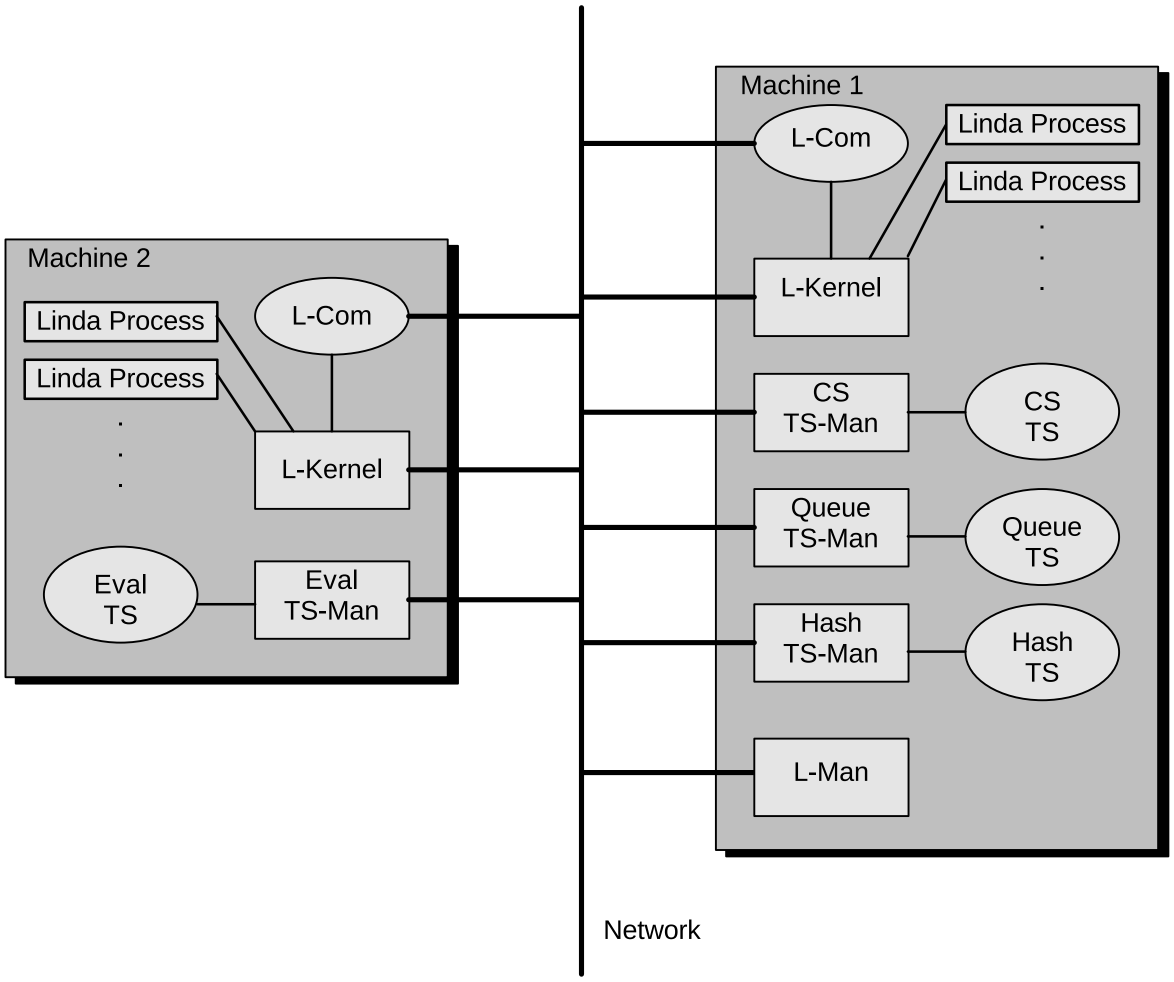


Figure 4.2 Physical Topology for Dual Workstation Testing

5. Experimental Results

This chapter presents the results obtained from the execution of the benchmark programs along with a brief discussion. Section 5.1 presents the results of the computationally intensive program jmatrix.cl. Section 5.2 describes the results of the input/output intensive program jblock.cl.

5.1 Jmatrix.cl Results

The computationally intensive program jmatrix.cl performs matrix multiplication on identical square matrices. An increase in performance is expected due to the input/output operations which are performed, providing the worker processes access to the row/column pairs for multiplication. Figure 5.1 and 5.2 present graphs illustrating the execution times resulting from the test runs of jmatrix.cl with the number of workers (Linda processes) held constant and the array dimensions varied. Figure 5.3 and 5.4 present graphs illustrating the execution times resulting from the test runs with the array dimensions held constant and the number of workers varied. The graphs indicate the following:

- The execution times for the single packet scheme are less than the execution times for the two packet scheme. This decrease in execution time is a result of the change from a two packet message to a single packet message. As stated previously, a receiving process has the potential to block using the two packet scheme, whereas the potential is removed using the single packet scheme. Figure 5.1 and 5.2 show a decrease in the overall execution times of approximately 25% and 28%, respectively. Figure 5.3 and 5.4 show a decrease in the overall execution times of approximately 27% and 30%, respectively.

- The execution times for the dual workstations are less than the execution times for a single workstation. Linda-LAN distributes the workers to the various workstations in a round-robin fashion. During processing on the dual workstations, the workers are evenly distributed among the workstations. This gives each worker more CPU time, thus decreasing the execution times. As a specific example, with the single packet scheme and 30 dimensional matrices, Figure 5.1 shows that the average execution time is approximately 27 seconds, while Figure 5.2 shows that the average execution time is approximately 18 seconds.
- In Figure 5.1 and 5.2, the execution times increase as the matrix dimension increases in a polynomial fashion due to (1) the increase in the number of computations that are performed and (2) the increase in the number of TS input/output operations that are performed. As a specific example, in Figure 5.1 with the single packet scheme, 20 dimensional matrices require 8,000 multiplications and 400 input/output operations, giving an average execution time of approximately 12 seconds. With 30 dimensional matrices, 27,000 multiplications and 900 input/output are required, giving an average execution time of approximately 27 seconds.
- In Figure 5.3 and 5.4, the execution times tend to increase as the number workers increase. A threshold exists at which point the time required for managing an additional worker outweighs the increase in performance gained by adding that worker. As workers are added beyond that threshold, the

execution times will increase. In Figure 5.3, the threshold is approximately three workers. In Figure 5.4, the threshold cannot be determined without further testing.

5.2 jblock.cl Results

The input/output intensive program jblock.cl outputs and inputs blocks of data to/from TS. Since the modifications to the message passing scheme directly affect the input/output operations, and increase in performance is expected. Figure 5.5 and 5.6 present graphs illustrating the execution times resulting from the test runs of jblock.cl. The graphs indicate the following:

- As with jmatrix.cl, the execution times for the single packet scheme are less by a constant factor than the execution times for the two packet scheme. Figure 5.5 and 5.6 show a decrease in the overall execution times of approximately 11% and 14%, respectively.
- The execution times are approximately the same for the single and dual workstations. For testing, the physical topology for the dual workstations is the same as that for the single workstation except that the Eval TS manager along with approximately half of the Linda processes are executed on a separate workstation. Since jblock.cl does not start any Linda processes, the Eval TS manager does not process any messages and thus the executing environment for both tests are essentially the same.

A regression analysis is used for analyzing the data due to the sporadic nature of the data points. This can be attributed to two factors. First, the smallest measured execution time is 3.906 milliseconds, with most of the measured execution times being a multiple of this value. This indicates that the smallest quantum of time for the test system is approximately 3.906 milliseconds. Second, the resulting raw data is averaged over the ten test runs. The averaging along with the imprecise measurements tend to increase the scattering of the data points, thus making it harder to analyze the results without the use of a regression analysis.

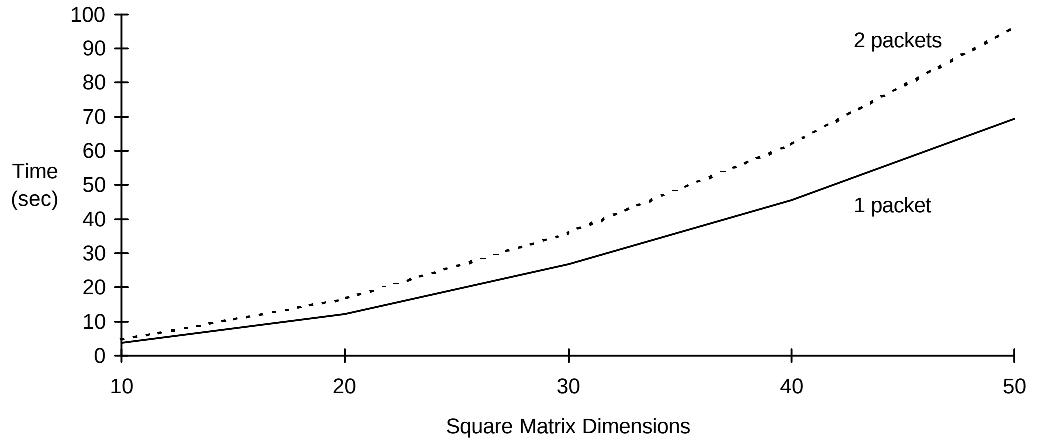


Figure 5.1 jmatrix.cl execution times (5 workers, 1 workstation)

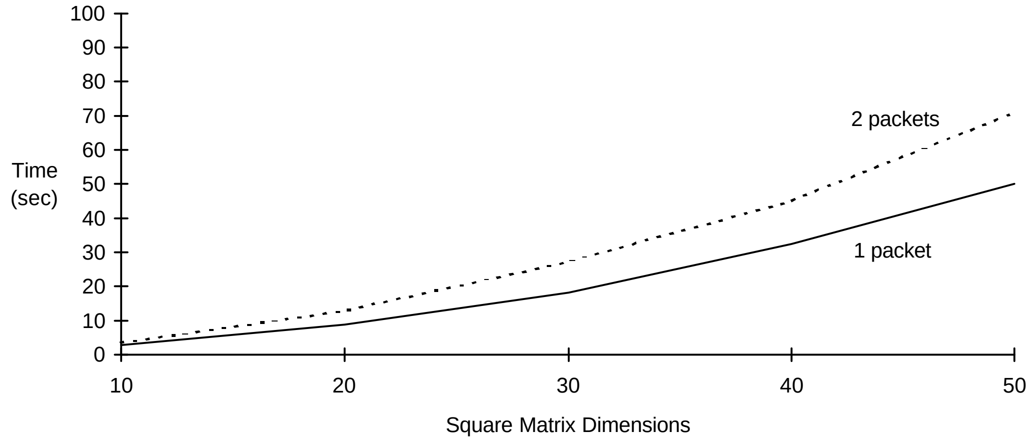


Figure 5.2 jmatrix.cl execution times (5 workers, 2 workstations)

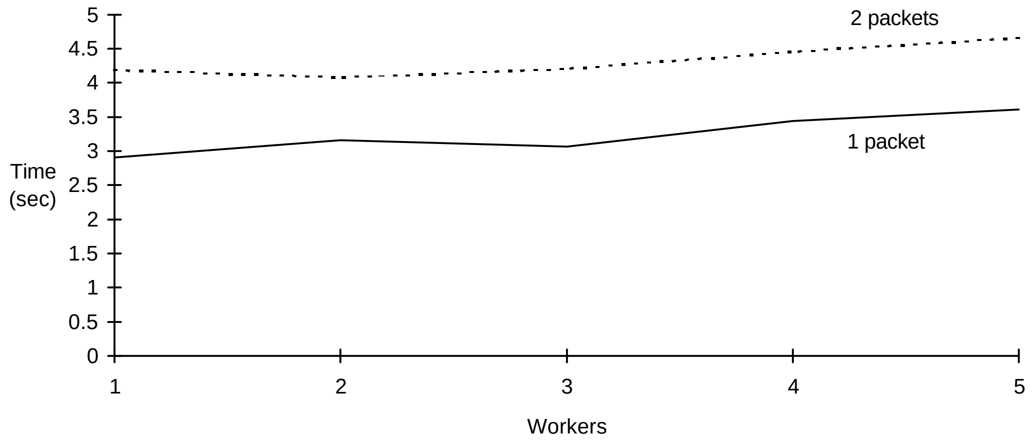


Figure 5.3 jmatrix.cl execution times (10 dimensional array, 1 workstation)

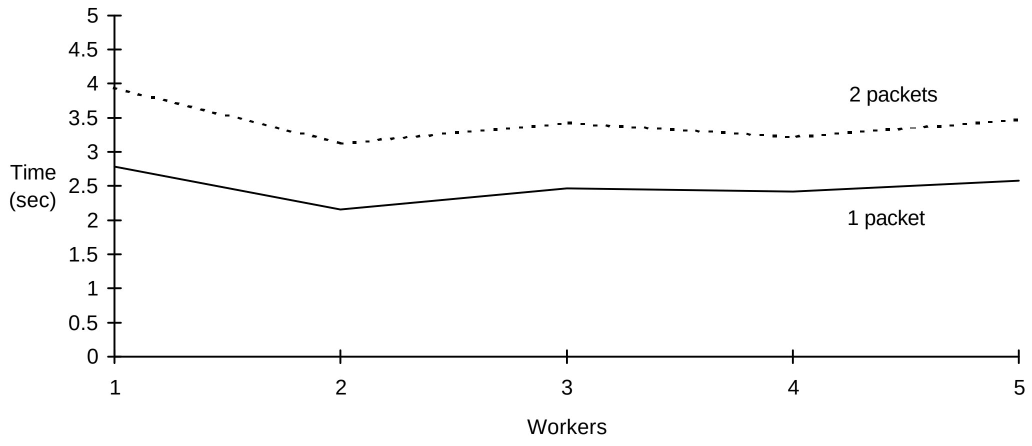


Figure 5.4 jmatrix.cl execution times (10 dimensional array, 2 workstations)

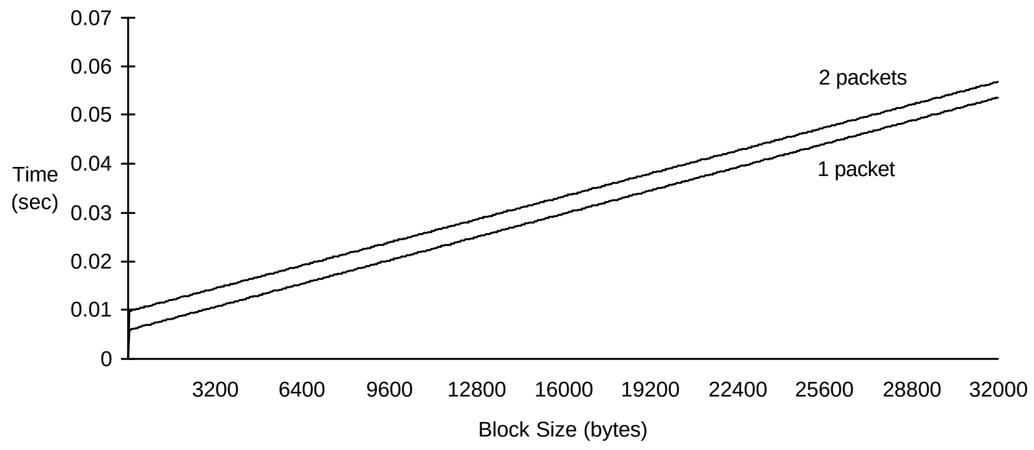


Figure 5.5 jblock.cl execution times (1 workstation; regression analysis)

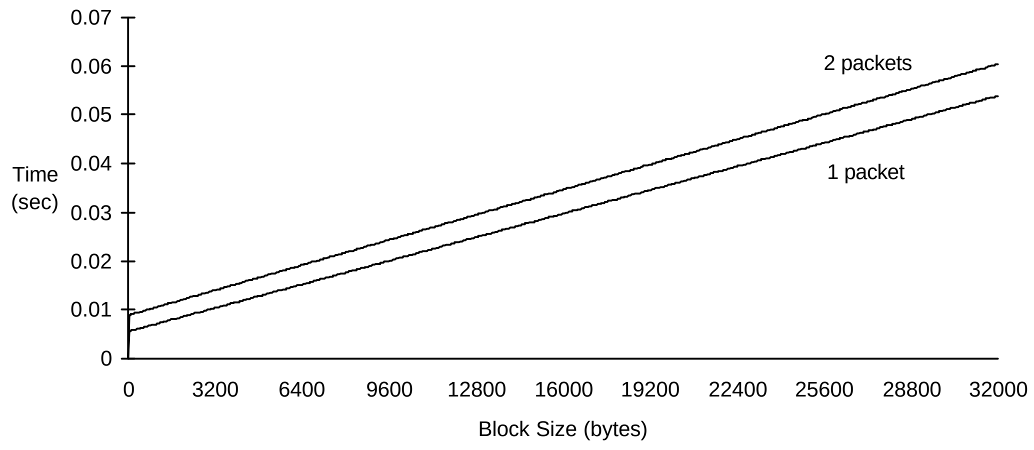


Figure 5.6 jblock.cl execution times (2 workstations; regression analysis)

6. Conclusion

The goal of this project is to increase the performance of C-Linda programs executed in the Linda-LAN environment. The performance increase is achieved by the modification of the current message passing scheme to use a single packet for the header and data as opposed to two separate packets. Using a single packet allows the buffering management to be handled by the TCP/IP networking protocol, thus permitting the header and data to be sent as a single packet, but read as two separate packets. Since the header and data are guaranteed to arrive at the destination at the same time, the potential for blocking while waiting for the data to arrive is removed. Hence, the time spent waiting for the data is nullified, thus producing an increase in performance.

Even though an increase in performance is obtained, there is always room for improvement. Topics still exist that hold the potential to further increase the performance of C-Linda programs. These topics relate specifically to interprocess communications.

- Data compression techniques could be used for decreasing the transmission time required to send the data through the network. This would also benefit the TS managers performance by storing the compressed tuple data in TS. Tuple matching could then be performed on the compressed data, reducing the comparison time [SCHU93].
- The L-Kernels could be eliminated to allow the Linda processes to communicate directly with the TS managers. This would decrease the number of network

transmissions by one-half, giving a decrease in data transmission time. The decrease in transmission time may, however, be at the expense of an increase in execution time since the burden of managing the socket connections would then fall on the TS managers [ROBI94].

- BSD sockets provide benefits such as portability and guaranteed packet delivery. There may be, however, other techniques for performing interprocess communications that are more efficient and better suited for the Linda-LAN environment [SCHU93].

Appendix A - Benchmark Program Listings

```
/*************************************************************************
*     Program: jmatrix.cl                                                *
*     Purpose: This program is a C-Linda program which multiplies two    *
*              square matrices together.  It is used to test the impact  *
*              of modifying the message passing structure such that the  *
*              header and data are passed as a single packet instead     *
*              of two separate packets.                                  *
*  Programmer: Jason L. Christian                                        *
*************************************************************************/

#include <stdio.h>
#include <stdlib.h>

/*******************************************************************
*  <<<<<<<<<<<<<<<<<<<<<<<<  Constant(s)  >>>>>>>>>>>>>>>>>>>>>>>>>  *
*******************************************************************/

#define MAX_DIM         256           /* maximum matrix dimension  */

#define FAIL            0

#define FALSE           0
#define TRUE            !FALSE

#define COLUMN          "column"      /* tuple data identifiers    */
#define DIM             "dimension"
#define RESULT          "result"
#define ROW             "row"
#define TASK            "task"

/*******************************************************************
*  <<<<<<<<<<<<<<<<<<  Function Prototype(s)  >>>>>>>>>>>>>>>>>>>>  *
*******************************************************************/

int worker( int num );

/*******************************************************************
*  <<<<<<<<<<<<<<<<<<<<<<<<<  real_main  >>>>>>>>>>>>>>>>>>>>>>>>>  *
*******************************************************************/

real_main( int argc, char **argv )

  {  /* real_main */

     char buff[32]           ;    /* temp buffer                     */
     int  column             ;    /* column index                    */
     int  dimension          ;    /* matrix dimensions               */
     int  num_results        ;    /* number of results               */
     int  num_workers        ;    /* number of workers               */
     int  result             ;    /* result of row/col mult.         */
     int  row                ;    /* row index                       */
     int  val[2*MAX_DIM]     ;    /* matrices values                 */
     int  x                  ;    /* loop index                      */



     if ( argc < 3 )
       fprintf(stderr,"Usage:  %s <dimension> <num workers>\n",argv[0]);
     else
```

```
        {
          /* get the number of workers and the matrix dimensions */

          dimension   = atoi(argv[1]);
          num_workers = atoi(argv[2]);
          /* eval the workers */

          fprintf(stderr,"eval() %d workers\n",num_workers);

          for ( x = 1; x <= num_workers; x++ )
            eval(worker(x));

          /* initialize matrices values */

          for ( x = 0; x < (2*dimension); x++ )
            val[x] = x + 2;

          /* out matrices rows and columns */

          fprintf(stderr,"out() %d dimension matrix\n",dimension);

          for ( x = 0; x < dimension; x++ )
            {
              out(ROW,x,&val[x]:dimension);
              out(COLUMN,x,&val[x]:dimension);
            }

          /* out multiplication tasks to be performed */

          fprintf(stderr,"out() multiplication tasks\n");

          for ( row = 0; row < dimension; row++ )
            for ( column = 0; column < dimension; column++ )
              out(TASK,row,column);

          /* out dimension which starts the workers */

          fprintf(stderr,"starting workers\n");

          out(DIM,dimension);

          /* start timer */
          start_timer();

          /* retrieve results from the workers */

          num_results = dimension * dimension;

          for ( x = 0; x < num_results; x++ )
            {
              in(RESULT,?row,?column,?result);
/*
              fprintf(stderr,"result(%d,%d) = %d\n",row,column,result);
*/
            }

          /* print timer results */

          sprintf(buff,"dim - %d, workers - %d",dimension,num_workers);
          timer_split(buff);
          print_times();
        }
```

```
  }  /* real_main */


/**********************************************************************
*     Function: worker                                                *
*      Purpose: This function is used in the multiplication of two   *
*               matrices.  The rows and columns of the matrices are  *
*               in tuple space.  A row from the first matrix and a   *
*               column from the second matrix are retrieved and      *
*               processed.  The resulting scalar value is placed     *
*               in tuple space for the main program to access.       *
* Parameter(s): none                                                  *
*    Return(s): none                                                  *
**********************************************************************/
int worker( int worker_num )

  {  /* worker */

     int  col[MAX_DIM]       ;    /* matrix column                 */
     int  col_index          ;    /* matrix column index           */
     int  dimension          ;    /* matrices dimensions           */
     int  result             ;    /* temporary result              */
     int  row[MAX_DIM]       ;    /* matrix row                    */
     int  row_index          ;    /* matrix row index              */
     int  x                  ;    /* temp loop variable            */
     int  xit                ;    /* loop exit flag                */


     xit = FALSE;

     /* get matrices dimension; this is also the start flag */

     rd(DIM,?dimension);

     fprintf(stderr,"worker %d started\n",worker_num);

     /* continue processing as long as there are rows and columns to */
     /* be multiplied                                                 */

     while ( !xit )
       {
         /* see if there are still columns and rows to be multplied */

         result = inp(TASK,?row_index,?col_index);

         if ( result == FAIL )
           xit = TRUE;
         else
           {
             /* get row and column data to be multiplied */

             rd(ROW,row_index,?row:dimension);
             rd(COLUMN,col_index,?col:dimension);

             /* do multiplication */

             result = 0;

             for ( x = 0; x < dimension; x++ )
               result += row[x] * col[x];

             /* out result */
```

```
                out(RESULT,row_index,col_index,result);
              }
          }

      }  /* worker */
```

```
/************************************************************************
*     Program: jblock.cl                                               *
*     Purpose: This program is a C-Linda program which out()'s and in()'s *
*              successively larger blocks of data.  It is used to test    *
*              the impact of modifying the message passing structure such *
*              that the header and data are passed as a single packet     *
*              instead of two separate packets.                           *
*  Programmer: Jason L. Christian                                        *
************************************************************************/

#include <stdio.h>

/*******************************************************************
*  <<<<<<<<<<<<<<<<<<<<<<<<  Constant(s)  >>>>>>>>>>>>>>>>>>>>>>>>  *
*******************************************************************/

#define MAX_SIZE 16384            /* max out() and in() buffers     */

/*******************************************************************
*  <<<<<<<<<<<<<<<<<<<<<<<<<  real_main  >>>>>>>>>>>>>>>>>>>>>>>>>  *
*******************************************************************/

real_main( int argc, char **argv )

  {  /* real_main */

     char buff[32]           ;    /* temp buffer                    */
     char iblock[MAX_SIZE]   ;    /* in() data block                */
     char oblock[MAX_SIZE]   ;    /* out() data block               */
     int  size               ;    /* size of block to out()         */
     int  step               ;    /* block step size                */
     int  total_blocks       ;    /* number of blocks to out()      */


     if ( argc != 3 )
       {
         printf("Usage: %s <blocks> <step>\n", *argv);
         exit(1);
       }

     /* get total blocks to use and the step size */

     total_blocks = atoi(argv[1]);
     step         = atoi(argv[2]);

     /* make sure that our test buffers are large enough */

     size = total_blocks * step;

     if ( size > MAX_SIZE )
       {
         printf("too many blocks and/or step size too large\n");
         printf("(blocks * step size < %d\n\n",MAX_SIZE);
       }
     else
       {
         size = step;

         printf("jblock -- blocks: %d, step: %d\n\n",total_blocks,step);

         start_timer();

         /* loop and out() and in() successively larger blocks */
```

```
      while( size <= (total_blocks*step) )
        {
          sprintf(buff,"size - %d",size);
          timer_split(buff);

          out("block",oblock:size);
          in("block",?iblock:size);

          size += step;
        }

      timer_split("done.");

      /* display results */

      print_times();
    }

}  /* real_main */
```

Appendix B - Porting Linda-LAN

Porting Linda-LAN to the DEC 5000 was straight forward. Only minor changes were necessary to the make files and to the source code to enable proper compilation and installation.

- The make files were reworked by adding macros and better organizing the dependency rules.
- Unix has an 'install' command that is used to install software in a given directory. Due to a user privilege problem caused by the 'install' command in the make files, it was replaced with the 'cp' command.
- Several Linda-LAN header files contained hard coded definitions indicating the Linda-LAN installation directories. The files were modified to use a single macro definition passed in by the make file.

One of the original goals of this project was to port the Linda-LAN software to the DEC Alpha. After creating the binaries on the Alpha with much trouble, Linda-LAN would not execute properly. Upon closer investigation into the source code for the C-Linda compiler, it was discovered that the code had been poorly designed. Memory allocations and many of the memory manipulating procedures were treating memory addresses as 32-bit integers. On a 32-bit machine, this will work properly since memory addressing is done using 32 bits. On a 64-bit machine such as the Alpha, memory addressing is performed using 64-bit addressing. Thus, when assigning a memory address to a 32-bit integer, many of the addresses are truncated, causing problems. Since 32-bit addressing was found throughout the code and the port was not the main focus of the project, Linda-LAN was instead ported to the DEC 5000 workstations.

One other problem found in the code was that certain file operations were implemented using the C data type long. The same data was then read as integer data. Again, on a 32-bit machine, this does not cause a problem but on a 64-bit machine it does.

References

[AMZA96] C. Amza, et. al., "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, Vol. 29, No. 2, February 1996, pp 18-28.

[BERN89] Donald J. Berndt, C-Linda Reference Manual DRAFT, Beta Version 2.3, Scientific Computing Associates, Inc., September 26, 1996.

[CLIN92] George Cline and James D. Arthur. "Linda-LAN: A Control Parallel Processing Environment", TR 92-37, Virginia Polytechnic Institute and State University, 1992.

[CLIN94] George Cline. "A Control Framework for Distributed (Parallel) Processing Environment", M.S. thesis, Virginia Polytechnic Institute and State University, 1994.

[GELE88] David Gelernter, "Getting the Job Done", BYTE Magazine, November 1988, pp 301-308.

[ROBI94] Patrick Robinson. "Distributed Linda: Design, Development, Characterization of the Data Subsystem", M.S. thesis, Virginia Polytechnic Institute and State University, 1994.

[SCHU91] Charles Schumann, Kenneth Landry, and James D. Arthur, "Comparison of Unix Communication Facilities Used in Linda", Proceedings of the 1991 Virginia Computer Users Conference, September 1991.

[SCHU93] Charles Schumann. "Distribution of Linda Across a Network of Workstations", M.S. thesis, Virginia Polytechnic Institute and State University, 1993.

[STON92] Harold S. Stone. High-Performance Computer Architecture. Reading, Massachusetts: Addison-Wesley Publishing Company, 1992.

Vita

Jason L. Christian was born December 4, 1965, in Richmond, Virginia. He spent his grade school years in the Richmond area, graduating from Thomas Dale High School in 1984. In the fall of that same year, he began attending Virginia Tech. It was there that he met his wife, Deborah. After graduating in 1988 with a BS in Electrical Engineering, he began working for Compute-Rx as a computer programmer. He was married on August 18, 1990 and a year later decided to return to Virginia Tech on a part-time basis for a Master's degree in Computer Science. It was during this time that his daughter, Catherine, was born (May 23, 1995). His Master's project “From Two Packets to One: Increasing the Performance of Linda-LAN” was completed in the spring of 1997.