A Control Framework for Distributed (Parallel) Processing Environments

by

George E. Cline

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute & State University

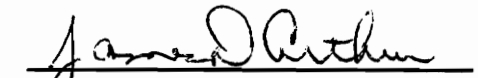
in partial fullfillment of the requirements for the degree of

Master of Science

in

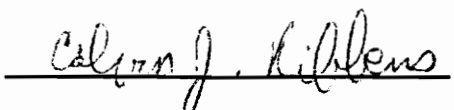
Computer Science

Approved:



James D. Arthur, Chairman

Dennis G. Kafura



Calvin J. Ribbens

August 1994

Blacksburg, Virginia

A Control Framework for Distributed (Parallel) Processing Environments

by

George E. Cline

James D. Arthur, Chairman

Virginia Polytechnic Institute & State University
Department of Computer Science
Blacksburg, VA 24061

Abstract

A control framework for distributed (parallel) processing environments is introduced. The control framework supplies a centralized control sub-system within a distributed environment to manage the resources available on a network of computers. Special attention is centered on the execution characteristics of the framework as reflected within the Linda paradigm. Linda is a conceptually simple coordination language that supports parallelism. Linda and the control framework are combined to create Linda-LAN, a distributed parallel programming environment that utilizes a local area network of computers to provide a low-cost parallel processing solution. This thesis presents the design of the Linda-LAN environment along with analysis on the applicability of the control framework within other distributed environments. In addition, the control sub-system's execution characteristics of stability and scalability are substantiated.

Key words and phrases:: Linda, Linda-LAN, Result Parallelism, Agenda Parallelism, Specialist Parallelism, Control Framework, Scalability, Stability

Acknowledgements

Special thanks go to my father, George. His constant support and wisdom throughout my life will be missed by his recent passing. I wish to thank my mother, Sue, and the rest of my family. Their love and support are never forgotten. My most cherished thanks go to my wife, Mojee. Her patience, understanding, caring and love has been invaluable throughout the years.

I also would like to thank my advisor, James "Sean" Arthur. His guidance and understanding are appreciated.

Table of Contents

Page

Page

List of Figures

Figure Page

List of Tables

1

INTRODUCTION

The computing problems of today are becoming increasingly more complex and time consuming. For example, mapping out the human genome requires a significant amount of computing power and time to analyze the approximately 100,000 genes in the human cell. In order to develop physical maps of each chromosome and to determine the sequences of various DNA chains, powerful parallel computers, vector processors and special-purpose computers are a necessity; however, the expense of such machines far exceeds the limits of many users. Alternative methods of achieving parallel performance at an economical price are desired. One such alternative exploits the idle CPU cycles existing on local area networks. With the increase in the computing power of workstations and their declining costs, the unused computing power attached to a local area network (LAN) can effectively be transformed into a parallel processing environment [THEIM89]. Exploiting such an environment requires a specification and operational framework that is portable, easy to use and efficient.

The Linda[1] parallel programming paradigm [CARRI89a, GELER85a and GELER85b] provides an effective parallel computational framework. In particular, Linda

- establishes a coordination language which allows a parallel program to perform process creation, synchronization, inter-communication and the sharing of distributed data structures through a logically shared object memory, called tuple space[BERND89],
- reduces the programmer's onerous task of dealing with the simultaneites of multiple executing processes within a parallel program, by allowing the programmer to develop each executing process independently from the rest,
- supports parallelism through replication as well as partitioning[2], and
- is portable; implementations of Linda systems exist on several different machine architectures and network configurations[3].

Although the Linda paradigm provides an environment with the benefits listed above, the paradigm is not a complete solution for a distributed environment. Users of a distributed environment require a tremendous amount of flexibility and control. Workstations are added and removed at various time intervals. Some workstations are assigned to particular individuals. Other workstations are dedicated to perform specific tasks. At other times, users require complete control over their workstations.

---

1 Linda is a trademark of Scientific Computing Associates.

2 While many parallel methodologies require a task to be partitioned into distinct and disjoint sub-tasks, which execute in parallel, Linda provides the added capability of replication. That is, a specific task can be duplicated many times, and each instance simultaneously operates on a different set of data.

3 Implementations include the Sequent and Encore multi-processor shared memory machines as well as VAX/VMS Ethernet networks.

In order for the basic Linda paradigm to succeed as a specification and operational framework within a distributed environment, additional requirements such as transparency, scalability and stability must be provided. Users of a distributed system often remain unaware of the number of processors being utilized; the existence of multiple networked computers are transparent (i.e. invisible) to users. Processors can be added and deleted from the network at various intervals; the environment must scale with those fluctuations. As with all computer systems, a certain degree of stability must be maintained; executing programs must behave in a predictable manner. To meet these requirements, network resources must be effectively and efficiently managed. By injecting a level of control into the specification and operational framework, the added requirements of transparency, scalability and stability can be met.

1.1 Motivations & Objectives

The focus of this thesis is the creation of a *control framework* for distributed parallel processing environments. In particular, the control framework is placed within a distributed parallel processing environment based on the Linda paradigm that operates on a local area network of low cost personal computers. The establishment of a controlled environment provides for the management of utilized network resources. The employment of Linda, a conceptually simple parallel programming paradigm, facilitates the utilization of such an environment and the application of low cost personal computers offers an economical solution to parallel processing. The objectives of this thesis are three-fold. First, the Linda-LAN[4] environment, especially its control sub-system is studied in detail. Second, the applicability of the control framework within other distributed processing environments is examined. Third, the desirable execution characteristics of scalability and stability are shown to exist within the

[4] Linda-LAN is a software-based framework designed by work from this thesis, but implemented by the Linda research group at Virginia Polytechnic Institute and State University. The LAN is the only "specialized" hardware needed to support the parallel environment which executes C-Linda programs.

control sub-system. We hypothesize that scalability and stability are achieved within the control sub-system by effectively and transparently controlling environment processors and the allocation of instantiated program processes to processors[5].

Scalability exists in two forms. The first form deals with the scalability of network processors and the second form deals with program processes. Network processors are added and removed arbitrarily during the life of the system. In order for the environment to remain useful to its users, the execution of the control sub-system must scale relative to the change in the number of processors. If relative scalability is not achieved within the control sub-system, users will not consider the environment to be an effective parallel processing alternative. The execution must also be able to scale with the change in the number of instantiated program processes. A dependency may exist between the number of program processes and program input. For instance, a parallel program to find occurrences of a string within a text file could be written to create a new search process for each line of data found within the text file. If this form of scalability is not attained, programmers may be burdened with the responsibility of controlling the number of utilized processes from within the program source code. The environment would not be regarded as being easy to use by its programmers.

A second execution characteristic that is desirable for any processing environment is stability. If an environment can produce predictable patterns of program execution, the environment is judged to be stable. Only a stable environment provides the usability and dependability required by its users. Stability is especially desirable to this research endeavor in that it will help to identify those areas of the control sub-system which require enhancement and possibly provide information on future research.

---

[5] The term effective refers to the efficient and appropriate selection of environment processors. Transparency refers to the hidden selection of a processor from an executing program so that the program is not required to be especially written, compiled, or linked for the environment, based on number of processors used.

Without having stability within the control sub-system, stability within the environment cannot be achieved.

1.2 Thesis Overview

This thesis begins in Chapter 2 by describing selected distributed processing environments for which the control framework can be applied. Background on the Linda paradigm, which was the chosen specification and operational framework for Linda-LAN, is given as well as the programming paradigms used with Linda. Chapter 3 presents a detailed overview of the current Linda-LAN environment and its design. The physical and logical topologies of the environment are shown. In addition, an examination of the control sub-system is supplied. Special attention is given to the control sub-system's allocation strategy of program processes to environment processors. Chapter 4 centers on the experimental results substantiating the stability and scalability of the control framework. Chapter 5 provides concluding remarks on the contributions of this thesis and addresses the advantages and disadvantages of the control framework and Linda-LAN. In addition, the control framework's relevance to other distributed environments is summarized.

2

BACKGROUND

One of the objectives of this chapter is to provide background information on a selection of programming languages and models of concurrency and parallelism. Environments based on the languages and models surveyed may be applicable to the control framework being introduced in this thesis. The background is limited to the areas of the programming languages and models that pertain to discussions on the control framework found in later chapters of this thesis. In particular, program instantiation and termination, process creation and the methods for inter-process communication are discussed.

Two additional objectives are to be met within this chapter. First, information on Linda and the Tuple Space model is conveyed. The background provided on Linda and its tuple space are necessary for the understanding of the Linda-LAN environment which is presented in later chapters. Second, a set of parallel programming paradigms used in the justification of scalability and stability within the control framework are discussed.

2.1 Survey of Distributed Environments

This section briefly describes selected programming languages and models of concurrency and parallelism upon which distributed environments may be based. For ease of discussion, the languages and models surveyed are divided into three general classes or categories. The first class consists of languages and models which are based on message passing. Concurrent Sequential Processes (CSP) [HOARE78] and occam [MAY83] are reviewed. The second category describes environments based on remote procedure calls (RPC) [BIRRE84 and BRINC78] and monitors[HOARE74]. Emerald [BLACK87], Ada[ADA83] and Concurrent Pascal [BRINC75, BRINC93] are surveyed. The third class includes distributed environments which are based on conceptually shared memory mechanisms. Orca[BAL90, BAL92] and Multilisp [HALST85] are reviewed. The Linda model also falls into this category, but is discussed in a section of its own.

2.1.1 Message Passing Environments

Numerous distributed environments support message passing. One of the most well-known models of message passing is Communicating Sequential Processes (CSP) introduced by C.A.R. Hoare. CSP greatly influenced the development of the occam language. Both CSP and occam provide communication between concurrent processes by means of synchronous message passing. The sending of a message from one process and the receiving of that message by another process are bound together as a single task. In order to send a message, the sending process must designate the receiving process and the receiving process must designate the sending process. If the sending process sends a message before the receiving process is ready to receive the message, the sender blocks until the message is received. The

converse is also true, if the receiving process tries to receive a message which has not been sent, the receiving process blocks until the sending process dispatches the message.

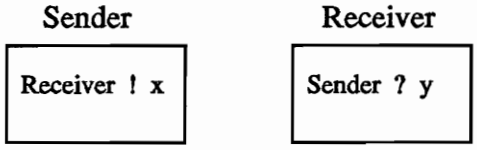


Figure 2-1. CSP inter-process communication.

```
CHAN channel :
PAR
   channel ! x        (Sender)
   channel ? y        (Receiver)
```

Figure 2-2. Occam inter-process communication.

Figures 2-1 and 2-2 illustrate the commands used for communications between CSP and occam processes, respectively. Notice that CSP designates the sending and receiving processes by name, while occam designates the sender and receiver by way of a channel. The communication paths between the processes can be determined prior to the execution of the program; either during compilation or dynamically before the program begins execution. The communication paths can also be determined at execution time, if the processes published their names and addresses on the distributed network.

CSP and occam are considered to be static languages. No new processes are dynamically created as a program executes. Processes may be allocated to processors at compile time or during execution. A program in CSP and occam is considered terminated when all processes become passive and messages

are no longer being transmitted between the processes. The above information becomes important in our discussions within the later chapters on how the control framework addresses program instantiation and termination.

2.1.2 Remote Procedure Call and Monitor Based Environments

Environments based on monitors and remote procedure calls comprise the second category of programming languages and models to be surveyed that support distributed environments. The languages and models of this class encompass a different form of message passing. Systems based on monitors and remote procedure calls provide communications between processes by means of a procedure call mechanism. To communicate between distributed processes, the first process invokes a procedure defined within a second process. When the first process invokes the procedure, a call is made to a stub procedure that packages a message containing the name of the procedure and its arguments. The packaged message is then sent to the second process, where another stub process is waiting for incoming messages. The receiving stub unpackages the message and invokes the procedure located within the receiving process. While the procedure is being executed on the remote machine, the calling stub and its associated process is suspended. When the process has completed execution of the remote procedure, a reply is packaged by the second stub and returned to the first process.

The details of locating the second process on the distributed network are transparent to the programmer. The stub procedures handle the interaction between the processes. Within RPCs, a global name server is used to locate called procedures. Monitor locations, on the other hand, are normally fixed and known by the operating system. In both cases, the processes that contain the remote procedures and monitor

procedures are normally started and stopped independently from the other calling processes. No constraints are placed on processes dynamically creating other new processes that call the remote procedures.

2.1.2.1 Ada

The Ada language possesses a form of remote procedure calls. Tasks[1] within Ada can be called by other tasks via **entries**. A task receives a call to one of its entries by executing an **accept** statement. A call to an entry corresponds with the sender process described previously and the accept statement corresponds with the receiving process. The main difference between the formal RPC and the Ada constructs deals with the location of the procedures or tasks. Tasks can be located on any given processor of a distributed network, while remote procedures are located at designated processors.

2.1.2.2 Concurrent Pascal and Emerald

Concurrent Pascal extends the sequential language Pascal with multiple processes and monitors. Unlike a parallel language, a concurrent language requires its processes to share the same address space. This requirement makes Concurrent Pascal ill-suited for a distributed environment, since all global data is defined within the same module. Languages similar to Emerald have been able to overcome the sharing of data within a distributed program. Emerald is an object-based language and it encapsulates shared information within a single entity, an object. An Emerald object consists of a unique network identifier,

---

[1] Tasks within Ada are considered entities whose exectuions proceed in parallel. Each task is executed by a logical processor of its own.

the data local to the object, the operations (procedures) that can be invoked on the object, and an optional process. When an Emerald object contains a process, it is termed active; otherwise, it is termed passive.

Active objects make invocations on the operations of other objects, and they in turn can invoke other objects. This series of invocations can continue to any depth. In addition, multiple concurrent invocations on the same object may occur. Synchronization on the object is provided by monitors. Remote invocations are similar to the example of the sender and receiver at the beginning of this sub-section. Objects within Emerald are mobile; they may migrate at any time to other locations on the distributed environment. Emerald also supports local invocations. Objects can be assigned permanent locations at compile time or prior to execution. Programmers of Emerald are also able to manipulate the locations of objects with primitives such as Locate, Move, Fix, Unfix, and Refix. Alternatively, the programmer can choose to ignore where objects are located and have the Emerald system determine the appropriate locations. Appropriate locations are determined during compile time and execution time.

Emerald objects which contain an optional process are started after the object's *initially* section is complete. Each object has an optional *initially* section which is a parameterless operation that executes exactly once when the object is created. When all active processes become passive, the program is terminated. Once again, the above information becomes more relevant when we discuss the control framework and how it can support these types of environments.

2.1.3 Conceptually Shared Memory Environments

The final class of languages and models focuses on distributed environments which conceptually provide shared memory. Environments in this class of languages and models provide a logical shared memory, which hides the actual physical location of the shared data. Processes executing on different processors share data objects or access a logical memory space. The physical locations of the shared memory can reside within a single physical address space or it can be replicated and distributed across the system or it can be divided into classes or types and distributed across the system.

2.1.3.1 Orca

Conceptually shared memory environments are less frequently encountered than the other two classes. Ocra is a conceptually shared memory language that provides the sharing of data by utilizing objects of abstract data types. The language is not entirely object-based. Orca merely provides objects for the sharing of data. All data-objects in Orca are passive and processes are active. By having passive objects, only processes are allowed to operate on a data-object. Orca replicates shared data-objects to the local memories of all processes. Each process directly reads its own copy of a shared data-object without having to do any inter-process communications. If available, physical multicasting is used to update the shared data-objects during write operations by a process [BAL90]. The multicasting speeds up the updating of shared data-objects by simultaneously broadcasting information to all relevant processes.

A program in Orca starts with a single process. Additional child processes are created explicitly using the `fork()` construct:

```
fork name(actual-parameters) [on (cpu-number)];
```

The `fork()` statement allows a process to be assigned to a specific processor. All processors are sequentially numbered and the **on** option is used to select the specific processor. If the **on** option is omitted, the child process is created on the same processor as the parent. Unlike Emerald, processes are not migrated implicitly by the system to other processors. The sharing of data is accomplished through the passing of data-objects as parameters from parent processes to child processes. When all processes become passive, the program is terminated.

2.1.3.2 Multilisp

Multilisp is a second conceptually shared memory language. Although this language is primarily designed for multiprocessor systems, others have modified its design to support distributed processing[KELLE84]. One shared name space is normally provided for all Multilisp tasks. The **future** construct is used to create Multilisp tasks and for the synchronization of tasks [LISTO88]. When the (future X) construct is executed, a concurrent evaluation of X begins immediately. The process which initiated the future does not block while waiting for the future to complete its evaluation. The future is considered *undetermined* until its value has been computed, at which point it becomes *determined.* While undetermined, the initiating process will block only if an operation (such as an addition) that needs the value of the future is encountered. When the future is determined, the process will continue. If the process does not encounter an operation that requires the determined future, the process will continue executing.

Multilisp also provides the **pcall** construct for introducing parallelism. The (pcall A B C) construct causes the concurrent evaluation of the expressions A,B and C. When all processes have completed their evaluations, the Multilisp program terminates. A Multilisp program is started by the execution of a

single task. The information presented in this section will be used in the discussions that appear later on the control framework and its application to other distributed environments.

2.1.4 Applicability of the Surveyed Environments

The languages and models of concurrency and parallelism surveyed in the previous sections provide a variety of approaches to process creation, inter-process communication, and program instantiation and termination. From a programmer's viewpoint, the mechanisms used to perform these tasks are quite different. However, when viewing these tasks from a system implementation viewpoint, the tasks become fundamentally the same. For instance, process creation involves the allocation of the process to a network processor. Inter-process communication requires the establishment of communications paths between active processes. In order to start a program, the executable code must be distributed or made available to the applicable processors. Program termination requires all executing processes to be terminated and all resources released. Although the tasks appear differently to the programmer of each programming language or model of concurrency and parallelism, the tasks are quite similar from an implementation viewpoint. These similarities are among the reasons for the development of a control framework.

A control framework is established within this thesis to support the tasks of process creation, inter-process communication, and program instantiation and termination. Many environments, which are based on the above languages and models of concurrency and parallelism, are applicable. The control framework injects a level of control into an environment. The objective is to provide for effective and efficient management of network resources. For example, the allocation of program processes to network processors can be more efficiently performed if global knowledge of the network is known. The control

framework provides this type of information. The following chapter presents the control framework and its implementation within the Linda-LAN environment. Discussions about applying the framework within the environments surveyed is also provided. The next section provides information on the Linda programming paradigm which is necessary for better understanding the Linda-LAN environment.

2.2 Linda

Linda is a coordination language rather than a complete parallel programming language. A coordination language [CARRI90 and ZENIT90] provides the primitives to create processes as well as coordinate communication among the processes. By virtue of being a coordination language, Linda primitives can be introduced into many base programming languages. The original implementation, using C as its base language, exploits a preprocessor approach which transforms Linda operations into C supported calls to the run-time system. Implementing the Linda primitives in this way is especially useful when parallel systems need to be developed in multiple languages. Linda has been embedded in a wide variety of other languages -- C++, FORTRAN, various Lisps, PostScript, Joyce, Modula-2 and soon Ada [GELER90].

When one discusses the Linda paradigm, two characteristics are often touted: its ease of use and its portability. From a conceptual standpoint, parallel programming within the Linda framework is intentionally high level, which is exactly why it is so flexible and powerful. Not so surprising, however, this high-level approach is at the root of Linda's greatest criticism - its questionable performance [DAVID89]. Linda does exhibit acceptable performance on both shared and distributed memory parallel (MIMD) machines [BJORN88, BJORN89 and CARRI87]. In addition, Linda has demonstrated its applicability on local area network platforms, although performance suffers for applications with tightly coupled processes [LELER88 and WHITE88].

Because Linda does not rely on any specific type of architecture, Linda programs are easily ported to a wide variety of machines with little or no modification. Machines currently hosting Linda include workstations such as Sun, DEC, Apple Mac II and Commodore AMIGA 3000UX. Linda has also been ported to parallel machines such as the Sequent, S/Net and the Hypercube.

The Linda approach supports process creation and inter-communication through a shared data/process repository called Tuple Space (TS). Linda provides operations to generate data tuples (`out`), to read data tuples (`rd`), and to remove them from TS (`in`). Tuple Space not only contains data tuples but also process tuples (created with the `eval` operation) which are often called "live tuples." These process tuples are instantiated and are eventually replaced by a data tuple when the instantiated process finishes executing. TS can also be used to share data structures among processes and to synchronize the order of actions that processes perform.

Figure 2-3 provides examples of the basic C-Linda operations. Note that the `eval()` operation creates a live tuple. When the eval is executed, the process issuing the `eval()` continues to execute. The issuing process does not wait for the `strlen` function to complete processing. The `strlen` function is considered to be a live tuple while executing concurrently with the issuing process. When the `strlen` function completes its execution and returns a value, the live tuple becomes passive and is replaced by the data tuple ("string", 5). The interactions of the operations with Tuple Space are associative. There is no physical addressing of TS. Two additional operations (`inp`) and (`rdp`) are predicate versions of the (`in`) and (`rd`) operations. These operations are non-blocking and return 1 if a matching tuple is retrieved from tuple space, or 0 otherwise. Figure 2-4 illustrates communication between sending and receiving processes.

```
int x, y;

out("string", 25);      /* A data tuple with two fields (a string
                           followed by an integer) is evaluated
                           and added to TS. */

out("string", 10);      /* Adds a second data tuple to TS. */

in("string", ?x);       /* Withdraws some arbitrary tuple from
                           TS that has two elements.  The first of
                           which is a string with the value
                           "string" and the second is an integer.
                           The variable x is assigned the value of
                           the integer (x will be 25 or 10).  If
                           no matching tuple exists in TS, the
                           process suspends until a matching tuple
                           becomes available. */

rd("string", ?y);       /* Behaves the same as in(), except the
                           tuple is not withdrawn from TS. */

eval("string", strlen("hello"));    /* Creates a live two element
                                       tuple.  The first element
                                       is a string and the second
                                       is the function strlen. */
```

Figure 2-3. C-Linda operations.

```
sender()
{
   out("message", "Hello Watson");
}

receiver()
{
   char msg[13];
   in("message", ?msg);
}
```

Figure 2-4. C-Linda inter-process communication.

As with Orca, a Linda program starts with a single process. Additional processes are created with the `eval()` statement. Unlike the modified `fork()` construct in Orca, the `eval()` does not provide the facility to designate the processor on which to create the new process. A Linda program terminates execution when the primary process becomes passive.

The Tuple Space concept is highly desirable within a network environment. Each of the TS primitives, initiated from within a Linda program, corresponds directly with a series of communication messages interacting with Tuple Space. Tuple Space may reside on a dedicated network machine or be distributed across the network onto any number of network machines. Activated Linda process tuples may also reside on any number of network machines. Once again, although the Linda paradigm is conceptually sound, there are recognized performance bottlenecks in accessing Tuple Space on a network platform;

however, these performance problems are being addressed and solved by current research[ARTHU91, SCHUM91, and ROBIN94].

A distributed Linda environment is described in the following chapter. The underlying tasks performed within the Linda-LAN environment are similar to those discussed in the previous section. The control framework has been embedded into the environment and is represented by the control sub-system of Linda-LAN. The next section provides background information on a set of parallel programming paradigms that are used in the justification of the stability and scalability of the control framework within the Linda-LAN environment.

2.3 Parallel Programming Paradigms

Nicholas Carriero and David Gelernter have introduced three basic parallel programming paradigms [CARRI90]. These paradigms establish unique ways of describing and analyzing parallelism within programs. These basic paradigms are result parallelism, agenda parallelism and specialist parallelism. Carriero and Gelernter admit that other paradigms may exist; however, the paradigms do cover a wide range of parallel programs. The building of a home is the typical analogy used in describing the programming paradigms. People are considered the processors or workers building a home. Additionally, sample C-Linda programs from [CARRI90] are used to illustrate the paradigms.

2.3.1 Result Parallelism

Result parallelism is based on the final outcome; the result being the finished home. The building of a home can be divided into many parts (i.e. framing the exterior walls, leveling the foundation, insulating the attic, installing the kitchen cabinets, etc.). Result parallelism centers on each worker being assigned one and only one task to complete. All workers simultaneously begin building the home in parallel. Some workers may have restrictions on when they can proceed. For instance, the worker installing the windows cannot begin until the exterior walls are framed. Once all the components have been finished, the result is complete. This describes the result parallel approach.

Figure 2-5 depicts a parallel program for finding prime numbers written in C-Linda. This program follows the result parallel paradigm. For each number between 2 and the upper limit, there exists a separate worker process (is_prime) that determines whether or not the number is prime. The `eval` statement is used to begin each worker process. All worker processes are started at the beginning of the program. Notice that each worker process requires the results, through the `rd("primes", i, ?ok)` statement, from workers that compute lower numbered primes. Also note that each worker process computes one and only one result. Once a worker process computes a prime, the worker process ends and does not perform any other task. After all the worker processes have completed, the program terminates.

```
#define UPPERLIMIT   1000

real_main(argc, argv)
int argc;
char *argv[];
{
   int count = 0, i, is_prime(), ok;

   /* For each number from 2 to limit, determine if it is prime */
   for (i=2; i <= UPPERLIMIT; ++i) eval("primes", i, is_prime(i));

   /* Read each number and its result from tuple space */
   for (i = 2; i <= UPPERLIMIT; ++i) {
       rd("primes", i, ?ok);
       /* Print number if prime */
       if (ok) {
          printf("Prime number : %d\n", i);  fflush(stdout);
          ++count;
       } /* end if */
   } /* end for */

} /* end main */

/* Worker Process */
int is_prime(me)
int me;
{
   int     i, limit, ok;
   double  sqrt();

   /* Find maximum divisor possible */
   limit = sqrt((double) me) + 1;

   /* Check each prime from 2 to max divisor */
   for (i=2; i < limit; ++i) {
       rd("primes", i, ?ok);
       /* If prime evenly dives into the number, then its not prime */
       if (ok && (me%i == 0)) return 0;
   } /* end for */

   return 1;

}  /* end is_prime */
```

Figure 2-5. Result parallelism paradigm -- prime finder program[2].

---

[2] This program is based on a sample program illustrated in [CARRI90].

2.3.2 Specialist Parallelism

Specialist parallelism utilizes a different approach in the building of homes. The specialist programming paradigm requires a collection of specifically trained or skilled workers (i.e. an electrician, a carpenter, a roofer, a plumber, etc.). Each worker is assigned a specific task to complete. All workers usually begin work at the same time; however, many of the workers remain idle during the initial stages of development. As one worker completes a task, the next worker or set of workers will begin their tasks. For example, once the carpenter has completed the frame of the home, the plumber can install the bathroom fixtures and the electrician can wire the house. The specialist paradigm is highly synonymous with pipelined jobs and monitors. This paradigm becomes clearer if we envision a block of homes being built; the carpenter can begin work on a second home while the plumber and electrician begin work on the first home. After a specialist worker completes a task, the worker can remain idle until another task becomes available or the specialist worker can terminate. If the specialist worker terminates and a new task becomes available, a new specialist worker can be called upon to perform the new task.

Figures 2-6(a) and 2-6(b) depict specialist parallelism within a program based on the Sieve of Eratosthenes, a prime-finding algorithm. The algorithm processes a sequence of numbers via a series of sieves. All integers pass through a 2-sieve which removes multiples of 2. The remaining integers pass through a 3-sieve which removes multiples of 3. Likewise, a 5-sieve is used and then a 7-sieve, and so forth for each prime. The integers that survive all the sieves are prime. Notice that each worker process within the program has a special and unique function. The source worker is used to perform the 2-sieve. The sink worker is used to perform the sieve on the current greatest prime. Each `eval` of the pipe_seg worker is used to sieve a specific prime and carry the list of remaining integers on to the next higher sieve. Specialist processes, like the sink worker, remain active until the program completes. Other

specialist processes, like the source worker, terminate after completing a particular task. Also notice that the source and sink processes are started at the beginning of the program, while new pipe_seg workers are created for each specific prime being processed.

```
#define UPPERLIMIT 1000

real_main(argc, argv)
int argc;
char *argv[];
{
    eval("source", source(UPPERLIMIT));
    eval("sink", sink(UPPERLIMIT));

    in("source", 0);
    in("sink", 0);
} /* end main */

/* Specialist worker */
int source(int limit)
{
   int i, out_index=0;

   for (i=5; i < limit; i+=2)
      out("seg", 3, out_index++, i);

   out("seg", 3, out_index, 0);

   return 0;
} /* end source */
```

Figure 2-6(a). Specialist parallelism paradigm -- prime finder program[3].

[3] This program is based on a sample program illustrated in [CARRI90].

```
/* Specialist worker */
int sink(int limit)
{
   int pipe_seg();
   int in_index=0, num, prime=3, prime_count=2;

   while(1) {
      in("seg", prime, in_index++, ?num);
      if (!num) break;
      if (num % prime) {
       ++prime_count;
       if (num*num < limit) {
          eval("pipe seg", pipe_seg(prime, num, in_index));
          prime = num;
          in_index = 0;
       } /* end if */
      } /* end if */
   } /* end while */
   printf("count: %d.\n", prime_count);

   return 0;
} /* end sink */

/* Specialist worker */
int pipe_seg(int prime, int next, int in_index)
{
   int num, out_index=0;

   while(1) {
      in("seg", prime, in_index++, ?num);
      if (!num) break;
      if (num%prime) out("seg", next, out_index++, num);
   } /* end while */

   out("seg", next, out_index, num);

   return 0;

} /* end pipe_seg */
```

Figure 2-6(b). Specialist parallelism paradigm -- prime finder program (continued).

2.3.3 Agenda Parallelism

Agenda parallelism centers on a list of tasks to be completed in the building of a home. A set of workers with no special skills or assignments begins work on whatever exists on the agenda of tasks. All workers begin work simultaneously and all are capable of performing any task on the list. Each worker consults the list for a task and goes about completing it. While building a home, a group of workers may work together and build the frame of the house. After completing the frame, some of the workers (arbitrarily chosen) may work on the windows, while others work on the roof. The agenda of tasks maintains control over the actions of the workers. The list of activities may have dependencies (e.g. do the first set of ten items and then proceed with the next set of items), or the list of activities may have no special ordering. Once a worker completes the assigned task, the worker returns to the list for another task. The workers always remain active or waiting until the entire list is completed.

Figures 2-7(a) and 2-7(b) depict agenda parallelism within a word count program. Each of the workers evaled in the program is identical and each is capable of counting the number of occurrences of a given word within each element of the array. The array acts as an agenda of tasks. The workers randomly and repeatedly access the list of tasks until the "done" task is listed in tuple space. All workers remain alive until the entire array has been processed. At this point, the workers finish executing. Notice that regardless of the number of workers defined in this program, the program will always execute in a similar manner. This approach is often called the replicated worker model or master-worker paradigm. Note that the replicated worker model is quite similar to the result parallel paradigm. The main difference between the two approaches is that the result parallel paradigm allows a worker process to perform only one task of work, while the replicated worker model requires each process to perform several tasks. For

instance, if there were 100 tasks to perform, the result parallel paradigm would require 100 processes, while the replicated worker model would make do with 10 processes.

2.3.4 Summarizing the Programming Paradigms

Result parallelism, specialist parallelism, and agenda parallelism provide three unique ways of describing and analyzing parallelism within programs. Combinations or mixtures of the paradigms are often used in order to solve a particular problem. The goals of this thesis utilize the paradigms separately in order to examine the stability and scalability of the control framework. The execution patterns of the basic programming paradigms are captured within a number of different programs and not the various combinations that are possible. By individually examining the paradigms through a set of programs, the control framework is shown to exhibit predictable patterns of execution.

```
#define  NUMWORKERS  5
#define  MAXLEN      100

real_main(argc, argv)
int argc;
char **argv;
{
   FILE *ifp;
   char word[MAXLEN + 1];
   char filenm[MAXLEN + 1];
   char scanline[MAXLEN + 1];
   int i, lines, cnt, total = 0;

   /* Get arguments */
   strcpy(word, argv[1]);
   strcpy(filenm, argv[2]);

   /* Open file for to search */
   ifp = fopen(filenm, "r");

   /* Start workers */
   for (i = 0; i < NUMWORKERS; i++)
      eval("worker", worker(s));

   /* Read each line of file and place into tuple space */
   lines = 0;
   while (fgets(scanline, MAXLEN, ifp) != NULL) {
      lines++;
      out("array", scanline);
   } /* end while */

   /* Close input file */
   fclose(ifp);

   /* Retrieve results of count per line */
   for (i = 0; i < lines; i++) {
      in("result", ?cnt);
      total += cnt;
   } /* end for */

   /* Terminate workers */
   out("done");

   /* Wait for workers to end */
   for (i = 0; i < num_workers; i++)
      in("worker", 1);

   printf("Count = %d\n", total);

} /* end main */
```

Figure 2-7(a). Agenda parallelism paradigm -- word count program.

```
int worker(s)
char s[MAXLEN + 1];
{
   int cnt, i;
   char line[MAXLEN + 1];
   char *temp, *strstr();

   /* Keep getting a task from tuple space until done */
   while (!rdp("done")) {
      /* Get an input line from tuple space */
      if (inp("array", ?line)) {
       cnt = 0;
       temp = line;
       /* Count occurrences */
       while ((temp = strstr(temp, s)) != NULL) {
          cnt++;
          temp += 1;
       } /* end while */
       out("result", cnt);
      } /* end if */
   } /* end while */

   /* End worker */
   return (1);

} /* end worker */
```

Figure 2-7(b). Agenda parallelism paradigm -- word count program (continued).

3

DESIGN OF THE LINDA-LAN ENVIRONMENT

A control framework for distributed (parallel) processing environments is established within this chapter. The control framework is introduced within Linda-LAN, a new distributed parallel processing environment. Linda-LAN encapsulates the control framework within its control sub-system to effectively and efficiently manage network resources. Linda-LAN is based upon the Linda paradigm, a conceptually simple specification and operational framework that supports an easy to use coordination (parallel) language. In addition, the multiple processors offered by a local area network (LAN) of personal computers are utilized by Linda-LAN to provide a low-cost parallel processing solution. The environment is primarily a software-based system and requires no specialized hardware. The system supports the compilation, distribution and execution of programs written in the C-Linda language.

By embedding the control framework within Linda-LAN, the environment gains the additional capabilities of transparency, scalability, and stability that are not supplied by the basic Linda paradigm. This chapter provides information on the historical origins of Linda-LAN, the philosophy used within its design, the physical and logical topologies, and detailed examination of the environment's control sub-system. The applicability of the control framework to other environments is also explored.

3.1. Historical Origins of Linda-LAN

Linda-LAN evolved from the Scientific Computing Association's (SCA) version of C-Linda. In the SCA version, the kernel (the set of routines used in supporting Linda operations) is encapsulated within a Linda program during compilation. During execution, each new program process started by the `eval()` operation contains a duplicate copy of the kernel. Figure 3-1 illustrates the interaction of multiple Linda processes with Tuple Space. Each Linda process accesses Tuple Space through its own individual copy of the kernel routines. Tuple Space resides in shared memory and the kernels synchronize their access through semaphores. Although the SCA approach of using multiple kernels attached to each Linda process is suitable for accessing Tuple Space which is located in the shared memory of a single machine, modifications to the kernel routines are required to work within a distributed environment. Changes are needed to the semaphore mechanisms used to access Tuple Space, as well as replacing the UNIX forking mechanisms used in starting new processes.

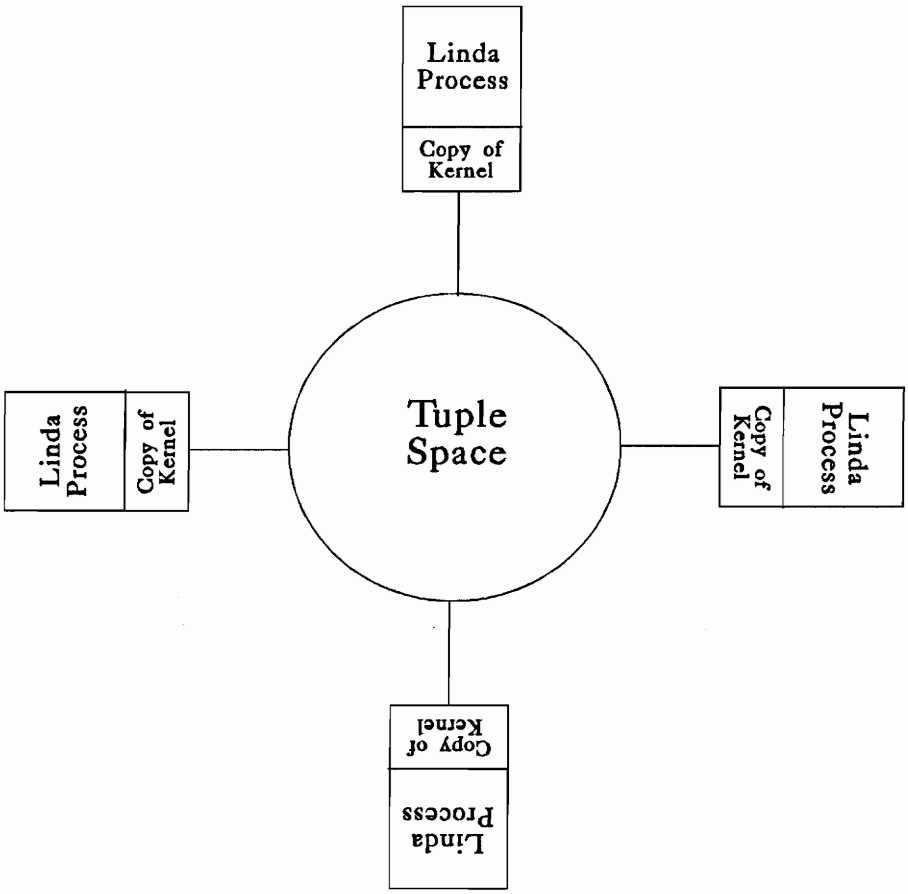


Figure 3-1. Tuple Space Access within the SCA Linda

Research by Schumann has taken an initial step towards distribution from the SCA Linda[SCHUM93]. Schumann isolated and extracted the kernel routines from the Linda program and placed them into a separate executing process. The new independent kernel process, dubbed d-kernel, assumed responsibility for all accesses to Tuple Space. The Linda program processes utilized stub procedures to communicate their requests to the d-kernel process. Figure 3-2 illustrates the interaction of Linda processes to Tuple Space through the d-kernel process. Although Schumann found the inter-communication between Linda processes and Tuple Space to be slowed by the d-kernel process, this advancement allowed Tuple Space to reside on a separately executing workstation.

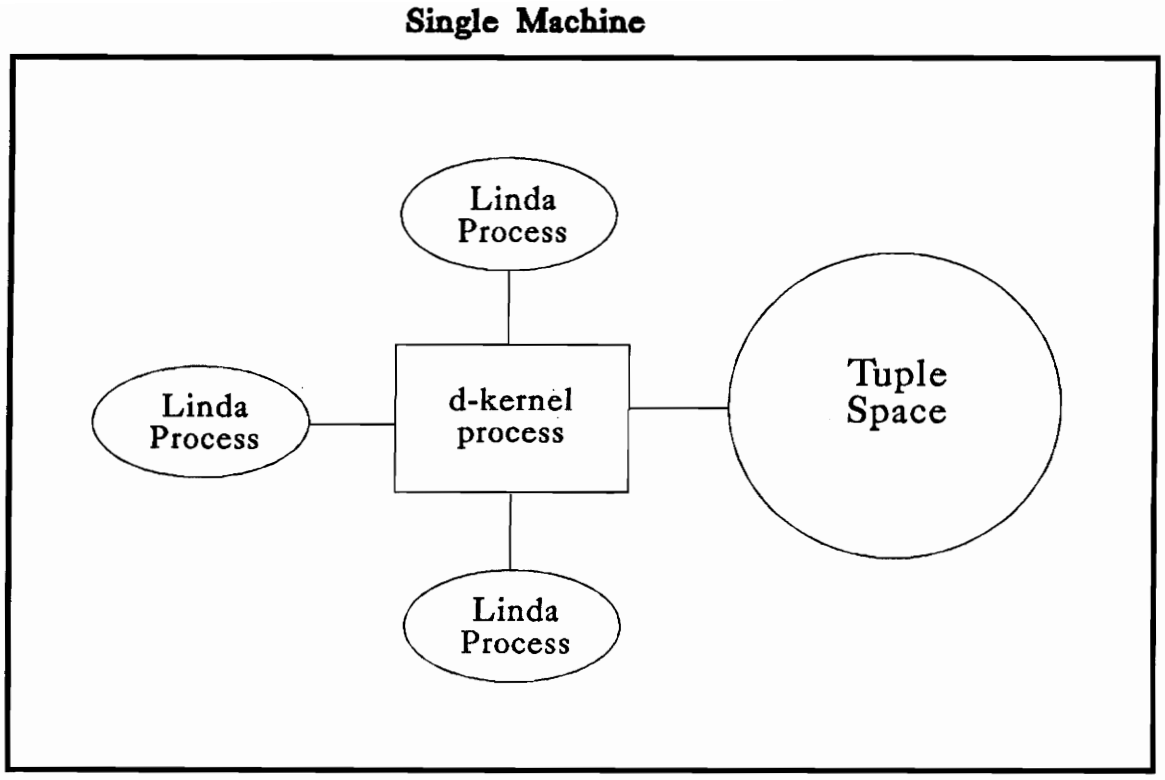


Figure 3-2. Tuple Space access through the d-kernel process.

This thesis takes the next logical step to create a distributed environment by replicating Schumann's kernel process across a network. Linda-LAN is founded upon this concept of multiple kernel-based processes. Figure 3-3 illustrates a typical Linda program executing on a distributed network of machines. Many advantages are provided by the multiple kernel-based processes. For instance, the kernel routines are no longer attached to each individual program process. The duplicate executable code is removed from each executing process, thus utilizing less disk space. Re-compilation of the program processes is no longer required unless changes are made to the stub procedures. Another advantage offered by the kernel processes is the opportunity to package multiple requests to Tuple Space as one message. Serialization of the Tuple Space requests can be performed at the workstation level. This is advantageous in that it offers a high degree of control over Tuple Space accesses. This may also be a drawback in that the serialization may slow down Tuple Space access.

Once the kernel processes are distributed, many questions are raised. For instance, how are the kernel processes replicated? Where should a new process be distributed onto the system? What machines should be used while executing the program? Should Tuple Space be located on one workstation or on several workstations? These questions can be answered in many ways. The design of Linda-LAN within this chapter answers many of these questions in a particular fashion. The next section describes the philosophy upon which the Linda-LAN environment is based.

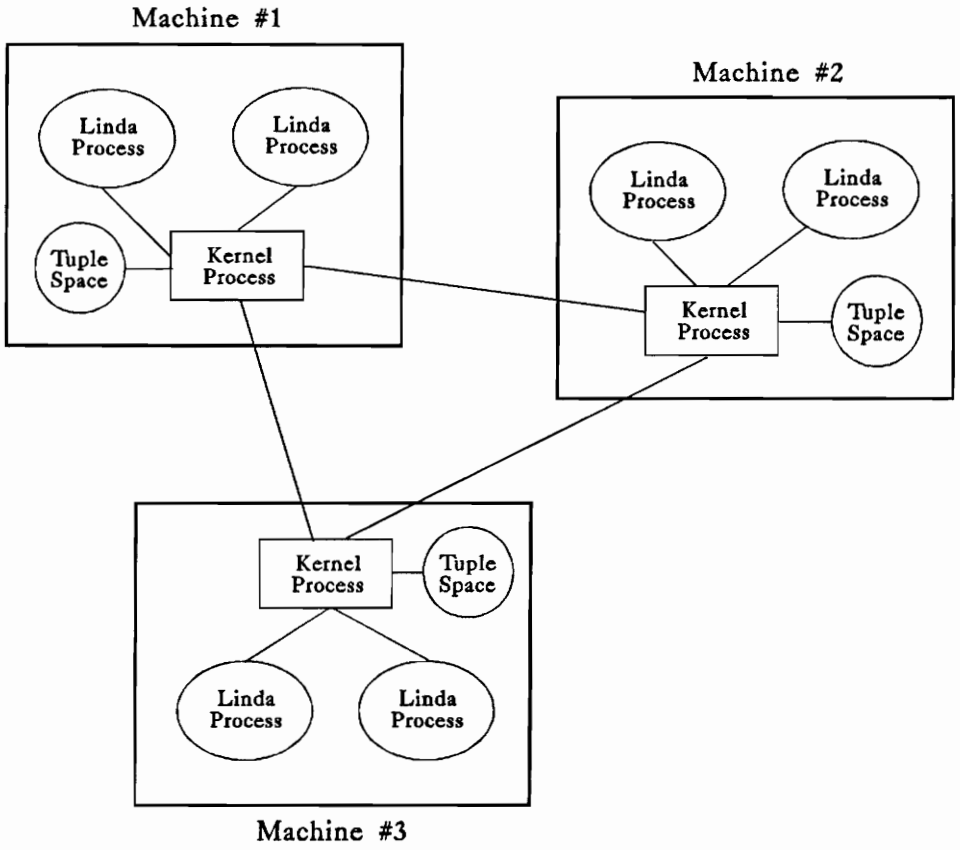


Figure 3-3. Network communications between a multiple kernel based Linda.

3.2. Philosophy of Linda-LAN

Many of the design decisions used in the creation of Linda-LAN are based on the KISS (Keep It Simple and Stupid) principle. The Linda paradigm provides a simple and easy to use framework for programming; the distributed environment which executes Linda programs should also be simple and easy to use. Although efficiency is important, simplicity and ease of use are given the higher priority. The initial Linda-LAN system is intended to provide a basis upon which future research can be conducted. For the above reasons, the environment is divided into a control sub-system and a data-subsytem. This division allows for a more focused analysis and implementation of the control functions and data functions. Control functions being the allocation of processes to processors, the scheduling of multiple programs, the designation of network processors, etc. The data functions are concerned with intercommunication of tuples between program processes and tuple space, the tuple matching within Tuple Space, and other data related duties.

In utilizing a network of personal computers (workstations), control over the network resources takes on a high priority. Workstations are generally assigned to individual users. These users normally demand that a greater portion of the processing time be allocated to them during their periods of work. When the workstations are idle, the processing time should be made available to the remotely executing Linda programs. In order to provide flexibility in the distribution of the processing time, a significant level of control must be injected into the system. For this reason, the allocation of processes to processors becomes important If a workstation is being taxed by an individual user, a new process should not be allocated to that workstation. Control is also needed over which workstations are participating in the execution of Linda programs. Some workstations may not be able to accept foreign processes. This may be due to security reasons, or user preference. Other network resources, such as workstation disk space,

must be properly managed. Only the executable code pertaining to a specific Linda program should be kept on each workstation. The C-Linda compiler and Tuple Space must reside only where they are desired.

The following two sections describe the physical and logical topologies of the Linda-LAN system. The philosophies discussed in this section have a direct impact on many of the design decisions made within Linda-LAN. Whereever possible, the KISS principle is applied and maintenance of control is applied.

3.3. Physical Topology

The physical topology of the Linda-LAN environment is illustrated in Figure 3-4. The environment utilizes an Internet network of UNIX-based workstations. Although present implementations require a homogeneous environment of Commodore AMIGA 3000UX workstations, plans are being made to migrate towards a heterogeneous environment. The Linda-LAN system is not limited to a local area network or to the physical Ethernet LAN as depicted in Figure 3-4. Machines may be connected to the environment over existing gateways or be interconnected through other physical network technology. This facility is provided through TCP/IP Internet Protocols. The current versions of Linda-LAN, however, do utilize a single physical Ethernet.

Linda-LAN is composed of multiple Linda-LAN Processors, Tuple Servers and the Communications Server. The attached workstations (nodes) within the environment can execute Linda program processes (live tuples), manage the shared Tuple Space, or manage the environment. As seen in Figure 3-4, the physical topology of the Linda-LAN environment allows for the distribution of multiple Linda processes initiated from within a single Linda program onto several Linda-LAN Processors (workstations) of a network. Linda-LAN Processors are able to support more than one executing Linda process; however, a

maximum of one Linda program may execute at any given time on the system. This limitation was introduced into the system for the benefit of other network users. Current network utilization by non-Linda users is extremely high. Minor modification of the system is required to increase the number of executing Linda programs. Future research may be directed towards the scheduling and execution of multiple Linda programs.

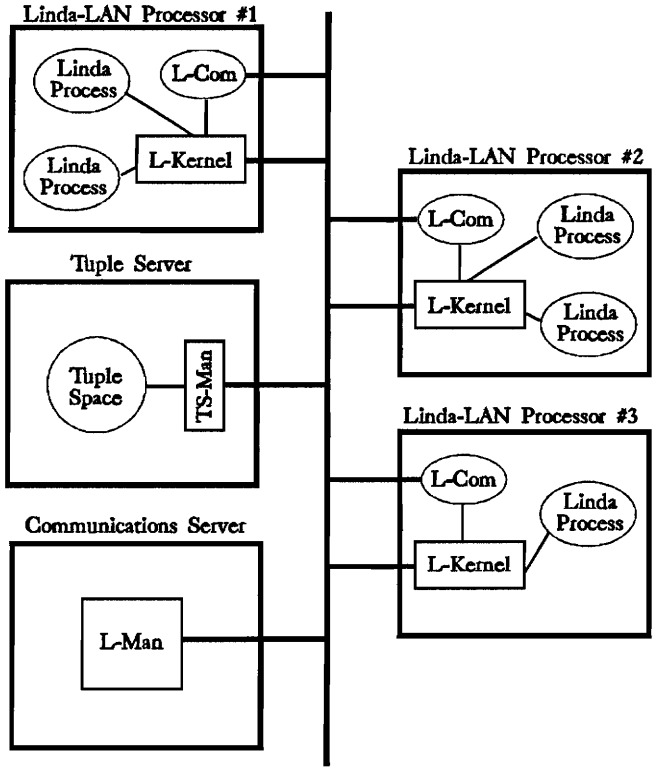


Figure 3-4. Physical Topology of Linda-LAN.

Although the current Linda-LAN system only executes a single program at any given time, multiple Linda-LAN systems are capable of executing at the same time over common network processors . This capability is advantageous for users with production and test environments. Linda programs that have not

been fully tested can execute over a test environment without affecting the programs executing on the production system. The control framework accomodates for multiple executing systems by externalizing the network information from the executing program.

The Linda program processes communicate with Tuple Space by a node resident process, called a Linda-LAN Kernel (L-Kernel). The L-Kernel packages each Tuple Space request and routes the request to a specified node containing another environment process, called a TS Manager (TS-Man), which manages and serializes all requests to Tuple Space. The TS Manager resides on a specified workstation of the environment, called the Tuple Server. Although a single Tuple Server is depicted in Figure 3-4, multiple Tuple Servers may exist. In fact, other versions of Linda-LAN have been implemented with multiple Tuple Servers [ROBIN94]. This flexibility offers enhanced performance to the system, by improving the access to Tuple Space. The communication demands of the system mainly occur between the workstations, which execute Linda processes, and the designated Tuple Server(s).

The communication between any two nodes of the system is point to point through the BSD *socket* communication protocols found within the System V Release 4 UNIX operating system. The point to point communications are made available by the system process, called the Linda-LAN Manager (L-Man), which executes on the last physical component of the Linda-LAN system called the Communications Server. Network address information on all participating workstations is maintained at the Communications Server. During system instantiation and program instantiation, the Communications Server relays needed network information to all executing system processes. Only one Communications Server exists per Linda-LAN system; however, multiple Linda-LAN systems may operate over common nodes of the network.

In addition to the network information, global system information is also managed at the Communications Server. The global information is collected from system processes, called Linda-LAN Communication Managers (L-Coms), which reside on each participating workstation that executes a Linda program process. Each L-Com periodically provides L-Man with utilization information on the processor on which the L-Com resides. Due to the wide fluctuations in workstation activity during the execution of a program, the utilization information is a necessity in the mapping of Linda program processes to idle Linda-LAN Processors (workstations). One of the duties of the Linda-LAN Communications Server is to determine the Linda-LAN Processor to which a Linda program process is to be distributed. Without the utilization information, effective program execution would not be possible. In addition, if not for the Communications Server, the primary user of a workstation may find executing Linda program processes consuming unavailable cpu resources.

Since the global information maintained at the Communications Server is highly important, the loss of the Communications Server would result in the failure of the system and any executing program. An outage of any other node, except for the Tuple Server, may not result in the immediate failure of the system. The Tuple Server is also a key component in the robustness of the system. The breakdown in a Tuple Server would result in the loss of shared program data and cause the system to fail. Future single Tuple Server versions of Linda-LAN can offer some hope in maintaining the integrity of the system by following existing fault tolerance mechanisms provided by today's file server technology. Although greater performance benefits are seen with the multiple Tuple Server versions of Linda-LAN, the robustness offered by the single Tuple Server versions of Linda-LAN may be found to be more desirable. These fault tolerance issues are an area of research which require further investigation.

3.4. Logical Topology

Figure 3-5 represents the logical topology of the Linda-LAN environment. The environment is composed of a control sub-system and a data sub-system. The control sub-system is responsible for maintaining network information, system instantiation and termination, program instantiation and termination, data sub-system instantiation and termination, program scheduling, process to processor allocation, network monitoring, and executable code distribution. The data sub-system is responsible for handling all Tuple Space requests, process instantiation and termination, and communication between program processes and the system.

Linda-LAN is specifically designed for the partitioning of the system into the control and data sub-systems. By logically and physically segregating the control functions from the data functions, a more simplistic view of the environment is established. In addition, research efforts have become more focused by permitting two distinct avenues of investigation. The first avenue deals with the management activities within the environment and the second avenue concentrates on the data interaction between the Linda program processes and Tuple Space. The division allows for modifications to the control sub-system to be performed independently from the changes to the data sub-system.

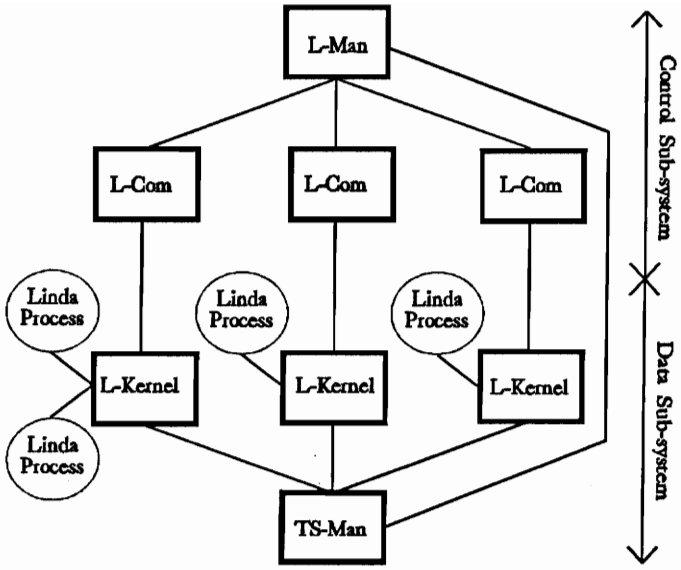


Figure 3-5. Logical topology of Linda-LAN.

This design choice is advantageous because the control sub-system can work with various implementations of the data sub-system. For instance, the data sub-system could provide a single Tuple Space, a partitioned Tuple Space, or even multiple Tuple Spaces. Regardless of the choice made within the data sub-system, the control sub-system can remain the same. The converse is also true. The control sub-system could be altered to use network broadcasting mechanisms rather than point-to-point communications. No change to the data sub-system is required. Any of the implemented data sub-systems could be used with any of the control sub-systems as long as the interfaces between the sub-systems remains intact.

The control sub-system is composed of a single Linda-LAN Manager (L-Man) and one or more Linda-LAN Communications Managers (L-Coms). The data sub-system is composed of the Tuple Space Manager (TS-Man) and a number of Linda-LAN Kernels (L-Kernels) equal in number to the L-Coms. Each of the control and data components are separate executing entities. This separation detracts slightly

from the single kernel-based process per workstation previously presented in Figure 3-3. Within Linda-LAN's architecture, the control and data components combine to provide the same functionality as the individual workstation kernels shown in Figure 3-3.

In order to supply the desired level of control over the environment, the Linda-LAN Manager (L-Man) and the Linda-LAN Communications Managers (L-Coms) were created. The L-Coms provide complete control over the individual workstations. Users must submit all of their requests to their local workstation's L-Com. For example, Linda programs can be instantiated and terminated by a request from the user to the local workstation's L-Com. Executable code can be distributed to other workstations by the issuance of a user request to its local L-Com. By being in control, the L-Coms are able to serialize the user requests. The L-Coms are also responsible for the instantiation and termination of the workstation's L-Kernel. If a user requests to terminate an executing program prematurely, an L-Com has the power to shutdown the local executing L-Kernel. By being in charge of the L-Kernels, the L-Coms also reduce resource utilization. The L-Kernels are only activated when they are needed. After performing their duties, the L-Kernels are terminated by the L-Coms.

The L-Coms are activated when the Linda-LAN environment is activated. For this reason, the L-Coms are light-weight processes that are interrupt-driven. The L-Coms respond to user and system requests only when needed. The L-Coms collect processor utilization information when requested by the L-Man. The information, collected by the L-Coms, is passed on to the Linda-LAN Manager (L-Man). Although a peer-to-peer relationship is feasible among the L-Coms, Linda-LAN is centrally controlled by the L-Man. As seen in Figure 3-5, the L-Man sits atop the environment and manages all the L-Coms and the data sub-system. L-Man centrally controls the environment by maintaining and analyzing global network information. The centralization of the L-Man enables the L-Man to control the nodes which participate in the execution of Linda programs.

The L-Man directly controls the Tuple Space Manager (TS-Man) and indirectly controls the Linda-LAN Kernels (L-Kernels) through the L-Coms. The TS-Man and the L-Kernels are only activated when a Linda program is executing. Within the design of Linda-LAN, the data sub-system is composed of the TS-Man and the L-Kernels. These components are only concerned with the execution of Linda programs. From Figure 3-5, the Linda program processes communicate all data requests through the L-Kernels.

The L-Kernels contain the logic for interacting with Tuple Space by way of the TS-Man. The L-Kernels are able to serialize the data requests to the TS-Man. The number of network connections from the Linda program processes to the Tuple Space are greatly reduced by the serialization through the L-Kernels. This reduces bottlenecks at the TS-Man. In addition, the L-Kernels can further reduce traffic to the TS-Man by packaging multiple requests as a single request to the TS-Man. As previously stated, the L-Kernels reduce the duplicated code within the Linda program processes and remove the restriction on requiring re-compilation of the processes when changes are made to the execution system.

Requests by a Linda program process to place data into Tuple Space are made through the L-Kernel. The L-Kernel packages each request and routes it to the TS-Man where the data is placed into Tuple Space. Once a request is passed to the L-Kernel, the program process immediately continues to execute. When a program process desires data from Tuple Space, the program process submits a request to the L-Kernel. The L-Kernel packages the request for the data and submits the request to the TS-Man for retrieval. After the data is retrieved by the TS-Man, the submitting L-Kernel is informed and the data is retrieved and passed to the requesting program process. The L-Kernel is an interrupt-driven process that waits for requests to arrive from its connections to program processes, the processor's attached L-Com, or the connection with the TS-Man. In the case of data retrieval, the L-Kernel would not wait (block) for the

TS-Man to retrieve desired data. The L-Kernel would process other requests until the TS-Man has retrieved the applicable data and sent it to the L-Kernel.

The L-Kernels also support the instantiation of new program processes. When an eval request is made by a Linda program process, the request is routed from the initiating L-Kernel to the TS-Man. The TS-Man places the eval request into Tuple Space and then informs the L-Man that an active tuple has entered Tuple Space. The L-Man responds by selecting an applicable system processor and routes the activate tuple request to the selected processor's L-Com. The L-Com passes the request to its associated L-Kernel, where a new program process is instantiated by way of the UNIX `execl()` mechanism [ROBIN94]. The new program process establishes a connection with the L-Kernel and submits a request through its L-Kernel to retrieve the new process's input parameters. This method of process instantiation adheres to the Linda paradigm. Active tuples are generated within Tuple Space.

Although it is possible for the L-Kernels to perform the functions necessary to manage Tuple Space, the TS-Man was created for reasons of simplicity. The TS-Man provides a simple transition from Schumann's previous work to supply a single Tuple Space to the environment. As time permits, the TS-Man can be improved to supply multiple distinct Tuple Spaces or supply a partitioned Tuple Space [ROBIN94]. As previously stated, the TS-Man is responsbile for all direct interaction with Tuple Space and is responsible for informing the L-Man of the arrival of new active tuples. The TS-Man is also an interrupt-driven process that only accepts and services requests from connected L-Kernels and the L-Man.

Figure 3-6 represents the interaction between the control sub-system and the data-sub-system as viewed by the control sub-system. The remainder of this chapter concentrates on the control sub-system components and the advantages found within it. In addition, the control sub-system's process to processor allocation strategy is discussed. The implementation details of the data sub-system and its components are not

specifically addressed within this thesis and are left for other publications to discuss [ROBIN94 and SCHUM93].

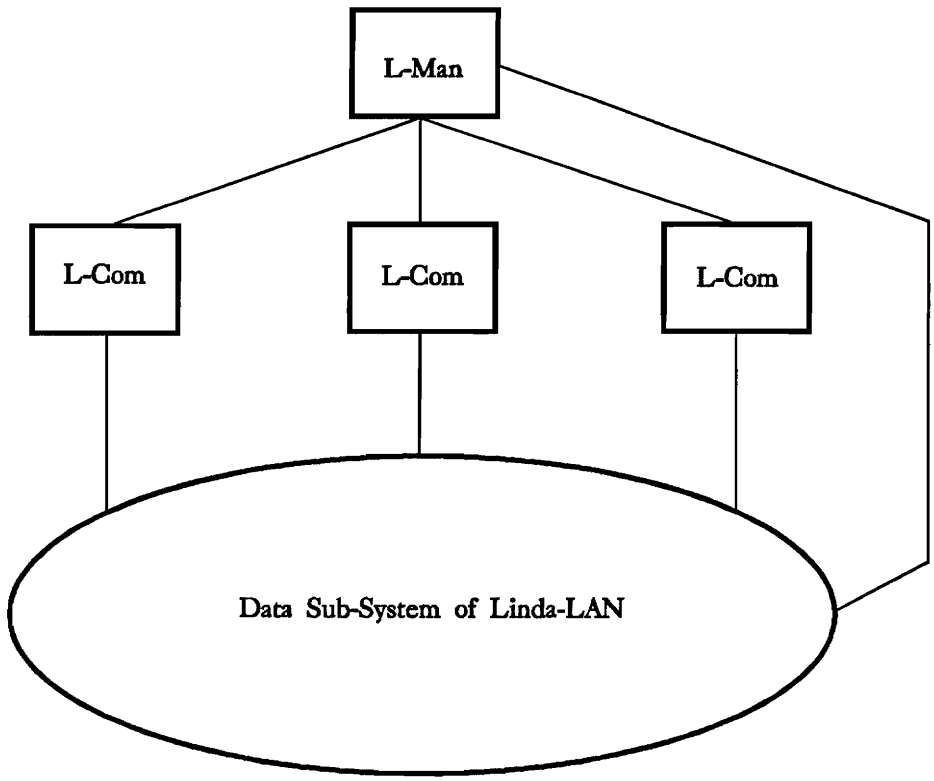


Figure 3-6. Linda-LAN as viewed by the control sub-system.

3.5. Control Sub-System

As noted earlier, the control sub-system of Linda-LAN consists of the Linda-LAN Manager (L-Man) and one or more Linda-LAN Communication Managers (L-Coms). One L-Com exists per Linda-LAN Processor, while a single L-Man resides on the Communications Server. As seen in Figure 3-5, a hierarchy exists between the L-Coms and L-Man. The L-Man is in complete control of the system, while each L-Com is responsible for the workstation on which it resides. The components work together to perform program instantiation and termination, data sub-system instantiation and termination, program scheduling, process to processor allocation, and executable code distribution. In addition, the L-Man performs system instantiation and termination, program scheduling, and network monitoring, as well as keeping track of all network information.

3.5.1 System Instantiation

System instantiation is performed at the Communications Server by a designated system administrator starting the Linda-LAN Manager by means of the *lman* (Linda-Lan Manager) command. The responsibility then shifts to the L-Man to instantiate all L-Coms at the participating Linda-LAN Processors (workstations) listed within a system table. The data sub-system components are instantiated after a program has been scheduled for execution. By having the system initiated by an administrator, centralized control over the availability of the system is established. In addition, the environment provides the administrator with control over which machines participate in the system.

The administrator is able to modify the system table of network information prior to the instantiation of the system. As each processor is added or removed from the system, the control sub-system scales

appropriately by adding or removing a L-Com. This ability to scale the number of participating processors is an important aspect in the design of Linda-LAN. Typical local area network environments have workstations added and deleted at various points in time. Linda-LAN provides a simplistic procedure for scaling processors by using the centralized system table of participating processors.

3.5.2 Program Compilation, Linking, Distribution and Execution

In order for a user to have a C-Linda program executed on the system, the user must follow a three step procedure. First, the user compiles and links the program at a participating Linda-LAN Processor. Second, the user submits a request to its local L-Com to distribute the executable code to all other Linda-LAN Processors. The request is first routed to the Linda-LAN Manager for approval. Once approved, the L-Man informs all other L-Coms to retrieve the executable code from the L-Com which originated the request. Finally, after the code has been distributed, the user must submit another request to its local L-Com for the program to be executed. This request is also passed on to the L-Man for approval. After being approved, the program is scheduled for execution and the L-Com, which originated the execution request, is informed by the L-Man to release the program into the system for execution.

Numerous advantages are offered by having the environment operate in this fashion. First, any Linda program process is able to execute on any participating Linda-LAN Processor. Second, once a program is compiled and linked for one Linda-LAN Processor, the same executable code is applicable to all other Linda-LAN Processors. There is no need for the program to be compiled or linked on the other homogeneous machines. This procedure must change when Linda-LAN is migrated to a heterogeneous environment; however, the compilation and linking of the source program will only be required on a non-homogeneous subset of the Linda-LAN Processors. For example, if the system were composed of twenty machines utilizing three different machine architectures, the source program would only require

compilation and linking on three machines which represent the three architectures. The remaining machines would copy the executable code from machines with an identical architecture. This new procedure is easily accomplished through the centralized control of the network information found at the L-Man.

Another advantage of having the user perform the above procedure is the ability to add new Linda-LAN Processors to the environment without requiring the re-compilation, re-linking, or complete re-distribution of the program. Workstations are merely added to the system table found at the L-Man and the executable code is distributed to the new workstations. The distribution process and the ability to scale participating processors, in combination with the Linda programming paradigm, promotes transparency within Linda-LAN. Programmers and users of Linda-LAN can remain unaware and unconcerned about the current number of environment processors. Finally, Linda-LAN only requires a program to be re-compiled, re-linked, and re-distributed when the actual source code is changed. Programs can be repeatedly executed with changes in the number of instantiated Linda program processes, the program input, or the number of Linda-LAN processors utilized.

A disadvantage with the current compile, link, distribute and execute procedure is the added step of distribution. This differs from the traditional serial programmer's duties of compile, link, and execute. At first, the added step of executable code distribution may seem awkward for the programmer, however, the objective should be understandable. One method of getting around the problem is to develop a UNIX *script* which encapsulates all of the necessary commands to compile, link, distribute, and execute the program. Another alternative would be to distribute the program when the program is actually executed. This alternative would hide the distribution process, but would increase the execution time. The repeated execution of the program does not require re-distribution of the program. Time-stamps on the executable code could be checked before transfering the code to a Linda-LAN Processor.

3.5.3 Data Sub-System Instantiation

Before a Linda program is actually released into the system for execution, the data sub-system is instantiated by the control sub-system. The instantiation of the data sub-system is simple. First, the TS-Man is initiated by the L-Man at the designated Tuple Server listed within a system table. Once the TS-Man is instantiated, it returns network addressing information to the L-Man. The L-Man routes this information to all participating L-Coms. Each L-Com instantiates a local Linda-LAN Kernel, passing it TS-Man's network addressing information. During each L-Kernel instantiation, a communications dialogue with the TS-Man is established. After a communications dialogue is established between the TS-Man and the L-Kernel on which the request for Linda program execution was initiated, the Linda program is released into the system.

Advantages are offered by having the data sub-system initiated at program instantiation. First, the data sub-system components are not required until a program has been executed. To have each workstation of the network manage unnecessary processes is wasteful. Second, the L-Coms are considered light-weight processes. Each L-Com utilizes a small amount of cpu processing and requires few system resources. The data sub-system components, on the other hand, require large amounts of system resources. Another advantage is found in the fact that the data sub-system components are not required to contain the code to manage themselves when there is no work to be done. Finally, the data sub-system components are alleviated from much of the communications setup. All network addressing information is supplied by the control sub-system. The data sub-system merely acquires the connections to the other needed components of the data sub-system and executing program processes.

One of the disadvantages of having the data sub-system instantiated at program start-up is the processing time required for the data sub-system to become active. This disadvantage is considered acceptable because of the processing and resource overhead in leaving the L-Kernels and TS-Man continually active.

3.5.4 Program Terminiation and System Termination

Program termination is another function of the control sub-system. An executing program informs a L-Kernel of its desire to terminate. A request for termination may result from normal or abnormal conditions. The L-Kernel routes the request to its local L-Com. The L-Com, in turn, sends the request on to the L-Man. The L-Man informs the TS-Man and all participating L-Coms of the program termination request. Each L-Com informs its local L-Kernel to terminate itself and all remaining Linda processes.

The data sub-system is completely terminated at program termination. The data sub-system components are no longer required and system resources are released. The program termination is well controlled under normal and abnormal termination. An alternative to the program request to terminate is provided by Linda-LAN. The user may execute a *lstop* (Linda-LAN stop) command to the local executing L-Com. The request is passed on to the L-Man, where termination messages are sent to all participating L-Coms.

To terminate the Linda-LAN system, the system administrator must issue a *lkill* (Linda-LAN kill) request to the Linda-LAN Manager. The L-Man aborts any active program through an abnormal program termination request and informs all L-Coms to terminate themselves. Once again, centralized control is made available to the system administrator.

3.5.5 Network Monitoring

Another managerial function of the system which is performed by the Linda-LAN Manager is the monitoring of the network. In order to determine the availability of a Linda-LAN Processor, the L-Man occasionally queries each L-Com for its workstation's cpu load utilization[1]. This information is returned to the L-Man and is used for the allocation of program processes to environment processors. If an L-Com does not respond, it can be removed from the table of participating Linda-LAN Processors. By removing the Linda-LAN Processor from the table, the environment assumes the machine is unavailable and continues processing requests. If a program is currently executing on the system, the current versions of Linda-LAN abort the program and the system. The main reason for operating in this fashion is to provide for some robustness within the system. Future research should be directed towards fault tolerance issues.

3.5.6 Process to Processor Allocation

Although the control sub-system has many essential duties, the most critical task of the control sub-system is process to processor allocation. Without an effective mapping of Linda program processes to available processors, inconveniences to the primary users of the network workstations may result and valuable program execution time may be lost. The Linda-LAN Manager is responsible for the mapping of Linda program processes to Linda-LAN Processors. The L-Man is informed by a request from the TS Manager to select a Linda-LAN Processor to eval (spawn) a new Linda program process. In the current versions of Linda-LAN, the L-Man utilizes a round-robin approach in determining the next Linda-LAN Processor to execute a Linda program process. As long as a Linda-LAN Processor has not surpassed its cpu load utilization limit set at system instantiation, it may be selected as the next workstation to execute a Linda

[1] Load utilization information is currently retrieved using the *uptime* command provided by the Commodore AMIGA 3000 AUX.

program process. If all Linda-LAN Processors have surpassed their cpu load utilization limit, the least recently selected workstation is chosen.

The main advantage of this mapping strategy is its simplicity and efficiency of selection. Although many different and sophisticated strategies are possible for the selection of Linda-LAN Processors, the current implementation does not require any additional information from a Linda-LAN Processor, except for the current cpu load utilization information. Other allocation methodologies may require knowledge on the termination of a Linda program process or on the expected execution time of a Linda program process or on the types of cpu processors available. Strategies, such as least-recently-used, most-recently-used, prioritized scheduling, and shortest-remaining-time scheduling can be implemented within the centralized Linda-LAN Manager [DIETE84]; however, future research must be directed towards these strategies to determine their viability.

Once a Linda-LAN Processor has been selected, the L-Man informs the appropriate L-Com. The L-Com, in turn, informs its local L-Kernel of its selection. The L-Kernel initiates (spawns) another executable version of the Linda program onto the workstation. The Linda program establishes a connection with the local L-Kernel and requests from the TS Manager, by way of the L-Kernel, for the name of the Linda program process to execute and any applicable process parameters. After the necessary information has been obtained from TS, the Linda program process begins execution.

Although the L-Kernels have been given the duty of initiating the Linda program processes, the L-Coms are also a likely candidate in the starting of program processes. The L-Kernels were chosen over the L-Coms because of the high degree of interaction that takes place between the program processes and the L-Kernels.

3.5.7 Summarizing the Control Sub-System

Overall, the Linda-LAN control sub-system centralizes the management activities of the system and gives control over the environment to a designated system administrator. The sub-system establishes monitoring capabilities for the entire system by providing for the collection of global system information. The control sub-system relieves the data sub-system from performing any management activities. Global scheduling of programs, processes, and processors is performed using knowledge of the entire system. Furthermore, the division of the system into a control sub-system and data sub-system has focused research efforts and has provided a more simplistic view of the system. The next section describes the application of the control framework to other environments.

3.6. Application of the Control Framework to Other Environments

The control sub-system presented in the previous section establishes a control framework that works well within the Linda-LAN environment. Applying this control framework to other environments is also desireable. By revisiting the classes of distributed environments discussed in Chapter 2, the feasibility of integrating the control framework within those environments can be examined. In particular, the application of the control functions to the classes of distributed environments are studied.

3.6.1 The Control Framework within Message Passing Environments

Environments based on pure message passing can utilize the control framework in many ways. First, message passing environments require the designation of the sending and receiving processes. In cases, such as CSP, process names are used to pass messages between processes. The control framework can

maintain the necessary network information in order for the processes to communicate. Process names can be aliased so that the locations of the sending and receiving processes are dynamically determined at program execution time. By dynamically determining the location of a process, the processes would not require re-compilation when the processes change network locations. Only the centralized system tables are altered to accomodate the changes in network locations.

The control framework determines the dynamic locations of processes. By maintaining cpu load utilization information on processors, the control framework provides for the selection of suitable processors for processes. For systems like CSP and occam, the control framework can determine all process locations at program start-up or on an as needed basis. The control framework can also be used to distribute the executable code for the processes. A command can be issued to distribute all executable code to the desired locations.

For languages, such as occam, the control framework can help in the termination of the distributed concurrent processes. Occam does not provide explicit mechanisms to terminate a program. The control framework can maintain information on the status of all processes and issue termination messages when necessary.

Although the control framework can provide the above capabilities to message passing systems, some programmers may desire the explicit control over the allocated processes and the behavior of the program. In these cases, the control framework would not provide many benefits and should not be used.

3.6.2 The Control Framework within RPC and Monitor Based Environments.

The class of distributed environments which compose of remote procedure calls and monitors are also potential candidates for utilizing the control framework. RPC environments normally maintain information on the locations of remote procedures through a global name server. The information can be handled within the control framework and allow for the dynamic selection of remote procedure locations by the control framework. This determination would be beneficial if more than one processor contains a copy of the remote procedure. The control framework would select a processor containing the remote procedure that has the lesser cpu load utilization. The control framework can also choose a remote procedure that is executing on the local processor of the process which called the remote procedure.

Monitor based environments are less likely to use the control framework. Although the control framework can select appropriate processors, the locations of monitor procedures are normally fixed and known by the operating system. Therefore, the implementation of the control framework is not advantageous in the case of fixed monitor procedure locations.

Systems which support the ADA language or Emerald language are definite candidates for embracing the control framework. ADA can take advantage of the control framework's selection strategy for newly created processes (tasks). Although Emerald provides for the selection of processors at compile time, Emerald also performs the selection of processors at execution time. When requested by Emerald, the control framework can help in the selection of processors during execution time. The control framework also enhances the termination of ADA and Emerald programs in normal and abnormal situations. The control framework's parental authority over executing program processes can enforce stable terminations of all processes.

3.6.3 The Control Framework within Conceptually Shared Memory Environments.

The class of environments based on conceptually shared memory are good candidates for using the control framework. For example, Orca assigns each processor an identification number. Processes refer to the processor identification numbers when a new process is to be created by the `fork()` mechanism within the Orca language. The control framework can be used to dynamically select appropriate processors for each identification number. The cpu load utilizations can be checked for all available processors and the least used processors can be selected at program instantiation. As an alternative approach, the control framework can select appropriate processors when needed. However, the control framework additionally is required to keep track of which identification numbers have already been selected.

Multilisp is another example of where the control framework can be applied. The `pcall` construct within Multilisp creates concurrent processes which may execute on any available processor. The control framework is again able to select the least used processors for the newly created processes. The control framework can also distribute the applicable executable code generated by Multilisp and Orca programs. Additionally, the control framework can provide for the termination of active Multilisp and Orca programs similar to the way in which Linda programs are terminated.

3.6.4 Summary of Applying the Control Framework to Other Environments

Although specifically designed for the Linda-LAN environment, the control framework is applicable to many of the environments described within this section. It can provide for effective control over environments other than Linda-LAN. The process to processor allocation capabilities of the control framework are beneficial, as well as the code distribution features. Additionally, the handling of program

instantiation and termination by the control framework is applicable. The following section summarizes the design of the Linda-LAN environment and the control framework presented in this chapter.

3.7. Summarizing the Design of Linda-LAN and the Control Framework

Linda-LAN is designed to provide an effective, efficient and easy to use parallel programming environment. The environment utilizes the available processors of a local area network and is not intended to dominate the processing power of LAN workstations. The control framework established within Linda-LAN only takes advantage of available and under-utilized workstations. Interference with the local processing of the primary users of the workstations is minimized by the control framework. Although efficiency is important, the control over the Linda-LAN environment takes on the higher precedence.

Linda-LAN and its control framework support scalability. Participating network workstations can be added and deleted at one central location. In addition, transparency is provided. Users and programmers can remain unaware of the changes in the number of processors. The Linda-LAN environment is physically and logically divided into a control sub-system and a data sub-system. This division simplifies the implementation of the execution system and focuses future research efforts.

Although Linda-LAN captures the conceptual simplicity of the Tuple Space model, the control framework can be extracted from the Linda-LAN environment and applied to other distributed, parallel languages and models. CSP, occam, RPC, Ada, Emerald, and Orca are examples of languages and environments that are able to utilize the control framework. The next chapter substantiates the scalability and stability of the control framework within Linda-LAN. Experimental results on the execution of C-Linda programs are examined.

4

SUBSTANTIATING SCALABILITY AND STABILITY

The efficiency and effectiveness of the Linda-LAN environment are influenced by both the execution of the control and data sub-systems. The results provided within this chapter are only applicable to the control sub-system of Linda-LAN. Results from the data sub-system are addressed in future publications. While many goals are desired of Linda-LAN, the execution characteristics of scalability and stability are of primary concern to the control sub-system. Without achieving these desired goals, the Linda-LAN environment would be considered an ineffective and unusable solution for parallel processing. The execution of the system would not be able to respond to changes in the number of utilized environment processors nor to changes in the number of executed program processes. In addition, executing programs

would be unstable, undependable and unpredictable. Linda-LAN would not be deemed a viable low cost parallel processing solution that operates within a conceptually simple parallel programming framework.

We hypothesize that the desired goals of scalability and stability, as defined in Chapter 1, are attained within the control sub-system of Linda-LAN by effectively and transparently controlling the environment processors and the allocation of instantiated program processes to processors. Linda-LAN allows for the alteration in the number of utilized environment processors through system tables located at the Linda-LAN Manager. The modification of the system tables is transparent to the users of Linda-LAN. Furthermore, the Linda-LAN system allows a program to vary the number of processes utilized by the program based on its input data. Since the control sub-system manages the environment processors utilized and the mapping of the program processes to processors, proper scalability and stability should be achievable.

This chapter begins by examining the patterns of control messages produced by executing Linda programs on the control sub-system of Linda-LAN. The patterns of control messages convey execution characteristics about Linda programs. The execution patterns are used to justify the stability and scalability of the control sub-system within the Linda-LAN environment. The chapter continues by showing the existence of two forms of stability within the control sub-system. Comparisons are made on the execution patterns of agenda parallel programs, and examinations are made on specialist and result parallel programs. The chapter concludes by comparing and relating additional control message information in the justification of two forms scalability within the control sub-system.

4.1. Patterns of Program Execution

The execution patterns of a Linda program are generated by the messages produced within the control sub-system of Linda-LAN [CLINE93]. Figure 4-1 (a) depicts the execution pattern of a Linda program based on agenda parallelism. Time intervals of ten seconds are used to track the control messages produced through the control sub-system. The ten second time interval is an arbitrarily chosen time interval that provides an abstract view of the patterns of control messages. The execution characteristics are represented by the various control messages produced by an executing program. Four control message types can occur during the execution of a program. Figure 4-1 (a) depicts three of the four control message types. Start-up messages refer to the control sub-system messages that occur after a user submits a request for program execution but before the program is actually released into the system. The eval messages refer to the control sub-system messages that occur during the instantiation of new program processes. The termination messages refer to the control sub-system messages that occur during program termination. CPU load utilization messages, not depicted in Figure 4-1, are the last of the four control message types. Utilization messages are constantly issued at approximately the same time intervals by the L-Man during system execution. These messages have been ignored for simplicity because they have no real impact on stability and scalability.

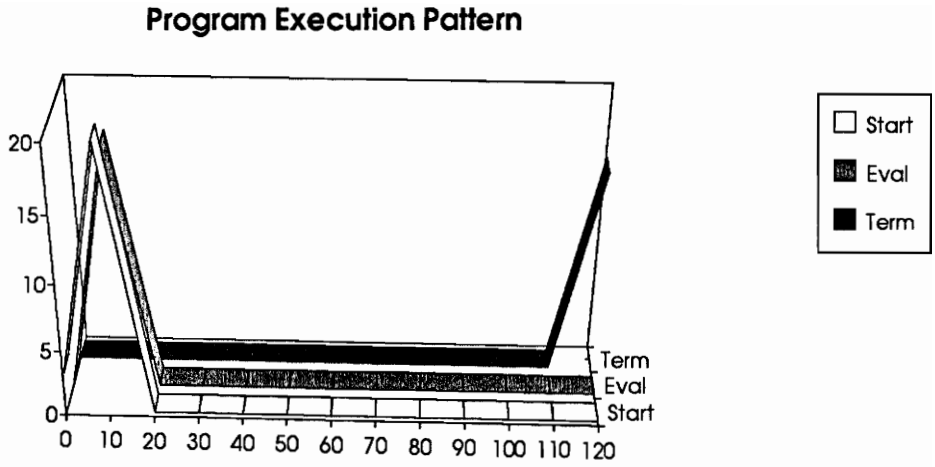


Figure 4-1 a. Execution pattern of a Linda program on the control sub-system of Linda-LAN.

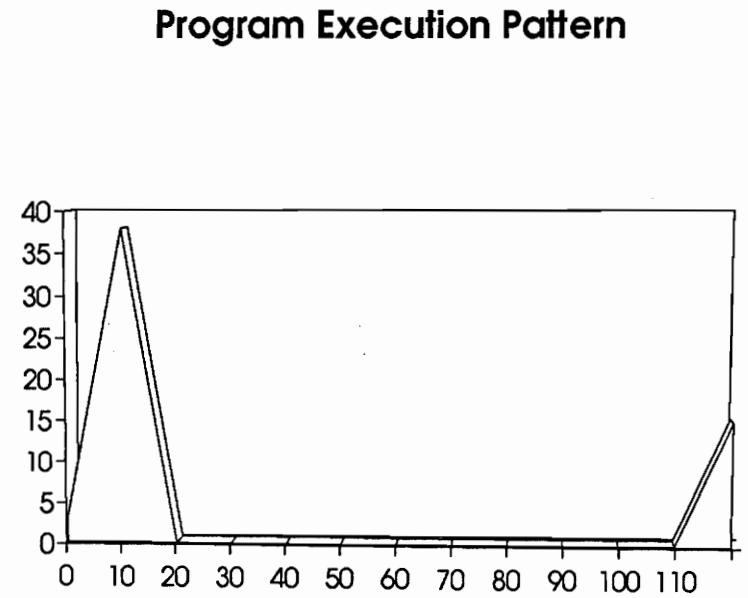


Figure 4-1 b. The execution pattern of a Linda program produced by the combination of control messsages.

Figure 4-1 (b) combines the control messages of Figure 4-1 (a) into a single execution pattern for the program. Within this chapter, the patterns of execution for a program are normally illustrated in the form presented by Figure 4-1(b). Notice that Figure 4-1(b) depicts a single large peak at the beginning of the execution pattern. The large peak is a result of the merger of the start-up and eval messages in Figure 4-1(a). The start-up of additional processors does not interfere with the eval of new processes at the beginning of a program. At times, the eval messages will be skewed farther to the right due to the delay of new processes being evaled. This will be reflected by two or more peaks at the beginning of the execution pattern. Combining the messages into a single pattern simplifies the comparison of different execution patterns between different programs. The following section addresses the stability of the control sub-system of Linda-LAN. Agenda, specialist and result parallel programs are examined.

4.2. Verification of Stability

In order for the control sub-system of Linda-LAN to be considered stable, two items must occur. First, the repeated execution of a program across Linda-LAN must generate the same execution characteristics on the control sub-system [CLINE93]. Second, programs that are similarly written should generate similar execution characteristics on the control sub-system, and therefore their execution patterns should be predictable. This section compares the execution patterns of an agenda program that is repeatedly executed under identical system conditions. The repeated executions of the agenda program will demonstrate that the control sub-system produces the same patterns of execution for the program. In addition, a comparison is made on the execution patterns for three agenda programs to verify that the system is stable for similarly written programs. The control sub-system is shown to produce predictable patterns of execution for the agenda programs. The section concludes by describing the execution patterns for result parallel and specialist parallel programs.

4.2.1 Stability within a Repeatedly Executed Program

Figure 4-2 illustrates the multiple executions of the same agenda program across the Linda-LAN system under identical conditions[1]. As can be seen, the program repeatedly executed with relatively the same execution pattern across the control sub-system. Therefore, stability is shown to exist within the control sub-system for the repeated execution of the same agenda program. By examining the execution patterns, the predictability of the patterns should be evident. The start messages attribute to the first few peaks of the execution patterns. Start messages are always generated at program instantiation. The start messages are composed of messages sent to instantiate the Tuple Space Manager and the Linda-LAN Kernels. Agenda programs generate new program processes during the initial stages of the program. The largest peak of each execution pattern consists of any remaining start messages and the eval messages generated by the creation of new program processes. Note that Figure 4-2 depicts two large peaks at the beginning of each execution pattern. This differs from Figure 4-1(b) because of the pre-processing performed by the main program process of Figure 4-2. New program processes in Figure 4-2 are not evaled until after the main process has filled Tuple Space with data. The new processes in Figure 4-1(b) are evaled at the start of program execution. The final peak within each execution pattern is formed by the processing of program termination messages.

Although Figure 4-2 represents the multiple executions of a single agenda parallel program, the results are indicative for all other agenda programs tested. Result and specialist parallel programs as described by Carriero and Gelernter in How to Write Parallel Programs were also tested. Predictable results for each type of program were achieved and the execution patterns are explained at the end of this section.

---

[1] Truly identical conditions on the network and the utilized workstations can not be achieved; however, steps were taken to ensure idle network and workstation conditions existed. In addition, identical program input data was used so that the number of instantiated processes for each execution was exactly the same.

The reasons for obtaining stability within the control sub-system are directly attributable to the centralized scheduling of program processes to processors. The data collected from the mapping of processes to processors showed identical distribution patterns by the Linda-LAN Manager component of the control sub-system. Exactly the same processors were selected for each instantiated process. The control sub-system behaved identically for each execution. The variability in termination times reflected in Figure 4-2 is attributable to conditions outside of the control sub-system, such as the data sub-system, network communications and the process scheduling of the UNIX operating system.

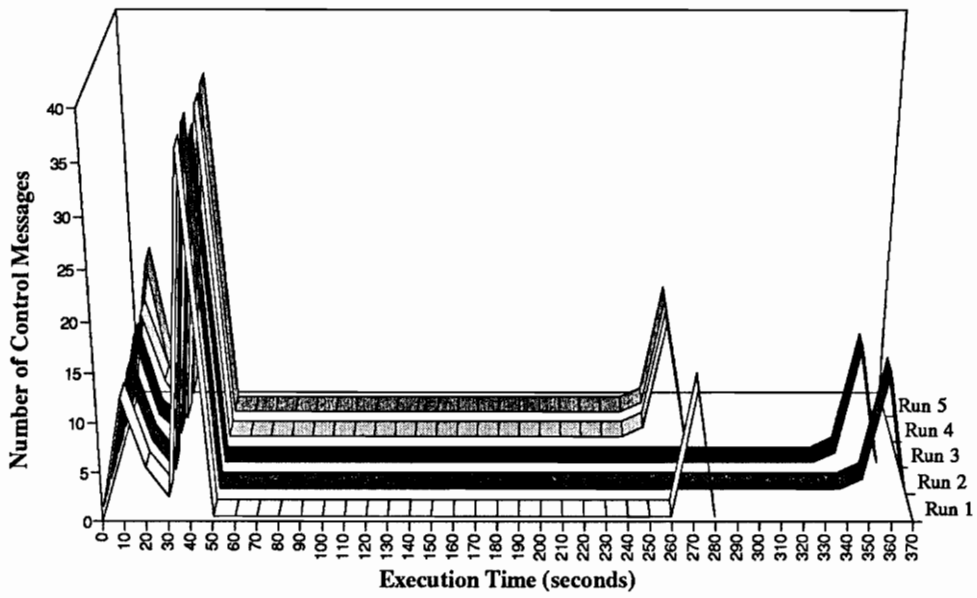


Figure 4-2. Execution patterns of a program repeatedly executed under identical conditions.

4.2.2 Stability across Similarly Written Agenda Parallel Programs

The second form of stability to be verified within the control sub-system of Linda-LAN deals with the predictable patterns of execution for programs that are similarly written. Linda programs that are based on agenda parallelism are considered to be similarly written. Figure 4-3 illustrates the execution patterns of three different agenda parallel programs. Although each program of Figure 4-3 solves a different problem, all the programs are based on agenda parallelism. Each program creates a set of identical worker processes to solve their respective problems. The execution patterns of the three programs convey the same predictable execution patterns on the control sub-system. The programs produce a large number of messages at the beginning of the program's execution. The beginning messages are followed by a period of execution time where no messages are produced. At the ends of the execution patterns a small number of messages are produced.

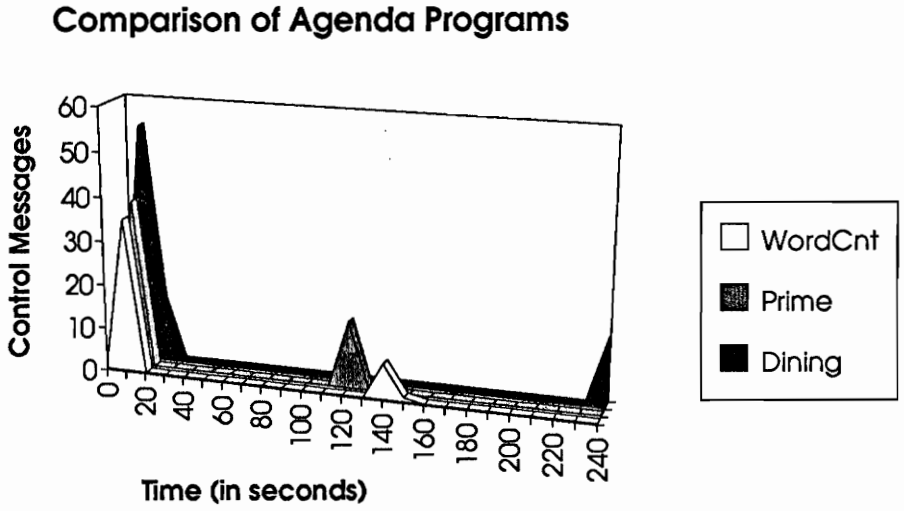


Figure 4-3. Stability of three agenda programs.

The reasons for this common pattern of execution are straightforward. The control messages produced at the beginning of the programs are a combination of start and eval messages. Agenda programs create their identical worker processes at the beginning of program execution. The worker processes normally remain active until the program terminates. A single large peak is often formed at the beginning of the execution pattern of agenda programs. The execution patterns of Figure 4-3 illustrate this single large peak of messages. The messages at the ends of the execution patterns are composed entirely of termination messages.

In some cases, a single large peak at the beginning of the execution pattern does not occur for agenda programs. Figure 4-2 illustrates the case where a single large peak does not occur at the beginning of the execution pattern for an agenda program. If an agenda program delays the creation of the identical worker processes, additional peaks are formed in the program's execution pattern. In Figure 4-2 the worker processes are not immediately created. The first two peaks are composed of start messages. The large peak in the execution pattern contains any remaining start messages and the eval messages for the created worker processes. A longer delay in the instantiation of new program processes results in peaks which are skewed further to the right.

One other case can occur where additional peaks are created in the execution pattern for an agenda program. When a large number of worker processes are created, the time required to process the numerous eval messages may exceed the ten second time interval. In these cases, additional peaks of messages are generated. The dining philosophers program in Figure 4-3 demonstrates an additional peak of control messages on the execution pattern after the large peak.

The differences in the heights of the peaks are dependent on the number of program processes generated by each program and the number of network processors utilized by the programs. Figure 4-4 compares the control messages produced by a variety of agenda parallel, result parallel and specialist parallel programs. The number of network processors for each program execution is fixed at six processors. Notice that the numbers of start messages for each program are equivalent, as well as the numbers of termination messages. Figure 4-4 demonstrates that the start and termination messages for any program that is executing on six processors are fixed and stable. If the programs of Figure 4-3 had executed on the same number of processors, there would be no difference in the height of the minor peaks at the ends of the execution patterns. The difference in the heights of the peaks at the beginning of the execution would still be uncertain.

The variability between the execution patterns of agenda programs is also dependent on the number of program processes that are instantiated by each program. Figure 4-5 further demonstrates the control sub-system's stability by comparing programs with a fixed number of instantiated processes as well as a fixed number of processors. Notice that the eval messages are identical for all programs, as well as the start and termination messages. The control sub-system remains stable regardless of the type of program executed. If the number of instantiated program processes and the number of environment processors were fixed, the agenda programs of Figure 4-3 would be quite similar. As mentioned previously, slight differences may occur based on the time that the new program processes are instantiated. Although slight variations may occur in the execution patterns of agenda programs, the overall execution pattern is predictable. The control sub-system is stable for agenda programs.

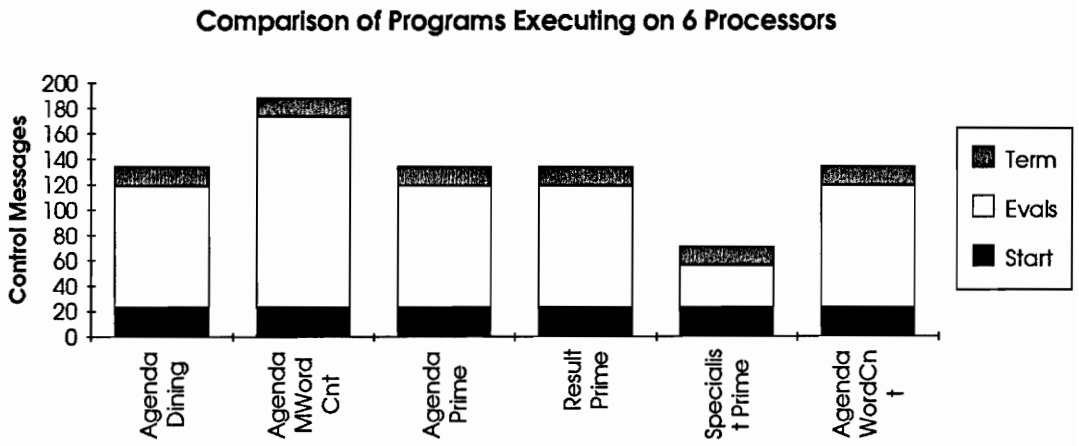


Figure 4-4. Comparison of programs demonstrating stability with fixed number of processors.

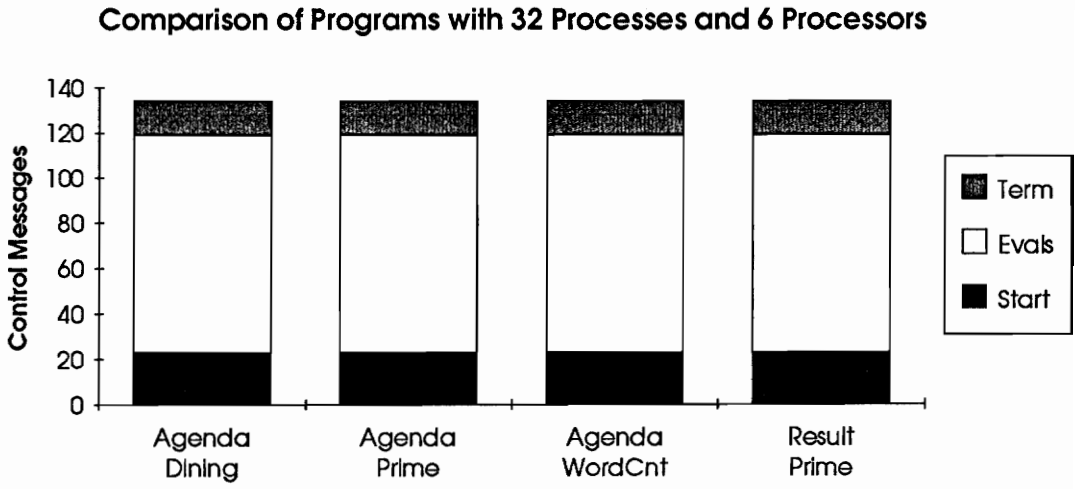


Figure 4-5. Comparison of programs demonstrating stability with fixed processors and processes.

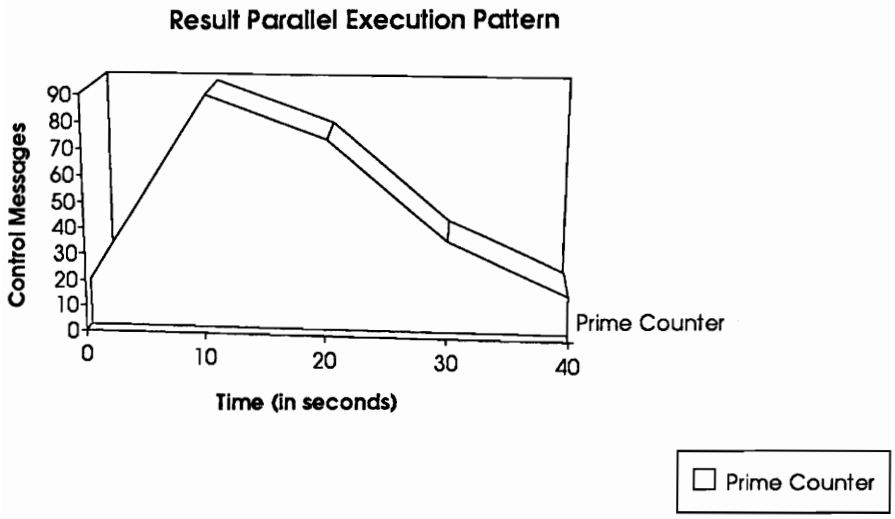


Figure 4-6. Execution pattern for a result parallel program.

4.2.3 Stability across Similarly Written Result Parallel Programs

The execution pattern of a result parallel program that counts prime numbers is depicted in Figure 4-6. The generated pattern varies from the agenda program presented in Figure 4-1(b). The result parallel execution pattern has minor peaks at the beginning and end of its execution pattern, similar to the agenda program. The result parallel execution pattern also has a large peak after the first minor peak. The difference between the result parallel execution pattern and the agenda parallel execution pattern lies in the additional peaks generated by the result parallel program. The additional peaks are predictable after examining the paradigm of result parallelism.

A result parallel program creates a new process for each component of work being performed. All work is started when the program begins execution. Therefore, all processes are created when the program begins execution. Once all the processes have completed their work, the program terminates and the result is yielded. As seen in Figure 4-6, the beginning of a result parallel program generates a large number of program processes. Eval messages are produced when program processes are created. The eval messages are responsible for the additional peaks that follow the first major peak of messages. As the program executes, the number of new processes generated is reduced due to the processing of previously created processes. The main program process, which performs all the evals, has a majority of the cpu time slices at the beginning of the program's execution. As time continues, processes created on the same workstation as the main program process gain more cpu time slices and reduce the number of time slices that the main program process receives. The main program process gradually generates fewer and fewer evals. A reduction in the number of eval messages occurs as the program executes. Figure 4-6 illustrates the reduction in messages. The peaks of messages reduce in height as the program executes.

As with agenda programs, differences can occur between different result parallel programs. The heights of the peaks are dependent on the number of processors used and the number of processes created. In addition, the final stages of result parallel programs can differ. If the lifetime of the result parallel processes is short, control messages will be produced throughout the life of the program. The main program process will eval new processes almost to the point at which the program terminates. As shown in Figure 4-6, control messages are produced throughout the life of the program. The work performed by each process of the prime counter program in Figure 4-6 is small. If the lifetime of the processes is long, a gap in the production of control messages will be present towards the end of the program. A segment of the execution pattern preceding the final peak of messages will appear with no messages.

Figures 4-4 and 4-5 also demonstrate the stability of a result parallel program when the number of processors and processes are fixed. The result parallel program in Figures 4-4 and 4-5 generated the identical number of start and termination messages within the control sub-system as the agenda programs. The result parallel program also generated the identical number of eval messages when the numbers of processes are fixed. Although the execution patterns of result parallel programs differ from agenda programs, the number of control messages matches that of agenda programs when the number of processors and processes are fixed.

4.2.4 Stability across Similarly Written Specialist Parallel Programs

Figure 4-7 depicts the execution pattern of a specialist parallel program that also counts prime numbers. The specialist program depicted in Figure 4-7 also has minor peaks at the beginning and end of its execution pattern. The start and termination messages are responsible for those minor peaks. The first large peak of the execution pattern is a common occurrence in all Linda programs seen thus far. A

parallel program should generate a large number of processes at the beginning of its execution. The objective of parallel programs is to utilize as many processors as possible for as long as possible.

The remaining peaks of the prime counter's execution pattern are not as predictable. Specialist programs create processes that perform unique and special functions. The specialist processes can be created at any point in time during a program's execution. Although many processes are called at the beginning of the program, this is not required. Each specialist program must be examined on an individual basis in order to determine the program's execution pattern. The source code for the program of Figure 4-7 is shown in Figures 2-6(a) and 2-6(b). In the prime counter program, all specialist workers are started near the beginning of the program. New versions of the "pipe_seg" process are started throughout the execution of the program. The remaining peaks of the prime counter's execution pattern are a result of these new versions. As seen in Figure 4-7, the last of the new versions of the "pipe_seg" process are created by the middle of the program's execution.

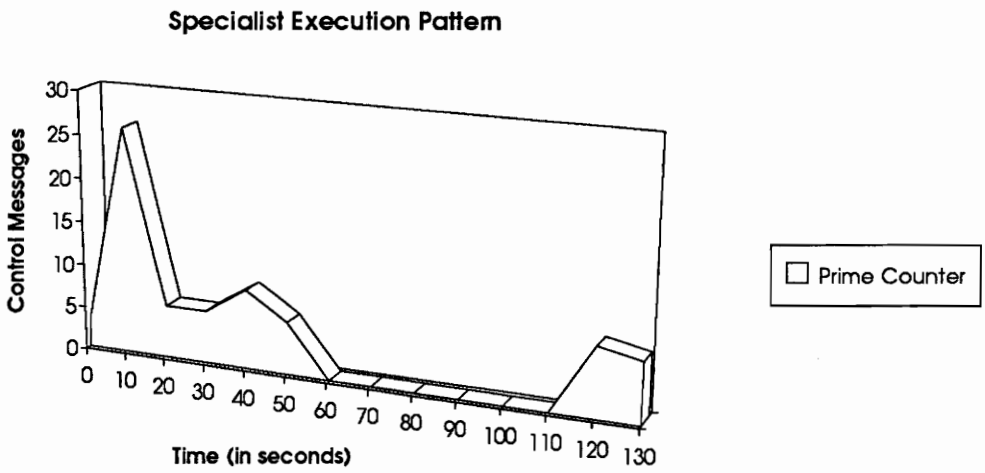


Figure 4-7. The execution pattern of a specialist parallel program.

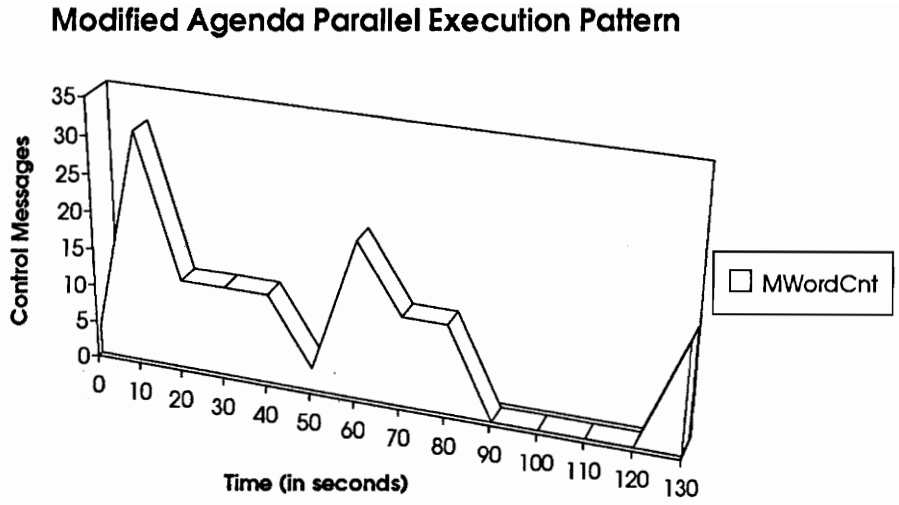


Figure 4-8. The execution pattern of a modified agenda parallel program.

Many specialist parallel programs produce patterns that are similar to agenda parallel programs. When all specialist processes are created at the beginning of the program, the execution pattern will be similar to the agenda programs. Otherwise, the patterns produced by a specialist program will be similar to the "random" patterns of Figure 4-7. The beginning and ending minor peaks will be present with a large peak occurring after the first minor peak. The remaining peaks of the execution pattern will be random and may occur throughout a major portion of the execution of the program. Figure 4-8 provides an example of an execution pattern from a modified agenda parallel program that mirrors the execution of a specialist program. Each worker process of the modified agenda program creates a new worker process. The new worker process replaces the parent worker process. By continuously creating new worker processes, the program generates eval messages throughout a major portion of the execution of the program.

Although specialist programs produce execution patterns that are fairly unpredictable, the number of start and termination messages produced when environment processors are fixed is identical to all other Linda programs. Figure 4-4 illustrates the control sub-system's stability of generating the identical number of start and termination messages regardless of the type of Linda program, when the environment processors are fixed. If the number of created processes is fixed for specialist programs, the results should be identical to those programs shown in Figure 4-5. Although a specialist program was not created which demonstrates this point, the program would behave no differently than any other Linda program.

In general, the execution patterns of the agenda, result parallel and specialist programming paradigms behave in a similar fashion. All three paradigms produce execution patterns that manage three peaks of control messages. The start messages are found at the beginning of the execution pattern. The eval messages are also normally distributed towards the beginning of the execution pattern. The termination messages are found at the end of the execution pattern. Although the depiction of a different number of peaks of messages can occur on the execution patterns, this can be attributed to the flavor of each written program. For instance, an agenda program that immediately evals new processes can cause one single peak to form at the beginning of the execution pattern. Another result parallel program can perform such a large number of evals that a series of peaks are formed on the execution pattern for the program. Overall, however, three peaks of control messages are produced by the control sub-system, which lends towards stability of the control framework.

The final section of this chapter examines the execution characteristic of scalability within the control sub-system of Linda-LAN. Two forms of scalability are shown to exist within the control sub-system.

4.3. Verification of Scalability

Scalability within the control sub-system is an important execution characteristic that must be attained by the Linda-LAN environment. Without scalability, the Linda-LAN system would be unable to cope with the ongoing fluctuations in a networked environment. Linda-LAN must handle the numerous additions and deletions of network processors and the many changes in the numbers of processes created by executing programs. The control sub-system must scale appropriately to be deemed a viable solution for parallel and distributed programming.

In this section, two forms of scalability are addressed. The first deals with the scalability of network processors. As previously mentioned in Chapter 3, the control framework of Linda-LAN allows for the environment to have network processors added and removed at one centralized location. Although the ability to modify the number of processors is important, this ability is useless if the system cannot handle the changes in a predictable fashion. The second form of scalability deals with the scalability of program processes. The Linda programming paradigm and the Linda-LAN environment supply the programmer with the ability to alter the number of program processes without re-compiling or re-distributing the program. This feature is also meaningless unless the system can handle the fluctuations in program processes. In the following sub-sections, both of the above execution characteristics of scalability are shown to exist within the control sub-system of Linda-LAN.

4.3.1 Scalability as the Number of Environment Processors Increases

Agenda, result and specialist parallel programs are examined in this sub-section to show that the control sub-system of Linda-LAN scales appropriately as the number of environment processors increases. Figure 4-9 illustrates the repeated execution of an agenda parallel program as the number of environment processors increases from one workstation to six workstations. The numbers of instantiated worker processes are constant for all executions of the program. As seen in Figure 4-9, the number of control messages increases as the number of workstations increases. The execution patterns scale appropriately as the number of workstations increases. The number of control messages produced by two workstations is only slightly higher than one workstation. Although the difference between one workstation and six workstations is high, the percentage change is still less than fifty percent.

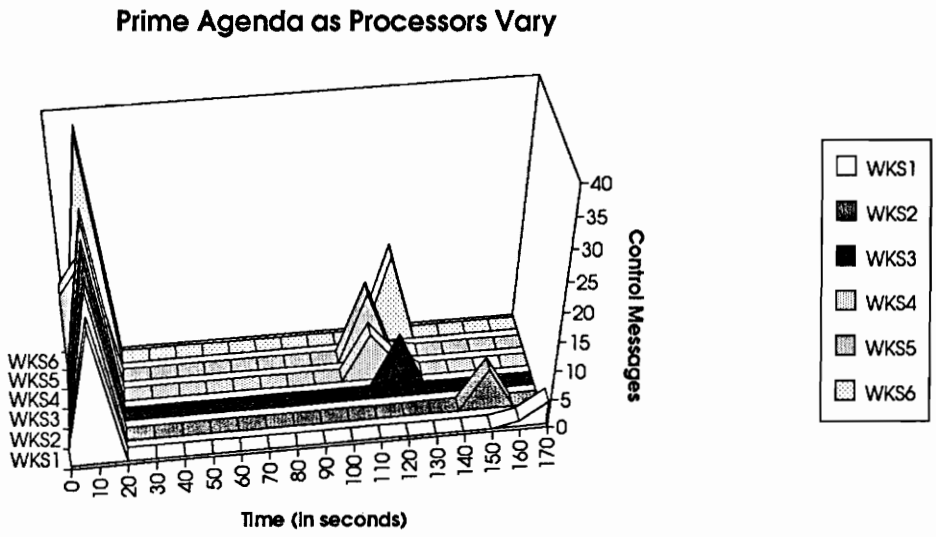


Figure 4-9. Execution patterns of a prime counter agenda program as the number of environment processors increases.

From Figure 4-9, a difference in the time at which the agenda program terminates is also noted. As expected, a decrease in the execution time occurred as the number of processors increased from one to two. The execution time continued to decrease until five processors were used. At this point, no further reduction in execution time occurred. The reduction in execution times demonstrates the advantages of using the Linda-LAN system. As seen from Figure 4-9, the Linda-LAN system is delivering performance benefits as the number of utilized processors is increased.

Figures 4-10 and 4-11 depict two other agenda parallel programs as the number of environment processors is increased. In both of these cases, the results are similar to the results of the prime agenda program of Figure 4-9. The execution patterns show the increases in the number of control messages as the environment processors are added. Figures 4-12 through 4-14 provide knowledge on the origins of the added messages. The control messages being added are a result of the additional start and termination messages needed to maintain the new processors.

Specifically, the number of start and termination messages increase linearly as the number of environment processors increases. As the number of processors increases, the L-Man must inform each L-Com of the instantiation of a new program for execution. Since there exists one L-Com per Linda-LAN Processor, a linear increase in the number of start messages is expected. The same argument is true for the linear increase in the number of termination messages. The control sub-system is not adding undue overhead to the system as the number of processors are increased.

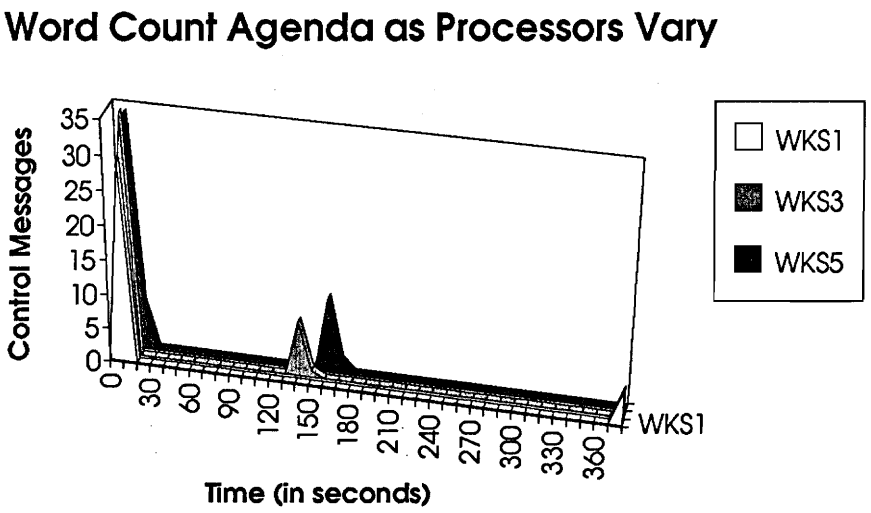


Figure 4-10. Execution patterns of a word counting agenda program as the number of environment processors increases.

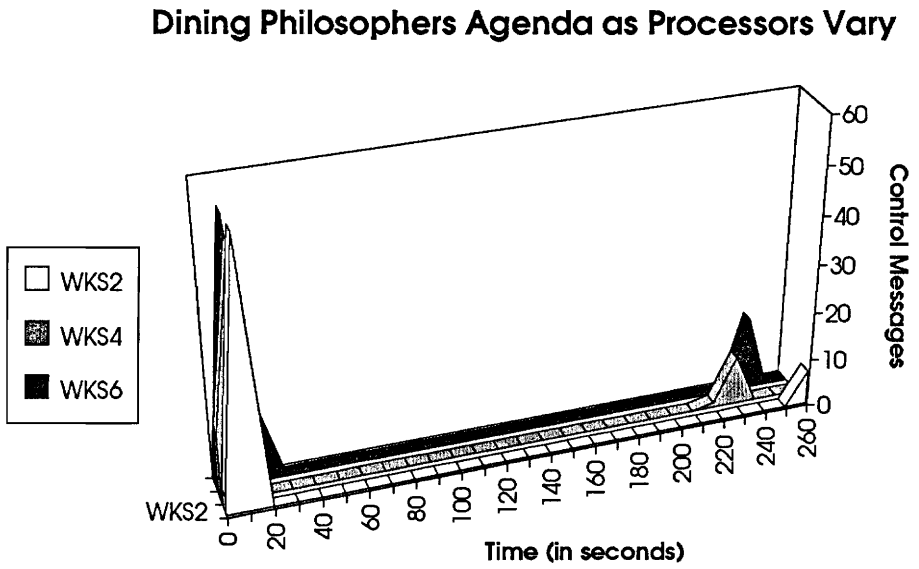


Figure 4-11. Execution patterns of a dining philosophers agenda program as the number of environment processors increases.

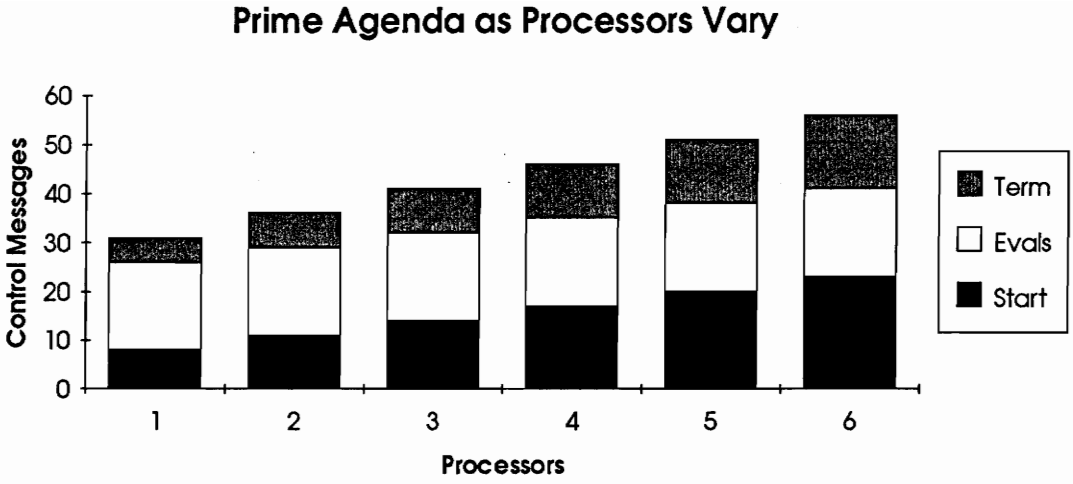


Figure 4-12 Control message totals for the prime counter agenda program.

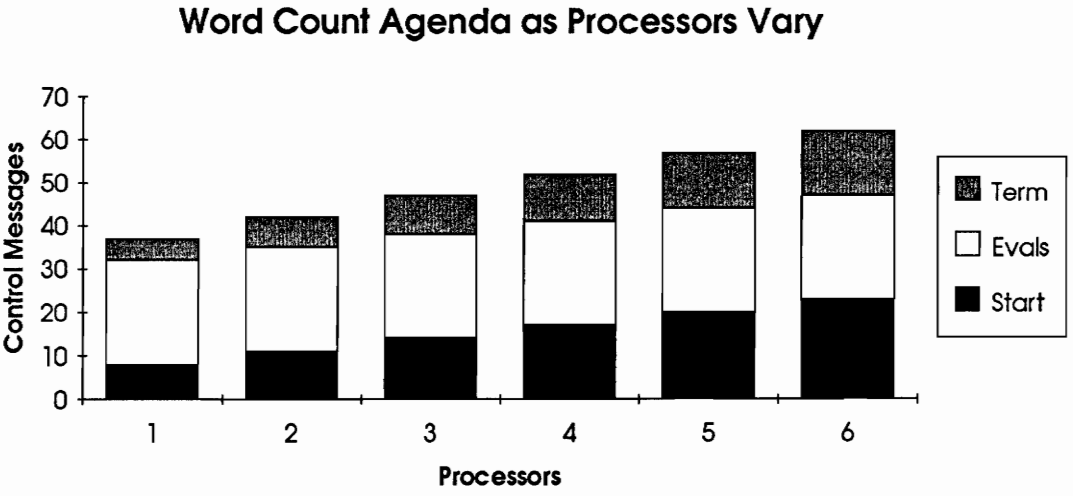


Figure 4-13 Control message totals for the word count agenda program.

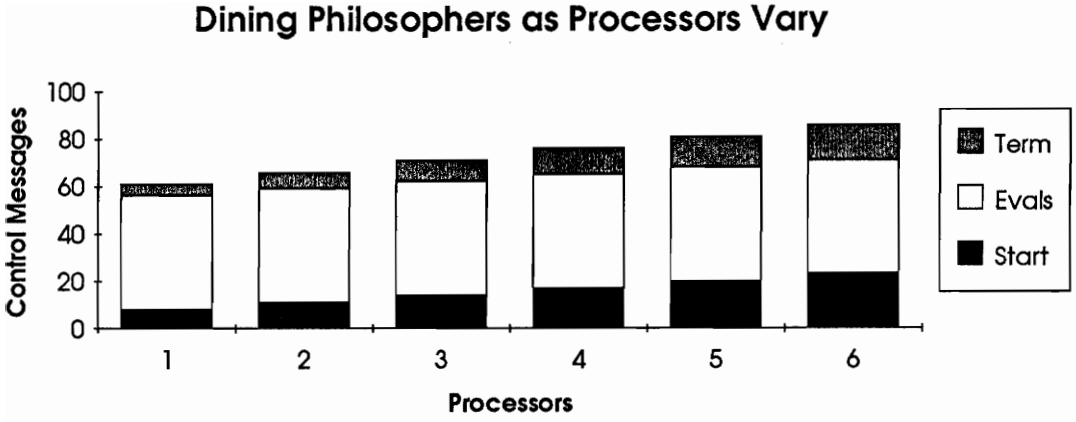


Figure 4-14 Control message totals for the dining philosophers agenda program.

The scalability of result parallel and specialist parallel programs are also shown in Figures 4-15 and 4-16. Results similar to the previous agenda programs are demonstrated. The result parallel program produced a reduction in execution time of twenty-five percent as the number of processors increased from one workstation to two workstations. However, the exact reduction in time was only ten seconds and only slight reductions in time were seen past two workstations. The specialist program produced a thirty-three percent reduction in execution time as the number of processors was increased from one workstation to six workstations.

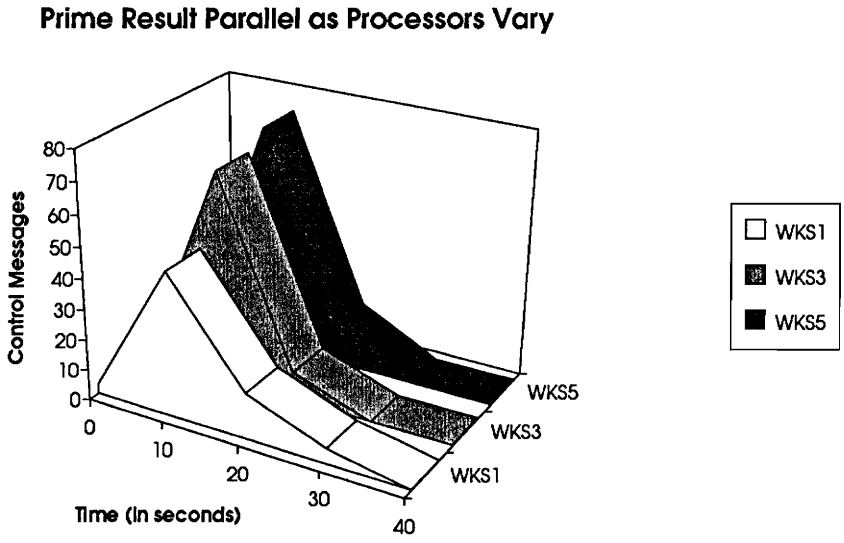


Figure 4-15 Execution patterns of a prime counter result parallel program.

The result parallel and specialist parallel programs also showed the same linear increase in start and termination messages as the number of processors increased. Figures 4-17 and 4-18 illustrate the total message counts as the number of processors were added. Like the agenda programs, the result and specialist programs showed no increase in the number of eval messages. The eval messages did not increase because of the constant number of instantiated processes. The following sub-section examines the scalability of the control sub-system as the number of instantiated program processes increases.

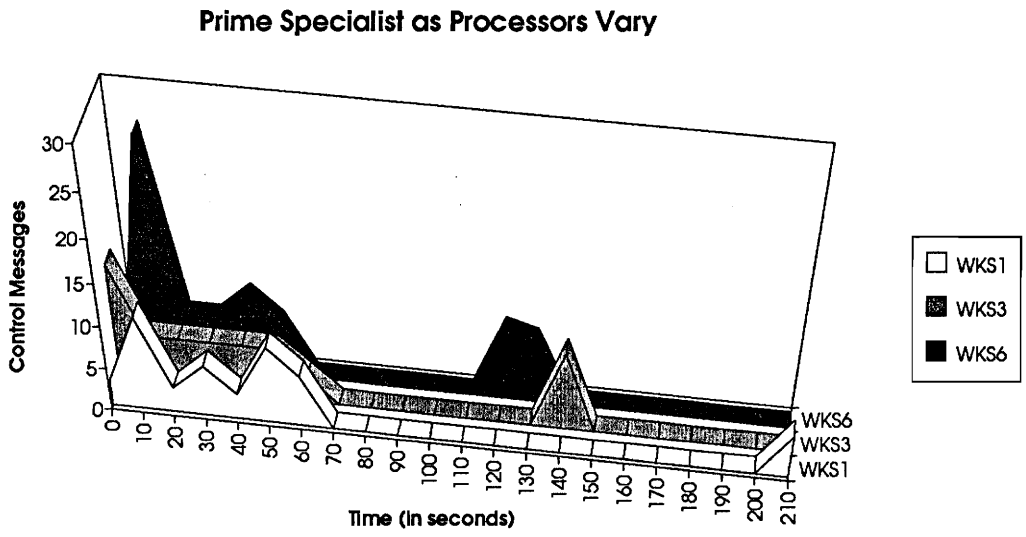


Figure 4-16 Execution patterns of a prime counter specialist parallel program.

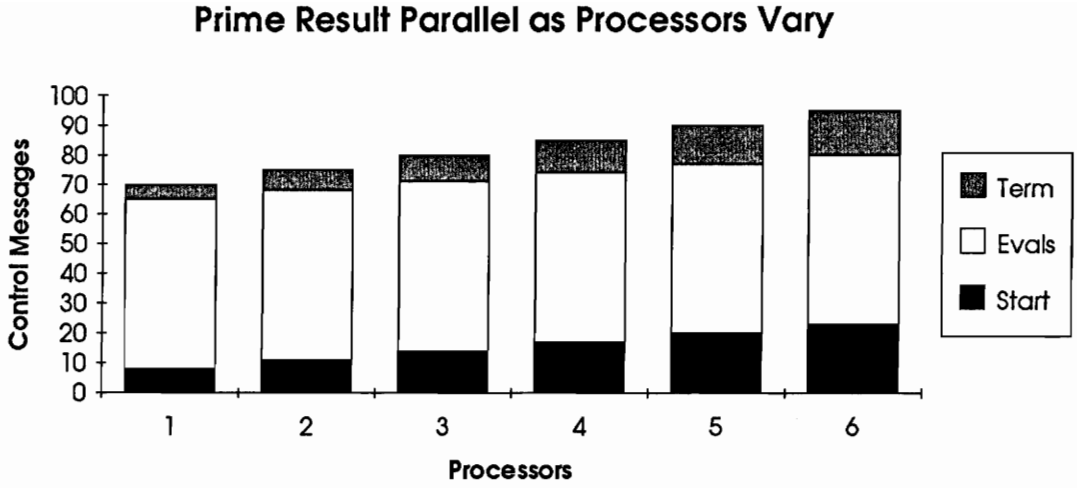


Figure 4-17 Control message totals for the prime counter result parallel program.

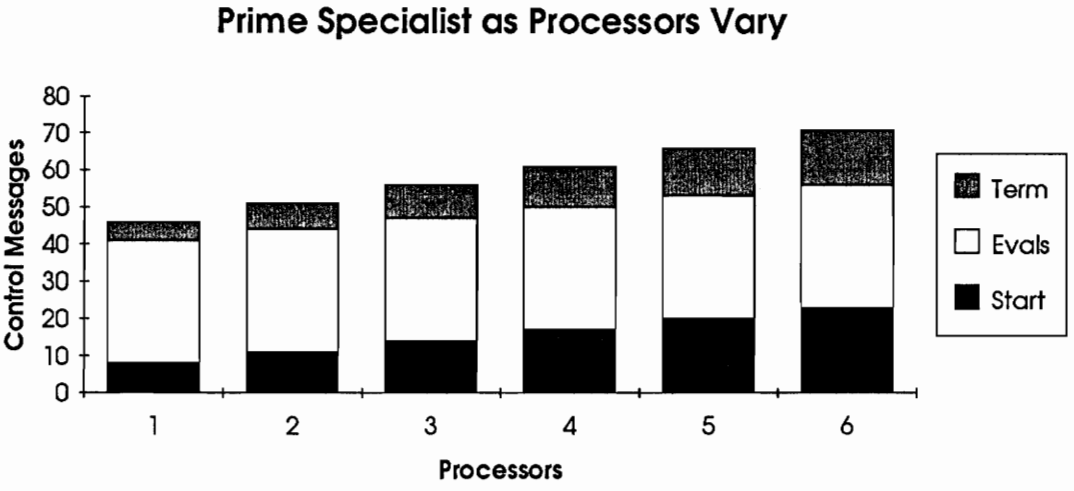


Figure 4-18 Control message totals for the prime counter specialist parallel program.

4.3.2 Scalability as the Number of Program Processes Increases

A program which increases the number of created processes may execute more efficiently and reduce its execution time. When a program increases the number of instantiated processes, the system overhead to manage the new processes increases. In order for an environment to be useful and effective, the environment must scale appropriately as the number of instantiated processes increases. If appropriate scale up is not achieved, the system behaves erratically and detrimental effects can occur. A system may incur drastic slowdown and fail in some cases. In this sub-section, a variety of programs are examined to show that the control sub-system of Linda-LAN demonstrates a linear increase in the number of control messages as the number of program processes increases.

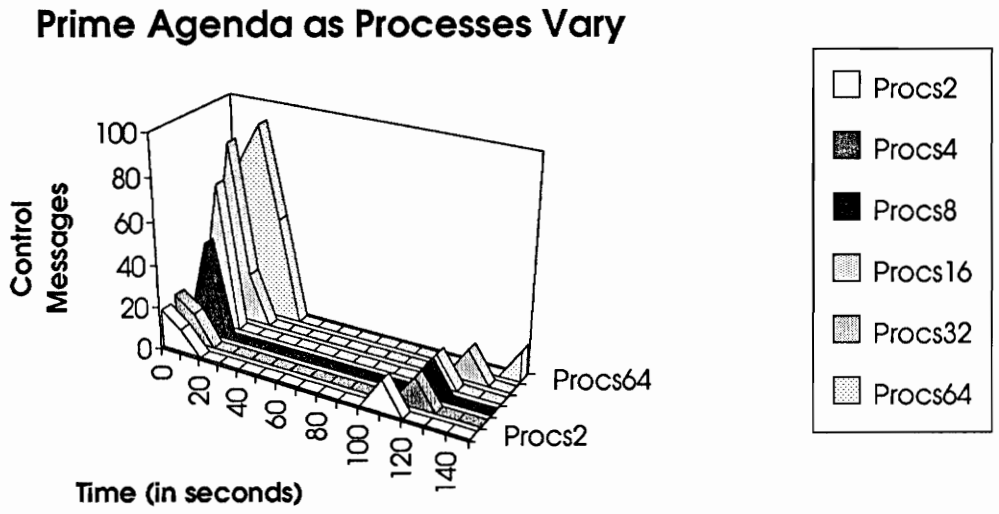


Figure 4-19 Execution patterns for the prime counter agenda program as the number of program processes increases.

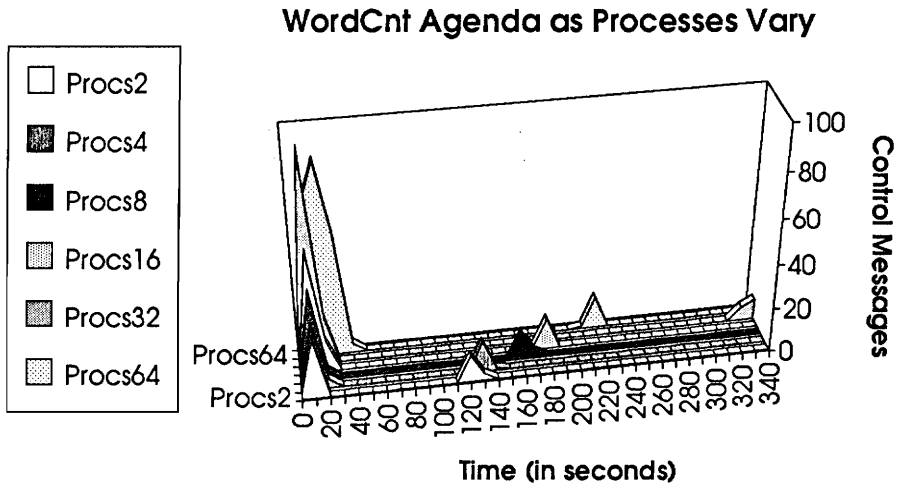


Figure 4-20 Execution patterns for the word count agenda program as the number of program processes increases.

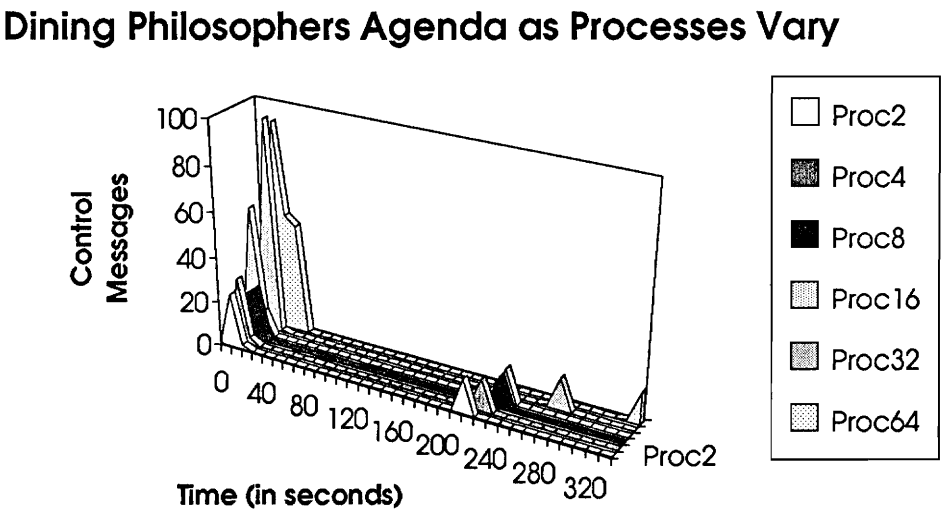


Figure 4-21 Execution patterns for the dining philosophers agenda program as the number of program processes increases.

4.3.2.1 Scalability of Agenda Parallel Programs as the Number of Processes Increases

Figure 4-19 depicts the execution patterns of an agenda parallel program as the number of program processes increased. Each of the initial peaks in execution patterns of the prime counter program increased as the number of instantiated processes rose. Additional peaks of messages are also generated as the number of processes increased. Figure 4-19 also shows that the program increased in execution time as the number of processes rose. No advantage was gained by increasing the number of program processes.

Two other agenda parallel programs are shown to have similar results. Figures 4-20 and 4-21 illustrate the word count program and the dining philosophers program while the number of program processes increased. The control messages gradually increased as the processes were added. The execution times also increased as the number of processes rose. The amount of processing being performed by the added processes does not warrant their usage. The additional messages in the control sub-system, the additional communication between the processes and Tuple Space, and possibly an insufficient amount of work to perform by the processes are contributing to the reductions in execution time. In addition, the overhead generated by the data sub-system's instantiation of a process on a Linda-LAN processor is high. The *forking* mechanism provided by the UNIX operating system and the retrieval of required start-up information from the Tuple Server are time consuming activities.

The linear relationship between the number of program processes and the number of control messages is illustrated in Figures 4-22 through 4-24. In each case, the number of eval messages doubled as the number of processes doubled. The start and termination messages were fixed as the number of processes increased. The numbers of workstations (processors) used for each execution were kept constant. The average times for the control sub-system to instantiate a new process are given in Table 4-1. The average

times varied between 1.0 and 2.0 seconds. No pattern of increase was shown to exist as the number of processes increased.

Agenda Program	2 Proc	4 Proc	8 Proc	16 Proc	32 Proc	64 Proc
Prime Counter	1.0	1.8	1.3	1.1	1.5	1.4
Word Count	1.5	1.8	1.6	2.0	1.6	1.9
Dining Philosophers	2.0	1.8	1.5	1.8	1.8	1.9

Table 4-1 Average times for instantiating agenda program processes within the control sub-system.

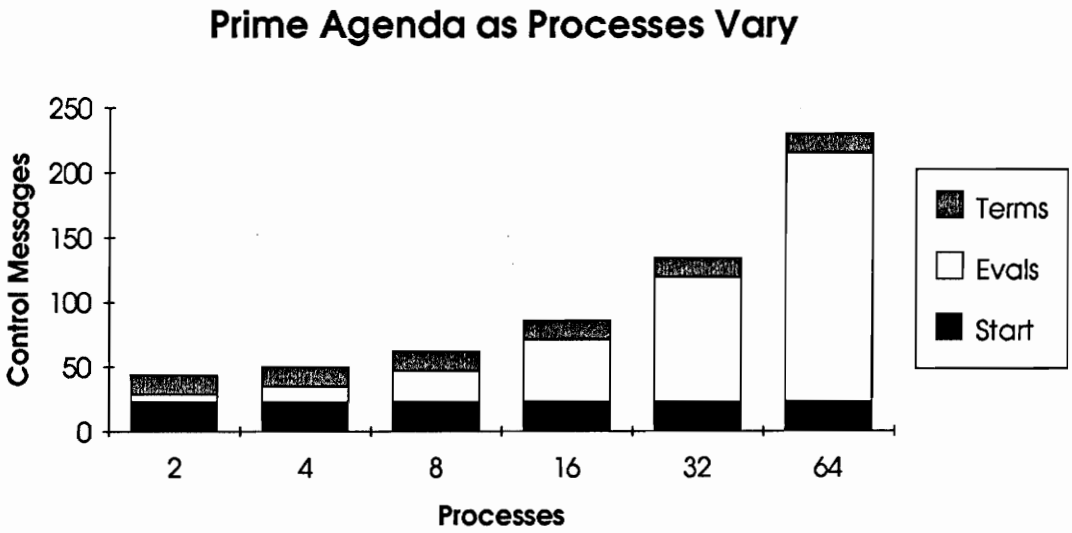


Figure 4-22 Message counts for the prime counter agenda program as the number of program processes increases.

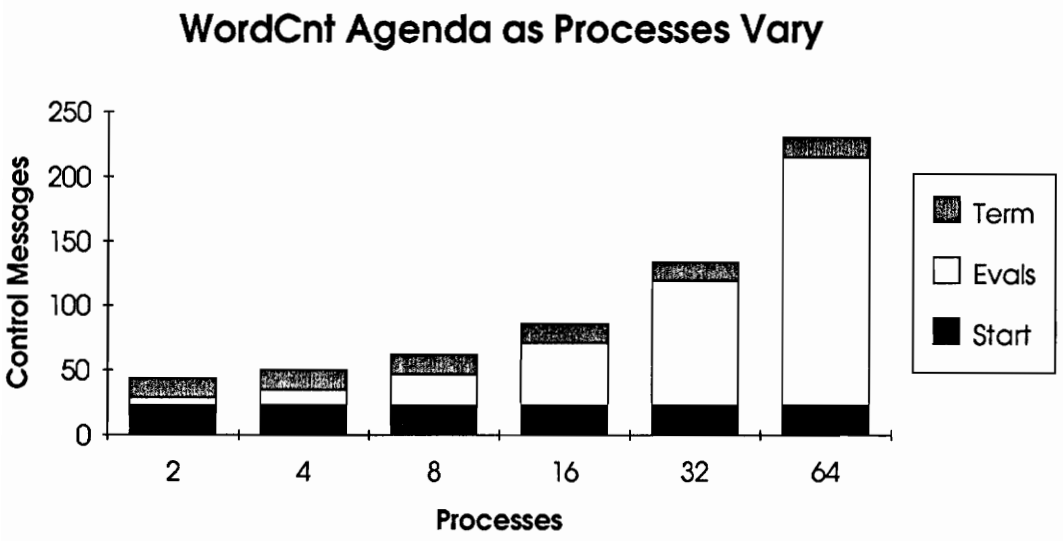


Figure 4-23 Message counts for the word count agenda program as the number of program processes increases.

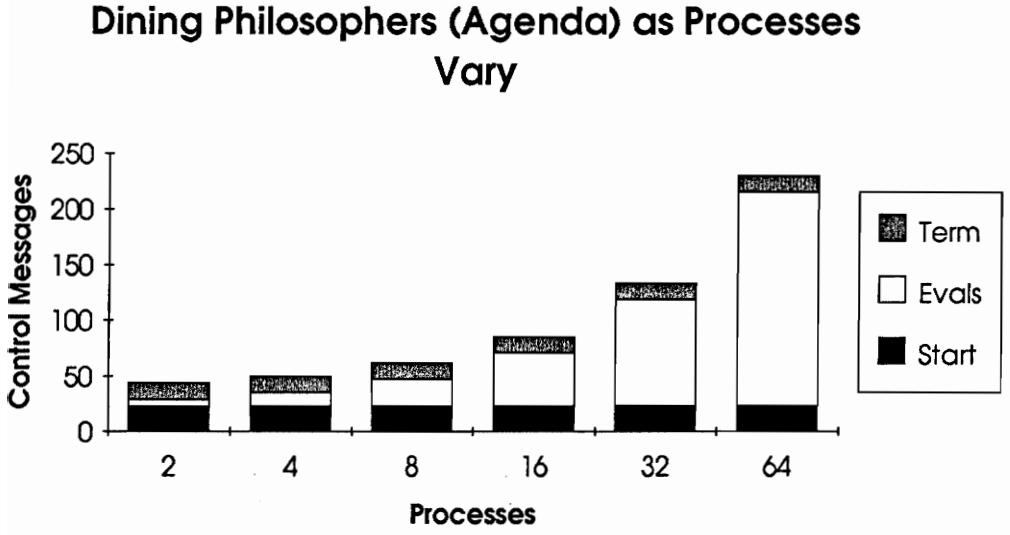


Figure 4-24 Message counts for the dining philosophers agenda program as the number of program processes increases.

4.3.2.2 Scalability of Result and Specialist Parallel Programs as the Number of Processes Increase

Programs that are based on the result parallel programming paradigm also display a linear relationship between the number of instantiated processes and the number of control messages. Figures 4-25 and 4-26 illustrate results similar to the agenda programs. The gradual increase in the peaks of control messages is depicted. The eval messages of Figure 4-26 doubled as the number of processes doubled. The execution times increased after the number of processes increased beyond thirty-two. However, no reduction or increase in execution times was noted when the processes increased from two processes to sixteen processes.

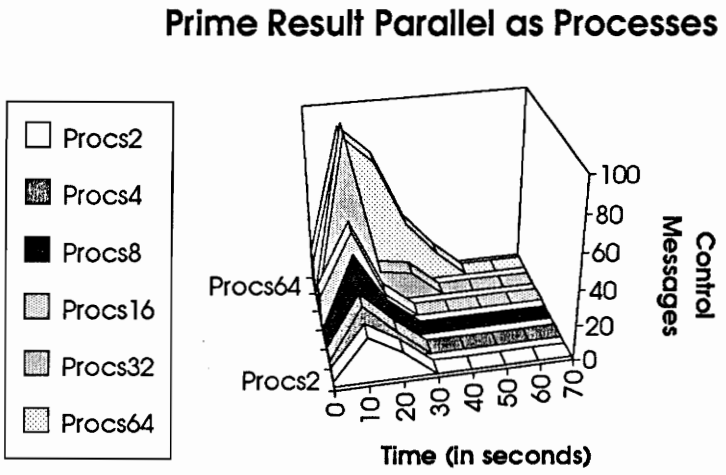


Figure 4-25 Execution patterns for the prime counter result parallel program as the number of program processes increases.

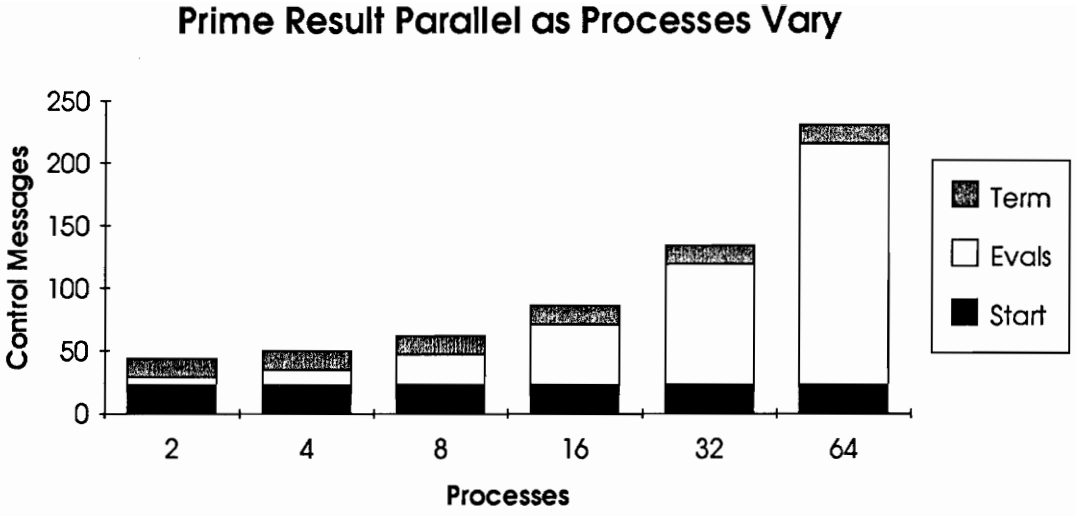


Figure 4-26 Message counts for the prime counter result parallel program as the number of program processes increases.

In order to increase the number of processes within a result parallel program, the input data must be altered. Result parallel programs create a process for each computed result. Additional processes imply additional results need to be calculated. In the result parallel primer counter program, additional numbers were checked for being prime.

The number of instantiated processes for specialist parallel programs cannot be increased in the same manner as agenda or result parallel programs. Each process of the specialist parallel program has a specific function. If the specialist processes are merely duplicated, the programming paradigm is fundamentally altered to resemble a specialist-agenda program. In order to illustrate the randomness in process creation that a specialist parallel program provides, the modified agenda parallel program is used. Figure 4-27 shows the execution patterns of the modified word count program as the number of processes is increased. The control messages increased as the number of processes increased. The execution time

decreased when processes were raised from two to four. However, the execution times increased once again when eight or more processes were used.

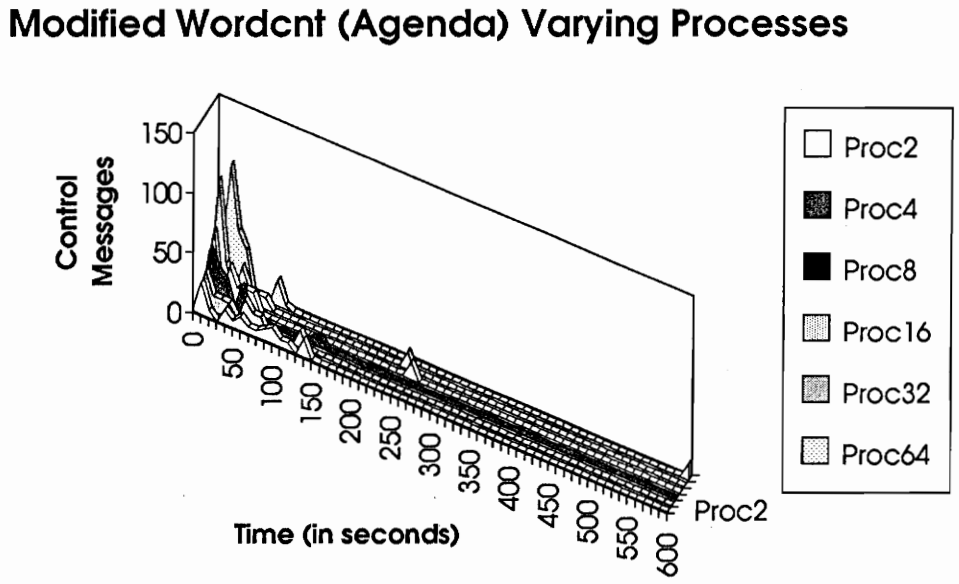


Figure 4-27 Execution patterns for the modified agenda program as the number of program processes increases.

Figure 4-28 shows the increase in the eval messages as the processes increased. The modified word count program differs from the prior programs presented in this sub-section. The modified word count does not create two, four, eight, etc., processes. The modified word count program begins by creating two, four, eight, etc., processes. The created processes continually create new processes as the program executes. Once a process has computed a set of values, the process evals a new replacement process and then terminates itself. For the above reasons, the modified agenda parallel program does not double the number of eval messages as the number of initial processes doubles. However, a linear relationship is exhibited between the number of eval messages and the increase in processes.

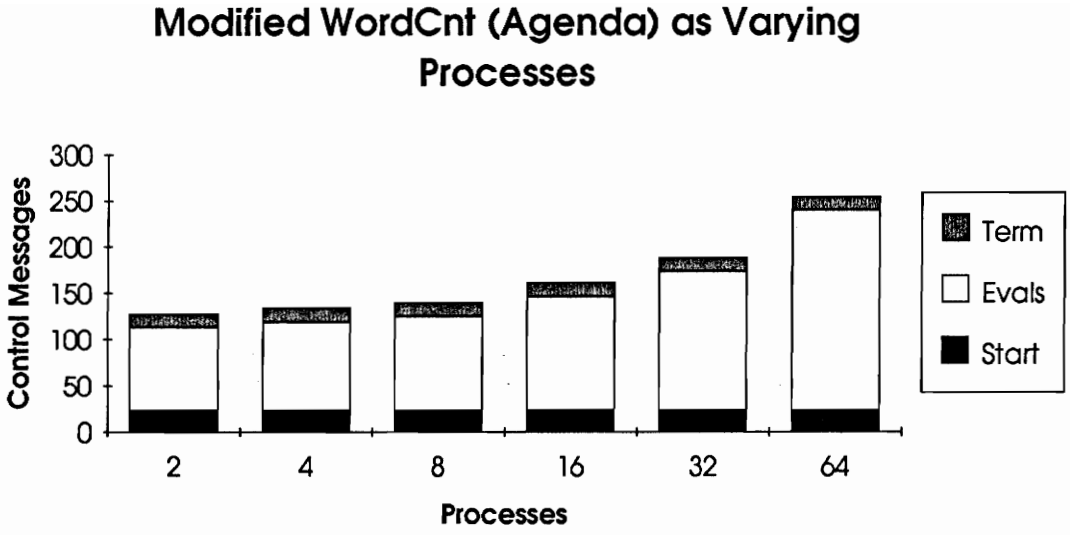


Figure 4-28 Message counts for the modified word count agenda parallel program as the number of program processes increases.

The centralized scheduling by the control sub-system attributes to the scalability of the control sub-system as the number of instantiated processes increases. Specifically, the eval messages that deal with process to processor allocation increase. Looking back at Table 4-1, the control sub-system's overhead in process instantiation is small and stable. Although a linear relationship is seen in the scalability of the control sub-system as the number of program processes increases, the scalability of the environment in its entirety may not hold. The amount of overhead generated by the data sub-system's instantiation of a process is unknown.

5

CONCLUSIONS

A control framework for distributed parallel processing environments is presented within this thesis. The control framework is introduced through a new distributed parallel processing environment called Linda-LAN. The Linda-LAN environment incorporates the control framework within its control sub-system. The environment is also based on the Linda parallel programming paradigm. The goals of the thesis are three-fold. First, the Linda-LAN environment, along with its control sub-system, are examined in detail. Second, the applicability of the control framework within other distributed environments is studied. Finally, the execution characteristics of stability and scalability are shown to exist within the control sub-system of Linda-LAN. The remainder of this chapter provides concluding remarks on the control framework and the Linda-LAN environment. Additional remarks are made on the application of the control framework to other distributed environments. Future research possibilities and a summary of the contributions of this research are also given.

5.1 Concluding Remarks on the Linda-LAN Environment and the Control Framework

With the increasing user demand for greater computational computing power, the Linda-LAN environment is a conceptually attractive configuration which provides

- a true parallel computational environment at an economical price,
- greater computing power than conventional uni-processor workstations,
- wider accessibility to a parallel environment,
- utilization of idle network CPU cycles, and
- centralized control of all management activities and system resources.

The Linda paradigm also provides the system with a portable and easy to use operational framework. The Linda language gives the programmer the ability to design, develop and implement parallel programs in a comprehensible and structured manner. By partitioning the system into the control and data sub-systems, research efforts have become more focused and the implementation of the system has been simplified. In addition, a more simplistic view of the environment is created.

The control framework supplies Linda-LAN with a control sub-system that centralizes system management activities. Global scheduling of programs, processes, and processors is provided, as well as global system monitoring and the collection of system statistics. The control sub-system empowers a designated administrator with control and flexibility over the environment. The control sub-system relieves the data sub-system from having to perform management activities. Furthermore, scalability and stability have been shown to exist for the control sub-system. A linear relationship exists between the

number of control sub-system messages and the number of program processes instantiated, as well as the number of environment processors used. Overall, the Linda-LAN system offers a viable parallel processing environment within a conceptually simple parallel programming framework that utilizes a local area network of low cost personal computers. In addition, the Linda-LAN system is well controlled and managed through a centralized control sub-system.

5.2 Additional Remarks on the Control Framework and Other Environments

Although the control framework is specifically designed for the Linda-LAN environment, the control framework can provide for effective and efficient management of other distributed environments. The control framework supports the centralized control over a distributed environment. CSP, occam, ADA, RPC, Emerald, and Orca are examples of languages and environments that may utilize the control framework. The process to processor allocation capabilities of the control framework are beneficial, as well as the code distribution features. The control framework's handling of program instantiation and termination can apply to other environments. The stability and scalability of the control framework are also important execution characteristics for all environments.

5.3 Future Research Possibilities

The research within this thesis on the Linda-LAN environment and the control framework provides a basis and reference for future work. This section attempts to outline new areas of research possibilities. The following items discuss the possible topics for the future.

1. The current Linda-LAN environment is composed of a homogeneous group of Commodore 3000 UX workstations. Future research should be conducted to migrate Linda-LAN to a heterogeneous group of workstations. A heterogeneous group of workstations would facilitate a more practical working environment.

2. Research addressing the area of fault tolerance within the Linda-LAN environment is needed. The current environment fails to support the loss of the Tuple Server or the Communications Server.

3. Additional work in the area of scheduling within the control framework is necessary. The present control framework within Linda-LAN only supports the execution of a single C-Linda program at any given time. The environment must support multiple executing programs in order to be a more usable environment.

4. The current Linda-LAN environment operates on a local area network of workstations. Future versions of the environment should explore the possibilities of operating over a wide area network of workstations. The current implementation does not limit the usage of network gateways, however, the practicality of using gateways should be examined.

5. New process to processor allocation strategies should be examined. The current implementations only support a round-robin allocation of processes to processors.

6. The ability to debug executing Linda-LAN programs should be addressed. Executing views of the environment's Tuple Space are needed to help in the development of parallel programs.

7. The building of functional libraries that are available to all executing programs should be examined. Many programs require common functionality. Reductions in the amounts of executable code and reusability of executable code are important for all environments.

8. The application of the control framework to other environments should be conducted. The results of the Linda-LAN environment should be compared and verified with the application of the control framework within another environment.

9. Finally, the area of performance should be addressed within Linda-LAN. One of the main drawbacks of the Linda paradigm is the questionable performance of the Tuple Space model. As workstation and communications technologies increase, new methods of accessing Tuple Space should be examined that take advantage of the new technologies.

The above list of research possibilities provides areas for future work. The next section summarizes the contributions of this thesis.

5.4 Contributions of this Thesis

In summary, the research conducted in this thesis supplies a control framework for parallel distributed environments. The control framework provides users with flexibility and centralized control over their networked environment. The control framework is encapsulated within a new parallel distributed environment, dubbed Linda-LAN. The design of the Linda-LAN environment and the implementation of its control sub-system were a direct result of this thesis. The Linda-LAN environment establishes a low-cost parallel processing solution that takes advantage of the unused processing power of local area networks. The Linda-LAN environment emphasizes the ease of use and portability of the Linda parallel programming paradigm.

References

[ADA83] U.S. Department of Defense, *Reference Manual for the Ada Programming Language*, 1983.

[ARTHU91] J. D. Arthur, G. Cline and K. Landry, "Linda-LAN: A Distributed Parallel Processing Environment Based Upon The Linda Paradigm," *A Research Proposal*, Computer Science Department, Virginia PolyTechnic Institute and State University, 1991.

[BAL90] H. Bal, M. Kaashoek, and A. Tanenbaum, "Experience with Distributed Programming in Orca," *Proceedings IEEE CS 1990 International Conference on Computer Languages* (New Orleans, LA), March 1990, pp. 79-89.

[BAL92] H. Bal and M. Kaashoek, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, March 1978, pp. 934-941.

[BERND89] D. Berndt, "C-Linda reference Manual (DRAFT) Beta Version 2.0," *Scientific Research Associates*, January 1989.

[BIRRE84] A. D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-59.

[BJORN88] R. Bjornson, N. Carriero, D. Gelernter and J. Leichter, "Linda, the Portable Parallel," *Research Report YALE/DCS/RR-520*, January 1988.

[BJORN89] R. Bjornson, N. Carriero, and D. Gelernter, "The Implementation and Performance of Hypercube Linda," *Research Report YALEU/DCS/RR-690*, March 1989.

[BLACK87] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, Vol. 13, No. 1, January 1987, pp. 65-76.

[BRINC75] P. Brinch Hansen, "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, June 1975, pp. 199-207.

[BRINC78] P. Brinch Hansen, "Distributed Processes: a Concurrent Programming Concept," *Communications of the ACM*, Vol. 21, November 1978, pp. 934-941.

[BRINC93] P. Brinch Hansen, "Monitors and Concurrent Pascal: A Personal History," *ACM SIGPLAN Notices*, Vol. 28, No. 3, March 1993, pp. 2-70.

[CARRI87] N. Carriero, "Implementation of Tuple Space Machines," *Research Report YALEU/DCS/RR-567* (PhD thesis), December 1987.

[CARRI88] N. Carriero and D. Gelernter, "Applications Experience with Linda," *Proc. ACM Symp. Parallel Programming*, July 1988.

[CARRI89a] N. Carriero and D. Gelernter, "Coordination Languages and their Significance," *Yale Tech Report*, YALEU/DCS/RR-716, July 1989.

[CARRI89b] N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, Vol. 32, No. 4, April 1989.

[CARRI90] N. Carriero and D. Gelernter, *How to Write Parallel Programs*. MIT Press, Cambridge, 1990.

[CLINE93] G. Cline and J. Arthur, "Linda-LAN: A Controlled Parallel Processing Environment," *IEEE International Phoenix Conference on Computers and Communications* (Scottsdale, AZ), March 1993.

[DAVID89] C. Davidson, "Technical Correspondence on *Linda in Context*,", Communications of the ACM, Vol. 32, No. 10, October 89, pp.1249-1252.

[DEITE84] H. Deitel, *An Introduction to Operating Systems*. Addison-Wesley Publishing Company, Reading, 1984.

[GELER85a] D. Gelernter, "Generative Communication in Linda," ACM *Transactions on Programming Languages and Systems*, Vol. 7, No. 1, January 1985, Pages 80-112.

[GELER85b] D. Gelernter, N. Carriero, S. Chandran and S. Chang, "Parallel Programming in Linda," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp.255-263.

[GELER90] D. Gelernter, "Ada-Linda: Motivation, Informal Description and Examples," *Yale Tech Report*.

[HALST85] R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 4, October 1985, pp. 501-538.

[HOARE74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol. 17, No. 10, October 1974, pp. 549-557.

[HOARE78] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

[KELLE84] R. Keller, F. Lin, and J. Tanaka, "Rediflow Multiprocessing," *IEEE COMPCON*, Spring 1984 (San Francisco), February 1984, pp. 410-417.

[LELER88] Wm. Leler, "PIX, the latest NeWS," *Cogent Technical Report*, Cogent Research, November 1988.

[LELER90] Wm. Leler, "Linda Meets Unix," *IEEE Computer*, February 1990, pp.43 - 54.

[LISTO88] B. Listov and L. Shrira, "PROMISES: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," *Proceedings of 88 Conference on Programming Languages Design and Implementation*, June 1988, pp. 260-267.

[MAY83] D. May, "OCCAM," *SIGPLAN Notices*, Vol. 18, No. 4, April 1983.

[ROBIN94] P. Robinson, "Distributed Linda: Design, Development, and Characterization of the Data Sub-System," Master's Thesis, Virginia Polytechnic Institute and State University, Virginia, 1994.

[SCHUM91] C. Schumann, K. Landry and J. D. Arthur, "Comparison of Unix Communication Facilities Used in Linda," *Proceedings of the 1991 Virginia Computer Users Conference.*

[SCHUM93] C. Schumann, "Distribution of Linda across a Network of Workstations," Master's Thesis, Virginia Polytechnic Institute and State University, Virginia, 1993.

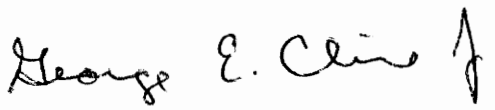
[THEIM89] M. Theimer and K. Lantz, "Finding Idle Machines in a Workstation-Based Distributed System," *IEEE Transactions on Software Engineering*, Vol. 15, No. 11, November 1989.

[WHITE88] R. Whiteside and J. Leichter, "Using Linda for Supercomputing On a Local Area Network," in *Proc. Supercomputing '88*, November 1988.

[ZENIT90] S. E. Zenith, "Linda Coordination Language; subsystem kernel architecture (on transputers)," *Research Report YALEU/DCS/RR-794*, May 1990.

Vita

George E Cline, Jr. was born in Augusta, Georgia on September 6, 1964. The majority of his childhood was spent in the small town of Cadiz, Ohio. His undergraduate studies in Computer Science and Applied Mathemethics were conducted at the University of Akron in Akron, Ohio. After obtaining his bachelor's degrees, he worked as a Systems Programmer for Westfield Companies. Three years were spent at Westfield before he decided to return to academia for graduate studies at Virginia Polytechnic Institute and State University. While at Virginia Tech, George married and began work at INCODE Corporation in Blacksburg, Virginia as a Software Engineer. After completing the Master's in Computer Science, George continued working at INCODE as the Project Leader of the Specification and Object Management Systems (SOMS) Group at INCODE.



George E. Cline, Jr.