1990

Linda Implementations Using Monitors and Message Passing

Alan L. Leveton
University of North Florida

LINDA IMPLEMENTATIONS
USING MONITORS AND MESSAGE PASSING

by

Alan L. Leveton

A thesis submitted to the
College of Computer and Information Sciences
in partial fulfillment of the requirements for the
degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
COLLEGE OF COMPUTER AND INFORMATION SCIENCES

December, 1990

The thesis "Linda Implementations Using Monitors and Message Passing" submitted by Alan L. Leveton in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

4/3/91

Thesis Adviser and Committee Chairman

Signature Deleted

4/3/91

Signature Deleted

4/4/91

Accepted for the College of Computer and Information Sciences:

Signature Deleted

4/4/91

Interim Dean

Accepted for the University:

Signature Deleted

4-4-91

Vice-President for Academic Affairs

ACKNOWLEDGEMENT

Naturally, nothing of this magnitude is accomplished without the tolerance and patience exerted by Lajeanne, Josh, and Mike.

CONTENTS

FIGURES

ABSTRACT

Linda is a new parallel programming language that is built around an interprocess communication model called generative communication that differs from previous models in specifying that shared data be added in tuple form to an environment called tuple space, where a tuple exists independently until some process chooses to use it. Interesting properties arise from the model, including space and time uncoupling as well as structured naming. We delineate the essential Linda operations, then discuss the properties of generative communication. We are particularly concerned with implementing Linda on top of two traditional parallel programming paradigms - process communication through globally shared memory via monitors, and process communication in local memory architectures through the use of message passing constructs. We discuss monitors and message passing, then follow with a description of the two Linda implementations.

Chapter 1

INTRODUCTION

1.1 Thesis Organization

Chapter 1 reviews the basic problem and the current research in tuple space coordination of parallel processes. Chapter 2 delineates our plan of attack: a background of the monitors and message passing paradigms that support our Linda implementations; a high-level design description; and those fundamental choices that influenced the design from the onset. We then present enough of the detailed design to allow the interested reader to judge, modify or enhance the implementations as he sees fit. Chapter 2 concludes with a description of four test cases: three variations on prime number generation and a parallel Linda solution to a semigroups problem. Finally, in Chapter 3 we evaluate the effort and conclude with recommendations for enhancement of the Linda model.

1.2 Problem Review

1.2.1 Linda Background

The abstract environment called tuple space forms the basis of Linda's model of communication. A process generates an

object called a tuple and places it in a globally shared collection of ordered tuples called tuple space. Theoretically, the object remains in tuple space forever, unless removed by another process [CAR89].

Tuple space holds two varieties of tuples. Process or 'live' tuples are under active evaluation, incorporate executable code, and execute concurrently. On the other hand, data tuples are passive, ordered collections of data items. For example, the tuple ("mother","age",56) contains three data items: two strings and an integer. A process tuple that is finished executing resolves into a data tuple, which may in turn be read or consumed by other processes [CAR89A].

Four operations are central to Linda: *out*, *in*, *rd* and *eval*. *Out*(t) adds tuple *t* to tuple space. The elements of *t* are evaluated before the tuple is added to tuple space [AHU86]. For example, if array[4] contains the value '10', *out*("sum",2,array[4]) adds the tuple ("sum",2,10) to tuple space and the process continues immediately.

In(m) attempts to match some tuple *t* in tuple space to the template *m* and, if a match is found, removes *t* from tuple space. Normally, *m* consists of a combination of actual and formal parameters, where the actuals in *m* must match the actuals in *t* by type and position and the formals in *m* are

assigned values in *t* [AHU86]. Thus, given the tuple noted above, *in*("sum",?i,?j) matches "sum", assigns 2 to i, 10 to j, and the tuple is removed from tuple space. *Rd* is similar to *in* except that the matched tuple remains in tuple space. Unlike the other operators, the executing process suspends if an *in* or *rd* fails to match a tuple.

Eval(t) is similar to *out*(t) with the exception that the tuple argument to *eval* is evaluated after *t* is added to tuple space. A process executing *eval* creates a live tuple and continues. In creating the active tuple, *eval* implicitly spawns a new process that begins to work evaluating the tuple [CAR89A]. For example, if the function *abs*(x) computes the absolute value of *x*, then *eval*("ab",-6,abs(-6)) creates or allocates another process to compute the absolute value of -6. Once evaluated, the active tuple resolves into the passive tuple ("ab",-6,6) which can now be consumed or read by an *in* or *rd*. *Eval* is not primitive in Linda and is actually constructed on top of *out* and provides Linda with a mechanism to dynamically create multiple processes to assist in a task. Implementations of Linda exist that do not recognize the *eval* operation [AHU86], including a network model based on worker replication - *n* nodes are given *n* copies of a program, thereby obviating the need for dynamic process creation.

Tuple members are usually simple data types: characters, one-dimensional strings, integers, or floats. In some Linda implementations tuples can include more complex data types (e.g., integer arrays) [CAR89A]. These data structures are removed from or added to tuple space just like the more fundamental types.

Operations which insert or withdraw from tuple space do so atomically. In theory, nondeterminism is inherent - it is assumed that the tuples are unordered in tuple space so that, given a template *m* and matching tuples *t1*, *t2* and *t3*, it can not be determined which tuple will be removed by *in*(m) [GEL85]. In practice, implementations of tuple space fall short of pure nondeterminism - some ordering is inescapable but remains implementation dependent. It is in the spirit of Linda programming not to presuppose any ordering of tuples in the underlying mechanism. Sequencing transactions upon tuple space is facilitated using a sequencing key as an additional tuple element [LEL90], a method employed to program distributed arrays in Linda. Thus the ith element of vector "A" is accessed via

in("A",i,<some_number>)

while the ith + 1 element is added to tuple space with

out("A",i+1,<some_number>)

We avoid the need for sequencing keys if some ordering of tuples is guaranteed in an implementation, but not without

costs - a programmer must be aware of the internal ordering mechanism, and the implementation loses orthogonality.

Several properties distinguish Linda. Generative communication simply means that a tuple generated by process *p1* has independent existence in tuple space until removed by some process *p2*. This property facilitates communication orthogonality because a receiver has no prior knowledge about a sender and a sender has none about the receiver - all communication is mediated through tuple space. Spatial and temporal uncoupling also mark Linda. Any number of disjoint processes may input tuples and tuples added to tuple space by *out* remain in tuple space until removed by *in* [GEL85].

A property called structured naming deserves special consideration. Given the operations out(t1) and in(m1), all actuals in *t1* must match the corresponding actuals in *m1* for matching to succeed. The actuals in *t1* constitute a structured name or key and, loosely speaking, make tuple space content addressable. For example, if ("sum",10,9) is a tuple in tuple space, then the success of the operation in("sum",?x,10) is predicated upon the structured name ["sum",10]. We are reminded both of the restriction operation in relational databases and instantiation in logic languages [GEL88]. The structured name should not be confused with the logical name, which is simply the initial

element in a tuple and must be an actual of any type. If *n1* is the logical name in template *m1*, and if any tuple in tuple space with *n1* as the first element successfully matches *m1*, then *n1* is said to be a single *non-unique* key. Restricting the key to a single element reduces search time if hashing is used to implement tuple space [LEL90]. In many of the examples that follow, the logical name is used as the key.

While Linda is best suited to building distributed data structure programs involving many worker processes attacking the structure simultaneously, it also works well with more traditional methods of employing parallelism [AHU86]. Furthermore, because it is a high-level programming tool, Linda can model both the shared memory as well as message passing style of programming regardless of the underlying architecture [LEL90].

1.2.2 The Problem

It was our desire to implement two versions of Linda - one to take advantage of a shared memory architecture, the other to utilize the resources of networked machines, offering an advantage in portability. Each implementation is based upon a different programming model. An abstract data structure called a monitor synchronizes access to shared data in shared memory architectures, whereas processes in disjoint

memory space communicate through message passing operations [BOY87].

Although shared memory seems a natural for tuple space, some means is required to make the operations on tuple space atomic. During the brief moment in which a process either places a tuple into tuple space or consumes a tuple, the process must be assured of being the sole process operating on the data. Monitors provide a coherent means to protect tuples from simultaneous access by processes executing in parallel. We developed the monitors model on an eight-processor Sequent Balance 8000, a shared memory multi-processing machine.

The message passing programming model provides a means for disjoint, loosely coupled processes to communicate solely through messages and is used to implement Linda in two environments: a shared memory machine that supports message passing primitives and a group of workstations that communicate over a local area network. We used an Ethernet network of Suns for the workstation environment, while the Sequent provided an excellent test bed for both implementations because it also support message-passing primitives.

Both programming paradigms are high level abstractions in themselves and provide an intelligent means to construct

parallel programs in diverse environments. The challenge was to bootstrap the approaches to a higher level of abstraction - that of the Linda model.

1.3 Literature Review

Boyle and others recognized the need for a set of portable tools to aid in parallel programming and describe their operation and applicability in several programs [BOY87]. Three multiprocessing paradigms are supported: (1) shared-memory multiprocessors; (2) a set of processors that communicate solely through messages (typically, a multiprocessor that does not support shared memory, or a group of workstations that communicate over a LAN); (3) communicating clusters - sets of large multiprocessing machines that communicate via message passing. The tools that support these paradigms achieve portability by hiding machine dependent details behind convenient macros (later, as their package evolved, the authors converted the macros to less cryptic functions).

Many of the properties of Linda were first described in [GEL85]. Gelernter introduces generative communication, which he argues is sufficiently different from the three basic kinds of concurrent programming mechanisms of the time (monitors, message passing, and remote operations) as to make it a fourth model. It differs from the other models in

requiring that messages be added in tuple form to the environment called tuple space where they exist independently until a process chooses to receive them. Generative communication became the basis for Linda, originally developed for the SBN network computer. Gelernter further elucidates the structured naming rules for tuples and some additional distinguishing properties - communication orthogonality, space uncoupling, time uncoupling, distributed sharing, and free naming. Carriero and others describe a Linda implementation designed at AT&T Bell Laboratories on the S/Net multicomputer [CAR86]. Of interest is the manner in which tuple space is implemented. Upon executing *out*(t), tuple *t* is broadcast to every node in the network, thus imposing a copy of tuple space on each node and forcing a delete protocol to handle *in*'s. If a matching tuple is found locally, an attempt is made to delete *t* across the entire network. All nodes must receive the delete message, and only one process attempting a deletion will succeed. The overhead to accomplish this protocol is surprisingly inexpensive because the nodes communicate over a fast, word-parallel bus. The costly storage requirements of replicated tuple space have spawned variations on the S/Net kernel. One attempt stores generated tuples locally and broadcasts templates to all nodes, a scheme which avoids the replication problem [CAR86].

Throughout the literature, the hardware usually dictates the complexity of the software implementations of Linda. Tuple space has an affinity with the notion of shared memory, so a Linda kernel for the Encore Multimax results in a simpler design than the S/Net or the Intel iPSC described in [AHU86]. Tuple space is implemented on the Intel as a distributed hash table where different hash bins are mapped to different nodes. Efforts are underway for Linda support in hardware that may overcome the communications overhead which results in a significant bottleneck as the number of nodes scales up.

The Linda Machine improves upon the software implementations in several respects [AHU88]. Each node in its processor grid has two parts, so internode communication is offloaded from the computation part to a Linda coprocessor which also serves as tuple storage manager. Furthermore, the architecture supports the peculiar semantics of tuples themselves, while a uniform distribution scheme across broadcast busses improves communication performance.

Finally, the work at Cogent Research takes the leap from applications programming to a version of Linda designed for system-level programming as the IPC for a parallel operating system [LEL90]. Their version of Linda, called Kernel Linda, supports multiple tuple spaces (discussed in Chapter 3) and language-independent data types. QIX, the name given

to their parallel, server-based operating system, is similar to Mach, except that QIX is based on Kernel Linda.

Chapter 2

METHODS AND PROCEDURES

2.1 Background

Before proceeding to interface and design details, we explain the notion of monitors and message passing that sustain our two Linda models. Boyle et al. [BOY87] originally implemented these abstract structures in a set of tools (hereafter called the P4 package) that were successfully developed for an automated reasoning system at Argonne National Laboratories. Eventually, they found wider applicability over a variety of architectures. For a detailed description of the algorithms see [BOY87].

2.1.1 Monitors

Programming multiprocessors in which processes can communicate with one another via globally shared memory requires that shared objects must be protected against unsafe concurrent access. One approach to programming such systems involves the use of an abstract data type called a monitor to synchronize access to shared objects. Monitors coordinate efficient use of locking mechanisms to guarantee exclusive access to shared resources and protect critical sections of code at any one time. They are responsible for

suspending processes that wish to enter the monitor prematurely, and releasing processes blocked on the condition queue when the resource is free and use of the monitor relinquished.

[BOY87] describes an implementation of monitor operations in terms of the following primitives:

(1) menter(<monitor-name>) - grants exclusive control of the monitor to a process.
(2) mexit(<monitor-name>) - relinquishes exclusive control.
(3) delay(<monitor-name>,<queue>) - suspends the process executing the delay and releases control of the monitor.
(4) continue(<monitor-name>,<queue>) - the process executing continue exits the monitor and awakens one of the processes in <queue>, which continues execution at the point where it was previously delayed.

P4's *create* and *g_malloc* (a shared memory version of C's *malloc* function) provide two other necessary mechanisms - process creation and shared memory allocation.

P4 includes high-level operations built on top of the low level primitives described above. One special-purpose mechanism is called the askfor monitor. A common pattern in multiprocessing, sometimes called agenda parallelism [CAR89A], focuses on a list of tasks to be performed and is epitomized in the master-worker paradigm. A master process initializes a computation and creates worker processes capable of performing, in parallel, a step in the computation. Workers repeatedly seek a task to be performed, perform the task, and continue to seek tasks

until an exhaustion state is reached. The askfor monitor manages just such a pool of tasks and is invoked with

```
askfor(<monitor-name>,<number-of-processes>,
       <get-problem>,<task>,<reset>)
```

where *monitor-name* is the name of the monitor, *number-of-processes* is the number of processes that share the task pool, *get-problem* is a user-defined function that provides the logic required to remove a task from the pool, *task* is the actual piece of work removed from the pool, and *reset* is the logic required to reinitialize the pool. *Askfor* includes the logic required to delay and continue processes if tasks cannot be taken from the pool. A set of support functions include *probend* and *progend*. Of special interest is *progend*, which signals program termination to all active processes. The peculiar use of two such askfors in our shared-memory implementation is introduced in section 2.3.

2.1.2 Message Passing

Message passing is the most widespread method for coordination of cooperating processes. In message passing, we create parallel processes and all data structures are maintained locally. Processes do not share physical memory, but communicate by exchanging messages. Processes must send data objects from one process to another through explicit send and receive operations. For algorithms that can be formulated as such, the P4 package includes the following primitives:

```
send(<id>,<type>,<size>)
receive_any(<id>,<type>,<size>)
```

where *id* is the process identification of the intended recipient of the message (for *send*) or the process id of the sender (for *receive_any*), *type* is the message type, and *size* is the length of the message. The message type actually points to a structure in which the message is 'packetized' and must be of a consistent specified format across all nodes that use the particular message type. *Sendr* (send with rendezvous), an alternative to *send*, forces the sending process to suspend until it receives acknowledgement from the recipient.

Two procedures are used to create processes in P4. While *create* is used to create processes in the shared memory implementation, *create_procgroup* is used to develop a network of processes (a process group) that communicate via messages.

2.2 Interface

Linda operations must adhere to a strict format in our implementations. The range of valid data types for tuples include integers, one-dimensional strings, floats (doubles), and aggregates (arrays of any of the other types). A format string or mask, typical for many 'C' functions that take variable length arguments (e.g., *printf*), must be present as the first argument to any of the Linda operations and is

not to be confused with the tuple elements themselves. The value of each element is formatted according to the codes embedded in the mask. For simple actuals (actuals that are not aggregates), the mask format specification is <%Type>, where Type is *d* (integer), *f* (double), or *s* (string). For aggregates the format specification is <:Type>. The Linda operations must distinguish between actuals and formals; thus a different type separator is used for simple formals: <?Type>, where type is again *d*, *f*, or *s*. Another restriction is that the first tuple element (the logical name) must be a string or integer actual.

Out is exemplified in the following code:

```
func()
{
  int i, num, big[10];
  int size = 10;
  char buf[20],mask[20];

  num = 100;
  strcpy(buf,"anything");
  for(i=0;i<20;i++)
    big[i] = i;
  .
  strcpy(mask,"%s%s%d:d");
  out(mask,"key",buf,num,big,size);
}
```

All tuple arguments to *out* are actuals, a necessary limitation of our model. Furthermore, the tuple contains one more element than type identifiers because aggregates require an integer dimension following the array name. When the parser recognizes the aggregate type separator, it automatically pops the dimension (*size*) off the argument stack.

Given the same declarations and assignments, when executing

```
in("%s?s?d:d","key",buf,&num,big,&size)
```

the parser interprets all arguments as formals, except the key. Since all formals are addresses of C variables, ampersands are required for the integers (names for strings and arrays are the addresses for these types). Note that the first tuple argument is the only one used for matching criteria. If we execute

```
in("%s?s%d:d","key",buf,2,big,&size)
```

then the structured name ["key",2] is used as matching criteria. One may wonder why the type separator for an aggregate formal (:) is the same as its actual counterpart. In our implementation, aggregate arguments to *rd* and *in* are restricted to formals and no distinguishing specifier is necessary.

Eval takes two arguments - a key and a pointer to a function. Any arguments to the function are passed via *out* and retrieved with *in*. A discussion of the constraints on our implementation of this operator is deferred until section 2.3.

A Linda program is not complete without requisite initialization and termination routines. *Mon_linda_init* intializes the monitors, creates the process pool, and sets up the environment. It take three arguments: PROCS, argc and argv. PROCS is a user-defined constant in *mon_linda.h*

and should be set to an optimum number of processes. One of the initialization procedures uses PROCS to create the process resource for *eval*. The termination routine *mon_linda_end* flushes the monitors and facilitates graceful termination of processes.

Initialization and termination routines for the message passing model are, respectively, *sr_linda_init* and *sr_linda_end*. The number of processes is not required as a parameter to the initialization function because it is defined separately with *create_procgroup*, which reads a "process group" file that contains the following fields:

1. the name of a remote machine on which slave processes are to be created.
2. the number of slaves that are to be created and share memory on the remote machine (since we make no use of the cluster model, this field defaults to 1.
3. the full path name of the slave program.

Each model requires a header file that declares the structures common to all processes. Both *sr_linda.h* and *mon_linda.h* allow for modification of the constant HANGER_SIZE, which defines the size of a string buffer used to store simple formals and actuals. The default size is 100 bytes, but the maximum size of a tuple is actually program dependent. If a tuple includes a large number of non-aggregate members or very large strings, this constant requires modification. Aggregates are dynamically allocated

in the monitor's model, but in the message passing model they are defined with a fixed maximum size. The definition of AGG_SIZE in *sr_linda.h* should scale with the expected aggregate size (the default is 300 bytes). If no aggregates are used, the programmer should set AGG_SIZE to one, minimizing communication overhead.

2.3 Basic design for the shared-memory implementation.

Tuples are stored in shared memory as self-contained data structures. The representation of tuples includes not only data, but also typing information required for matching and retrieving the tuple. The first element of the tuple structure, called the hanger, contains the data - formals or actuals that constitute the tuple. The tuple mask is the second element and contains the typing information required to process the tuple.

Given the type mask "%s%d:d", and the statement

```
out("%s%d:d","key",10,array,5)
```

where *array* points to some local array of length 5 with elements (1..5), Figure 1 shows what the four element tuple looks like when stored in shared memory. Note that all elements are actuals, a necessary restriction placed on *out* in our implementation. Actuals that are integers, floats, or simple strings are copied into the hanger. For actuals that are aggregates, a global copy is made and a pointer to the

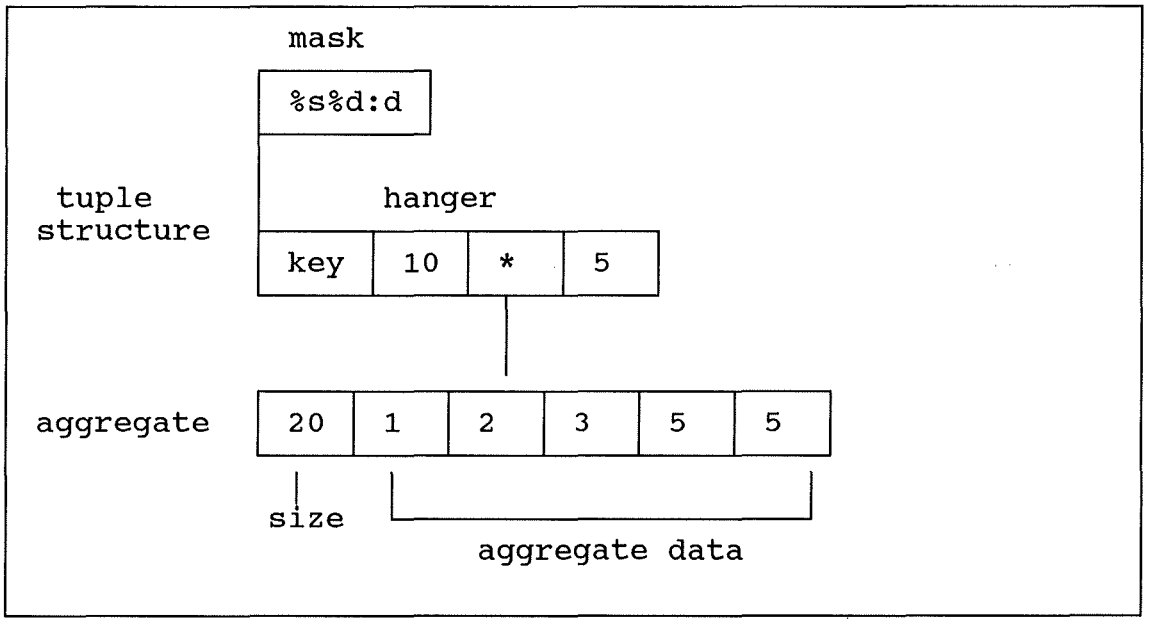

Figure 1: A tuple in shared memory.

copy is stored in the tuple hanger. The tuple structure is hashed into any one of 256 linked lists. These hash lists, in their entirety, are at any time the physical embodiment of tuple space.

The four basic Linda operations are implemented as functions in the shared-memory model. A single monitor protects two resources: a queue of unevaluated functions and the linked list representation of tuple space. Two asfors control respective access to tuple space and process-to-task initiated by *eval*.

Out is relatively easy to process. A statement of the form out(mask,arg1,arg2,...argN) invokes a function which examines each argument for its type based on the relative position in mask. The mask informs the function how to build

the hanger. All that remains is to claim access to the monitor with *menter*, link the tuple structure to the appropriate hash list, and relinquish the monitor with *continue*. *Continue* is preferred over *mexit* because it releases a suspended process from the monitor's delay queue. The activated process is now free to reexamine tuple space for a matching tuple.

In and *rd* are more complicated because a process must suspend if no tuple matching occurs. A statement of the form in(mask, arg1, arg2 .. argN), where the arguments are a collection of actuals and formals, invokes a function that constructs a local template based on typing information in mask. The process must now gain exclusive access to the tuple space monitor to search for a matching tuple. Neither *menter* nor *mexit* will help us here because we need some means to obtain a task (a matched tuple) from a task pool (a linked list of tuple structures), but block if none is found. The askfor monitor provides the answer. Remember that one of the parameters to askfor is <get-problem>, a pointer to a routine whose purpose is to return a task from a pool of work. In our case that routine includes the following logic:

(1) search the appropriate hash list for a matching tuple.
(2) if a match is found, delete the tuple structure from the hash list and return success to *askfor*.
(3) if no match is found, return failure to *askfor*.

Two characteristics of *askfor* are crucial to the Linda operations. If a match is found, the matched tuple is returned in <task>, another of the parameters to *askfor*. If no match is found, the askfor monitor automatically delays the process on a monitor queue. *Rd* initiates a similar process, except that the tuple structure is not deleted from the hash list.

Eval's basic design is best explained by example. Suppose we have defined the a function to compute the number of primes within the range 2 to *x*. If *primes* is a pointer to a function, eval("some_tag",primes) spawns a process that calls the function. Arguments to the function are passed via tuple space - the process executing the *eval* adds the arguments to tuple space; the process allocated by *eval* removes the arguments from tuple space. The example is coded in our system as

```
main()
{
  int primes();
  .
  /* masks are omitted for
     convenience */
  out("prime_arg",3);
  eval("some_tag",primes);
  /* collect primes */

 }

primes()
{
  int i,result;

  in("prime_arg",i)
  /* compute the result */
  out("some_tag",result);
}
```

With these restrictions in mind, the design of *eval* only has to consider allocating processes to unevaluated functions. A separate askfor is used to this end. *Eval* is basically a three step operation: enter the evaluation monitor, add the function name to the pool of tasks (a linked list of pointers to functions), and exit the monitor. Note that we have slightly altered the traditional semantics of *eval*. Heeding the caveat, process creation is not cheap, we decided to create *n* processes up front where *n* is the user-provided argument (PROCS) to *mon_linda_init*. This permits us to "reuse" processes rather than repeatedly create them. The initialization function uses P4's *create*, which spawns a new process that begins execution at a procedure with a twofold purpose: invoke an *askfor* that manages the assignment of unevaluated functions to available processes, and then invoke the function retrieved from the pool. The <get-problem> parameter to *askfor* pops the function off the list and returns the pointer to the function in <task>. If there are no functions on the list, the process delays. Processes continue to attack the pool of functions until the main procedure invokes *progend*, signalling an exhaustion condition.

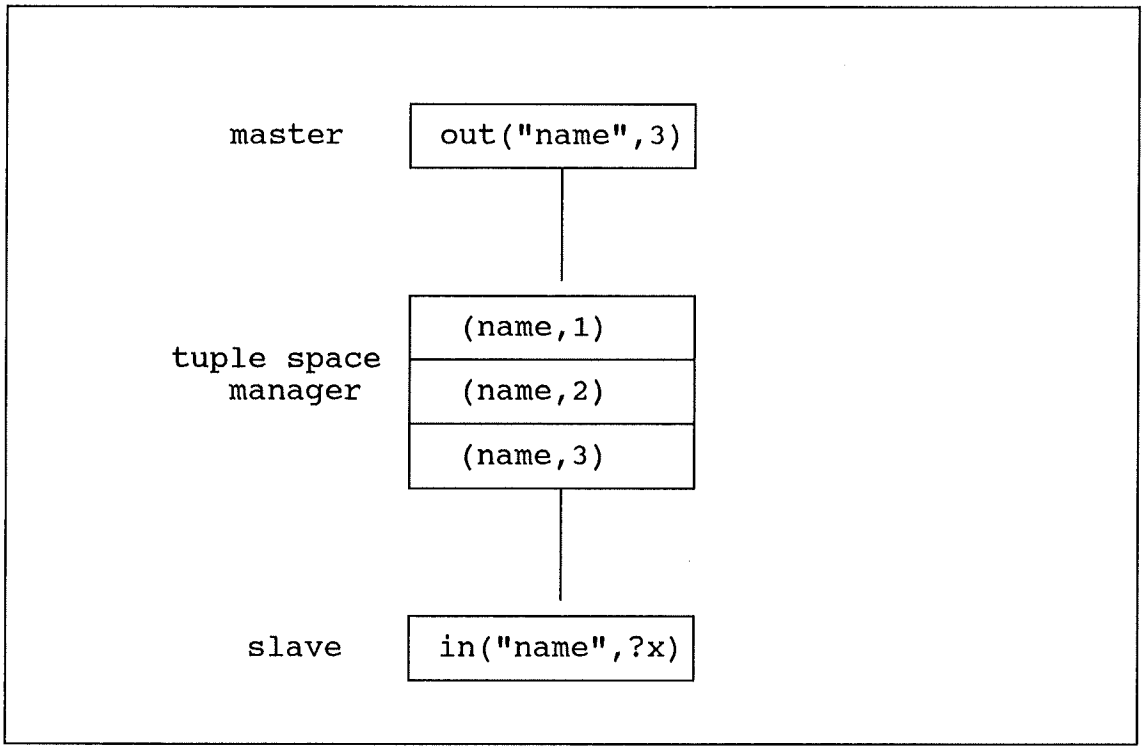

Figure 2: Communication mediated through the tuple space manager.

2.4 Basic Design for the Message-Passing Implementation

A Linda model based on message passing requires a minimum of three processes: a master process to initialize the environment, at least one slave process to assist in doing work, and a process to act as tuple space manager. All communication between the master process and the slaves is mediated through Linda operations and tuple storage handled by the manager. Figure 2 depicts a master process communicating a tuple to a slave process through the tuple space manager.

Tuples are stored as structures in the local memory of the tuple space manager. A tuple structure includes the following elements: a mask contains the typing information; the hanger contains the data corresponding to simple data types; a type identifier indicates whether a request is IN, RD, or OUT; size identifiers store the tuple and aggregate lengths; and a separate area stores aggregate data. Note that all data, including aggregates, are copied into the tuple structure's data areas - pointer storage is meaningless in disjoint memory space. Once again, a tuple structure is hashed into any one of 256 linked lists. A similar structure, which we call the tuple channel, serves as the primary message type through which processes communicate tuple information to the tuple manager.

The initial steps of *in* and *rd* require argument examination and template construction. The tuple channel is used to send the template to the tuple space manager and to receive the actual tuple from tuple space. The two statements

```
send(manager_id,tuple_channel,size)
receive_any(id,tuple_channel,size)
```

not only communicate a matched tuple to the process executing the *in* or *rd*, but suspend the process until a match is found. A process retains a copy of the template, and defers the assignment of actuals to formals until receiving a matched tuple. Send was preferred to sendr because the dialogue between a Linda process and the manager

uses self-synchronizing pairs - a *send* is immediately followed by a *receive_any* in any process executing *rd* or *in*.

Out examines the argument list, populates the tuple channel and uses *send* to communicate the information to the tuple manager. *Sendr* is unnecessary because the sender does not require prior knowledge of the process that will ultimately *in* or *rd* the tuple. This is in the spirit of communication orthogonality, in which the tuple manager plays the role of mediator.

The tuple manager takes the place of the monitor in the message passing implementation. It's sole job is to receive a request on tuple space, process the request dependent on the tuple type, and iterate. If the tuple type is RD or IN, the manager searches the appropriate hash list. If a match is found, data is packed into the tuple channel and returned to the suspended process. When no match is found the identity of the requester, the tuple type and the template are linked to a wait queue. Upon receipt of a tuple of type OUT, the manager first searches the wait queue, satisfying all pending requests (there may be several *rd*'s waiting on the same tuple) until the first matched *in* is encountered or the search is exhausted. If no *in* is encountered, the information in the tuple channel is copied into a tuple space structure and linked to the appropriate hash list.

The manager serves requests until it receives a special tuple of type END which signals termination.

2.5 Design Considerations

We wanted to design a Linda model, not a complete Linda kernel; hence, the fundamental decision to code the Linda operations as functions. We were further justified by the fact that much of what is standard in 'C' (i.e. the library of I/O functions) are procedures built on top of a minimal set of instructions and we simply viewed the linda primitives as an extension of this standard. This decision resulted in certain limitations on *eval* and *out*.
A Linda kernel cited in [CAR89B] allows eval tuples to have more than two elements. For example, a typical *eval* may appear as

```
eval("key",i,primes(i))
```

which spawns a process to compute whether or not *i* is prime. After the tuple is evaluated, the tuple ("key",i,<result>) is added to tuple space. In our implementation it is impossible to defer the evaluation of *primes*(i) - the function will return a value prior to process creation. Instead we use

```
out("another_key",i)
eval("key",primes)
```

where primes is a pointer to a function and a separate tuple maps arguments to the function via tuple space. A declaration of such a function is superfluous and is treated

simply as an integer type, the default in most C implementations.

With this in mind, we considered two possible implementations for *eval* in the monitors model. One method dynamically creates processes as needed: eval("key",func) invokes *create*(func). Although successful, indeterminate calls to *eval* result in costly process creation overhead. Instead, we decided on the process queue method discussed above.

In both models, the arguments to *out* are restricted to formals. Some Linda kernels allow for inverse structured naming, in which formals are permitted as elements in tuple space. Although the monitors model can be enhanced to include a restricted form of inverse naming (the formals would have to be shared variables), without special locators or distributed pointers this is all but impossible to implement in a loosely coupled world.

Another fundamental decision affected the implementation of tuple space in the message passing model. We opted for a single tuple manager verses a distributed or replicated tuple space because the latter methods require building fast deletion and broadcast protocols, an effort beyond the scope of the project. For an interesting discussion of these schemes see [CAR86A].

2.6 Detailed Design

Both Linda implementations are coded in C. In the detailed discussion, key C functions are italicized and explained in the follow-up discussions. Variables are capitalized for emphasis. Although we begin with the monitors implementation, we preserve a common syntax where similar algorithms carry over to the message passing implementation. Readers not interested in detailed design considerations may wish to skip the remainder of Chapter 2.

2.6.1 Detailed Design in the Monitor Based Implementation

In, *out*, and *rd* initially parse varying length list of tuple elements through a call to

Parse(Tuple_mask,Type,Buffer,Tuple_list)

where Tuple_mask is the string of type specifiers; Type is a constant indicating whether the calling function is *in*, *out* or *rd*; Buffer is a string buffer that will contain the resultant template (if type is IN or RD) or hanger (if type is OUT); and Tuple_list is the argument stack. A parse of the mask yields the type separator and the data type for each argument on Tuple_list. To parse an integer actual we use

```
if(Mask_ptr == '%') {
  if(Mask_ptr[1] == 'd' {
    Integer = va_arg(Tuple_list,int);
    sprintf(Token,"%d ",Integer);
    strcat(Buffer,Token);
  }
```

ANSI C's *va_arg* (and related functions) allows one to iteratively access the elements of varying length argument lists, given knowledge of the data type for each element. C's *sprintf* formats and writes the values of C variables to a string token before it is concatenated to the data buffer. *In* and *rd* require storing address pointers in Buffer for later actual-to-formal assignment, thus

```
if(Mask_ptr == '?') {
  if(Mask_ptr[1] == 'd' {
     Int_ptr = Va_arg(Tuple_list,int *);
     sprintf(Token,"%d ",Int_ptr);
     strcat(Buffer,Token);
  }
```

pops the address of an integer off the argument stack and appends it to the buffer. Addresses of all types are formatted as integers, but are properly recast during instantiation. The only remaining problem is to process aggregates. To place an integer array into tuple space we use

```
if(Mask_ptr == ':') {
  if(Type == OUT) {
     if(Mask_ptr[1] == 'd' {
       Int_ptr = va_arg(Tuple_list,int *);
       Size = va_arg(Tuple_list,int);
       Aggreg_ptr = (struct aggregate *)
                 g_malloc((sizeof(Int_ptr)
                 * size + sizeof(int));
       bcopy(Int_ptr,&(Aggreg_ptr->data),
             (sizeof(Int_ptr) * size));
       Aggreg_ptr->size = size;
       sprintf(Token,"%d ",Aggreg_ptr);
       strcat(Buffer,Token);
     }
  }
}
```

Two elements are popped off the argument stack: a pointer to the array, followed by the number of elements in the array.

G_malloc allocates shared memory for the aggregate structure, while *bcopy* copies from one memory buffer (the array) to another (the aggregate), an efficient means to build the data portion of the aggregate structure. If the operator type is IN or RD, and Mask_ptr points to 'd' (the aggregate formal is an integer array variable), then a series of statements of the form

```
Int_ptr = va_arg(Tuple_list,int *);
Global.size_ptr = va_arg(Tuple_list,int *);
sprintf(Token,"%d ",Int_ptr);
strcat(Buffer,Token);
```

places the address of the array formal into the data buffer. Restricting Linda operations to only one aggregate formal permits us to place the address of the expected array size into a global structure.

With *parse* defined, the code for *out* is straight forward. Although a call to *out* is made with a variable number of parameters, the function takes the first parameter as its only argument. *Va_start* sets a pointer to the top of a stack containing the remaining arguments:

```
out(Tuple_mask)
.
va_start(Tuple_mask,Tuple_list);
parse(Tuple_mask,Type,Hanger,Tuple_list)
stok(Hanger,Key)
Hashnum = hash(Key)
menter(&((Glob->TS).m);
[allocate space for space_node]
strcpy(Space_node->hanger,Hanger);
strcpy(space_node->mask,tuple_mask);
[link Space_node to tail of linked list of
Space_nodes based on Hashnum]
continue(&((glob->TS),m),0);
```

The buffer constructed in *parse* is passed to *out* through Hanger. *Stok* take two arguments: a source (Hanger) and a target token (Key). *Stok* picks off the first space-delimited token from the source string and copies it into the target string. A hashing algorithm suggested by [PEA90] efficiently maps variable length text strings onto small integers. The spread of integers is uniform, and experiments with the function rarely yield collisions. *Menter* and *continue* takes as arguments the address of the monitor declared in *mon_linda.h*. The monitor is continued and not strictly exited so that a process blocked on an *in* or *rd* is released from the delay queue before the process executing *out* exits the monitor. Since all processes share the data stored in tuple space, allocation for a space node uses *g_malloc* instead of *malloc*.

In and *rd* search the list of tuple structures before matching actuals to formals. The algorithm is as follows:

```
in(tuple_mask)
.
va_start(Tuple_mask,Tuple_list);
Type = IN;
parse(Tuple_mask,Type,Template,Tuple_list);
strcpy(Global.template,Template);
Global.type = Type;
strcpy(Global.mask,Tuple_mask);
Rc = askfor((&(Glob->TS),Glob->procs,t_match,Hanger);
instantiate(Tuple_mask,Template,Hanger);
```

T_match is invoked from within *askfor*, and passes the matched data to Hanger from the linked list of Space_nodes. Before invoking *askfor*, Template, Type and Tuple_mask are copied into global storage because any procedure that gets a

problem from the pool (in this case, *t_match*) is restricted to only one argument - the address through which a task is passed to the function executing *askfor*. The algorithm for *t_match* follows:

```
t_match(Hanger)
  .
  found = FALSE;
  Rc = 1;
  stok(Key,Global.template)
  Hashnum = hash(Key)
  Space_node = Tuple_space[Hashnum]
  while (!found and Space_node != NULL) {
    if match(Space_node->hanger,Global.template,
                Global.mask)
        found = TRUE;
    else [get next Space_node in list]
  }
  if(found) {
     strcpy(Hanger,Space_node->hanger);
     if(Type == IN)
        [deallocate space_node]
     Rc = 0;
  } return(RC);
```

Match (not shown) returns TRUE if the node hanger matches the relative actuals embedded in the template. If the search is exhausted before a match is found, *askfor* suspends the process on a monitor queue and returns a -1 in rc. If the search succeeds, *t_match* removes the affected structure from the linked list and frees its memory. Instantiation reverses the parse and match stages. Whereas *t_match* compares actuals (the structured name) in a template to actuals in the candidate hanger, *instantiate* ignores the structured name and focuses on formals. Instantiation of an integer proceeds as follows:

```
instantiate(Tuple_mask,Template,Hanger)
{
.
stok(Template_tok,Template);
stok(Hanger_tok,Hanger);
if(Mask_ptr == '?') {
   if(Mask_ptr == 'd') {
        sscanf(Template_tok,'%d',
                &Ptr);
        sscanf(Hanger_tok,"%d",Integer);

        Int_ptr = (int *) Ptr;
        *Int_ptr = Integer;
    }
 }
```

Instantiate does actual-to-formal assignment. During instantiation *sscanf* reverses *sprintf*. It reads characters from the template, then converts and stores them in C variables according to the specified format in Tuple_mask. In the case of formals, *sscanf* yields an address of a particular type, and the actual (Hanger_tok) is assigned to that address. Since any address is buffered as an integer, it is recast to the necessary type prior to assignment. In the case of formals that reference aggregates, *bcopy* is used to copy the data to a local address referenced in the template, as shown here:

```
if(Mask_pointer == ':') {
   if(Mask_pointer is 'd') {
     sscanf(Template_tok,"%d",&Ptr);
     Int_ptr = (int *) Ptr;
     sscanf(Hanger_tok,"%d",&Ptr);
     Aggr_node = (struct aggregate *) Ptr;
     bcopy(&(Aggr_node->data),Int_ptr,
           sizeof(int) * Aggr_node->size);
     *Global.size_ptr = Aggr_node->size;
   }
}
```

C's *sizeof* returns the number of bytes for a given type, which is factored against the aggregate size to determine

the exact number of bytes to be copied. Two assignments are made for every aggregate instantiation - the data and the number of elements in the aggregate. We saved the address of the target array size in a global structure and the last statement assigns the actual size to this address.

Finally, *eval* is implemented using a second askfor monitor. Initially, *eval* simply stores a pointer to an integer function in a string buffer. Remember from the discussion above that one of the parameters to *eval* is a pointer to the function to be evaluated. That pointer is linked to a list of Eval_nodes accessible to processes spawned during initialization:

```
eval(Tuple_mask)
.
Eval_node = alloc_eval_node();
strcpy(Tuple_mask,"%s%d");
/* get tuple name and function ptr */
Key = va_arg(Tuple_list, char *);
sprintf(Buffer,"%s ",Key);
strcat(Eval_node->work,Buffer);
Ptr_to_IntFunction = va_arg(Tuple_list,int *);
sprintf(Token,"%d ",Ptr_to_IntFunction);
strcat(Eval_node->work,Token);
strcpy(Eval_node->mask,Tuple_mask);
[enter the monitor
 link Eval_node to pool of work
 continue the monitor]
```

During initialization, *create* receives one argument that is a pointer to a function and creates a new process that executes the indicated function (*work*, described below). Visualize any new process as hovering around an askfor monitor in an attempt to retrieve an Eval_node from the task pool:

```
work()
.
Rc = askfor(&(Glob->TS,Num_procs,getfunc,Func)

while((Rc == 0) || (Rc != -1))
{
   if(Rc == 0) {
      Eval_node = (struct work_struct) Func;
      if(Eval_node->mask[3] == 'd') {
       sscanf(Eval_node->work,"%s%d",Key,
            Ptr_to_IntFunction)
       (*Ptr_to_IntFunction)();

Rc = askfor(&(Glob->TS,Glob->procs,getfunc,Func)
}
```

Getfunc, and hence *askfor*, return success if an Eval_node is successfully removed from the task pool. If a process enters the monitor and finds no tasks (the list of Eval_nodes in the pool queue is empty), *getfunc* returns a 1 in Rc, and the process is put on a delay queue. The function *progend* (not shown) signals processes delayed in the askfor monitor that the program has ended and they exit the monitor with RC set to -1. The function buffered in Eval_node is called without any arguments, as it is incumbent upon the programmer to *out* the function arguments to tuple space prior to invoking *eval*.

2.6.2 Detailed Design for the Message Passing Implementation

In the message passing Linda model, the algorithms for *out*, *in*, and *rd* are similar to those in the monitors implementation. Unlike the previous model, communication with the tuple space is accomplished through send and receive operations and tuple space is a local memory manager

of these operations. All operations access the logical communication channel. A local process feeds the channel as these essential statements for *out* show:

```
out(Tuple_mask)
.
va_start(Tuple_list,Tuple_mask)
parse(Tuple_mask,Type,Hanger,Tuple_list)
strcpy(Tuple_channel.hanger,Hanger);
strcpy(Tuple_channel.mask,Tuple_mask);
Tuple_channel.type = OUT;
[calculate Tuple_size]
send(Manager_Id,Tuple_channel,Tuple_size)
```

Parse differs from its relative in the monitors model only in how aggregates are handled. In the monitors model, *parse* dynamically allocates separate structures for aggregates, and only stores the address in a hanger. In the message-passing model, the data and size for an aggregate are part of the tuple channel, and parser *bcopy*'s directly into the channel structure. *Instantiate* also differs from its relative in the monitors model - it takes one less argument because the data used for actual-to-formal assignments are accessed via the channel structure, as these statements for *in* show:

```
in(Tuple_mask)
.
va_start(Tuple_list,Tuple_mask)
parse(Tuple_mask,Type,Template,Tuple_list)
strcpy(Tuple_channel.hanger,Template);
strcpy(Tuple_channel.mask,Tuple_mask);
Tuple_channel.type = IN;
[calculate Tuple_size]
send(Manager_Id,Tuple_structure,Tuple_size);
receive(Manager_Id,Tuple_channel,Tuple_size);
/* Tuple_channel.hanger now has actuals
   for instantiation */
instantiate(Tuple_mask,Template);
```

The process immediately blocks after a send until receive is satisfied. *In* and *rd* are identical on the master and slave processes, differing only in how the tuple manager processes them. The main module for the tuple manager includes

```
receive(Proc_id,Tuple_channel,Tuple_size)
while(Tuple_channel.type != END) {
  if(Tuple_channel.type == IN ||
     Tuple_channel.type == RD)
     serve_in_or_rd(Proc_id,Tuple_channel.type,
                    Tuple_size);
  else if(Tuple_channel.type == OUT)
       if(!(check_wait(Proc_id,Tuple_size)))
            serve_out(Proc_id,Tuple_size);
  receive(Proc_id,Tuple_channel,Tuple_size)
```

The manager receives and processes tuples until *sr_linda_end* transmits a terminal tuple channel with tuple type set to END. We present the algorithm for serve_out:

```
serve_out(Proc_id,Tuple_size)
.
stok(Key,Tuple_channel.hanger)
Hashnum = hash(key)
[allocate space for a Space_node]
[copy all elements of Tuple_structure into
 Space_node]
[link Space_node to the tail of the Space_queue]
```

Serve_out uses the same hashing algorithm as that found in the monitors model. As will be shown below, an *out*'d tuple is not always hashed directly into a tuple space list. If there are pending *in*'s or *rd*'s, *check_queue* processes newly arriving tuples. But first, we present the algorithm for *serve_in_or_rd*:

```
serve_in_or_rd(Proc_id,Type,Tuple_size)

stok(Key,Tuple_channel.hanger);
hashnum = hash(Key);
Space_node = Tuple_space[hashnum];
found = FALSE;
while(!found && Space_node != NULL)
  if(match(tuple_channel.hanger,Space_node->hanger,
```

```
                tuple_channel.mask))
              found = TRUE;
        else [get next Space_node in list]

    if(found) {
        if(Tuple_channel.type == IN)
           [delete Space_node from list]
        [copy elements of Space_node into Tuple_channel]
        send(Proc_id,Tuple_channel,Space_node.tuple_size)
        free(Space_node)
    }
    else { /* put on a wait queue */
        [allocate space for a Wait_node]
        [Copy elements of Tuple_channel into Wait_node]
        Wait_node->id = Proc_id
        [Link the Wait_node to the tail of the
             Wait_queue]
```

If a match is found, the request is satisfied and the manager sends the entire tuple to the suspended process, identified by Proc_id. A null condition on a hash list signals the manager to queue the process to a linked list of Wait_nodes. A wait node contains the process id of the waiting process in addition to the tuple type, mask and template. If the tuple manager receives a structure of type OUT, it first searches the wait queue for any pending *in*'s or *rd*'s. Thus,

```
check_wait(Proc_id,Tuple_size)
.
Found_in = FALSE
Found = FALSE
While (wait_node != NULL and !Found_in){
  if(match(Tuple_channel.hanger,
          Wait_node->template,Wait_node->mask)
      Found = TRUE
  else [get next Wait_node in wait queue]
}
if(Found) {
   send(Wait_node->id,Tuple_channel,Tuple_size);
   [remove Wait_node from wait queue]
   if(Wait_node->type == IN)
        Found_in = TRUE
```

```
        else [get next Wait_node in wait queue]
           [deallocate Wait_node]
        }
        return(Found_in)
```

The standard matching algorithm is used to compare templates to hangers. It is important to note that if the manager matches a template of type IN, the search ends and the tuple is never added to tuple space. The manager adds the tuple to tuple space if only RD's are matched, or the search ends without any match.

2.7 Demonstration and Applicability

Carriero explores many conceptual classes of parallel programs and advances each with variations on finding all the primes within a specified range[CAR89A]. Testing our implementations on these programs proved applicability over several categories of parallelism and at the same time verified the code. Significant interaction among processes justifies primes-finding as a test case, but the interaction is fairly constant throughout execution time. In contrast, a settling property, in which process interaction decreases relative to time, characterizes a semigroups problem and makes it an excellent candidate for the message passing implementation where communication overhead is a often a critical factor. Appendix C includes the source code for the test cases.

2.7.1 Primes Finder I.

The first test case, run in the Sequent's shared memory environment using the monitors model, is an example of result parallelism using a live data structure method. A result vector is defined and each process is assigned to compute one element of the vector. Furthermore, it uses a method known as live data structures in which each element of the resulting data structure is an active process that yields the element upon termination. If ("primes",n,ok) is one element of the distributed result vector, where *n* is the index into the vector and *ok* is 1 if *n* is prime, then the couplet

```
eval("%s%d","primes",prime)
out("%s%d","primearg",n)
```

implicitly creates a process to compute the nth element of the result vector, adding the tuple ("result",n,ok) before termination. As explained above, our implementation deviates from the ideal - first, the programmer must explicitly *out* the evaluated tuple before exiting prime; secondly, if there are 100 elements to resolve, we do not create 100 processes; instead a fixed number of processes are reused as needed from the process queue.

With slight modification the program succeeds under the message passing model in the absence of *eval*. First, *prime* is replicated across *n* nodes, where *n* is the process group

size. After the master collects all of the primes, it *outs* *n* special tuples to signal termination.

While this exercise is natural, simple and proves the correctness of the manager and process monitors, it is nevertheless highly inefficient: the process creation overhead combined with small granularity obviates speedup expected from parallelizing in the first place. Carriero offers a large grain approach that improves speed at the expense of simplicity.

2.7.2 Primes Finder II.

In the first primes finder a vector was actually constructed in tuple space. Tuple space acted like shared memory and, in fact, the program works just as well in the absence of true shared memory, but just as inefficiently. An alternative is to use agenda parallelism in which workers focus on a list of tasks to be performed. The master assigns the following task to a worker: find all of the primes within a specific range where the block size is a programmer-defined constant. The master process constructs the distributed global table where all primes are stored. Slaves store in local tables only those primes required to construct a new block. The master *in*'s completed blocks and expands the resultant primes table. A full explanation can

	Primes Finder I	Primes Finder II
Number of Processes		Grain = 2000 Limit = 300000
1	85200	11000
3	15500	3850
4	13554	3050
5	12300	2725
8	15080	2600

Figure 3: Time vs Processes for Primes Finding

be found in [CAR89A]. Run under monitor control, this version showed significant speedups over the live data structure method (figure 3) while also validating the storage and retrieval of aggregates. Speedup was also evident when tested under the message passing model, although the size of the message channel for the tuple structure was a limiting factor in grain size.

2.7.3 Primes Finder III

In many parallel programs the concurrent processes perform the same task, a pattern we call function homogeneity (note that this is not the same as the instruction homogeneity exhibited by SIMD machines). Many programs require a heterogenous mix of functions to be executed in parallel. Thus, our final primes case proves interesting if only because more than one type of function is *eval*'d to do the work, a programming method Carriero calls specialist

parallelism [CAR89A]. Based on the sieve of Eratosthenes algorithm, the program starts off with two pipes: a source that generates integers; and a sink that removes multiples of the last known prime. As the sink discovers a new greatest primes, it *eval*'s a function (*pipe_seg*) that sieves multiples of this prime. Again, for an in depth discussion of the algorithm, see [CAR89A]. Run under the monitors model, three functions are evaluated, proving the robustness of the function queue and its overseer, the evaluation monitor.

This case raises the following question: in the absence of *eval*, how does one achieve function heterogeneity? One solution is to partition network nodes among the functions. In the pipes example, delegate one node as the master, another to *source*, and the remainder to *sink* and *pipeseq*. Evaluation is now inherent in the architecture, and nodes communicate as usual through the medium of tuple space using the fundamental Linda operators.

Without dynamic pointers or locators our only other alternative is to replicate all functions across all nodes and simulate *eval* with tuples. In our pipes example the entry point for all slave nodes begin with a function filter.

```
while(1) {
  in("eval",type);
  if(!strcmp(type,"source"))
    source();
  else if(!strcmp(type,"sink"))
```

```
    sink();
  else if(!strcmp(type,"pipeseg"))
    pipeseg();
  else break; /* type = end_token */
 }
```

When *sink* detects the final prime, it outs a termination token to all slaves, including itself.

As a test case for exercising *eval*, primes finder III proved invaluable. As an efficient parallel program, it ranks unfavorably when compared with the agenda program, though not as inefficient as the 'live' data structure example. The methodology applied to a coarse-grained problem may prove advantageous.

2.7.4 A Semigroups Problem

There exists a class of programs in which communication costs decrease as execution time increases. The semigroups problem falls into this category, and thus is a very good candidate for Linda's message passing implementation. A short discussion of an algorithm suggested by [BUT88] follows the problem description.

The program is given as input a set of words and an operation table that defines how to build new words from existing ones. The object is to build a unique set of words by applying the operation table to the original set and any newly derived words. The set of all possible words is

usually very large when compared with the solution set. For example, if there are six unique values for a character in a word, and a 6x6 operation table defining the product of a character pair, for a 36 element word one can derive 6 to the 36th words. Eliminating duplicates yields a solution set of only 224 words.

A Linda parallel solution to the problem requires a master and any number of slaves. For efficiency, all slaves are required to build local copies of the word list and no two slaves can receive the same piece of work, represented by an index into the local word list; thus, it is incumbent upon the master to communicate new words to slaves via tuple space. To meet this requirement, new-word tuples are indexed by slave. Initially the master must communicate unique id's to each slave by placing into tuple space *n* tuples of the form ("id",i) where *n* is the number of slaves and *i* is some arbitrary integer. After the master places the operation table and initial word list into tuple space, it *in*'s tuples of the form

("master",&type,&id,word);

where *type* takes the value Candidate (a slave found a word he thinks is new) or Work_request (a slave needs an operand from which to generate new words). If the master *in*'s a candidate that is indeed a new word, it adds the word to the master list and *outs* the tuple

(id,type,word,idx)

	TIME	
Number of Processes	Word size = 25	Word size = 36
1	1250	11000
3	660	4400
4	575	3430
5	600	3330
8	1400	5800

Figure 4: Time vs Processes for Semigroups Problem

where *type* is New_word, *id* is the unique id of the target slave, and *idx* is an indication of where *word* is to be placed in the local list.

Slave processes *in* tuples of the form

(id,&type,word,&idx)

where *type* contains one of two flags: New_word, which informs the slave to add *word* to its local list; or Work, which prompts the slave to generate new words from the word pointed to by *idx*. If a derived word exists locally, it is discarded.

If a derived word is not in the local list, the slave outs the tuple

("master",type,id,word)

where *type* is Candidate. The master now searches the primary list for the word. If the master discovers the word is truly

new, he adds it to the primary list and outs *n* copies into tuple space, where *n* is the number of slaves.

Communication costs are substantially curtailed by maintaining a master list and several local lists. If each slaves list is a subset of the master list, a slave can eliminate as many duplicates a possible on a local level, rather than communicate all generated tuples to the master. For a complete discussion of the semigroups algorithm, see [But88].

Results on 36-element words are recorded in figure 4 for 1 and 3 processes. The results are promising for loosely coupled processors because, as execution time increases, generated words are more likely found in local lists, and only request type tuples are communicated through tuple space.

Chapter 3

RECOMMENDATIONS AND CONCLUSIONS

3.1 Recommendations for Future Enhancement

The Linda implementations provide the minimal set of Linda operations in *out*, *in* and *rd*. Boolean versions of these primitives can perform existence tests on tuples in tuple space. *Inp* and *rdp* would attempt to locate a matching tuple and return 0 if they fail; otherwise they return a 1 and perform the usual matching of actuals to formals that are found in a normal *in* or *rd*. Constructing these predicate versions on top of *in* and *rd* requires minimal modification to the existing code.

Our hashing scheme works best when tuples are restricted to a single unique key. Once such a key is identified in tuple space, the tuple will match any template with the same key. If the hash distribution is good, this translates into a match with the first tuple in the hash list. Unfortunately, not all tuples fall into this category. In problems where the matching criteria include two tuple elements (the logical name and one or more additional actuals) hashing on a combination of these elements should result in a faster search for a matching tuple. Our hashing method is less than optimum for tuple patterns like these, and we therefore

recommend experimentation with concatenated index schemes to alleviate potential search bottlenecks.

Finally, there is the issue of multiple tuple spaces. Suppose we wished to add two matrices "A" and "B". To inform matrix "A" of its row index and data we write

```
out("A",index,data).
```

The logical "A" identifies a specific vector, while *index* points to a specific element of the vector. An element is retrieved by matching on the first two tuple members:

```
rd("A",index,&data).
```

The amount of searching can be reduced if we placed vector "A" in its own tuple space, thus eliminating the need for combined keys. In the message passing model, this translates into multiple tuple managers. A distributed askfor, or use of several monitors, may provide the answer to multiple tuple spaces in the monitors model. A Linda kernel described in [LEL90] implements multiple tuple spaces.

3.2 Evaluation and Conclusion

Facilities such as interprocess communication and protection of shared resources were added to operating systems to support multiprogramming and have since been adapted to exploit explicit multiprocessing within the scope of two models - the shared-memory model and the distributed (message-passing) model. Application programmers working

within a traditional multiprogramming environment are typically shielded from the details of the underlying mechanisms because multiple processes are rarely used in a single program. In contrast, when multiprocessors are used for explicit parallelism, the difference between the models is exposed to the programmer (LEL90]. The P4 tool set was originally developed to buffer the programmer from painful synchronization problems while offering an added advantage in portability. Nevertheless, two dialects are still needed to communicate parallel algorithms. Our attempt to build a single high-level programming model on top of the existing paradigms in the hope that the same semantics can be used regardless of the underlying model was successful with the exception of the eval operation. While the three primary Linda operators remain semantically consistent, the eval operator remains non-portable between the shared memory and message passing implementations. More importantly, the fundamental properties associated with generative communication remain intact, and the distinction between shared and disjoint memory is blurred in the light of this fourth model - that of tuple space synchronization.

REFERENCES

[AHU88]
Ahuja, S., et al. "Matching Language and Hardware for Parallel Computation in the Linda Machine", IEEE Trans. Computers 37,8 (Aug. 1988), pp. 921-929.

[AHU86]
Ahuja, S., Carriero, N. and Gelernter, D. "Linda and Friends", IEEE Computer 19,8 (Aug. 1986), pp 26-34.

[BOY87]
Boyle, J., et al. Portable Programs for Parallel Processors, Holt, Rinehart and Winston, Inc., New York NY, 1987, 272 pages.

[BUT88]
Butler, R., and Karonis, N. "Exploitation of Parallelism in Prototypical Deduction Problems". Ninth International Conference on Automated Deduction, 1988, pp. 333-343.

[CAR88]
Carriero, N., and Gelernter, D. "Applications experience with Linda". In Proceedings of the ACM Symposium on Parallel Programming, July, 1988.

[CAR89A]
Carriero, N., and Gelernter, D. "How to Write Parallel Programs", ACM Computing Surveys 21,3 (Sept. 1989), pp. 323-356.

[CAR89B]
Carriero, N., and Gelernter, D. "Linda in Context", Commun. ACM. 32,4 (April, 1989), pp. 444-458.

[CAR86A]
Carriero, N., and Gelernter, D. "The S/Net's Linda Kernel", ACM Trans. Comput. Syst. 4,2 (May 1986), pp. 110-129.

[CAR86B]
Carriero, N., Gelernter, D., and Leichter, J. "Distributed data structures in Linda". Proceedings of the ACM Symposium on Principles of Programming Languages, Jan. 1986.

[GEL85]
Gelernter, D. "Generative Communication in Linda", ACM Trans. Prog. Lang. Syst. 7,1 (Jan. 1985), pp. 80-112.

[LEL90]
Leler, Wm. "Linda Meets Unix", IEEE Computer. 23,2 (Feb. 1990), pp. 43-54.

[PEA90]
Pearson, Peter K. "Fast Hashing of Variable-Length Text Strings", Commun. ACM. 33,6 (June 1990), pp. 677-680.

APPENDIX A

Monitors Model Listing

```
/* LINDA.COMM.MON is the program kernel for the monitors
model run in a shared memory environment.  The routines are
described in detail in the main body of the thesis.
*/


/* The header file MON_LINDA.H includes the common
structures used in the monitors model.  Of primary concern
are the structures space_q and aggregate: space_q contains a
tuple as it appears in tuple space, while aggregate holds
complex tuple elements.
*/

/* MON_LINDA.H: */

#include <stdarg.h>
#include "p4.h"
#include "p4_compat.h"

#define HANGER_SIZE 80
#define KEY_SIZE 80
#define IN 0
#define RD 1
#define OUT 3

int *pinum2;

struct globals{
  char template[80];
  int type;
  char mask[80];
  } global;

struct space_q {
    char hanger[HANGER_SIZE];
    char mask[80];
    struct space_q *next;
    };

struct aggregate {
    int size;
    char data;
}*ag_ptr;

struct globmem {
    struct space_q *tuple_space[256], *space_tails[256];
```

```
    struct work_struct *pool, *pool_tail;
    int numprocs;
    struct askfor_monitor TS;
}*glob;



struct work_struct {
  char work[80];
  char mask[20];
  struct work_struct *next;
  };

struct space_q *head_avl_nodeq;
```

```
/* LINDA.COMM.MON: */

#include <stdio.h>
#include "mon_linda.h"


slave()
{
    work('s');
}


/* Function RESET is an optional parameter to askfor. It is
not used in the implementation */

reset()
{
}


/* Function GETFUNC is used by askfor.  It is the logic
required to take an unevaluated function from the function
queue (glob->pool) */

getfunc(p)
    int *p;
{
    int rc = 1;

    if (glob->pool != NULL)
    {
     /* return function from pool */
     *p = (int) glob->pool;
     glob->pool = glob->pool->next;
     rc = 0;
    }
    else
    {
     glob->pool_tail = NULL;
     glob->pool = NULL;
    }
    return (rc);

}

/* Function WORK iteratively calls an askfor that attempts
to take an unevaluated function from the function queue.  If
the queue is empty, askfor provides the logic to suspend the
process.  If askfor returns success, then the function
pulled from the queue is evaluated */

work(who)
char who;
```

```
{
    int rc;
    struct work_struct *eval_tuple;
    int func;

    int ptr;
    int result;
    int (*int_func_ptr) ();
    char key[80];

    ptr = 0;
    printf("entered work\n");
    rc = askfor(&(glob->TS), glob->numprocs, getfunc, &func,
                reset);
    while ((rc == 0) || ((rc != -1) && (who == 's')))
    {
     if (rc == 0)
     {
         eval_tuple = (struct work_struct *) func;

         if (strcmp(eval_tuple->mask, "%s%d") == 0)
          sscanf(eval_tuple->work, "%s%d", key,
                &int_func_ptr);


         (*int_func_ptr) ();


     }
     rc = askfor(&(glob->TS), glob->numprocs, getfunc,
            &func, reset);
    }
}


/* Function OUT passes a variable length argument list to
PARSE, which returns the tuple elements in hanger. OUT then
enters the monitor, hashes the tuple structure (space_node)
to an appropriate linked list, and exits the monitor with a
continue statement */

out(tuple_mask)
char tuple_mask[80];

{
    va_list tuple_list;
    int hashnum, type;
    char hanger[HANGER_SIZE], key[KEY_SIZE];
    struct space_q, *space_node;
    struct space_q *alloc_tuple_struct();

    type = OUT;
    va_start(tuple_list, tuple_mask);
    parse(tuple_mask, type, hanger, tuple_list);
```

```
    va_end(tuple_list);

    stok(key, hanger);
    hashnum = thash(key);
    menter(&((glob->TS).m));
    if (glob->tuple_space[hashnum] == NULL)
    {
     space_node = alloc_tuple_struct();
     strcpy(space_node->hanger, hanger);
     strcpy(space_node->mask, tuple_mask);
     space_node->next = NULL;
     glob->tuple_space[hashnum] = space_node;
     glob->space_tails[hashnum] = space_node;
    }
    else
    {

     space_node = alloc_tuple_struct();
     strcpy(space_node->hanger, hanger);
     strcpy(space_node->mask, tuple_mask);
     space_node->next = NULL;
     glob->space_tails[hashnum]->next = space_node;
     glob->space_tails[hashnum] = space_node;
    }
    cont(&((glob->TS).m), 0);

}


/* Like OUT, function IN first calls PARSE, which returns a
template for matching.  IN then invokes askfor, which either
returns in hanger the matched tuple or suspends the process
if no match occurs.  If askfor succeeds, INSTANTIATE does
the actual to formal assignments. */

in(tuple_mask)
char tuple_mask[80];

{

    va_list tuple_list;
    int rc, type;
    char template[HANGER_SIZE];
    char hanger[HANGER_SIZE];

    type = global.type = IN;

    va_start(tuple_list, tuple_mask);
    parse(tuple_mask, type, template, tuple_list);
    va_end(tuple_list);
    strcpy(global.template, template);
    strcpy(global.mask, tuple_mask);
    rc = 1;
    while ((rc != -1) && (rc != 0))
```

```
    {
     rc = askfor(&(glob->TS),glob->numprocs,t_match,hanger,
                 reset);
    }

    if (rc == 0)
     instantiate(tuple_mask, template, hanger);
}


/* Function RD is identical to IN, except that askfor does
not remove the matched tuple from tuple space */

rd(tuple_mask)
char tuple_mask[80];

{
    va_list tuple_list;
    int rc, type;
    char hanger[HANGER_SIZE];
    char template[HANGER_SIZE];

    type = global.type = RD;

    va_start(tuple_list, tuple_mask);
    parse(tuple_mask, type, template, tuple_list);
    va_end(tuple_list);
    strcpy(global.template, template);
    strcpy(global.mask, tuple_mask);
    rc = 1;
    while ((rc != -1) && (rc != 0))
     rc = askfor(&(glob->TS),glob->numprocs,t_match,hanger,
                 reset);
    instantiate(tuple_mask, template, hanger);

}


/* Function EVAL places a pointer to an (unevaluated)
function on a function queue (glob->pool) protected by a
monitor. */

eval(tuple_mask)
char tuple_mask[80];

{
    va_list tuple_list;
    char *mask_ptr, *key;
    int *int_func_ptr;
    char buffer[80];
    struct work_struct *alloc_eval_node();
    struct work_struct *eval_node;

    va_start(tuple_list, tuple_mask);
```

```
    key = va_arg(tuple_list, char *);
    eval_node = alloc_eval_node();
    sprintf(eval_node->work, "%s ", key);
    strcpy(eval_node->mask, "%s%d");

    int_func_ptr = va_arg(tuple_list, int *);
    sprintf(buffer, "%d ", int_func_ptr);
    strcat(eval_node->work, buffer);
    va_end(tuple_list);
    menter(&((glob->TS).m));

    eval_node->next = NULL;
    if (glob->pool == NULL)
    {
     glob->pool = eval_node;
     glob->pool_tail = eval_node;
    }
    else
    {
     glob->pool_tail->next = eval_node;
     glob->pool_tail = eval_node;
    }
    cont(&((glob->TS).m), 0);
}


/* Function T_MATCH provides the logic to compare actuals in
a template with actuals in a tuple structure's hanger.  If
the template matches some tuple, T_MATCH returns success to
askfor along with the matched hanger; otherwise T_MATCH
returns failure to askfor. T_MATCH calls MATCH, which
actually performs the comparison. */

t_match(hanger)
char *hanger;

{
    int rc;
    struct space_q *space_node, *pred;
    struct aggregate *agnode;
    int stok();
    int hashnum;
    int found;
    char key[80], tuple_tok[80];


    found = 0;
    rc = 1;
    stok(key, global.template);
    hashnum = thash(key);
    pred = space_node = glob->tuple_space[hashnum];
    while ((!found) && (space_node != NULL))
    {
     if (match(space_node->hanger, global.template,
```

```
            global.mask))
        {
            found = 1;
        }
        else
        {
            pred = space_node;
            space_node = space_node->next;
        }
    }                       /* end while !NULL */
    if (found)
    {
     strcpy(hanger, space_node->hanger);
     if (global.type == IN)
     {
         if (space_node == glob->tuple_space[hashnum])
          glob->tuple_space[hashnum] = space_node->next;
         else if (space_node->next == NULL)
         {
          pred->next = NULL;
          glob->space_tails[hashnum] = pred;
         }
         else
          pred->next = space_node->next;
         space_node->next = head_avl_nodeq;
         head_avl_nodeq = space_node;
     }
     rc = 0;
    }
    return (rc);
}


/* Function PARSE builds the tuple structure's hanger using
information supplied by the tuple mask.  PARSE pops
arguments off the argument list (tuple_list) and converts
the argument into a formatted string.  The string is
formatted according to the type information found in the
mask. */

parse(tuple_mask, type, buffer, tuple_list)
char *tuple_mask, *buffer;
int type;
va_list tuple_list;

{
    int rc;
    char *mask_ptr;
    struct aggregate *ag_ptr;
    double flt, *flt_ptr;
    int *int_ptr, size, integer;
    char *char_ptr;
    char tok[80];
    rc = 1;
```

```
char_ptr = va_arg(tuple_list, char *);
sprintf(buffer, "%s ", char_ptr);
for (mask_ptr = tuple_mask + 2; *mask_ptr; mask_ptr++)
{
 if (*mask_ptr == '%')
      switch (mask_ptr[1])
      {
      case 's':
       char_ptr = va_arg(tuple_list, char *);
       sprintf(tok, "%s ", char_ptr);
       strcat(buffer, tok);
       break;

      case 'd':
       integer = va_arg(tuple_list, int);
       sprintf(tok, "%d ", integer);
       strcat(buffer, tok);
       break;

      case 'f':
       flt = va_arg(tuple_list, double);
       sprintf(tok, "%f ", flt);
       strcat(buffer, tok);
       break;
      }
 else if (*mask_ptr == '?')
      switch (mask_ptr[1])
      {
      case 'd':
       int_ptr = va_arg(tuple_list, int *);
       /* need to try %p here */
       sprintf(tok, "%d ", int_ptr);
       strcat(buffer, tok);
       break;

      case 's':
       char_ptr = va_arg(tuple_list, char *);
       sprintf(tok, "%d ", char_ptr);
       strcat(buffer, tok);
       break;

      case 'f':
       flt_ptr = va_arg(tuple_list, double *);
       sprintf(tok, "%d ", flt_ptr);
       strcat(buffer, tok);
       break;
      }
 else if (*mask_ptr == ':')
 {

      if (type == OUT)
      {
       switch (mask_ptr[1])
       {
```

```
case 'd':
    int_ptr = va_arg(tuple_list, int *);
    size = va_arg(tuple_list, int);
    ag_ptr = (struct aggregate *)
             g_malloc((sizeof(int) * size) +
                      sizeof(int));
    if (ag_ptr == NULL)
    {
     printf("agmal failed\n");
     exit(1);
    }
    ag_ptr->size = size;
    bcopy(int_ptr, &(ag_ptr->data), (sizeof(int) *
          size));
    sprintf(tok, "%d ", ag_ptr);
    strcat(buffer, tok);
    break;

case 's':
    char_ptr = va_arg(tuple_list, char *);
    size = va_arg(tuple_list, int);
    ag_ptr = (struct aggregate *)
     g_malloc((sizeof(char) * size) +
              sizeof(int));
    if (ag_ptr == NULL)
    {
     printf("agmal failed\n");
     exit(1);
    }
    ag_ptr->size = size;
    bcopy(char_ptr, &(ag_ptr->data), (sizeof(char)
          * size));
    sprintf(tok, "%d ", ag_ptr);
    strcat(buffer, tok);
    break;

case 'f':
    flt_ptr = va_arg(tuple_list, double *);
    size = va_arg(tuple_list, int);
    ag_ptr = (struct aggregate *)
             g_malloc((sizeof(double) * size) +
             sizeof(int));
    if (ag_ptr == NULL)
    {
     printf("agmal failed\n");
     exit(1);
    }
    ag_ptr->size = size;
    bcopy(flt_ptr, &(ag_ptr->data),
        (sizeof(double) * size));
    sprintf(tok, "%d ", ag_ptr);
    strcat(buffer, tok);
    break;
```

```
            }
           }                    /* end if type out */
           else
           {
            switch (mask_ptr[1])
            {
            /* pinum2 stores the global ptr to size */
            case 'd':
                int_ptr = va_arg(tuple_list, int *);
                pinum2 = va_arg(tuple_list, int *);
                sprintf(tok, "%d ", int_ptr);
                strcat(buffer, tok);
                break;

            case 's':
                char_ptr = va_arg(tuple_list, char *);
                pinum2 = va_arg(tuple_list, int *);
                sprintf(tok, "%d ", char_ptr);
                strcat(buffer, tok);
                break;

            case 'f':
                flt_ptr = va_arg(tuple_list, double *);
                pinum2 = va_arg(tuple_list, int *);
                sprintf(tok, "%d ", flt_ptr);
                strcat(buffer, tok);
                break;

            }
           }
      }
     }
}


/* Function INSTANTIATE does actual to formal assignments.
It assigns actuals in hanger to appropriate formals in
template. */

instantiate(tuple_mask, template, hanger)
char *tuple_mask, *template, *hanger;

{
    int rc;
    char *mask_ptr;
    struct space_q *space_node;
    struct aggregate *aggr_node;
    int stok();
    int tokint;
    int *int_ptr;
    char *string_ptr, *char_ptr;
    char template_tok[80], hanger_tok[80], tokchr[80];
    char *pout, *pin;
```

```
int *generic_ptr;
double *flt_ptr, tokflt;

pin = template;
pout = hanger;

for (mask_ptr = tuple_mask + 2, stok(template_tok, pin),
       stok(hanger_tok, pout); *mask_ptr; mask_ptr += 2)
{
 pout = pout + strlen(hanger_tok) + 1;
 pin = pin + strlen(template_tok) + 1;
 stok(template_tok, pin);
 stok(hanger_tok, pout);
 if (*mask_ptr == '%')
 {
 }
 else if (*mask_ptr == '?')
     switch (mask_ptr[1])
     {
     case 's':
      printf("found ?s\n");
      sscanf(template_tok, "%d", &generic_ptr);
      sscanf(hanger_tok, "%s", tokchr);
      char_ptr = (char *) generic_ptr;
      strcpy(char_ptr, tokchr);
      break;

     case 'd':
      sscanf(template_tok, "%d", &generic_ptr);
      sscanf(hanger_tok, "%d", &tokint);
      *generic_ptr = tokint;
      break;

     case 'f':
      sscanf(template_tok, "%d", &generic_ptr);
      sscanf(hanger_tok, "%lf", &tokflt);
      flt_ptr = (double *) generic_ptr;
      *flt_ptr = tokflt;
      break;
     }

 else if (*mask_ptr == ':')
     switch (mask_ptr[1])
     {
     case 'd':
      sscanf(hanger_tok, "%d", &generic_ptr);
      aggr_node = (struct aggregate *) generic_ptr;
      sscanf(template_tok, "%d", &generic_ptr);
      bcopy(&(aggr_node->data), generic_ptr,
             (sizeof(generic_ptr) * aggr_node->size));
      *pinum2 = aggr_node->size;
      free(aggr_node);
      break;
```

```
            case 's':
             sscanf(hanger_tok, "%d", &generic_ptr);
             aggr_node = (struct aggregate *) generic_ptr;
             sscanf(template_tok, "%d", &generic_ptr);
             char_ptr = (char *) generic_ptr;
             bcopy(&(aggr_node->data), char_ptr, (sizeof(char)
                   * aggr_node->size));
             *pinum2 = aggr_node->size;
             free(aggr_node);
             break;

            case 'f':
             sscanf(hanger_tok, "%d", &generic_ptr);
             aggr_node = (struct aggregate *) generic_ptr;
             sscanf(template_tok, "%d", &generic_ptr);
             flt_ptr = (double *) generic_ptr;
             bcopy(&(aggr_node->data), flt_ptr, (sizeof(double)
                  * aggr_node->size));
             *pinum2 = aggr_node->size;
             free(aggr_node);
             break;
            }
    }

}


/* Function STOK picks a space-delimited token off a source
string and returns it in tok.  It is primarily used during
the instantiation phase in IN and RD, where the actuals
embedded in a hanger and the formals embedded in a template
are stripped off a string buffer before the actual is
assigned to the formal. */

int stok(tok, source)
char *tok, *source;

{
    int i;

    for (i = 0; (source[i] != ' ') && (source[i] != '\0');
        i++)
     tok[i] = source[i];
    tok[i] = '\0';
    if (source[i] == '\0')
     return (1);
    else
     return (0);
}
```

```
/* ALLOC_TUPLE_STRUCT dynamically allocates a node for a
tuple structure. */

struct space_q *alloc_tuple_struct()
{
    struct space_q *node;

    if ((node = head_avl_nodeq) == NULL)
    {
     node = (struct space_q *) g_malloc(sizeof(struct
     space_q));
     if (node == NULL)
         printf("Malloc failed for space_q\n");
    }
    else
    {
     head_avl_nodeq = node->next;
    }

    return (node);
}

/* ALLOC_EVAL_NODE dynamically allocates a node for a work
structure in the pool of unevaluated functions. */

struct work_struct * alloc_eval_node()
{
    struct work_struct *node;

    node = (struct work_struct *) g_malloc(sizeof(struct
            work_struct));
    if (node == NULL)
     printf("malloc failed for worknode\n");

    return (node);
}


/* THASH is the hashing function used by OUT. */

thash(word)
char word[80];

{
    int h;
    int i;

    h = 0;
    for (i = 1; word[i] != '\0'; i++)
    {
     h = t[h ^ word[i]];
    }
    return (h);
}
```

```
/* MON_LINDA_INIT initializes the environment, and creates
the appropriate number of slaves. */

mon_linda_init(procs, ac, av)
int procs, int ac;
char **av;

{
    extern slave();
    int i;

    initenv(ac, av);
    glob = (struct globmem *) g_malloc(sizeof(struct
           globmem));
    for (i = 0; i < 256; i++)
    {
     glob->tuple_space[i] = NULL;
     glob->space_tails[i] = NULL;
    }
    glob->pool = NULL;
    glob->pool_tail = NULL;
    head_avl_nodeq = NULL;
    askfor_init(&(glob->TS));
    glob->numprocs = procs;

    for (i = 1; i <= procs; i++)
    {
     create(slave);
    }

}


/* Function MATCH returns success if a template matches a
hanger */

match(template, hanger, mask)
char *template, *hanger, *mask;

{
    int i, k, j, count;
    int flag = 1;

    count = 0;
    k = 0;
    j = 0;
    for (i = 0; (*(mask + i) != '\0') && flag; i += 2)
    {
     if (*(mask + i) == '%')
     {
         for (; (*template != ' ') || (*hanger != ' ');
                template++, hanger++)
         {
          if (*template != *hanger)
```

```
            {
                flag = 0;
                break;
            }
           }
      }
      else
      {

          for (; *template != ' '; template++);
          for (; *hanger != ' '; hanger++);
      }
      template++;
      hanger++;
     }
     return (flag);
}


/* The termination procedure */

mon_linda_end()
{
    progend(&((glob->TS).m));
    wait_for_end();
}
```

APPENDIX B

Message Passing Model Listing

```
/* Many of the routines included in LINDA.COMM.SR are
similar to those described in LINDA.COMM.MON; therefore,
comments listed below strive to point out the differences in
the two comm files. */

/* SR_LINDA.H defines global constants. Values for AG_MAX
and HANGER_SIZE are program dependent. */


#include <stdio.h>
#include <stdarg.h>
#include <p4.h>
#include <p4_compat.h>

#define END 0
#define IN 1
#define RD 2
#define OUT 3
#define AG_MAX 100
#define HANGER_SIZE 80

/* end of SR_LINDA.H */

/* The globals used throughout the comm file functions
include global structures required to store tuples
(space_q), communicate tuple information to the tuple
manager (tuple_msg_type) and suspend processes waiting for a
matching tuple (wait_queue). */

struct tuple_msg_type
{
    int type;
    char mask[20];
    char hanger[HANGER_SIZE];
    int aggreg_size;
    int tuple_size;
    char aggreg_data[AG_MAX];
};

struct tuple_msg_type tuple_channel;

struct space_q
{
    struct space_q *next;
    char mask[20];
    char hanger[128];
```

```
    int aggreg_size;
    int tuple_size;
    char aggreg_data[AG_MAX];
} *head_avl_nodeq, *tuple_space[256], *space_tails[256],
  *head;

struct wait_queue
{
    int id;
    int type;
    char mask[20];
    char hanger[128];
    struct wait_queue *next;
} *head_avl_waitq, *wait_head, *wait_tail;

struct globals
{
    int aglen;
    int *size_ptr;
} global;

int t_master;


/* The comm file routines listed here were those tested on a
shared memory machine.  The proc_group used by such a
configuration assumes that all slave process name their
entry points with the common name slave().  Since only one
of the slave processes can assume the role of tuple manager,
function SLAVE performs manager tasks only if the process
executing SLAVE has a process id equal to 1.  Otherwise
SLAVE invokes LSLAVE, which is assumed to be a worker
process in a Linda program. */

slave()
{
    if (get_my_id() == 1)
     tm();
    else
     lslave();
}


/* Function TM is the main routine for the tuple manager.
It receives tuple information via the message channel
(tuple_channel) and satisfies the appropriate linda
operation, iterating until the termination routine
(SR_LINDA_END) signals end of program. */

tm()
{

    int id, ln, i;
```

```
    wait_tail = wait_head = head_avl_waitq = NULL;
    for (i = 0; i < 256; i++)
    {
     tuple_space[i] = NULL;
     space_tails[i] = NULL;
    }
    head_avl_nodeq = NULL;
    g_recv_any(&id, &tuple_channel, &ln);


    while (tuple_channel.type != END)
    {
     if ((tuple_channel.type == RD) || (tuple_channel.type
           == IN))
         serve_in_or_rd(id, tuple_channel.type, ln);
     else if (tuple_channel.type == OUT)
     {

         if (check_wait(id, ln) == 0)
         {
          serve_out(id, ln);
         }
     }
     g_recv_any(&id, &tuple_channel, &ln);
    }
}

/* Before a tuple is actually placed in tuple space,
function CHECK_WAIT examines the wait queue, which contains
the process id and template belonging to any process waiting
for a matching tuple.  */

check_wait(id, ln)
int id, ln;

{

    int tln, qid, found, found_in, hold_type;
    struct wait_queue *wait_node, *pred, *saveq;

    pred = wait_node = wait_head;
    found_in = 0;
    found = 0;

    while ((wait_node != NULL) && (!found_in))

    {
     qid = wait_node->id;
     if (match(tuple_channel.hanger, wait_node->hanger,
           wait_node->mask))
         found = 1;
     else
     {
         pred = wait_node;
```

```
            wait_node = wait_node->next;
            found = 0;
        }
        if (found)
        {
            hold_type = wait_node->type;

            saveq = wait_node;

            tuple_channel.type = hold_type;
            tln = tuple_channel.tuple_size;
            g_send(qid, &tuple_channel, tln);


            if (wait_node == wait_head)
            {
             wait_head = wait_node->next;
             wait_node = wait_head;
             if (wait_head == NULL)
                 wait_tail = wait_head;
            }
            else if (wait_node == wait_tail)
            {
             pred->next = NULL;
             wait_node = wait_tail = pred;
            }
            else
            {
             pred->next = wait_node->next;
             wait_node = pred;
            }


            if (hold_type == IN)
             found_in = 1;
            else
            {
             pred = wait_node;
             wait_node = wait_node->next;
             found = 0;
            }


            saveq->next = head_avl_waitq;
            head_avl_waitq = saveq;

        }
    }

    return (found_in);
}
```

```
/* Function SERVE_OUT adds the tuple to tuple space under
two conditions: no process on the wait_q matched the tuple
or processes waiting exclusively for RD's matched the tuple
*/

serve_out(id, ln)
int id, ln;

{


    struct space_q *temp_node, *space_node;
    struct space_q *alloc_space_q();
    int tln, hashnum;
    int i;
    char key[20];

    stok(key, tuple_channel.hanger);
    hashnum = thash(key);

    if (tuple_space[hashnum] == NULL)
    {
     space_node = alloc_space_q(ln);
     bcopy(&(tuple_channel.mask), &(space_node->mask), ln -
         sizeof(tuple_channel.type));
     space_node->next = NULL;
     tuple_space[hashnum] = space_node;
     space_tails[hashnum] = space_node;
    }
    else
    {
     space_node = alloc_space_q(ln);
     bcopy(&(tuple_channel.mask), &(space_node->mask), ln -
         sizeof(tuple_channel.type));
     space_node->next = NULL;
     space_tails[hashnum]->next = space_node;
     space_tails[hashnum] = space_node;
    }

}

/* Function SERVE_IN_OR_RD manages the Linda operations in
and rd.  MATCH returns success if the template matches a
hanger in tuple space.  If MATCH fails, the process id of
the waiting process, the template, and the tuple type are
assigned to a wait_queue structure (wait_node) and linked to
the wait queue. */

serve_in_or_rd(id, type, ln)
int id, type, ln;

{
    struct space_q *space_node, *pred;
    struct wait_queue *wait_node, *alloc_wait_q();
```

```
int stok();
int found;
int hashnum;
char key[80];


found = 0;

stok(key, tuple_channel.hanger);

hashnum = thash(key);

pred = space_node = tuple_space[hashnum];
while ((!found) && (space_node != NULL))

{
 if (match(space_node->hanger, tuple_channel.hanger,
          tuple_channel.mask))
 {
     found = 1;
 }
 else
 {
     pred = space_node;
     space_node = space_node->next;
 }
}

if (found)
{
 if (type == IN)
 {
     if (space_node == tuple_space[hashnum])
      tuple_space[hashnum] = space_node->next;
     else if (space_node->next == NULL)
     {
      pred->next = NULL;
      space_tails[hashnum] = pred;
     }
     else
      pred->next = space_node->next;

 }

 bcopy(&(space_node->mask), &(tuple_channel.mask),
        space_node->tuple_size);
 g_send(id, &tuple_channel, space_node->tuple_size);

 free(space_node);
}
```

```
    else
    {

     wait_node = alloc_wait_q();
     wait_node->id = id;
     wait_node->type = tuple_channel.type;
     strcpy(wait_node->mask, tuple_channel.mask);
     strcpy(wait_node->hanger, tuple_channel.hanger);
     wait_node->next = NULL;
     if (wait_head == NULL)
     {
         wait_tail = wait_head = wait_node;
     }
     else
     {
         wait_tail->next = wait_node;
         wait_tail = wait_node;
     }
    }

}


/* ALLOC_SPACE_Q returns an available node for use in tuple
space */

struct space_q *alloc_space_q(t_ln)
int t_ln;
{
    struct space_q *node;

    node = (struct space_q *) g_malloc(t_ln);
    if (node == NULL)
     printf("Failed malloc in node\n");

    return (node);
}


/* ALLOC_WAIT_Q returns an available node for use on the
process wait queue. */

struct wait_queue *alloc_wait_q()
{
    struct wait_queue *node;

    if ((node = head_avl_waitq) == NULL)
    {
     node = (struct wait_queue *) g_malloc(sizeof(struct
            wait_queue));
     if (node == NULL)
         printf("Failed malloc in wait\n");
    }
```

```
    else
    {
     head_avl_waitq = node->next;
    }

    return (node);
}

/* The primary difference between OUT and its cousin in the
monitors model is that after the argument list is parsed and
the hanger packed, OUT sends the information to the tuple
manager for processing.  In other words, the tuple manager
takes the place of the monitor in the message passing model.
*/

out(tuple_mask)
char tuple_mask[80];

{
    va_list tuple_list;
    char *p;
    int type, tln, id;
    char *char_ptr;
    char temp[80];
    char hanger[128];

    type = OUT;
    global.aglen = 1;

    tuple_channel.aggreg_size = 0;
    tuple_channel.aggreg_data[0] = '0';
    va_start(tuple_list, tuple_mask);
    strcpy(tuple_channel.mask, tuple_mask);
    parse(tuple_mask, type, hanger, tuple_list);
    va_end(tuple_list);

    strcpy(tuple_channel.hanger, hanger);
    tuple_channel.type = OUT;
    id = get_my_id();
    tln = sizeof(tuple_channel.aggreg_size) +
                sizeof(tuple_channel.type) +
                 sizeof(tuple_channel.mask) +
                 sizeof(tuple_channel.hanger) +
                sizeof(tuple_channel.tuple_size) +
                 global.aglen;

    tuple_channel.tuple_size = tln;
    g_send(1, &tuple_channel, tln);
}
```

```
/* See the comment on OUT above. */

in(tuple_mask)
char tuple_mask[80];

{
    va_list tuple_list;
    int  tln, id;
    char template[128];
    char temp[80];
    int rc, type;

    type = IN;
    va_start(tuple_list, tuple_mask);
    strcpy(tuple_channel.mask, tuple_mask);
    parse(tuple_mask, type, template, tuple_list);
    va_end(tuple_list);

    strcpy(tuple_channel.hanger, template);
    tuple_channel.type = type;
    tln = tuple_channel.tuple_size = sizeof(tuple_channel);
    g_send(1, &tuple_channel, tln);
    id = get_my_id();
    g_recv_any(&t_master, &tuple_channel, &ln);
    instantiate(tuple_mask, template, tuple_channel.hanger);


}


/* See the comment on OUT above. */

rd(tuple_mask)
char tuple_mask[80];

{
    va_list tuple_list;
    int tln;
    int id;
    char template[128];
    char temp[80];
    int  rc, type;

    type = RD;
    va_start(tuple_list, tuple_mask);
    strcpy(tuple_channel.mask, tuple_mask);
    parse(tuple_mask, type, template, tuple_list);
    va_end(tuple_list);

    strcpy(tuple_channel.hanger, template);
    tuple_channel.type = RD;
    id = get_my_id();
```

```
    tln = tuple_channel.tuple_size = sizeof(tuple_channel);
    g_send(1, &tuple_channel, tln);
    g_recv_any(&t_master, &tuple_channel, &ln);
    instantiate(tuple_mask, template, tuple_channel.hanger);

}


/* PARSE is almost identical to its cousin in the monitors
model.  The primary difference is that we do not dynamically
allocate memory to store aggregate data.  Rather, we copy
the contents of the aggregate into the message channel. */

parse(tuple_mask, type, buffer, tuple_list)
char *tuple_mask, *buffer;
int type;
va_list tuple_list;
{

    char *mask_ptr;
    int *int_ptr,integer;
    char *char_ptr;
    char tok[80];
    double flt;
    double *flt_ptr;

    if (tuple_mask[1] == 's')
    {
     char_ptr = va_arg(tuple_list, char *);
     sprintf(buffer, "%s ", char_ptr);
    }
    else
    {
     integer = va_arg(tuple_list, int);
     sprintf(buffer, "%d ", integer);
    }
    for (mask_ptr = tuple_mask + 2; *mask_ptr; mask_ptr++)
    {
     if (*mask_ptr == '%')
         switch (mask_ptr[1])
         {
         case 's':
          char_ptr = va_arg(tuple_list, char *);
          sprintf(tok, "%s ", char_ptr);
          strcat(buffer, tok);
          break;

         case 'd':
          integer = va_arg(tuple_list, int);
          sprintf(tok, "%d ", integer);
          strcat(buffer, tok);
          break;

         case 'f':
```

```
        flt = va_arg(tuple_list, double);
        sprintf(tok, "%f ", flt);
        strcat(buffer, tok);
        break;
      }
  else if (*mask_ptr == '?')
      switch (mask_ptr[1])
      {
      case 'd':
        int_ptr = va_arg(tuple_list, int *);
        /* need to try %p here */
        sprintf(tok, "%d ", int_ptr);
        strcat(buffer, tok);
        break;

      case 's':
        char_ptr = va_arg(tuple_list, char *);
        sprintf(tok, "%d ", char_ptr);
        strcat(buffer, tok);
        break;

      case 'f':
        flt_ptr = va_arg(tuple_list, double *);
        sprintf(tok, "%d ", flt_ptr);
        strcat(buffer, tok);
        break;
      }
  else if (*mask_ptr == ':')
  {
      if (type == OUT)
      {
        switch (mask_ptr[1])
        {

        case 'd':
            int_ptr = va_arg(tuple_list, int *);
            size = va_arg(tuple_list, int);
            global.aglen = sizeof(int) * size;

            tuple_channel.aggreg_size = size;
            bcopy(int_ptr, &(tuple_channel.aggreg_data),
                 global.aglen);
            sprintf(tok, "%d ", int_ptr);
            strcat(buffer, tok);
            break;

        case 's':
            char_ptr = va_arg(tuple_list, char *);
            size = va_arg(tuple_list, int);
            global.aglen = sizeof(char) * size;
            tuple_channel.aggreg_size = size;
            bcopy(char_ptr, &(tuple_channel.aggreg_data),
                global.aglen);
            sprintf(tok, "%d ", char_ptr);
```

```
                strcat(buffer, tok);
                break;

            case 'f':
                flt_ptr = va_arg(tuple_list, double *);
                size = va_arg(tuple_list, int);
                global.aglen = sizeof(double) * size;
                tuple_channel.aggreg_size = size;
                bcopy(flt_ptr, &(tuple_channel.aggreg_data),
                     global.aglen);
                sprintf(tok, "%d ", flt_ptr);
                strcat(buffer, tok);
                break;
            }
           }
           else
           {
            switch (mask_ptr[1])
            {
            case 'd':
                int_ptr = va_arg(tuple_list, int *);
                global.size_ptr = va_arg(tuple_list, int *);
                sprintf(tok, "%d ", int_ptr);
                break;
            case 's':
                char_ptr = va_arg(tuple_list, char *);
                global.size_ptr = va_arg(tuple_list, int *);
                sprintf(tok, "%d ", char_ptr);
                break;
            case 'f':
                flt_ptr = va_arg(tuple_list, double *);
                global.size_ptr = va_arg(tuple_list, int *);
                sprintf(tok, "%d ", flt_ptr);
                break;
            }

            tuple_channel.aggreg_size = 0;
            tuple_channel.aggreg_data[0] = '0';
            strcat(buffer, tok);

           }
      }

    }
}


/* Again, INSTANTIATE is similar to its cousin in the
monitors model. */

instantiate(tuple_mask, template, hanger)
char *tuple_mask, *template, *hanger;
{
    va_list tuple_list;
```

```
char *mask_ptr;
int stok();
int integer, tokint, tln;
int *int_ptr;
char *char_ptr;
char template_tok[80], hanger_tok[80], tokchr[80];
char *pout, *pin;
int *generic_ptr;
double tokflt, *flt_ptr;

pin = template;
pout = hanger;
for (mask_ptr = tuple_mask + 2, stok(template_tok, pin),
    stok(hanger_tok, pout); *mask_ptr; mask_ptr += 2)
{
 pout = pout + strlen(hanger_tok) + 1;
 pin = pin + strlen(template_tok) + 1;
 stok(template_tok, pin);
 stok(hanger_tok, pout);
 if (*mask_ptr == '%')
     continue;
 else if (*mask_ptr == '?')

     switch (mask_ptr[1])
     {
     case 's':
      sscanf(template_tok, "%d", &generic_ptr);
      sscanf(hanger_tok, "%s", tokchr);
      char_ptr = (char *) generic_ptr;
      strcpy(char_ptr, tokchr);
      break;

     case 'd':
      sscanf(template_tok, "%d", &generic_ptr);
      sscanf(hanger_tok, "%d", &tokint);
      int_ptr = (int *) generic_ptr;
      *int_ptr = tokint;
      break;

     case 'f':
      sscanf(template_tok, "%d", &generic_ptr);
      sscanf(hanger_tok, "%lf", &tokflt);
      flt_ptr = (double *) generic_ptr;
      *flt_ptr = tokflt;
      break;
     }

 else if (*mask_ptr == ':')
     switch (mask_ptr[1])
     {
     case 'd':
      sscanf(template_tok, "%d", &generic_ptr);
      int_ptr = (int *) generic_ptr;
      tln = sizeof(int) * tuple_channel.aggreg_size;
```

```
            bcopy(&(tuple_channel.aggreg_data), int_ptr, tln);
            *global.size_ptr = tuple_channel.aggreg_size;
            break;

           case 's':
            sscanf(template_tok, "%d", &generic_ptr);
            char_ptr = (char *) generic_ptr;
            tln = sizeof(char) * tuple_channel.aggreg_size;
            bcopy(&(tuple_channel.aggreg_data), char_ptr,
                  tln);
            *global.size_ptr = tuple_channel.aggreg_size;
            break;

           case 'f':
            sscanf(template_tok, "%d", &generic_ptr);
            flt_ptr = (double *) generic_ptr;
            tln = sizeof(double) * tuple_channel.aggreg_size;
            bcopy(&(tuple_channel.aggreg_data), flt_ptr, tln);
            *global.size_ptr = tuple_channel.aggreg_size;
            break;
           }

    }

}


/* STOK is identical to its cousin in the monitors model. */

int stok(tok, source)
char *tok, *source;

{
    int i;
    for (i = 0; (source[i] != ' ') && (source[i] != '\0');
        i++)
     tok[i] = source[i];
    tok[i] = '\0';
    if (source[i] == '\0')
     return (1);
    else
     return (0);
}

/* SR_LINDA_END sends a special string to the tuple manager
signalling end of program.  It then waits for all other
slave processes to die. */

sr_linda_end()
{

    struct tuple_msg_type tuple_channel;

    strcpy(tuple_channel.hanger, "endstring");
```

```
    strcpy(tuple_channel.mask, "%end");
    tuple_channel.type = END;
    g_send(1, &tuple_channel, sizeof(tuple_channel));
    wait_for_end();
}

/* THASH is identical to its counterpart in the monitors
model */

thash(word)
    char word[80];

{
    int h;
    int i;

    h = 0;
    for (i = 1; word[i] != '\0'; i++)
    {
     h = t[h ^ word[i]];
     /* printf("%d ",h); */
    }
    return (h);

}

/* MATCH is identical to MATCH in the monitors model */

match(template, hanger, mask)
    char *template, *hanger, *mask;

{
    int i, k, j, count;
    int flag = 1;
    count = 0;
    k = 0;
    j = 0;
    for (i = 0; (*(mask + i) != '\0') && flag; i += 2)
    {
     if (*(mask + i) == '%')
     {
         for (; (*template != ' ') || (*hanger != ' ');
             template++, hanger++)
         {
          if (*template != *hanger)
          {
              flag = 0;
              break;
          }
         }
     }
     else
     {
```

```
            for (; *template != ' '; template++);
            for (; *hanger != ' '; hanger++);
        }
        template++;
        hanger++;
      }
      return (flag);
}

/* SR_LINDA_INIT initializes the environment and creates the
process group. */

sr_linda_init(ac, av)
int ac;
char **av;

{

      initenv(ac, av);
      create_procgroup();

}
```

APPENDIX C

Sample Linda Programs

```
/* A note on the following primes finding programs - The
three primes finding programs are variations on similar
programs found in [CAR89A].  Furthermore, only versions
based on the monitors model are shown here.  Minor
modifications are required for execution under the message
passing model. */

/* PRIMES FINDER I: */

#include <stdio.h>
#include "mon_linda.h"

#define NUM_PROCS 4

main(argc,argv)
int argc;
char **argv;

{
  int primes();
  int last,i,ok,limit;

  mon_linda_init(NUM_PROCS,argc,argv);

  limit = 100;

  for(i=2;i<limit;++i)
   {
     out("%s%d","primeargs",i);
    eval("%s%d","primes",primeptr);
   }

  for(i=2;i<limit;++i)
   {
     rd("%s%d?d","primes",i,&ok);
     if(ok == 1)
      last = i;
   }
  printf("greatest prime is %d\n",last);

  mon_linda_end();

}
```

```
int primes()

{
  int me,i,limit,ok;
  double sqrt();

  in("%s?d","primeargs",&me);
  limit = sqrt((double) me) + 1;
  for(i=2;i<limit;++i)
   {
     rd("%s%d?d","primes",i,&ok);
     if((ok) && (me%i == 0))
       {
        out("%s%d%d","primes",me,0);
        return(0);
       }
   }

  printf("slave %d found prime = %d\n",get_my_id(),me);
  out("%s%d%d","primes",me,1);
  return(1);
}
```

```
/* PRIMES FINDER II */

/* Master process */

#include <stdio.h>
#include "mon_linda.h"

#define NUM_PROCS 3
#define GRAIN 2000
#define LIMIT 100000
#define NUM_INIT_PRIME 15

int prime[LIMIT / 10 + 1] = {2, 3, 5, 7, 11, 13, 17, 19, 23,
                             29, 31, 37, 41, 43, 47}
int pp[LIMIT / 10 + 1] =
      {4, 9, 25, 49, 121, 169, 289, 361, 529, 841, 961,
       1369, 1681, 1849, 2209};

long time_start, time_end;


main(argc, argv)
int argc;
char **argv;

{
    int eot, size, new, first_num, i, num, num_primes;
    char formal[80];
    int new_primes[GRAIN], np2;
    int it;
    int timestart, timeend;
    int worker();

    mon_linda_init(NUM_PROCS, argc, argv);

    timestart = clock();
    for (i = 0; i < NUM_PROCS; ++i)
    {
     eval("%s%d", "worker", worker);
    }


    num_primes = NUM_INIT_PRIME;
    first_num = prime[num_primes - 1] + 2;
    out("%s%d", "next_task", first_num);

    eot = 0;
    newptr = new_primes;
    for (num = first_num; num < LIMIT; num += GRAIN)
    {
     in("%s%d:d", "result", num, new_primes, &size);
```

```
        for (i = 0; i < size; ++i, ++num_primes)
        {
            prime[num_primes] = new_primes[i];
            if (!eot)
            {
             np2 = new_primes[i] * new_primes[i];
             if (np2 > LIMIT)
             {
                 eot = 1;
                 np2 = -1;
             }
             out("%s%d%d%d", "primes", num_primes, .
                  new_primes[i], np2);
            }
        }
       }
       for (i = 0; i < NUM_PROCS; ++i)
        in("%s?d", "worker", &it);

       timeend = clock();
       printf("Time: %d\n", timeend - timestart);

       printf("%d: %d\n", num_primes, prime[num_primes - 1]);

       mon_linda_end();
}


/* worker process */

int worker()

{
       int xprime[LIMIT / 10 + 1] = {2, 3, 5, 7, 11, 13, 17,
                          19, 23, 29, 31, 37, 41, 43, 47 }
       int xpp[LIMIT / 10 + 1] = {4, 9, 25, 49, 121, 169, 289,
                361, 529, 841, 961, 1369, 1681, 1849, 2209};
       int count, eot, i, limit, num, num_primes, ok, start;
       int my_primes[GRAIN];

       num_primes = NUM_INIT_PRIME;

       eot = 0;
       while (1)
       {


        in("%s?d", "next_task", &num);
        if (num == -1)
        {
            out("%s%d", "next_task", -1);
            break;
        }
        limit = num + GRAIN;
```

```
        out("%s%d", "next_task", (limit > LIMIT) ? -1 : limit);

        if (limit > LIMIT)
            limit = LIMIT;

        start = num;
        for (count = 0; num < limit; num += 2)
        {
            while (!eot && num > xpp[num_primes - 1])
            {
             rd("%s%d?d?d", "primes", num_primes,
              &xprime[num_primes], &xpp[num_primes]);
             if (xpp[num_primes] < 0)
                 eot = 1;
             else
                 ++num_primes;
            }

            for (i = 1, ok = 1; i < num_primes; ++i)
            {
             if (!(num % xprime[i]))
             {
                 ok = 0;
                 break;
             }
             if (num < xpp[i])
                 break;
            }
            if (ok)
            {
             my_primes[count] = num;
             ++count;
            }
        }
        out("%s%d:d", "result", start, my_primes, count);
    }
    out("%s%d", "worker", 1);
}
```

```
/* PRIMES FINDER III: */

#include <stdio.h>
#include "mon_linda.h"

#define LIMIT 200
#define NUM_PROCS 6
long time_start, time_end;

main(argc, argv)
int argc;
char **argv;

{
    int source();
    int sink();
    int i, end;
    int timestart, timeend;

    mon_linda_init(NUM_PROCS, argc, argv);
    timestart = clock();
    eval("%s%d", "source", source);
    eval("%s%d", "sink", sink);
    in("%s?d", "sink", &end);
    timeend = clock();
    mon_linda_end();
    printf("time is %d\n", timeend - timestart);

}


int source()

{
    int i, out_index = 0;

    for (i = 5; i < LIMIT; i += 2)
     out("%s%d%d%d", "seg", 3, out_index++, i);
    out("%s%d%d%d", "seg", 3, out_index, 0);

}

int sink()
{
    int in_index = 0, num, prime = 3, prime_count = 2;
    int pipe_seg();

    while (1)
    {
     in("%s%d%d?d", "seg", prime, in_index, &num);

     in_index++;
     if (!num)
         break;
```

```
        if (num % prime)
        {
            ++prime_count;
            if ((num * num) < LIMIT)
            {
             eval("%s%d", "pipeseg", pipe_seg);
             out("%s%d%d%d", "psegargs", prime, num, in_index);

             prime = num;
             in_index = 0;
            }
        }
    }
    printf("count: %d.\n", prime_count);
    out("%s%d", "sink", 1);

}

int pipe_seg()

{
    int prime, next, in_index, num, out_index = 0;

    in("%s?d?d?d", "psegargs", &prime, &next, &in_index);

    while (1)
    {

     in("%s%d%d?d", "seg", prime, in_index, &num);
     in_index++;
     if (!num)
     {
         out("%s%d%d%d", "seg", next, out_index, num);
         return;
     }
     if (num % prime)
     {
         out("%s%d%d%d", "seg", next, out_index, num);
         out_index++;
     }
    }
}
```

```
/* A note on the SEMIGROUPS PROBLEM - The program is a
modification of one referenced in [BUT88] and was tested
under the message passing model.  Minor modifications are
required for it to run under the monitors model. */

/* SEMI.H - header file for the semigroups problem */

#define BOOL   int
#define TRUE   1
#define FALSE  0

#define DEFAULT_MEM_SZ 10000
#define MAX_INIT_PROB_SZ 10
#define MAXTBLSZ 20
#define MAXWORDSZ  125
#define MAXSLAVES  9
#define MAXOPER 9
#define HASH_TBL_SZ 9973

/* message types */

#define INITDATA    0
#define REQWORK     1
#define WORK        2
#define CANDIDATE   3
#define NEWWORD     4
#define TERMINATE   5
#define ACK         6

#define PERFORM_OPERATION(W1,W2,W3,WORDSZ)    \
     {                                \
     int i;                                \
                                      \
          for (i=0; i < WORDSZ; i++)     \
               W3[i] = operation_tbl[W1[i]][W2[i]]; \
          W3[i] = '\0';  /* add string terminator */    \
     }

typedef char WORD[MAXWORDSZ+1];

#define DATA_REC     \
        int  type;   \
        WORD word;   \
        BOOL ack;    \
        int idx;

struct data_struct
{
     DATA_REC
};

struct hash_node
{
     WORD *wrd_ptr;
```

```
     struct hash_node *next;
};

struct newword
{
     struct newword *next;
     WORD word;
     int idx;
};


struct init_data_struct
{
     int type;
     int master;
     int my_id;
     int word_sz;
     int oper_tbl_sz;
     char operation_tbl[MAXOPER][MAXOPER];
};
struct work_struct
{
     DATA_REC
};
struct cand_struct
{
     DATA_REC
};
struct neww_struct
{
     DATA_REC
};
struct ack_struct
{
     DATA_REC
};



/* SEMIGROUPS PROBLEM - MASTER PROCESS: */

#include <stdio.h>
#include "semi.h"
#include "sr_linda.h"


int my_id = 0;
BOOL more_work;
WORD word;
char s[80];
int type;
int wait_idx;
struct data_struct newword, work, candidate;
int next_mast_entry;
```

```
struct newword *head_avl_newword;
struct hash_node *head_avl_hash_node;

int msg_type, i, idx, size, next_idx, last_idx, hash_idx,
       nslaves;
int num_rows_in_mast_tbl;
long time_start, time_end;
struct hash_node *hash_tbl[HASH_TBL_SZ], *p,
*alloc_hashnode();

main(argc, argv)
int argc;
char **argv;
{
    struct newword *newword_queue[MAXSLAVES], *qp,
                   *alloc_newword();
    int id, word_sz, oper_tbl_sz;
    char operation_tbl[MAXOPER][MAXOPER];
    char mytab[MAXOPER][MAXOPER];
    int wait_queue[MAXSLAVES];
    WORD *master_table[100];
    int timestart, timeend;
    int ln, op_sz;
    int tempint;
    int i, j;

    sr_linda_init(argc, argv);

    next_mast_entry = 2;
    printf("master before Malloc\n");
    master_table[0] = (WORD *) g_malloc((sizeof(WORD) *
                  MAXTBLSZ));
    printf("master after Malloc\n");
    if (master_table[0] == NULL)
    {
     printf("first Malloc failed in master\n");
     exit(9);
    }
    printf("master before Malloc\n");
    master_table[1] = (WORD *) g_malloc((sizeof(WORD) *
                  MAXTBLSZ));
    printf("master after Malloc\n");
    if (master_table[1] == NULL)
    {
     printf("second Malloc failed in master\n");
     exit(9);
    }

    for (i = 0; i < HASH_TBL_SZ; i++)
     hash_tbl[i] = NULL;

    for (i = 0; i < MAXSLAVES; i++)
     newword_queue[i] = NULL;
```

```
read_input(master_table, hash_tbl, newword_queue,
          &nslaves,
 &num_rows_in_mast_tbl, &word_sz,
 &oper_tbl_sz, operation_tbl);

bcopy(operation_tbl, mytab, 36);
last_idx = num_rows_in_mast_tbl - 1;
head_avl_newword = NULL;
head_avl_hash_node = NULL;

for (i = 1; i <= nslaves; i++)
{
 out("%s%d", "id", i);
}
strcpy(word, "init");
op_sz = sizeof(operation_tbl);
printf("sizeof oper_tbl is %d\n", op_sz);
for (i = 1; i <= nslaves; i++)
 out("%s%d%d%s:s", "initdata", i, word_sz, word,
      operation_tbl, op_sz);

more_work = TRUE;
wait_idx = -1;
next_idx = 0;

timestart = clock();
while (more_work)
{
 in("%s?d?d?s?d", "master", &type, &id, word, &idx);
 if (type != REQWORK && type != CANDIDATE)
 {
     exit(99);
 }
 if (type == CANDIDATE)

 {
     if (!word_exists(word, hash_tbl, word_sz))
     {
      last_idx++;
      if (last_idx >= (next_mast_entry * MAXTBLSZ))
      {
          master_table[next_mast_entry] = (WORD *)
                 g_malloc((sizeof(WORD) * MAXTBLSZ));
          if (master_table[next_mast_entry] == NULL)
          {
           exit(9);
          }
          next_mast_entry++;
      }
      strcpy(master_table[last_idx / MAXTBLSZ] +
             (last_idx % MAXTBLSZ), word);
      hash_idx = hash(word, word_sz);
      p = alloc_hashnode();
      p->next = hash_tbl[hash_idx];
```

```
        p->wrd_ptr = master_table[last_idx / MAXTBLSZ] +
                     (last_idx % MAXTBLSZ);
        hash_tbl[hash_idx] = p;
        for (i = 0; i < nslaves; i++)
        {
            qp = alloc_newword();
            qp->next = newword_queue[i];
            strcpy(qp->word, master_table[last_idx /
                  MAXTBLSZ] + (last_idx % MAXTBLSZ));
            qp->idx = last_idx;
            newword_queue[i] = qp;
        }           /* endfor */
        if (wait_idx > -1)
        {
            dump_queue(wait_queue[wait_idx],
                       newword_queue, nslaves);
            type = WORK;
            out("%d%d%s%d", wait_queue[wait_idx], type,
                 word, idx);
            wait_idx--;
            next_idx++;
        }           /* endif */
    }               /* endif */
    dump_queue(id, newword_queue, nslaves);

    type = ACK;
    out("%d%d%s%d", id, type, word, idx);
}
else
{
    dump_queue(id, newword_queue, nslaves);
    if (next_idx > last_idx)
    {
     if (wait_idx == nslaves - 2)
     {
         for (i = 1; i <= nslaves; i++)
         {
          type = TERMINATE;
          out("%d%d%s%d", i, type, word, idx);
         }          /* endfor */

         more_work = FALSE;
     }
     else
     {
         wait_idx++;
         wait_queue[wait_idx] = id;
     }          /* endif */
    }
    else
    {
     idx = next_idx++;
```

```
            type = WORK;
            out("%d%d%s%d", id, type, word, idx);
           }
        }
       }

       timeend = clock();
       sr_linda_end();

       printf("Time was %d\n", timeend - timestart);
       printf("Ending table with %d entries\n", last_idx + 1);
       /*
        * print_table(master_table,last_idx,word_sz);
        * display_hash_table(hash_tbl);
        */
       printf("Time was %d\n", timeend - timestart);
}

display_hash_table(tbl)
struct hash_node *tbl[];

{

       int i, count, total_count;
       struct hash_node *p;

       printf("\nDisplay of Hash Table\n");
       total_count = 0;
       for (i = 0; i < HASH_TBL_SZ; i++)
       {
        for (count = 0, p = tbl[i]; p != NULL; p = p->next,
            count++);
        if (count)
        {
            printf("Hash idx = %d has %d nodes\n", i, count);
            total_count += count;
        }
       }

       printf("\nTotal nodes found in hash table = %d\n",
               total_count);

}                      /* end display_hash_table */

print_table(table, table_length, word_length)
WORD *table[];
int table_length, word_length;

{
       int i;
       int tbl_idx;

       for (tbl_idx = 0; (tbl_idx * MAXTBLSZ) < table_length;
                   tbl_idx++)
```

```
      for (i = 0; i < MAXTBLSZ && (tbl_idx * MAXTBLSZ + i) <=
                  table_length; i++)
          print_word(table[tbl_idx] + i, word_length);

    return;

}                   /* end print_table */

print_word(word, length)
WORD word;
int length;
{
    int i;

    if (my_id)
    {
     printf("slv%d: -->", my_id);
    }
    else
    {
     printf("master -->");
    }

    printf("   ");
    for (i = 0; i < length; i++)
     printf("%c", word[i] + '0' - 1);
    printf(" %x", word[i]);

    printf("\n");

    return;

}                   /* end print_word */


read_input(master_tbl, hash_tbl, slv_q_tbl, nslaves,
   init_prob_sz, word_sz, oper_tbl_sz, operation_tbl)

WORD *master_tbl[];
int *nslaves, *init_prob_sz, *word_sz, *oper_tbl_sz;
struct hash_node *hash_tbl[];
struct newword *slv_q_tbl[];
char operation_tbl[MAXOPER][MAXOPER];

{
    int i, j, hash_idx;
    struct hash_node *p;
    struct newword *alloc_newword();
    struct newword *qp;

    scanf("%d %d %d", nslaves, word_sz, init_prob_sz);
    /*
     * printf("nslaves=%d word_sz=%d init_prob_sz=%d
     * \n",*nslaves,*word_sz,*init_prob_sz);  */
```

```
    if (*init_prob_sz >= MAX_INIT_PROB_SZ || *word_sz >
        MAXWORDSZ)
    {
     printf("problem too big - increase size of init_tbl or
           word size\n");
     exit(3);
    }
    for (i = 0; i <= (*init_prob_sz) - 1; i++)
    {
     scanf("%s", master_tbl[0] + i);
     convert(master_tbl[0] + i, master_tbl[0] + i, word_sz);

     hash_idx = hash(master_tbl[0] + i, *word_sz);
     p = alloc_hashnode();
     p->next = hash_tbl[hash_idx];
     p->wrd_ptr = master_tbl[0] + i;
     hash_tbl[hash_idx] = p;
     for (j = 0; j < *nslaves; j++)
     {
         qp = alloc_newword();
         qp->next = slv_q_tbl[j];
         qp->idx = i;
         strcpy(qp->word, master_tbl[0] + i);
         slv_q_tbl[j] = qp;
     }
    }

    printf("Initial table with %d entries:\n",
            *init_prob_sz);
    print_table(master_tbl, (*init_prob_sz) - 1, *word_sz);

    scanf("%d", oper_tbl_sz);
    while (getchar() != '\n');
    printf("\nOperation table of dimension %d:\n",
             *oper_tbl_sz);
    for (i = 1; i <= (*oper_tbl_sz); i++)
    {
     for (j = 1; j <= (*oper_tbl_sz); j++)
     {
         operation_tbl[i][j] = getchar();
         printf("%c", operation_tbl[i][j]);
         operation_tbl[i][j] = operation_tbl[i][j] - '0' +
                               1;
         printf(" %x", operation_tbl[i][j]);
     }
     while (getchar() != '\n');
     printf("\n");
    }

}
```

```
convert(source, target, ln)
WORD source, target;
int *ln;
{

    int i;

    for (i = 0; i < *ln; i++)
     target[i] = source[i] - '0' + 1;

    return;
}

dump_queue(id, q_tbl, nslaves)
int id;
struct newword *q_tbl[];
int nslaves;

{
    struct newword *qp, *qp1;
    int i, mark;


    for (i = 1, mark = (-1); i <= nslaves && mark == (-1);
          i++)
    {
     if (id == i)
         mark = i - 1;
    }

    if (mark == (-1))
    {
     printf("Master unable to locate %d i\n", id);
     exit(3);
    }

    for (qp = q_tbl[mark], qp1 = NULL; qp != NULL; qp1 = qp,
         qp = qp->next)

    {
     strcpy(word, qp->word);
     idx = qp->idx;
     type = NEWWORD;
     out("%d%d%s%d", id, type, word, idx);
    }

    if (qp1 != NULL)
    {
     qp1->next = head_avl_newword;
     head_avl_newword = q_tbl[mark];
    }
    q_tbl[mark] = NULL;
    return;
}                   /* end dump_queue */
```

```
struct hash_node *alloc_hashnode()
{
    int i;
    struct hash_node *p, *qp, *qp1;

    if (head_avl_hash_node == NULL)
    {
     qp = (struct hash_node *) g_malloc((sizeof(struct
            hash_node)) * 100);
     if (qp == NULL)
     {
         printf("can't alloc hashnode\n");
         exit(9);
     }
     for (i = 1, head_avl_hash_node = qp; i < 100; i++, qp =
          qp->next)
     {
         qp->next = qp + 1;
     }
     qp->next = NULL;
    }
    p = head_avl_hash_node;
    head_avl_hash_node = head_avl_hash_node->next;
    return (p);
}                   /* end alloc_hashnode */

struct newword * alloc_newword()
{
    int i;
    struct newword *p, *qp, *qp1;

    if (head_avl_newword == NULL)
    {
     qp = (struct newword *) g_malloc((sizeof(struct
            newword))
           * 100);
     if (qp == NULL)
     {
         printf("cant alloc newword\n");
         exit(9);
     }
     for (i = 1, head_avl_newword = qp; i < 100; i++, qp =
          qp->next)
     {
         qp->next = qp + 1;
     }
     qp->next = NULL;
    }
    p = head_avl_newword;
    head_avl_newword = head_avl_newword->next;
    return (p);
}                   /* end alloc_newword */
```

```
hash(word, word_sz)
WORD word;
int word_sz;
{
    int aligned_buffer[(MAXWORDSZ / sizeof(int)) + 1];
    int i, left, right, j;
    unsigned int accum, ored_word;
    int *l;
    char *c;


    strcpy((char *) aligned_buffer, word);
    l = aligned_buffer;
    accum = 0;
    for (i = (word_sz / (3 * sizeof(int))); i; i--)
    {
     ored_word = 0;
     for (j = 0; j < 3; j++)
     {
         ored_word |= (*l << j * 3);
         l++;
     }
     accum = (accum << 1) ^ ored_word;
    }
    return (accum % HASH_TBL_SZ);
}                   /* end hash */

word_exists(word, hash_tbl, word_sz)
struct hash_node *hash_tbl[];
WORD word;
int word_sz;

{
    int i, rc;
    struct hash_node *p;

    for (rc = 0, i = hash(word, word_sz), p = hash_tbl[i]; p
         != NULL && rc == 0; p = p->next)
    {
     if (strcmp(word, p->wrd_ptr) == 0) /* if equal words */

         rc = 1;
    }

    return (rc);
}                   /* end word_exists */
```

```
/* SEMIGROUPS PROBLEM - SLAVE PROCESS: */

#include <stdio.h>
#include "semi.h"

int my_id;
char s[80];
WORD word;
int idx, type;
int word_sz, oper_tbl_sz;
struct hash_node *hash_tbl[HASH_TBL_SZ], *p,
*slave_alloc_hashnode();
struct hash_node *slave_head_avl_hash_node;
char operation_tbl[MAXOPER][MAXOPER];
char *op_tbl;

lslave()
{
    int id, myid;
    BOOL more_work, waiting_for_ack, first_with_idx;
    int i, work_idx, msg_type, hash_idx, loc_tbl_idx,
        local_idx;
    WORD *local_table[100], newword;
    int next_local_idx;
    int ln, j;

    slave_head_avl_hash_node = NULL;
    next_local_idx = 2;
    local_table[0] = (WORD *) g_malloc((sizeof(WORD) *
                      MAXTBLSZ));
    if (local_table[0] == NULL)
    {
     printf("First Malloc failed in a slave\n");
     exit(9);
    }
    printf("s: local table 1 defined\n");
    local_table[1] = (WORD *) g_malloc((sizeof(WORD) *
                      MAXTBLSZ));
    if (local_table[1] == NULL)
    {
     printf("Second Malloc failed in a slave\n");
     exit(9);
    }
    printf("s: local table 2 defined\n");
    for (i = 0; i < HASH_TBL_SZ; i++)
     hash_tbl[i] = NULL;

    in("%s?d", "id", &id);

    strcpy(s, "%s%d?d?s:s");
    in("%s%d?d?s:s", "initdata", id, &word_sz, word,
                      operation_tbl, &oper_tbl_sz);
    myid = get_my_id();
    type = REQWORK;
```

```
strcpy(word, "dummy");
idx = 0;

out("%s%d%d%s%d", "master", type, id, word, idx);

more_work = TRUE;
while (more_work)
{
 in("%d?d?s?d", id, &type, word, &idx);
 switch (type)
 {

 case TERMINATE:
     more_work = FALSE;
     break;

 case NEWWORD:
     for (local_idx = idx / MAXTBLSZ; local_idx >=
          next_local_idx; next_local_idx++)
     {
      local_table[next_local_idx] = (WORD *)
              g_malloc((sizeof(WORD) * MAXTBLSZ));
      if (local_table[next_local_idx] == NULL)
      {
          exit(9);
      }
     }
     strcpy(local_table[local_idx] + (idx % MAXTBLSZ),
            word);
     hash_idx = hash(word, word_sz);
     p = slave_alloc_hashnode();
     p->next = hash_tbl[hash_idx];
     p->wrd_ptr = local_table[idx / MAXTBLSZ] + (idx %
                  MAXTBLSZ);
     hash_tbl[hash_idx] = p;
     break;

 case WORK:
     work_idx = idx;
     loc_tbl_idx = 0;
     first_with_idx = TRUE;
     while (generate_a_word(local_table, &loc_tbl_idx,
          work_idx, word_sz,
          &first_with_idx, newword,
          operation_tbl))
     {
      if (!word_exists(newword, hash_tbl, word_sz))
      {
          strcpy(word, newword);
          /*
           * printf("NEWER: ");
             print_word(word,word_sz);
           */
          type = CANDIDATE;
```

```
                out("%s%d%d%s%d", "master", type, id, word,
                     idx);
                waiting_for_ack = TRUE;
                while (waiting_for_ack)
                {
                 in("%d?d?s?d", id, &type, word, &idx);
                 if (type != NEWWORD && type != ACK)
                 {
                     exit(99);
                 }

                 if (type == NEWWORD)
                 {
                     for (local_idx = idx / MAXTBLSZ;
                          local_idx
                           >= next_local_idx;
                           next_local_idx++)
                     {
                      local_table[next_local_idx] = (WORD *)
                            g_malloc((sizeof(WORD) *
                            MAXTBLSZ));
                      if (local_table[next_local_idx] == NULL)
                          exit(9);
                     }
                     strcpy(local_table[local_idx] + (idx %
                            MAXTBLSZ), word);
                     hash_idx = hash(word, word_sz);
                     p = slave_alloc_hashnode();
                     p->next = hash_tbl[hash_idx];
                     p->wrd_ptr = local_table[idx / MAXTBLSZ]
                                  + (idx % MAXTBLSZ);
                     hash_tbl[hash_idx] = p;
                 }
                 else
                 {
                     waiting_for_ack = FALSE;
                 }
                }
             }
            }
            type = REQWORK;
            out("%s%d%d%s%d", "master", type, id, word, idx);
            break;

        default:
            printf("s: exiting due to invalid type %d\n",
                    type);
            exit(99);
            break;
         }              /* end switch */
        }                    /* endwhile */

}                        /* end main */
```

```
generate_a_word(local_table, loc_tbl_idx,
    data_rec_idx, word_sz, first_with_idx,
    newword, operation_tbl)
WORD *local_table[], newword;
int *loc_tbl_idx, data_rec_idx, word_sz;
BOOL *first_with_idx;
char operation_tbl[MAXOPER][MAXOPER];

{
    char *w1, *w2;
    int rc;

    rc = 1;                /* word generated */
    w1 = local_table[*loc_tbl_idx / MAXTBLSZ] +
                    (*loc_tbl_idx
                     % MAXTBLSZ);

    w2 = local_table[data_rec_idx / MAXTBLSZ] +
                    (data_rec_idx
                     % MAXTBLSZ);
    if (*first_with_idx && *loc_tbl_idx <= data_rec_idx)
    {
     PERFORM_OPERATION(w1, w2, newword, word_sz);
     *first_with_idx = FALSE;
    }
    else
    {
     if (*loc_tbl_idx < data_rec_idx)
     {
         PERFORM_OPERATION(w2, w1, newword, word_sz);
     }
     else
     {
         rc = 0;           /* word not generated */
     }
     *first_with_idx = TRUE;
     (*loc_tbl_idx)++;
    }

    return (rc);
}

struct hash_node *slave_alloc_hashnode()
{
    int i;
    struct hash_node *p, *qp, *qp1;

    if (slave_head_avl_hash_node == NULL)
    {
     qp = (struct hash_node *) g_malloc((sizeof(struct
           hash_node)) * 100);
     if (qp == NULL)
     {
         printf("Error in malloc for hash nodes in
```

```
                slave\n");
            exit(9);
        }
        for (i = 1, slave_head_avl_hash_node = qp; i < 100;
              i++,
             qp = qp->next)
        {
            qp->next = qp + 1;
        }
        qp->next = NULL;
    }
    p = slave_head_avl_hash_node;
    slave_head_avl_hash_node =
      slave_head_avl_hash_node->next;
    return (p);
}                       /* end slave_alloc_hashnode */
```

VITA

Alan Leveton currently works as a systems analyst for the Naval Aviation Depot in Jacksonville, Florida. He received a B.A. in Education from the University of Florida in 1974. A teaching career that spanned over ten years included a variety of courses in Literature and Mathematics. Alan plans to graduate in May, 1991.