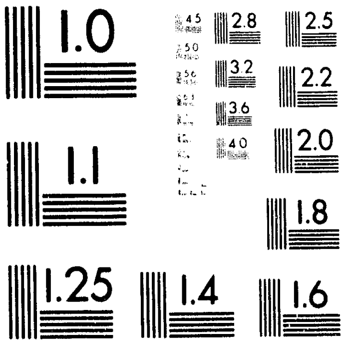
1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

1 of 1

p4-Linda: A Portable Implementation of Linda

Ralph M. Butler *
Alan L. Leveton

College of Comp. and Inf. Sci.
University of North Florida
Jacksonville, FL 32224
rbutler@sinkhole.unf.edu

Ewing L. Lusk †

Math. and Comp. Sci. Division
Argonne National Laboratory
Argonne, IL 60439
lusk@mcs.anl.gov

Abstract

Facilities such as interprocess communication and protection of shared resources have been 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. When multiprocessors (or networks of heterogeneous processors) are used for explicit parallelism, the difference between these models is exposed to the programmer. The p4 tool set was originally developed to buffer the programmer from synchronization issues while offering an added advantage in portability, however two models are often still needed to develop parallel algorithms. We provide two implementations of Linda in an attempt to support a single high-level programming model on top of the existing paradigms in order to provide a consistent semantics regardless of the underlying model. Linda's fundamental properties associated with generative communication eliminate the distinction between shared and distributed memory.

1 Introduction

We have implemented two compatible versions of Linda on top of the p4 portable parallel programming system, one to take advantage of shared-memory architectures, the other to utilize a network of heterogeneous processors, offering an advantage in portability. Each implementation is based on a different programming model: an abstract data structure called a monitor synchronizes access to shared data in shared-memory architectures, whereas processes in distributed-memory space communicate through message-passing operations. 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.

Although shared-memory seems natural for Linda's 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.

The message-passing programming model provides a means for distributed, loosely-coupled processes to communicate solely through messages. It supports an implementation of Linda that works on both shared-memory machines and distributed machines that communicate over a network. This model may run on a large multicomputer, or on a collection of heterogeneous machines, including a network of workstations. It provides a more portable system at the possible risk of suffering some loss in performance.

2 Linda Background

Linda is described in [8]. 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 an

*This work was partially supported by National Science Foundation grant CCR-9121875.

†This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

environment called tuple space where they exist independently until a process chooses to receive them.

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 tuples called tuple space. Theoretically, the object remains in tuple space forever, unless removed by another process [6].

Tuple space holds two varieties of tuples. Process or "live" tuples are under active evaluation, incorporate executable code, and execute concurrently. 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 [6].

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 [1]. 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 [1]. 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 [6]. 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 [1], 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) [6]. 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) [8]. 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 [10], 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>)
```

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 processes may retrieve tuples, and tuples added to tuple space by out remain in tuple space until removed by in [8].

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 [8]. The structured name should not be confused with the logical name, which is simply the initial

3 p4 Background

p4 [4] [2] [9] is a set of parallel programming tools designed to support portability across a wide range of multiprocessor/multicomputer architectures (hence the name "Portable Programs for Parallel Processors"). Three parallel processing paradigms are supported:

- shared-memory multiprocessors;
- a set of processors that communicate solely through messages (typically, a distributed-memory multiprocessor, or a group of machines that communciate over a network);
- communicating clusters (sets of large multiprocessors that communicate via shared-memory locally and via message-passing remotely).

The tools that support these paradigms achieve portability by hiding machine dependent details inside C procedures.

Programming multiprocessors in which processes can communicate 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.

p4 includes high-level monitor operations built on top of low-level, machine-dependent primitives. One special-purpose mechanism is called the *askfor* monitor. A common pattern in multiprocessing, sometimes called agenda parallelism [6], 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>,<num_processes>,
       <get_problem>,<task>,<reset>)
```

where monitor_name is a unique name of the monitor, num_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.

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:

```
p4_send(<type>,<id>,<msg>,<size>)
p4_recv(<type>,<id>,<msg>,<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), 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. *p4_sendr* (send with rendezvous), an alternative to send, forces the sending process to suspend until it receives acknowledgement from the recipient.

Processes are created in p4 via *p4_create_procgroup*(). It reads a file, called the *procgroup* file, to determine on which machines processes are to be started, and the number on each machine.

4 Interface to p4-Linda

Linda operations must adhere to a strict format in our implementations. In particular, a format string or mask, must be present as the first argument to some of the Linda operations; it should not be confused with the tuple elements themselves. This mask is unusual in our implementation, but is typical for many C libraries that contain functions which accept variable length argument lists (e.g., printf). The range of valid data types for tuples include integers, one-dimensional strings, floats (doubles), and aggregates (arrays of any of the other types). 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;
  ...
  out("%s%s%d:d","key",buf,num,big,size);
}
```

A necessary limitation of our model is that tuple arguments to out must be actuals. Furthermore, a tuple may contain 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 matching criteria consists of "key" and "2". 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.

p4-Linda requires that the user program include a header file and invoke initialization and termination procedures. Processes are created as part of the initialization procedure, by reading a procgroup file that includes the following information:

- the name of each (remote) machine on which processes are to be created
- the number of processes that are to be created and share memory on each remote machine
- the full path name of the remote program on each machine

We wanted to design a Linda model, not a complete Linda kernel; hence, the fundamental decision to code the Linda operations as functions. Further, we observed 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 p4-Linda primitives as an extension of this standard. This decision resulted in certain limitations on eval and out. A Linda kernel cited in [7] 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("prime_args",i)
eval("key","primes")
```

where "primes" is the name of a function which is found in a table supplied by the user at initialization. The primes procedure would then obtain its arguments by doing:

```
in("prime_args",&i)
```

Also, our implementation of eval does not place a tuple in tuple space, rather the invoked procedure (primes in this case) is responsible for doing an out operation when it completes.

In p4-Linda, the arguments to out are restricted to actuals. 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 would be quite difficult to implement in a loosely-coupled environment.

5 Design of 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, i.e. formals or actuals that constitute the tuple. The tuple mask is the second element and contains the typing information required to process the tuple. 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 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.

We considered two possible implementations for eval in the shared-memory model. One method dynamically creates processes as needed, i.e. eval("key",func) would cause a new process to be created. The other method would cause a set of processes to be created at initialization. These processes would then share the task of handling any new work that is generated, remaining active until termination of the user's program. The latter approach was selected because it follows the established p4 model which assumes that process create/destroy may be an expensive operation on many machines.

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 askfors 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, link the tuple structure to the appropriate hash list, and relinquish the monitor.

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 then gain exclusive access to the tuple space monitor to search for a matching tuple. The askfor monitor provides the answer. Recall that one of the parameters to askfor is $< get_{p}roblem >$, 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:

- search the appropriate hash list for a matching tuple
- if a match is found, delete the tuple structure from the hash list and return success to askfor
- if no match is found, return failure to askfor

Two characteristics of askfor are crucial to the p4-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 a function to compute the number of primes within the range 2 to N. 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()
{
  /* masks omitted for convenience */
  out("prime_arg",3);
  eval("prime_test","primes");
  /* collect primes with */
  /* "is_prime" tag      */
}
```

```
primes()
{
  int i,result;

  in("prime_arg",i)
  /* compute result and */
  /* put in tuple space */
  out("is_prime",result);
}
```

With these restrictions in mind, the design of eval only has to assign unevaluated live tuples to waiting processes. 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 may be expensive on many machines, we decided to create N processes up front where N is the number of processes specified by the user in the procgroup file. This permits us to "reuse" processes rather than repeatedly create them. The p4 procedure *p4_create_procgroup*() spawns processes which begin execution at a procedure that invoke an askfor that manages the assignment of unevaluated tuples to available processes, and then invoke the function in the tuple retrieved from the pool.

6 Design of the Message-Passing Implementation

A p4-Linda program based on message-passing requires a minimum of two processes: a master process to initialize the environment and a process to act as tuple space manager. Of course, if there are not other processes, then the master process will be the only process to alter tuple space. All communication between the master process and slaves is mediated through p4-Linda operations and tuple storage handled by the manager.

A fundamental decision in the message-passing model was whether tuple space should be centralized, distributed, or even replicated. We opted for a centralized tuple space because the alternative methods require building fast deletion and broadcast protocols, an effort beyond the scope of the project. For an interesting discussion of these schemes see [5].

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 distributed-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:

```
p4_send(type,manager_id,channel,size)
p4_recv(type,from_id,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 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.

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.

7 An Example Program

As an example, we present a simple program whose mainline procedure puts MAXVAL items into tuple space. For each item inserted, it evals the procedure named consumer to process the item, and then extracts an acknowledgement from tuple space indicating that the item was processed. To process an item, consumer simply removes it from tuple space and outs the acknowledgement.

```
#include "sr_linda.h"

#define MAXVAL 1000

main(argc,argv)
int argc;
char **argv;
{
  int primes();
  int last,i,ok;
  struct linda_eval_tbl linda_eval_funcs[2];

  linda_eval_funcs[0].ptr = consumer;
  strcpy(linda_eval_funcs[0].name,"consumer");
  linda_eval_funcs[1].ptr = NULL;

  linda_init(&argc,argv,linda_eval_funcs);

  for (i=0; i <= MAXVAL; i++)
  {
     out("%s%d","msg",i);
     eval("%s","consumer");
     in("%s%d","ack",i);
  }

  printf("mainline exiting\n");

  linda_end();
}


int consumer()
{
  int i,val;

  in("%s?d","msg",&val);
  out("%s%d","ack",val);
  return(1);
}
```

This program works with both versions of the code if we merely replace the include for *sr_linda.h* with *mon_linda.h*. When the program executes *linda_init*, p4 will spawn some number of processes to participate in the execution. The number of processes spawned will be determined by the contents of the p4 procgroup file. The sr_linda version will use one of those processes to manage the tuple space. The mon_linda version will use all processes to evaluate live tuples and coordinate their access to tuple space via monitors; each process will be in a loop looking for live tuples to evaluate. Thus, note that the mainline program does one eval for each number to be examined. Each eval causes the procedure primes to be invoked as part of the evaluation.

To reiterate an important point however, note that if the program is run in a message-passing environment, it can run on a shared-memory machine, and p4 will handle message-passing through the shared-memory. The program could even run on a network of shared-memory machines, and p4 would use shared-memory when possible, passing messages over the network only when necessary.

Table 1 contains the run times for three executions of the program, one in which all communications are handled via monitors, one in which communications are handled via message-passing through shared-memory, and one in which all communications are handled via message-passing over a network. Note that the message-passing versions are slower because all ins and outs must be handled by an extra process, the tuple space manager.

Synchronization Method	Communication Medium	Time in Seconds
Monitors	Shared-memory	3
Message-passing	Shared-memory	25
Message-passing	Ethernet	70

Table 1: Times for Example Program Executions

8 A Semigroups Problem

There exists a class of programs in which communication costs decrease as execution time increases. The *semigroups problem* [11] falls into this category, and thus is a very good candidate for p4-Linda's message passing implementation. A short discussion of an algorithm suggested by [3] follows the problem description.

As input, the program is given 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 223 words.

A p4-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 it 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)
```

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 slave's 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.

Some results for problems of two different word sizes are recorded in Table 2. All processes were running on a shared-memory Sequent Symmetry. The results are promising for loosely-coupled processors also because, as execution time increases, generated words are more likely found in local lists, and communication through tuple space only occurs infrequently.

Number of Processes	Word Size 25	Word Size 36
1	3.5	31.5
2	2.3	19.6
4	2.3	11.3

Table 2: Time (seconds) for Semigroup Problem

9 Future Directions

The p4-Linda implementations provide a minimal set of Linda operations: eval, out, in, and rd. Boolean versions of the primitives might prove useful to 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 would 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 would require 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 a distributed tuple space. 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 one segment of 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 distributed tuple spaces in the monitors model. A Linda kernel described in [10] implements multiple tuple spaces.

10 Conclusions

We have implemented two compatible versions of Linda on top of the p4 portable parallel programming system, one to take advantage of shared-memory architectures, the other to utilize resources of networked machines, offering an advantage in portability. We have described the advantages and disadvantages of each implementation and methods in which the performance of each might be enhanced. We view these implementations as being prototypes and suggest that if there is sufficient interest, we would like to further develop them. The code for these systems is available in the pub/p4 directory at info.mcs.anl.gov.

11 Bibliography

References

[1] S. Ahuja, N. Carriero, and D Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, 1987.

[3] R. Butler and N. Karonis. Exploitation of parallelism in prototypical deduction problems. In *Ninth International Conference on Automated Deduction*, pages 333–343, 1988.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, Mathematics and Computer Science Division, October 1992.

[5] N. Carriero and D. Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[6] N. Carriero and D. Gelernter. How to write parallel programs. *ACM Computing Surveys*, 21(3):323–356, September 1989.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[8] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Language Systems*, 7(1):80–112, January 1985.

[9] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with Upshot. Technical Report ANL-91/15, Argonne National Laboratory, Mathematics and Computer Science Division, August 1991.

[10] W. Leler. Linda meets unix. *IEEE Computer*, 23(2):43–54, February 1990.

[11] E. Lusk and R. McFadden. Using automated reasoning tools: A study of the semigroup f2b2. *Semigroup Forum*, 36(1):75–88, 1987.

DATE
FILMED
10/13/93
END