Design and Implementation of Linda on Transputers

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of
Master of Engineering
in
COMPUTER SCIENCE & ENGINEERING

By
Shekhar K H

Department of Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012

October 1993

Acknowledgements

The project "Design and Implementation of Linda on Transputers" has been successfully completed because of the willing contribution and cooperation of many people associated with the project.

I am very much indebted to my project advisor Dr. Y. N. Srikant for his encouragement and suggestions right through the course of this project.

I am also grateful to Prof. V. Rajaraman for permitting me to work on Transputer based system and for providing excellent computing facilities at KBCS.

My sincere thanks to all my friends who made my stay at IISc a pleasurable one. I wish to thank Ravi for his comments on the preliminary version of this report. I am also thankful to Pawa, Dutta, Joshi and Yeshwant Rao for their company while working at KBCS during my project semester. Special thanks to Manjunath for his delightful company for one and half years.

Abstract

The Linda memory model is becoming a popular inter process communication paradigm for concurrent systems. It is an elegant and simple parallel processing paradigm based on the concept of *generative communication*. In Linda, the processes interact via a shared, global associative memory called *tuple space*.

In this report, design and implementation of C-Linda Preprocessor and Kernel for a four node transputer machine is described. General design issues for Linda systems are outlined, and possible approaches to a distributed implementation are discussed in detail.

Most of the Linda implementations on distributed memory MIMD systems are not flexible. They assume fixed topology. The strategies used in the design are not modifiable. Our implementation is highly modularized and each of the modules can be modified for new application easily.

Many Linda systems on distributed memory computers do not support the eval() operator of Linda. Even if some do, they do not allow recursive functions to use eval() operator. Our implementation is unique in the sense that it does not suffer from these limitations. A detailed design and implementation of this is given.

Until recently, Linda implementations were available for machines which could provide the underlying operating system support. In a new effort, we have written the kernel supporting the eval() operator. The implementation is scalable with the architecture.

Contents

List of Figures

Chapter 1

INTRODUCTION

With the increase in use and importance of Parallel/Distributed computing, the dearth for new, powerful parallel programming languages has increased. The use of Parallel Programming Languages was less initially, due to the success of the parallelizing compilers and "nightmares" of developing and debugging parallel programs. With the advent of better parallel programming environments and availability of powerful constructs for parallel thread creation, communication between parallel threads and synchronization, the use of Parallel Programming Languages has increased.

The lacuna between software and hardware has enlarged further in parallel computers when compared to sequential ones. Given a parallel algorithm, the problem of getting a working program on a parallel computer does not have a satisfactory solution. Researchers have worked out several basic approaches for programming parallel computers. All these approaches can be classified into three conceptual classes for exploiting parallelism and three programming paradigms for implementing parallel programs. The conceptual classes are *result parallelism*, which centers on parallel computation of all elements in a data structure, *agenda parallelism*, which specifies an agenda of tasks for parallel execution, and *specialist parallelism*, in which specialist agents solve problems cooperatively. The programming paradigms use live data structures which transform themselves into result data structures, distributed data structures that are accessible to many processes simultaneously, and message passing in which all data objects are encapsulated within explicitly communicating processes. Programming language for parallel computer should be able to support the above concepts and paradigms. Linda is one such language, in which one can express the parallel algorithms easily [9].

1.1 Why Linda ?

Linda is a simpler, powerful and elegant model of parallel programming. It has been compared head-to-head with other parallel programming models based on message passing, concurrent object oriented programming, concurrent logic languages and concurrent functional programming systems [?] to prove the above point. Linda is a good choice for many reasons.

1. Linda is General: It is general as it goes with the existing languages to produce a hybrid language.
2. Linda is Flexible: It supports all the three programming methods described in the previous paragraph in a straightforward fashion.
3. Linda is Portable: Efficient Linda implementations are available on a wide variety of parallel computers, thus making parallel software developed in Linda portable.
4. Linda is Scalable: Linda abstracts away from architectural details, which means no rewriting of the application if more number of processors are added.
5. Real-life applications have been developed and are running satisfactorily.

1.2 Linda in Detail

Parallel programs when executing have to create and coordinate multiple execution threads. Depending on the programming languages these activities may be done by the underlying system automatically or user has to specify them explicitly in a program. Pure parallel functional languages fall into former category, while Linda belongs to the latter.

Linda is a language which incorporates high level primitives for process creation, communication and coordination. It was initially developed at the YALE university by David Gelernter. Now it enjoys widespread use as its study has disseminated to research institutions. In Linda, all processes communicate through a shared data space called **tuple space**, in contrast to using messages or shared memory locations.

Linda is based on *tuple space model* of parallel programming. It is obtained by adding tuple space operators to any sequential base language. Base language may belong to any of the programming paradigms like Object Oriented, Logic, Functional or Imperative. Thus Linda is unique in the sense that it is compatible with any of the base languages. Base language is used for specifying the computation part of a program. The Tuple Space operators are used for process creation and coordination. Thus the Linda model does not bother about the computation done by each process; it worries only about managing the execution threads. Since the number of Linda operators is very small and as they are easy to use and comprehend, one can swiftly adapt to writing programs using Linda [?].

Two fundamental notions associated with Linda are *Tuples* and *Tuple Spaces*. A tuple is an ordered collection of fields, which is similar to a record in relational database theory. Each field has a **type** associated with it, which has to be one of the valid types allowed in the base language. Tuple Space is a collection of tuples. Many copies of the same tuple can exist in a Tuple Space. It is a global shared object and every process has access to it.

The Tuple Space model allows generative communication. If two processes need to communicate, the sender process creates a tuple and sends it to the Tuple Space. The receiver process picks up the tuple from the Tuple Space. Creating a new process is also done in a similar way. A process that needs to create a second, creates a "live tuple" and sends it to the Tuple Space. The live tuple carries out some computation of its own and then gets converted to an ordinary tuple.

1.2.1 Tuple and Tuple Space

Let,
$\mathcal{T}$: Operator mapping objects(variables or values) to their types.
$\mathbf{T}$: Set of all primitive(i.e. not derived) types allowed in the base language.
V_t: Set of all legal values for an object of type t.
$\mathcal{V}(x)$: Operator mapping the object x to the value associated with it. Then, we can define a tuple of arity 'n' as,
$T_n = \langle t_1, t_2, \ldots, t_n \rangle$ where,
each t_i corresponds to objects in the base language &
$\mathcal{T}(t_i) \in \mathbf{T}$.
$\mathcal{V}(t_i) \in V_{\mathcal{T}(t_i)}$
If the value of the object is not defined, then it is known as *formal* else *actual*. The tuples explained above are known as passive tuples. An **active tuple**(also known as

a live tuple) is one that has a function call along with its arguments as a field.

1.2.2 Tuple Space Operators

Linda - 2 supports six operators. Linda - 3 is extended with two more operators [?]. Since our implementation is for Linda - 2(hence onwards we refer Linda - 2 as Linda), Linda - 3 is not dealt with here. The operators that are included in Linda are for removing tuples from the Tuple Space, putting back a tuple into the Tuple Space, creating a process to execute a procedure in parallel etc. The operators are described in detail.

out(tuple)

This inserts the tuple into the Tuple Space. The process executing this operator is not blocked. When a process executes several out() operators, the temporal order in which tuples arrive at the Tuple Space may be different to the sequence in which out() were executed.

Example: The operation,

```
out( "check", "left", 1.0, "right", 5.0)
```

will place the tuple ⟨ "check", "left", 1.0, "right", 5.0 ⟩ into the Tuple Space.

in(tuple)

This operator is used to extract the matched tuple(Refer page 6) from the tuple space. The process executing this operator gets blocked until the matched tuple is obtained from the Tuple Space. This operator is a destructive operator as it removes the matched tuple from the Tuple Space. If multiple copies are matched, one of them is picked arbitrarily.

Example: In the sample program,

```
main()
{
        int max, min;
        out("key1", 20 , "compute", 10);
        in( "key1", ?max, "compute", ?min);
              ⋮
}
```

the tuple, $\langle$ "key1", 20, "compute", 10 $\rangle$
is placed in the Tuple Space by out() operator, and in() operator removes the same tuple from the Tuple Space. After executing in(), the values of **max** and **min** are 20 and 10. The symbol '?' is used to specify the corresponding variable as a formal parameter.

rd(tuple)

This operator is similar to the in() operator except that the matched tuple is not removed from the Tuple Space. It is a non- destructive version of in() operator. This operator is useful when multiple processes have to share data.

inp(tuple) and rdp(tuple)

These are the predicate versions of the in() and rd() operators. They are non blocked operators and return 1 whenever a matched tuple is found in the Tuple Space, otherwise 0. These operators are useful for processes to communicate asynchronously.

eval(tuple)

This operator resembles the out() operator except that it creates an active tuple. This operator helps in creating multiple processes which can be executed in parallel with the process executing this operator. It evaluates each field in parallel after placing the tuple in the Tuple Space. The result of creating parallel computing tasks. The other operators described earlier are used for coordinating and controlling the processes.

Example: The operation,

```
eval( "nodeinfo", compute(i,j), generate(l,m))
```

will evaluate the procedures 'compute' and 'generate' in parallel and the returned value of these procedures is placed in the tuple. These types of tuples are known as *live tuples* and they help in building live data structures which finally become the passive ones.

1.2.3 Matching of the Tuples

All operators that extract tuples from the Tuple Space are defined in terms of 'Tuple Matchings'[?]. If $\mathcal{R}$ is the tuple template in an extracting operator and $\mathcal{S}$ is the corresponding tuple residing in Tuple Space, then we can say $\mathcal{R}$ matches with $\mathcal{S}$ only if:

- Arity of $\mathcal{R}$ and $\mathcal{S}$ coincide.
- $\mathcal{T}\ (r_i)$ is same as $\mathcal{T}\ (s_i)$ for all fields where $\mathcal{T}$ is a **type of** operator.
- r_i and s_i match in the following condition.

  1. if r_i and s_i are *actuals*, they match iff their values are equal.
  2. if r_i is *actual* and s_i is *formal*, they match unconditionally. The value of r_i is discarded.
  3. if r_i is *formal*, and s_i is *actual*, they match. The value of s_i is copied to r_i if all other fields also match.
  4. if r_i and s_i are *formal*, they do not match.

- $\mathcal{R}$ and $\mathcal{S}$ match only if each of r_i and s_i match.

As explained earlier, if multiple tuples are matched, one of them is selected randomly.

1.3 Elegance and Simplicity of Linda

The elegance and simplicity of Linda is due to message uncoupling. It has both temporal message uncoupling and spatial message uncoupling. This helps while developing parallel programs. The messages are temporally uncoupled as there is no need for both sender and receiver process to be in a state of execution at the same time. The tuple space acts as a temporary dump when the tuples are exchanged. The messages are also spatially uncoupled as the sender process need not know the receiver process's identity and vice-versa. The identity is known indirectly through the tuple contents. Due to spatial uncoupling of the messages, one can use any type of communication from one-to-one to broadcasting.

1.4 A Complete Example

In this section we give a simple example, to illustrate the use of Linda operators. The program is to convert the lower case characters to an upper case characters. The Linda main program creates a process which converts a lower case character to upper case character. Then it sends each character to the created process and receives the upper case character. All the upper case characters are printed on the screen.

```
#include <stdio.h>
#include <ctype.h>
main()
{
    int low_chr, upr_chr;
    extern int lower_upper();
    low_chr = getchar();
    /* Create lower_upper() process. */
    eval("Low_to_Upr", lower_upper());
    for(; low_chr != EOF; low_chr = getchar()) {
        out("lower", low_chr);
        in("upper", ?upr_chr);
        putchar(upr_chr);
    }
}


int lower_upper()
{
    int low_alpha;
    int upr_alpha;
    in("lower", ?low_alpha);
    upr_alpha = toupper(low_alpha);
    out("upper", upr_alpha);
}
```

1.5 Current Status

Linda has been implemented both on shared memory and distributed memory multi computers. Linda operations have been extended to several sequential languages. Linda operators have been added to languages like C, Fortran, C++, Postscript, Scheme and Modula. It has been ported to wide range of parallel machines: shared memory multi-computers like the Encore Multimax, Sequent Balance, Symmetry and Alliant FX/8; Distributed-memory computers like Intel iPSC/2, S/Net, VAX/VMS based local area networks and Transputer based systems. Several real time applications have also been developed using Linda.

Chapter 2

LINDA PREPROCESSOR

In this and the next chapter, our implementation of Linda on a four node Transputer attached to IBM PC/AT is dealt with. The complete Linda environment consists of Linda Preprocessor, Run time Kernel, Debugger and Linda Interface. We have implemented the Preprocessor and the Linda Kernel. In this chapter, details of the Preprocessor implementation is given. We have selected 'C' as the Base Language and all Linda operations are added to it. The flow of getting executable module of a Linda program is shown in the figure 2.1

2.1 A Note on Transputers

The **INMOS** transputer is a **VLSI** design building block for concurrent processing systems, with **OCCAM** as the associated design formalism. It has processor, memory and communication links for direct connection to other transputers on a single chip. Concurrent systems can be constructed from a collection of transputers, which operate concurrently and communicate through links. Since our Linda design runs on **IMS T800**, we will give its hardware and software characteristics.

2.1.1 Hardware Characteristics

IMS T800[?] has an architecture similar to its predecessor **IMS T414**, with better capabilities. It has on chip microcoded Floating Point Unit(FPU) along with memory, links and CPU. The architecture is RISC based and hence it has small a number of instructions, all with the same format.

In T800, the FPU operates concurrently with the CPU. This means that it is possible to perform an address calculation in the CPU while the FPU performs the floating point operation. The communication links are point-to-point serial type and this provides maximum speed with minimum hardware communication. It supports 4 pairs of links, each link being a simplex channel. It has memory block move instructions which are useful in Graphics Operations.

2.1.2 Software Environment

We will explain the software characteristics of Transputer, keeping in view, the implementation of Parallel C on transputers [?]. Transputers support a number of concurrent processes [?]. It executes one process at a time. The process can execute at one of the two priority levels - *urgent* and *less urgent*. The processor will execute an urgent process in preference to a less urgent process, if both are active. It maintains 2 queues for processes, one for each priority. The CPU, time slices between the less urgent processes, but urgent processes are not interrupted until they cannot proceed because of communication. A process can be preempted by executing the preempting instruction by the process itself.

Communication between processes is achieved by using channels. Channels between processes in the same transputer are implemented using memory locations, and channels between processes in different processors by point-to-point links. A process can be compiled and linked without its knowledge of whether the channels are connected to other processes on the same transputer or different transputer.

The complete application can be viewed as a collection of one or more concurrently executing tasks. Each task has its own region of memory for code and data. Tasks placed on same node can have any number of interconnection channels, but on different nodes, it is limited by only physical wires. All the communication channels between tasks have to be defined at compile time itself.

Tasks support multiple threads. The new execution thread then starts executing the code of the pointed-to function concurrently with the thread or task which created it. The new thread runs on the same transputer of the parent task or thread. It inherits the priority, the global memory and the heap of the parent. Parent thread has no control over the created thread.

Parallel C [?] provides *semaphore* functions in its run time library. Semaphores can be used to synchronize the operation of threads and their implementation is such that they do not consume CPU cycles.

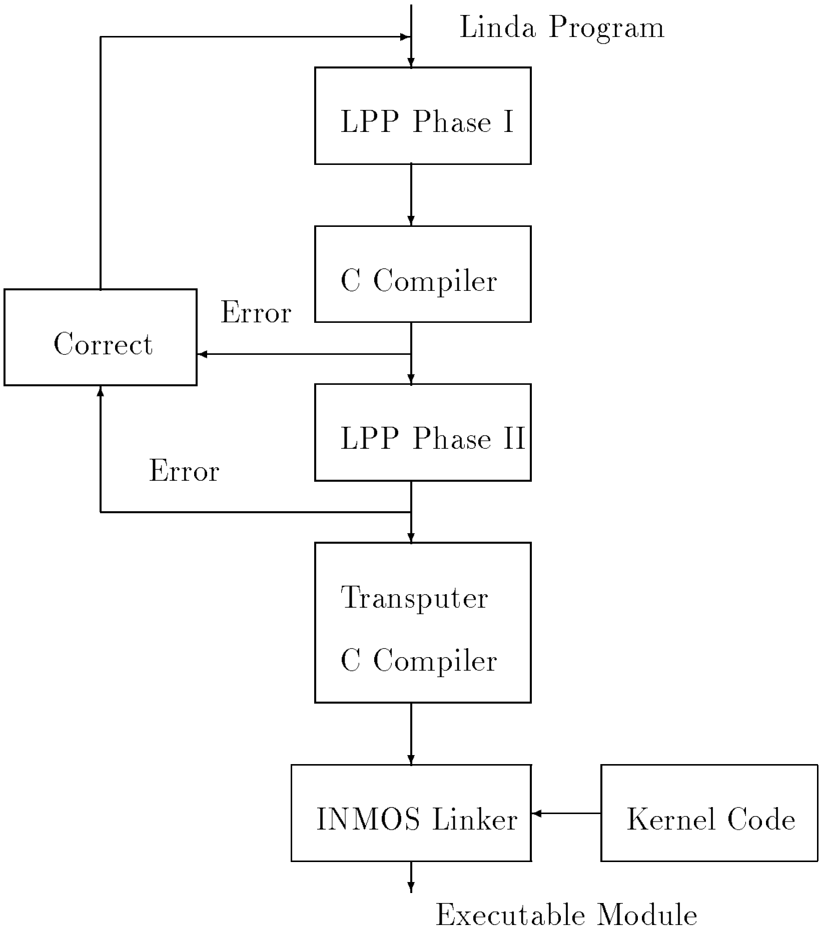

Figure 2.1: Flow of Executable Load Module Creation

2.2 Linda Preprocessor

As explained in the previous chapter, Linda is obtained by extending a base language with new operators. To implement this there are two approaches. One is to write a compiler for the extended language containing Linda operators. Second is to implement Linda operators as function calls to Linda Kernel. Here Linda kernel will be executing as a separate task. In the second implementation, one needs a preprocessor to convert the Linda program to equivalent 'C' program. Then the resulting 'C' program can be compiled with a native 'C' compiler producing the object code for Transputers. Linda compiler is definitely superior and faster when compared to preprocessor, but it takes time to write. Linda Preprocessor (LPP) which introduces an additional pass before getting the executable module, is easier and faster to develop. As our main aim was to develop a reliable test-bed for running Linda programs, we took the second approach.

2.2.1 Linda Preprocessor Functionality

LPP is used mainly for code translation to 'C' code. It does many other functions. The LPP checks for Syntax and Semantic errors in the usage of Linda operations, generates tuple usage information which may be used for Load Balancing, produces the Configuration file for the Transputer 'C' Compiler[?], creates **eval manager** code etc. Each of these functions along with their implementation is explained below.

Syntax and Semantic Error Reporting.

LPP reports syntactic mistakes only with respect to usage of Linda operators. It does'nt check for syntax errors in non Linda statements of the program. Hence it calls the native 'C' compiler for syntax checking before preprocessing. The 'C' compiler checks for all syntax errors in non-Linda statements and if there are no errors the input program is passed to LPP.

The errors identified by LPP are with reference to our implementation of Linda. The error checks made are described below.

1. Our Linda implementation expects the first field in each tuple to be a string. This restriction is because the first field is used for hashing.

2. LPP generates a warning if there is a formal variable in out() operator, as it is very unlikely that one needs a formal variable in out() operator.

3. Maximum number of fields in each tuple is fixed currently at eight. If arity of a tuple exceeds eight, an error is given.

4. The **type** of each field has to be one of the fundamental types supported by 'C' language. Pointers can be passed only as a integer and not as a pointer to some object. Only pointers to **char** are allowed(i.e. strings can be used as fields). All derived types are forbidden.

5. Each function call in an eval() operator, when processed internally, takes one field for each argument and two fields for the function name. An additional field is used to indicate the number of arguments. Thus, for each function call in eval(), LPP adds 3 additional fields in the tuple formed after translation. Even in this case, if the total number of fields exceed eight, an error is reported.

All errors related with undeclared variables, missing of comma, semicolon, opening and closing of parentheses etc. in Linda operators are caught by the native 'C' compiler. This is possible as Linda operators are provided as function calls.

2.2.2 Linda Preprocessor Implementation

The preprocessor generates 'C' program with transformations made to Linda operators. For example, it replaces the character '**?**', which represents a formal field in tuple with '&' which generates an address of the variable. Linda kernel uses this address when it has to fill up the formal variable value at the end of in() or other extracting tuple operators.

Another transformation is, it constructs a string containing **type** information of each field in the tuple similar to printf() routine of 'C'. It uses '%' for representing actual variables and '?' for representing the formal variables. This string is placed as a first field in the transformed tuple. In the second field, it places the name of the Linda operator. From the third field onwards, original field elements of the initial tuple are placed. Each Linda operator which is a function call is replaced with a call to the function oprtr(). The function oprtr() is a generic function provided by Linda kernel and this function handles all the Linda operators in an uniform way.

Example:
The in() operator in the following program,

```
main()
```

```
{
    int ival;
    float fnum;
        ⋮
    ival = 100;
    in("key1", 10.1, ival, ?fnum);
        ⋮
}
```

gets transformed to,

```
oprtr("%s%f%i?f",IN, "key1", 10.1,
                     ival, &fnum);
```

where, '%f', '%i', '%s' in the first string indicates that the corresponding fields in the tuple are of *float* type, *int* type and *string* type. The first string is added in the transformed tuple to facilitate variable number of arguments processing.

The transformation done for eval() operator is slightly different. Along with arguments of the function call, it places the function name and its return value **type** in the transformed tuple.

Example: The eval() operator in the following partial program,

```
double compute(a, b)
int a; float b;
{
      ⋮
}

main()
{
    int ival;
    float fnum;
```

```
        ⋮
    eval("ans", compute(ival, fnum));
        ⋮
}
```

gets transformed to,

```
oprtr("%s%p%d%i%i%f", EVAL,"ans","compute",
                        DOUBLE-TYPE, 2, ival, fnum);
```

where, '%p' in the first string indicates that the corresponding field is a function call and return value is of **double** type. Number of arguments of the function call is placed as another field in the transformed tuple(in the example transformation above, number 2 denotes this).

The LPP is implemented using YACC and 'C'. All 'C' preprocessor directives (like # include) are evaluated in input Linda program as a separate pass and then it is passed to LPP. The LPP expects input program with no syntax errors. The restriction can be removed in future, if better error recovery tokens are introduced in the parser.

LPP places each variable in the Symbol Table along with its **type**. All the symbols entered once in the Symbol Table are not deleted, as it is used even after parsing.

2.2.3 Future Extentions

Whenever the parser comes across a Linda operator, it has to do the following.

1. Creating a "proto-buffer" which contains information about the number of fields in the tuple, their types, the addresses of formal variables, values of actual variables and the corresponding line number of the operator. This information is useful for generating the **trace information** while running the program or while debugging the program at the Linda operator level.

2. It has to place the **tuple type** in the " load balance buffer" if it is not present there along with the function name in which the tuple is present. This information has to be kept with respect to the Linda operator and it should also

store whether the Linda operator is within a loop or not. The tuple type can be identified with **type** and arity of the tuple. At the end of parsing, the " load balance buffer" has to contain information regarding tuple usage for the entire program. The Tuple Space Load balancer can use this information.

3. For every function, within an eval() operator, it has to make an entry in " eval buffer".

4. For every **tuple type** within an out() operator, there should be a corresponding in(), rd(), inp() or rdp() operator having same tuple type. Otherwise, a warning has to be generated, to indicate that there is no matching operator.

5. The functions within an eval() operator may be evaluated anywhere in the network. Since, the environment of the parent which creates a concurrent thread using eval(), is not transferable, no global variables should be allowed in such routines. LPP has to find out these types of errors also.

6. A post-amble code has to be written which works on "load balance buffer" for identifying unmatched out()-in(), out()-inp(), out()-rd() or out()-rdp()) pair of Linda operators.

Currently, LPP runs on the host IBM PC/AT. The code for the Eval Manager to evaluate a function within an eval() operator is generated without using "eval buffer" in an ad hoc fashion. In future it has to be modified to use the "eval buffer".

Chapter 3

LINDA KERNEL

Our main design goal behind the Linda Kernel Development on Transputers has been to develop a modularized kernel. The Kernel design is such that one can easily modify the protocol, load balancer, tuple distribution strategy and routing information to suit any new application on any new machine configuration. All modules in the kernel are logically interconnected and changes in one module may necessitate some changes in other modules for better results.

The main components of the Linda kernel are,

1. Listener, which acts as a gateway with user's main task and processes created by the eval() operator.
2. Channel Coordinator, which is responsible for managing a heap of channels and providing them on request at run time.
3. Protocol Manager, which is responsible for maintaining the tuple space.
4. Eval Manager, catering to the need of evaluating functions that are within eval() operator.
5. Router, for routing the tuples and messages to other transputers.
6. Load Balancer, one for tuple distribution and the other for Parallel evaluation of functions.

All these components are present at every node. The logical interconnection between the components is shown in figure 3.1. The user's task is run on the node to which the host is attached. The process created by the Eval Manager also communicates with the Listener. These Computational processes are shown in the figure 3.1

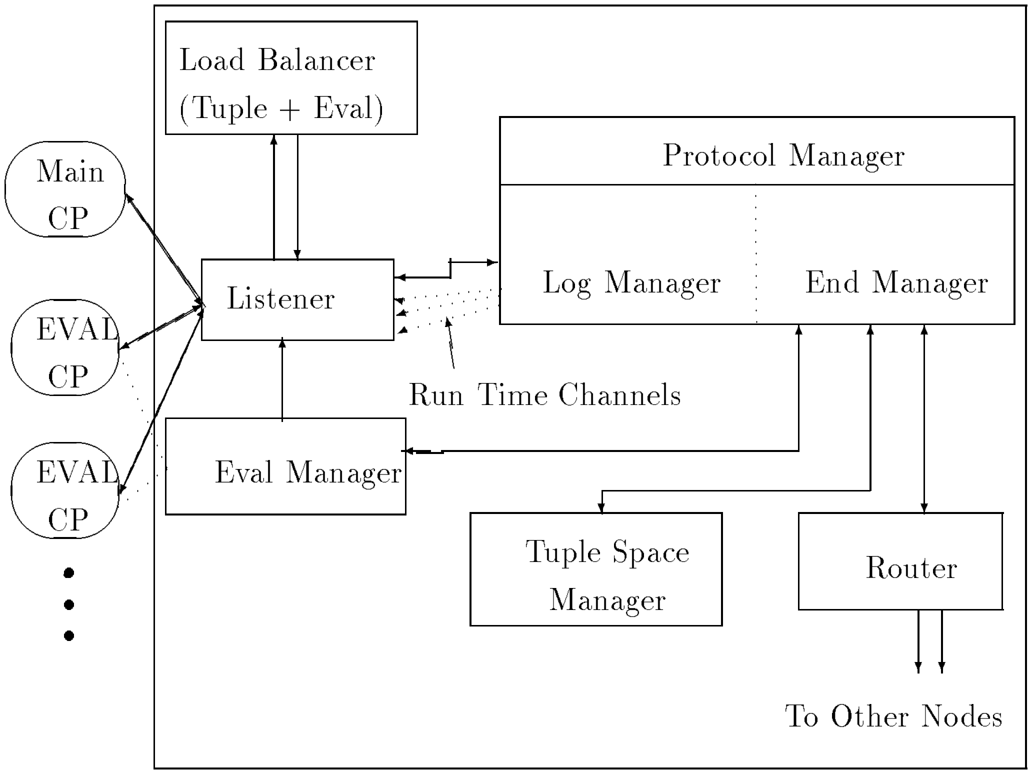

Figure 3.1: Kernel Components that run at each node.

as 'CP'. Only one of them is the user's main process, all the rest are created by the kernel. The kernel consists of several components. The components are implemented using **threads** and **tasks** of transputers. References for additional information on the threads and tasks on transputers is given at the end of this report[**?**,**?**,**?**]. Since tuple is the main object exchanged between kernel components, its structure is described below.

3.1 Tuple Structure

Tuple Structure contains both Control Information and data. The structure is shown in figure 3.2. Tuple data is stored after the Control Word and Type Signature structure.

3.1.1 Control Word Structure

The Control Word is used to store all details about tuple. The contents are,

- Tuple Size: This field stores the total size of the tuple which includes Control Word and Type Signature.
- Tuple Operator: It contains the Linda operator like IN, OUT etc.
- Tuple Arity: Number of fields in the tuple are stored here.
- Source and Destination Node id: Source node id is the node at which the tuple was constructed. Destination node id field is filled up by the Load balancer.
- 1 bit flags for recording whether each tuple field is a formal or actual variable.
- Bits to store additional information like log table and tuple space bucket number and key.
- A field to store the type of tuple, which indicates whether it is a request tuple or the acknowledgement tuple etc.

In our implementation, the Control Word structure takes eight bytes.

Control Word 8 Bytes	Type Signature 4 Bytes	Data Maximum 1010 Bytes

Figure 3.2: Structure of the Tuple

3.1.2 Type Signature Structure

This structure contains **type** information of every field in the tuple. The possible types are the one supported by ‘C’ and some additional ones like FUNCTION-TYPE and STRING-TYPE. Presently, type information of tuples of arity up to eight are supported.

3.1.3 Storing format of the Tuple Data

Tuple data is stored after Control Word and Type Signature. The data is stored in its original format. Integers and Floating point numbers take four bytes and double-precision numbers take eight bytes. Strings are stored with null termination. Tuples are sent and received from other processes as messages in transputers.

In the following we describe in detail the user’s process and its interaction with the Linda kernel.

3.2 User’s Computing Processes

The Linda program consists of one main() function of ‘C’. Multiple execution threads are obtained by using eval() operator. The Preprocessor textually replaces every Linda call with function oprtr() 2.2.2. All eval() processes communicate with the Listener over two pairs of **in** and **out** type of channels which are fixed at compile time.

Oprtr() routine is the one which constructs the tuple. It fills up most of the fields in Control Word and Type Signature Structure, like tuple size, arity, operator etc. It builds up temporary structures storing the addresses of the formal variables of in(), rd(), inp() or rdp() operators. This structure is used for assigning the values when the requested tuple is received.

The **eval processes** created by Eval Manager also call oprtr() routine for transferring tuples. There are two channels fixed at compile time for eval processes to send and receive tuples from the Listener. Since, there can be multiple processes waiting to communicate with the Listener, **semaphores** are used for locking and unlocking of the channels. All eval processes do not wait on the channels fixed at compile time for in() type of operators. They get a channel index from the Listener, at which they wait. Since, Listener and eval processes are having same ancestor, channels at run time can be created and used 2.1.2. Listener gets the channel number from " Channel Coordinator". The pseudocode of the routine oprtr() is,

[This routine is called by the user's process for sending and receiving tuples from the Linda Kernel.]

Pseudocode 3.1 *Function Oprtr(type-string, tuple-operator, tuple-fields)*

OUT1 and IN1: *Channels(ports) over which the main user's process communicates with the Kernel(Listener).*

OUT2, IN2 and OUT3, IN3: *Channels for eval processes to communicate with the Listener.*

MAIN-FLAG: *Set to TRUE, if this routine is called by main user's process, otherwise FALSE.*

1. *Build a list containing information about each tuple field of the passed tuple.*
   (a) *Store the* **type** *of each variable passed.*
   (b) *If the passed variable is an actual one store its value else store the address if it is a formal variable.*
2. *Construct the tuple.*
3. *If (tuple-operator = OUT or EVAL) then*
   (a) *if (MAIN-FLAG = TRUE)*

   i. *Send tuple over the port OUT1.*

   (b) *else*

   i. *Wait and lock one of the pairs of ports IN2, OUT2 or IN3, OUT3.*
   ii. *Send Tuple over the selected port.*
   iii. *Unlock the ports.*

   (c) *exit*

4. *If (tuple-operator = IN or RD)*

   (a) *If (MAIN-FLAG = TRUE)*

   i. *Send tuple over OUT1 port.*
   ii. *Wait for the tuple at IN1 port.*

   (b) *else*

   i. *Wait and Lock one of the pairs of ports.*
   ii. *Send tuple over the selected port.*
   iii. *Receive a channel number from which actual tuple can be received.*
   iv. *Relinquish the ports*
   v. *Wait at new channel number for the tuple.*

   (c) *Send TRUE or FALSE to waiting user's process, if the operator is a predicate operator. Fill up the formal variables if tuple is received.*

   (d) *exit.*

In the following we describe each component of the Kernel in detail. The method of executing the Linda operator by the Kernel is explained in the next section.

3.3 Run-time Channel Manager

The channel manager maintains a pool of channels which is globally accessible to all the threads of the Kernel. Whenever the Listener needs a channel, a request is sent to the Channel Manager. The Channel Manager, sends an index of the array of channels in return. Once the communication is over, the channel is relinquished by sending the index number to the Channel Manager. Then Channel Manager adds the returned channel to the original pool.

Since, any process in the kernel can ask for a dynamic channel to the Channel Manager, the request and relinquish ports are provided with locks.

3.4 Listener

As explained earlier, Listener is the interface routine between user's process and kernel. It communicates with the main Linda process, eval processes and the Protocol Manager. It calls the Tuple Space Load Balancer for the node numbers to which the tuples or tuple requests can be sent. For eval() operators, Eval Load Balancer is requested for a set of nodes. The algorithm for Listener is given below.

[This routine is a interface routine with the user's process and Linda Kernel].

Pseudocode 3.2 *Function Listener()*

IN1, OUT1: *Communication ports with Main User's Process.*

IN2, OUT2, IN3, OUT3: *Communication ports with eval processes.*

OUT4: *Channel over which it sends tuple to Log Manager.*

MAIN-FLAG: *Set to TRUE if tuple is received from IN1, otherwise FALSE.*

INDEX1, INDEX2: *Two run-time channels supplied by the Channel Manager.*

1. *Wait on input from IN1, IN2, IN3.*
2. *Receive the tuple from the selected input port.*
3. *If (tuple-operator = EVAL)*
    (a) *Get a node from Eval Load Balancer to which tuple has to be sent.*
4. *else*
    (a) *Get set of nodes from tuple space Load Balancer to which the tuple has to be sent.*
5. *if (tuple-operator = OUT or EVAL)*
    (a) *Send tuple and nodes to Protocol Manager via OUT4.*

6. *if (tuple-operator = IN or RD or INP or RDP)*
   (a) *INDEX1 ← Request and Get a index from Channel Manager.*
   (b) *if (MAIN-FLAG = FALSE)*
      i. *INDEX2 ← Request and Get a Index from channel manager.*
   (c) *Send tuple, nodes and INDEX1 to Protocol Manager via OUT4.*
   (d) *If (MAIN-FLAG = TRUE)*
      i. *Create a thread to receive the tuple or Boolean value(in case of Predicate operators) from INDEX1. and send it to user's process via OUT1.*
   (e) *else*
      i. *Create a process to receive the tuple from INDEX1 and send it to eval process via INDEX2.*
7. *Repeat from the beginning for ever.*

Threads created for communicating the tuple, relinquish INDEX1 and INDEX2 after sending the tuple. They die out after finishing their computation. Thus Listener process consists of a main process and several light weight transient processes.

3.5 Protocol Manager

It is an important part of the Kernel. The working of the Protocol Manager is quite complex. The main function of the Protocol Manager is to implement the protocol of getting the tuples from other nodes for Linda tuple request operators. The Protocol Manager consists of two concurrently executing Processes. One is the Log Manager and the other is the End Manager.

3.5.1 Log Manager

The Log manager is responsible for getting tuples from nodes for tuple request operators. It implements the protocol. The protocol we have followed is two level modified selective protocol.

The Log Manager maintains a Log Table. It is used to record the request for a tuple. It calls Install() routine for placing the request in the Log. Other operation done on the log table is to search for a request entry when a reply comes back from other nodes.

Structure of the Log Table

It is implemented as a hashed table. Each hash bucket points to a linked list of structures containing the log entries. Each log entry structure contains a list of nodes (referred as node-list hence onwards) to which the request is sent and a list of channel numbers (channel-list). Each channel number corresponds to one user process waiting for a tuple at that channel. If multiple requests come for same type of tuple, a single entry is made in the log table and a new channel number is attached to the channel-list. Every entry in the hash bucket header along with every node in the complete structure is provided with semaphore locks, as the complete structure can be concurrently accessed by many processes. Each log entry structure in a list attached to hash table has a unique key. This is provided for saving time when searching the entry for the second time whenever acknowledgement comes. Each node-list element contains the state of the request associated with that node. The Protocol Manager stores the requests and replies of nodes for a tuple in the corresponding node entry. Log Manager uses this information whenever it restarts the protocol.

Log manager implements the protocol in two levels. The Log Manager sends the requests for the tuple to all nodes identified by the Load Balancer. Corresponding nodes send acknowledgement regarding the presence of tuple in them. Then the Log Manager requests for the actual tuple from one of them. If actual tuple is obtained, it is sent to the waiting process. If negative acknowledgement is received, it restarts the protocol either at the first or at the second level.

In the following, the algorithm of the Log Manager is explained. The Log Manager communicates with the Listener and the End Manager. Acknowledgements are received from the End Manager and they may be the first level or second level ones.

[This routine is responsible for implementing the protocol of exchanging the tuple.]

Pseudocode 3.3 *Function LogManager()*

IN1: *Port number from which it receives tuple from Listener.*

IN2, OUT2: *Communication ports connected to End Manager.*

LEVEL1-ACK, LEVEL2-ACK: *Boolean variable, indicating the first or second level acknowledgement.*

INDEX1: *Run-time channels supplied by the Channel Manager.*

1. *Wait for communication at IN1 and IN2.*
2. *If request from IN1*
   (a) *Receive the tuple.*
   (b) *Receive the set of nodes.*
   (c) *If (tuple-operator = OUT or EVAL)*
      i. *Construct the tuple.*
      ii. *Send the tuple to all nodes received via OUT2.*
   (d) *else If (tuple-operator = IN or INP or RD or RDP)*
      i. *INDEX1 ← Get the Channel number from IN1.*
      ii. *Call Install() routine with tuple, nodes and INDEX1. Install() routine returns the bucket-no and key of the log table entry.*
3. *If request from IN2*
   (a) *Receive the acknowledgement from End Manager.*
   (b) *set LEVEL1-ACK or LEVEL2-ACK based on acknowledgement type.*
   (c) *If (ack-type = LEVEL1-ACK)*
      i. *Call UpdateLogFirst with received tuple.*
   (d) *else If (ack-type = LEVEL2-ACK)*
      i. *Call UpdateLogSecond with received tuple.*

The Install() routine itself sends the First Level request for the tuple if the log entry for the corresponding tuple type is not already present. UpdateLogFirst() Routine and UpdateLogSecond() routine depending on the acknowledgement type of YES or NO, further initiates or restarts the protocol from the beginning.

Install() routine is used for log entry of the requests. Install() routine calls a routine TupleCompleteMatch() for checking whether the log entry tuple is the same as the new tuple request received. Such type of requests are possible when Linda operators are in a loop. The Pseudocode for TupleCompleteMatch() is given below.

[This routine is called for matching two tuple requests, one residing in the log table and the new request tuple.]

Pseudocode 3.4 *Function TupleCompleteMatch(new-tuple, stored-tuple)*

1. *Separate the Control Information from Control Word of each tuple.*
2. *Tuple arity of new-tuple and the stored tuple must be same.*
3. *Tuple operator pairs IN-RD and INP-RDP are compatible with one another. Tuple operators in old and new tuples must be compatible.*
4. *Size of new and resident tuple should match.*
5. *Type of each field in both the tuples must be same.*
6. *Position of formal and actual fields should coincide.*
7. *All values in actual field of both the tuples should agree.*
8. *If all previous conditions are satisfied return TRUE else return FALSE.*

The pseudocode of the Install() routine is given below. If same type of tuple request had not come earlier, it starts the protocol by sending the First Level request to all the nodes. Otherwise, it just updates the channel-list, and UpdateLogFirst() or UpdateLogSecond() routine takes care of getting the tuple for the latter requests.

[This routine is used to enter the request into log table.]

Pseudocode 3.5 *Function Install(tuple-request, nodes, port-no)*

BUCKET-NO: *Log table bucket entry number, computed using hashing.*

1. *BUCKET-NO ← Hash value obtained by hashing on tuple-request contents. Hashing uses Control Word, Type Signature and first field of request-tuple.*
2. *For every element in the list in log-table[BUCKET-NO] do*
   (a) *Check whether the request-tuple matches with the tuple stored in the entry, by calling TupleCompleteMatch() routine.*
   (b) *If they match*
      i. *Append the port-no at the end of channel-list.*
      ii. *Return the bucket-no and key of the log entry made.*

3. *If no tuple match in entire list*
   (a) *Construct a new log table entry structure and append it to end of list.*
   (b) *Create a node-list and transfer all nodes to node-list. Attach node-list to entry structure.*
   (c) *Create channel-list consisting of one element. Store the port-no and then attach it to entry structure.*
   (d) *Get and store the new key at the log entry.*
   (e) *Send First Level Request for tuples to all nodes in node-list. Control Word in each tuple is entered with bucket-no and key so that when ack. comes, finding out the log entry is easier.*
   (f) *Return the bucket number and key of the log entry made.*

whenever an acknowledgement of LEVEL1 comes the UpdateLogFirst() routine is called . The reply may be positive or negative. There are also chances of more channel numbers being added to channel-list requesting a particular type of tuple. If it receives the last acknowledgement(irrespective of LEVEL1 or LEVEL2) it may restart the protocol from LEVEL1 or clean up the log table entry. The algorithm for this routine is given below.

Pseudocode 3.6 *Function UpdateLogFirst(ack-tuple, log-entry-node, Bucket-No)*

NODE-NO: *Node number from which ack-tuple has come.*

ACK-TYPE: *Type of acknowledgement. Like TUPLE-AVAILABLE and TUPLE-NOT-AVAILABLE.*

LAST-ACKN: *Boolean variable indicating whether ack-tuple is the last acknowledgement to come or not.*

TOT-WAIT-PROCESS: *Stores the size of channel-list, which indicates the number of waiting processes for same type of tuple.*

TOT-YES: *Used for storing the number of nodes that have the requested tuple*

TOT-LEVEL2-REQ: *Number of nodes to which second level requests are sent and reply is not received yet.*

1. *NODE-NO ← Get node number from ack-tuple Control Word.*
2. *ACK-TYPE ← Get the type of acknowledgement from Control Word of ack-tuple.*
3. *Store ACK-TYPE in node-list element whose node number is NODE-NO.*
4. *Find out the values of LAST-ACKN, TOT-WAIT-PROCESS, TOT-YES, TOT-LEVEL2-REQ by going through the node-list attached to log-entry-node.*
5. *If (ACK-TYPE = TUPLE-AVAILABLE)*
   (a) *If (TOT-WAIT-PROCESS > TOT-LEVEL2-REQ)*
      i. *Send the Second Level Request for the tuple to the node number NODE-NO.*
   (b) *If (LAST-ACKN = TRUE and TOT-WAIT-PROCESS = 0)*
      i. *Clean the log entry structure completely.*
6. *If (ACK-TYPE = TUPLE-NOT-AVAILABLE and LAST-ACKN = TRUE)*
   (a) *Call Function ProcessLastAckn() with ack-tuple, tuple-entry-node, and other details found.*

Function ProcessLastAckn() is called whenever the last acknowledgement comes. The function clears the log entry if the number of channels in it is zero. That is, all the requests are satisfied much earlier and Log Manager was waiting for just the last acknowledgement to complete the protocol. If all the requests for a tuple are not satisfied, then it resends the requests for the tuple. The algorithm of this routine is given below.

Pseudocode 3.7 *Function ProcessLastAckn(ack-tuple, TupleEntryNode, TOT-YES, TOT-LEVEL2-REQ)*

TOT-YES: *Used for storing the number of nodes that have the requested tuple*

TOT-LEVEL2-REQ: *Number of nodes to which second level requests are sent and reply is not received yet.*

TOT-WAIT-PROCESS: *Stores the size of channel-list, which indicates the number of waiting processes for same type of tuple.*

1. *If (TOT-WAIT-PROCESS = 0)*
   (a) *Clean up the Log entry.*
2. *If (TOT-WAIT-PROCESS > TOT-LEVEL2-REQ)*
   (a) *If (TOT-YES $\neq$ 0)*
      i. *Find out the node that has a requested tuple.*
      ii. *Send Second Level request for the tuple to the node found in previous step.*
      iii. *exit.*
   (b) *else If (tuple-operator = IN or RD)*
      i. *Send First Level Request for the tuple to all the nodes in node-list.*
      ii. *exit.*
   (c) *else If (tuple-operator = INP or RDP)*
      i. *Send FALSE through all the channels in channel-list.*
      ii. *exit.*

The routine UpdateLogSecond() is called for level2 acknowledgements. If the reply is positive, the tuple is sent to all waiting processes and tuples are sent through the channel numbers stored in the channel-list. If the reply is the last acknowledgement, special processing is done by calling the routine ProcessLastAckn(). The Pseudocode of this routine is,

Pseudocode 3.8 *Function UpdateLogSecond(ack-tuple, log-entry-node, bucket-no)*

NODE-NO: *Node number from which ack-tuple has come.*

ACK-TYPE: *Type of acknowledgement like TUPLE-AVAILABLE and TUPLE-NOT-AVAILABLE is stored.*

LAST-ACKN: *Boolean variable indicating whether ack-tuple is the last acknowledgement to come or not.*

TOT-WAIT-PROCESS: *Stores the size of channel-list, which indicates the number of waiting processes for same type of tuple.*

TOT-YES: *Used for storing the number of nodes that have the requested tuple*

TOT-LEVEL2-REQ: *Number of nodes to which second level requests are sent and reply is not received yet.*

1. *NODE-NO ← Get node number from ack-tuple Control Word.*
2. *ACK-TYPE ← Get the type of acknowledgement from Control Word of ack-tuple.*
3. *Store ACK-TYPE in node-list element whose node number is NODE-NO.*
4. *Find out the values of LAST-ACKN, TOT-WAIT-PROCESS, TOT-YES, TOT-LEVEL2-REQ by going through the node-list attached to log-entry-node.*
5. *If (ACK-TYPE = TUPLE-AVAILABLE)*
   (a) *Send the tuple to waiting processes through the ports in channel-list.*
6. *If (LAST-ACKN = TRUE)*
   (a) *If (TOT-WAIT-PROCESS = 0)*
      i. *Free the log entry node.*
   (b) *else*
      i. *Call the ProcessLastAckn() to restart the protocol if required.*

3.5.2 Protocol End Manager

This routine acts as a router for receiving and sending the requests and acknowledgements to to concurrently executing tasks. It communicates with the Log Manager, the Tuple Space Manager and the Router. Its working is simple. The pseudo code for this routine is,

Pseudocode 3.9 *Function ProtocolEndManager()*

IN1, OUT1: *Communication ports with Log Manager.*

IN2, OUT2: *Communication ports with Router.*

IN3, OUT3: *Channels for communication with Tuple Space Manager.*

OUT4: *Send port with Eval Manager.*

1. *Wait for any message at IN1, IN2, IN3*
2. *If communication is at IN1*
   (a) *Receive the tuple*
   (b) *If destination node-id is same as local node-id*
      i. *Send tuple to Tuple Space Manager or Eval Manager based on tuple operator is OUT or EVAL.*
   (c) *else*
      i. *Send tuple to Router.*
3. *IF communication is at IN2*
   (a) *Receive the tuple.*
   (b) *If tuple type is ACK-TYPE*
      i. *Send tuple to Log Manager.*
   (c) *else*
      i. *Send tuple to Tuple Space Manager or Eval Manager based on the tuple operator is OUT or EVAL.*
4. *IF communication is at IN3*
   (a) *Receive the tuple.*
   (b) *If destination node-id is same as local node-id*
      i. *Send tuple to Log Manager.*
   (c) *else*
      i. *Send tuple to Router.*

3.6 Tuple Space Manager

The Tuple Space Manager (hence onwards TSM) is in charge of maintaining the tuples in Tuple Space and coordinating the requests for the tuples. Tuples are stored in a hash table. Each tuple is stored in a tuple structure. Each entry in hash table points to a linked list of tuple structures. Tuple structure contains fields for storing the tuple, number of such tuples, key to identify the structure uniquely. It has also a semaphore variable.

TSM basically is responsible for two operations. One is to place tuples into tuple space and other is to search for a tuple whenever a request comes.

Whenever TSM receives a tuple to be placed in the Tuple Space, it finds out the bucket number by hashing the tuple. Tuple arity, types of each field and first string value of the tuple is used for hashing. Then, it searches for the tuple in the list of tuple structures attached to hash table at bucket number entry. If a tuple of same type is found, then it increments the count field, else it places the tuple at the end of linked list.

TSM searches for a tuple in tuple space whenever requests for it comes. Routine SearchAndDelete() takes care of searching for a tuple in the Tuple Space. If the request for the tuple is first level one, it uses the same hashing method used while placing the tuple to find out the bucket number. It searches for the tuple in the list of tuple structures attached at the bucket number entry of the hash table. If the tuple is found, it sends a positive acknowledgement, after placing the bucket number and key in Control Word of the acknowledgement. If a second level request comes, is uses the bucket number and key stored in Control Word of the tuple and searches for the tuple structure having the same key in the list attached to hash table at bucket number. If the tuple is not found, it sends a negative reply. If it is found, tuple is sent back. If IN or INP is the tuple operator needing the tuple, it decrements the count field of corresponding tuple structure entry. The algorithm for SearchAndDelete() is given below.

Pseudocode 3.10 *Function SearchAndDelete(tuple)*

HASH-NO: *Stores the hash number obtained after hashing the tuple.*

FOUND: *Boolean value storing whether search for a tuple is successful or not.*

TUPLE-TYPE: *Variable to store whether the request is first level or second level.*

KEY: *Variable to store the unique associated with each node in every list attached to hash table.*

1. *TUPLE-TYPE ← Get type of tuple from Control Word.*

2. *if (TUPLE-TYPE = first level request)*
   (a) *HASH-NO = Hashing(tuple)*
   (b) *Search for the tuple in tuple structure list attached to table[HASH-NO]. Set the value of FOUND*
   (c) *if FOUND*
      i. *Store HASH-NO and key of tuple structure in the tuple.*
      ii. *Send the tuple to Protocol Manager.*
3. *if (TUPLE-TYPE = second level request)*
   (a) *HASH-NO, KEY ← Get from Control Structure of the tuple.*
   (b) *Search for a tuple structure having its key equal to KEY in the list attached to table[HASH-NO]. Set the value of FOUND.*
   (c) *if FOUND and tuple-operator = IN or INP*
      i. *decrement the count field in tuple structure.*
      ii. *Return the tuple.*
4. *Store information like tuple-not-available and acknowledgement level in Control Word of acknowledgement.*
5. *return the acknowledgement*

The tuple match algorithm used while searching for a tuple is similar to one explained in Pseudocode number reftup:match. The algorithm checks for whether the arity of two tuples is same or not. Type of each field must be same in both the tuples. For corresponding actual fields data part should coincide. Two formal fields never match. If all the above conditions are met, then tuple match algorithm is successful and it returns TRUE.

3.7 Eval Manager

Implementing the eval() operator, as described earlier is the toughest one on distributed memory MIMD systems. Our implementation for the eval() does not support the specification of the original eval() operator completely. In fact, no implementations of Linda supports this operator completely [?].

The functions within an eval() operator can be executed anywhere in the network. Hence, one cannot use free variables(static or global) within the procedures. Similarly pointers are not allowed.

Input to the Eval Manager is the tuples corresponding to an eval() operator. It contains at least a function name. Function may have some arguments and other fields in tuple may contain some other values. The Eval Manager organizes each such tuple in a Queue and evaluates each tuple in FIFO manner. Each tuple is further broken down to units which can be evaluated independently. The broken unit contains either a function along with its arguments or an ordinary value. Another concurrently executing thread, evaluates each function in parallel. The routine which is created as a thread to evaluate the function is generated by the Preprocessor. The third concurrent process collects the result of the evaluation and forms the tuple. The formed tuple is sent to the Listener and the data structures are cleaned. All the three concurrently executing functions are described in detail below.

The routine eval_manager() is the one which receives the tuple from from Protocol Manager for evaluation. It creates the other two routines, one which evaluates each function concurrently and other which collects the result of evaluation. All these routines are synchronized using semaphores. The eval_manager() appends the received tuple at the end of Queue and calls the routine which breaks the tuple into independent units ideal for evaluation. Each tuple can have multiple function calls with different arguments with different type of return values.

Each independent unit created for every tuple, has fields to store the unit type(like FUNCTION-TYPE or ordinary field), name of the function, return value of function if unit is for a function. In case of function, arguments are also stored as a list attached to unit structure. The algorithm to build this structure from a tuple is simple. Once, the structure is built, a semaphore " READY " is set. Parallel evaluator which waits on this semaphore, starts evaluating.

The function EvalFuncParll() evaluates the functions stored in unit structures concurrently. It initially waits for something in the Queue. Once a node is there in the Queue, it waits on READY semaphore of the first node in the Queue. READY semaphore is set, only after breaking of the tuple to unit structure. Once a Queue node is ready for evaluation, it starts evaluating each unit structure one after another. Whenever it finds a node of FUNCTION-TYPE, it creates a thread to evaluate the corresponding function in parallel. The function eval_function() executes the function and places the returned result of the function in the same structure where it picked up name of the function. Since eval_function() executes in parallel with EvalFuncParll() many eval_function() routines will be created evaluating each function concurrently. The routine eval_function() is generated by the Preprocessor. In the following, a simple example of 'C' code generated for an eval() operator by the Preprocessor is given to illustrate the principle.

```
main()
{
    int a;  float b;
    extern int compute();
    extern float generate();
            ⋮
    eval(compute(a),generate(b));
            ⋮
}

int compute(a)
int a;
{
    ⋮
}

float generate(b)
float b;
{
    ⋮
}
```

The code generated for eval_function() consists of,

```
void eval_function(node)
EvalTupleNode * node;
{
```

```
    extern int compute();
    extern int generate();

    if(strcmp(node->str, "compute")== NULL){
        int a1; int ret_value;
        Get_Arg(node, 1, &a1);
        ret_value = compute(a1);
        StoreRetVal(node, ret_value);
        thread_stop();
    } else if(strcmp(node->str,"generate" == NULL){
        float a1; float ret_value;
        Get_Arg(node, 1, &a1);
        ret_value = generate(a1);
        StoreRetVal(node, ret_value);
        thread_stop();
    }
}

int compute(a)
int a;
{
    ⋮
}
float generate(b)
float b;
{
    ⋮
}
```

In the above, node is a pointer to a structure containing information regarding the function to be evaluated. The routine Get_Arg() gets the argument from the node and the routine StoreRetVal() stores the returned value of the function in the node structure. These two routines are library routines of the Linda Kernel. Whenever, eval_function() is called with a corresponding node of FUNCTION-TYPE, it evaluates the function and places the return value in the node. All the routines that are within eval() operator are copied to file containing the routine eval_function() by the preprocessor.

The routine EvalFuncParll() executes the routine eval_function() for every unit structure containing function. The pseudocode of EvalFuncParll() is given below.

Pseudocode 3.11 *Function EvalFuncParll()*

READY: *Binary semaphore indicating that a particular entry in Queue of tuples is ready to be evaluated.*

FINISHED: *Binary semaphore which is set once the evaluation of particular tuple is complete(Tuples may have several functions).*

1. *Wait for at least one entry in Queue.*
2. *Select first node in the Queue.*
3. *Wait on READY semaphore in selected node of the Queue.*
4. *For each node in unit structure of tuple do*
   (a) *if (node-type = FUNCTION-TYPE)*
      i. *thread-start(eval_function, node)*
5. *Set semaphore FINISHED*
6. *Select the next node from Queue*
7. *Repeat from 3 to previous step for ever.*

The function FreeUsedTp() waits on the semaphore FINISHED. Once the parallel evaluation of each function in the tuple is over, FreeUsedTp() goes through the nodes of evaluated tuple and collects the returned values. It forms a tuple and sends it to the Listener. It then frees the complete structure including the Queue entry. The routines eval_manager(), EvalFuncParll() and FreeUsedTp() execute in a locked fashion waiting for each other to complete.

3.8 Implementation of the Linda Operators

Most of the implementation details of the Linda operators are covered in previous and present chapter. Here we give how each operator affects the Kernel components.

The user process which executes the out() operator, sends a tuple to the Listener. The Listener gets a set of nodes from the Load Balancer and sends the tuple and nodes to the Protocol Manager. The Protocol Manager forms as many tuples as there are nodes obtained from the Listener and sends it to local or other node Tuple Space Managers based on node id via the Router. Whenever a OUT tuple is received from the Router at any node, it is sent to the Tuple Space Manager to place it in hash table.

The user process which executes the in() operator gets blocked until it receives the matched tuple. The oprtr() routine stores the details about the call made. It then forms the request tuple and sends it to the Listener. The Listener gets a set of nodes from the Load Balancer to which the request tuple has to be sent. If the in() operator was executed by eval() process, then it gets a channel from Channel Manger and sends the tuple, nodes and channel to Protocol Manager. It then waits on at the channel for the tuple.

Protocol Manager, sends the requests to all the nodes given by the Load Balancer and waits for the tuple. It uses the log table for entering the status of enquiry to each node for the tuple. Once the tuple is received, it is sent to waiting process through the channel number which was provided earlier. The Tuple Space Manager is at the other end of request for the tuple. It searches the tuple space and sends the tuple if it is found or a negative acknowledgement. In case of inp() operator, if for the first time, the issuing Protocol Manager receives a negative reply from all the nodes, it returns FALSE to waiting process. It does not restart the protocol.

Rd() and rdp() operators are implemented in a similar way as that of in() and inp() operators. The only difference is that the Tuple Space Manager does not delete the matched tuple from the Tuple Space.

The Eval() operator implementation differs with other operator implementations. The Listener calls a different Load Balancer for eval() operators. This Load Balancer is responsible for maintaining equal load on all nodes and also it should try to match the execution of function on the node at which all the required tuples are present locally. Eval() tuples are not sent to Tuple Space Manager. It is maintained and evaluated by the Eval Manager. Once the evaluation is over, the formed tuple is sent to the Listener.

3.8.1 Recursion

Most of the Linda implementations do not support recursive functions having eval() operators. Our implementation overcomes this limitation. This is because, all the functions that are evaluated are copied to every node.

There is also a possibility of a function calling itself within an eval() operator. This is not exactly recursion, as the calling procedure is not going to wait for the eval() operator to complete. This we call **Pseudo Recursion**. Linda system may create a new instanciation of the same procedure somewhere in the network. Working of this is similar to the ones within eval() operators. One needs this facility for many applications involving distributed tree data structures.

3.9 Router

The main job of the Router is to identify whether a tuple it receives is for the current node or to some other node. The Router is implemented with the help of router table. The Router has five inputs and five outputs channels. Four pairs of channels are connected to other nodes one in each direction. The fifth pair is used to communicate with the Protocol Manager of the local node. The router table gives the channel number when it is provided with the source and destination node of the tuple. For different configuration of the transputer processors only router table information has to be modified.

Chapter 4

CONCLUSION

We have implemented Linda on a four node transputer board connected to IBM PC/AT. Our design is quite flexible in the sense that it can be easily ported to bigger machines. Most of the current implementations of Linda on distributed memory MIMD systems do not provide the eval() facility. Even if some do, they do not allow recursive routines to have the eval() operator. Our implementation is unique in the sense that it allows recursion within eval() operators.

The design methodology we have followed is quite modular. Most of the current Linda implementations assumes a fixed topology. At the same time the strategies used for Tuple Space Distribution, Load Balancing, Routing, Protocol enhancement etc. are alos not modifiable. The disadvantage of such system is that some applications developed in Linda may not run efficiently. This is because, some of the schemes as enumerated above depend on the actual implementation. The design methodology we have followed is quite modular. One can easily modify and fine tune ny component to suit any new application swiftly.

4.1 Future Work

Since our main idea was to get a test bed for programming in Linda, more onus was given on reliability and working of the Linda environment. There are many places in the implementation where it can be still fine tuned especially in concurrent operations on linked lists.

Currently, Kernel implementation is complete. Preprocessor has been designed and is in the final stages of implementation, which will be ready within a short time. One can further add Linda debugger and User Interface to complete the Linda environment.

No current implementations as reported in Literature supports variables of derives types withing Linda operators completely. Our implementation supports only array of char for string types. Some of the implementations support array of base types. One can extend the Linda system to support the complete derived types. For this, every user type definitions have to be made available on every node.

The Preprocessor is implemented over a commercial 'C' Compiler. If all the Linda operators are included within a compiler, the system will be faster.

Bibliography

[1] Zenith,E.,S. Linda Coordination language; subsystem kernel architecture (on transputers). Yale University Department of Computer Science Research Report 794(May 1990)

[2] MacDonald, N,.B. A Distributed Linda Kernel. Edinburgh Concurrent Supercomputer Project Research Report ECSP-TN-33.

[3] Leler., W. Linda meets Unix. IEEE Computer., Feb. 1990.

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

[5] Gelernter, D. Generative Communication in Linda. ACM Trans. Prog. Lang. Sys. 7,1(1985),pp.80-112

[6] Carriero,N., and Gelernter, D. Linda in context. Comm. of the ACM. 32, 4 (Apr.1989), pp. 444-458

[7] Bjornson,R. et.al. Linda, the Portable Parallel. Yale University Department of Computer Science Research Report 520(Feb. 1987)

[8] Gelernter., D. Multiple tuple spces in Linda., invited paper at PARLE 89.

[9] Carriero,N., and Gelernter, D. How to write parallel programs: A Guide to the perplexed. Yale University Department of Computer Science Research Report 628 (Nov.1988)

[10] Whiteside, R,.A., Leichter, S., A. Using Linda for Supercomputing On a Local Area Network. Yale University Department of Computer Science Research Report 638(Feb. 1987)

[11] Carriero,N., and Gelernter, D. The S/Nets's Linda Kernel. ACM Trans. Comput. Sys.(May 1986),pp. 110-129.

[12] Enbody,J.,R. and DU., C.,H. Dynamic Hashing Schemes. ACM Compu. Sur. Vol. 20,2,(Jun.1988).

[13] Pearson K., P. Fast Hashing of Variable Length Text Strings, Comm. of the ACM.,33,6.(Jun.1990).

[14] Arango., M., and Berndt., J., D. TSnet: A Linda Implementation for Networks of Unix-based Computers., YALEU/DCS/RR-739.,(Aug. 1989).

[15] Shekhar.,K., H., Design and Implementation of Linda on Transputers., ME Project Report, CSA, IISc, Bangalore.(Jan 1990).

Appendix A

EXAMPLE

Here we give a complete example of Linda program. The example is picked from Numerical Analysis. It is Numerical Integration based on the Trapezoidal Rule. The integration is done using the following formula,

$T = \int_a^b f(x)dx \approx h(\frac{1}{2}f(a) + f(x_1) + f(x_2) + \cdots + f(x_n) + \frac{1}{2}f(b))$,

where, $h = \frac{b-a}{n}, x_0(= a), x_1, x_2, \ldots, x_n(= b)$ are end points of sub intervals of integration. 'n' is the number of intervals. Each $x_j = x_o + jh$.

Our implementation of the same is given below. The main Linda program creates 'n' functions in parallel. Each concurrently executing function evaluates the function 'f' at the sub interval passed as parameter. The main process collects the result of evaluation and finally adds the value. The program is given below.

```
main()
{
    double low_limit,upr_limit;
    double fst_val, last_val;
    double sub_value;
    int j;
    int no_of_intervals;
    double step_size, value;

    low_limit = 1.0;
    upr_limit = 10.0;
    no_of_intervals = 20;
    step_size = (upr_limit - low_limit)/no_of_intervals;
```

```
    for( j=0; low_limit + j * step_size <= upr_limit + step_size ;
                          j++)
        eval("int", j, math_func(j, low_limit + j * step_size));

    for(j = 1; j <= no_of_steps; j++) {
        in("result",j,&sub_value);
        value += sub_value;
    }
    in("result",0,&fst_val);
    in("result",j+1, &last_val);
    value = step_size * (fst_val/2.0 + value + last_val/2.0);
    printf("Value after Integration is:%d",value);
}

int math_func(count,val)
int count;
double val;
{
    double ret_val;
    ret_val = (power(val, 4.0) + power(val, 2.0)) / power(val,3.0);
    out("result", count, ret_val);
    return(0);
}
```

A.1 Results