A DEVELOPMENT ENVIRONMENT FOR PARALLEL ALGORITHMS BASED ON LINDA

By

MOHAMMED A. AL-ABDULKAREEM

Bachelor of Science in Computer Science

King Saud University

Riyadh, Saudi Arabia

1988

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1993

A DEVELOPMENT ENVIRONMENT FOR PARALLEL ALGORITHMS BASED ON LINDA

Thesis Approved:

[illegible]

Thesis Advisor

Blayne E. Mayfield

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGMENT

I wish to express sincere appreciation to Dr. K. M. George, my advisor for his scientific guidance and continuos encouragement throughout my graduate program. I thank Dr. B. Mayfiled and Dr. K. A. Teague for serving on my graduate committee and for their scientific guidance. My thanks also go to Dr. Miller for his help and suggestions.

I am also thankful to my respected parents and all my family members for their love and support. A special thank goes to my wife, Muna, for her encouragement and sacrifices.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

In the past decade parallel computers have become more important and popular because of the need to process huge and complicated scientific computations. Therefore, parallel programming is gaining the same popularity and importance. Many parallel programming languages are proposed and implemented which are designed to support software development in a parallel environment. Consequently, parallel algorithm development has emerged as an important area of research. Attention is also focused on the development of tools and environment for the development of parallel programs.

This thesis focuses on the design and implementation of a development tool for parallel algorithms written using the Linda™ approach[1]. This tool is referred by the name "Linda Tool" in this thesis. It is implemented on IBM® personal computers with Microsoft® Windows™. Tools related to the Linda model of parallelism have been developed on other systems also. But, to my knowledge this work is the first one on personal computer environment. The design of Linda Tool includes several subsystems. These include an abstract machine, a debugger and a user interface.

[1] Linda is a registered trademark of Scientific Computing Associates.
IBM is a registered trademark of International Business Machines Corporation.
Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

This tool is designed to help the user understand the algorithm behavior in parallel processing environment and assist in the design of correct algorithms. "Linda Tool" may be used as a teaching tool to allow experimentation. It can help the students understand the concepts of parallel and concurrent processing in general, and the Linda approach in particular.

This thesis contains seven chapters including this. The second chapter reviews some of the work related to the thesis. It covers the following areas: parallel languages, simulation of parallel machines, and debugging tools. The third chapter presents the high-level design and the instruction set architecture of the abstract machine to be simulated. Chapters four and five, describe the design and implementation of the system including the simulator and the debugger. Chapter four covers the simulator design and implementation for the processors and tuple space. Chapter five covers the debugger design and implementation including the programming of Windows interface. The sixth chapter describes the system from the user point of view. It includes the description of the main and sub menus and the child windows. The last chapter provides suggestions for future improvement of the system and conclusion.

CHAPTER II

LITERATURE REVIEW

In this chapter the work related to this project are reviewed. The areas reviewed are broadly classified as parallel languages, simulation and debugging tools.

Parallel Languages

In parallel computing we need computers that have more than one processor to do the computations in parallel. Parallel computers may do parallel computation in different ways. Some can execute the same instruction on different data at the same time. For this purpose, sophisticated compilers may be used to make sequential programs run on parallel computer. However, the power of compilers is limited, and programming the solutions of difficult problems on the parallel machines reuires parallel programming languages. Willie Schatz [Wil89] calls it a good news that several parallel machines are being developed and at least a few programs, mainly applications, have been converted to run on these machines. Mani and Kesselman [KMa91] predict that in the future, because of the continuos advance in VLSI, parallel software and communication technology, it will be less difficult to develop efficient parallel program.

Many parallel programming languages have been developed by researchers, some of which are extensions of sequential languages such as C and Pascal with parallel or concurrent statements. There are others built to be parallel languages such as Ada. The CSP programming notation (1987) provides a theoretical model for parallel programs. CSP has influenced the development of several programming languages such as Occam which is a derivative of CSP. Occam is a parallel programming language associated mainly with transputer and contains a small number of mechanisms: assignment, input and output. Unlike Occam, SR (Late 1970s) contains a large variety of mechanisms. SR can be used to implement algorithms for both distributed and shared memory computations. Lo and others [VLo90] described LaRCS, a language that enables users who program parallel algorithms to specify information about static and dynamic communication behavior of the algorithm. Information provided by the user is used in mapping parallel computations to processors.

A radically different approach to parallel programming is proposed by Gelernter in Linda [Nic89]. Linda is not a programming language rather it is an approach for parallel programming. According to Sudhir and others [Suh86] Linda operators when added to any language turns that language into a parallel programming language. Linda centers on a special memory model called the tuple space "TS", the unit of storage is called a tuple rather than a byte or bit or a full word. The units of this memory are accessed by logical names and not by addresses. A logical name is any combination of tuple values. The tuple space consists of two types of tuples. Gregory Andrews [Ger91] describes the two types as passive data tuples and active process tuples. Process tuples are routines executed asynchronously. The active tuple is generated by the operator *eval* which creates a process tuple. The following example shows the usage of the eval operator :

Consider the statement

eval("tag",$expr_1$,$expr_2$,....,$expr_n$)

In the above statement, "tag" is a string of characters and each expression can be either a statement or a function. The result of the evaluation of the functions will become later a passive data tuple and stays in the tuple space until a process removes it using the get operation. The other operations are related to data tuples only. They are[2] :

1. readt : This operation reads a tuple from the tuple space, and the tuple is not removed from the tuple space. If the matching tuple is not in the tuple space the process execution is suspended until a matching tuple become available in TS.

2. put : This operation places a new tuple in the tuple space, TS may contain more than one tuple with the same contents.

3. get : This operation removes a tuple from TS. If no matching tuple is found the execution of the process is suspended until the tuple become available in TS.

Figure 1 illustrates the three operation and the following are examples of the usage of these operations:

readt("tag",$Item_1$,$Item_2$,..,$Item_n$)

put("tag",$Item_1$,$Item_2$,..,$Item_n$)

get("tag",$Item_1$,$Item_2$,..,$Item_n$)

[2] In this thesis, put(..) is used instead of out(..), get(..) instead of in(..) and readt(..) instead of read(..).

Furthermore, the tuples in Linda can not be updated [Sud86] but they may be removed, updated and then returned to TS. This makes it possible for many processes to access TS simultaneously.

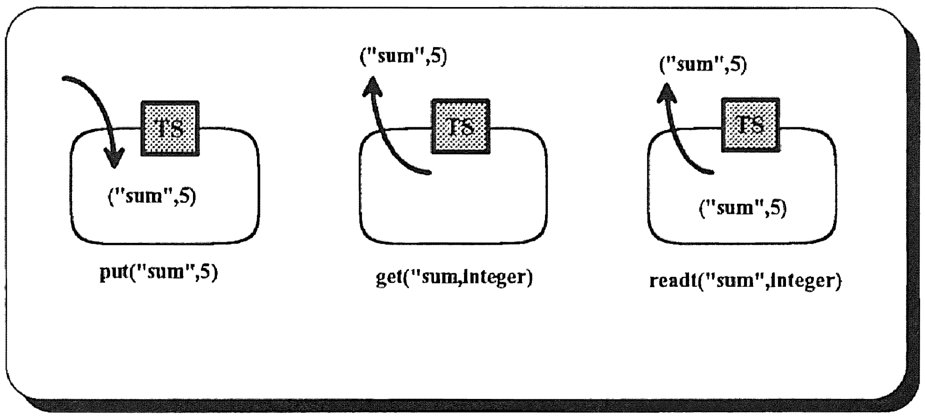


Figure 1. The Three Operations put, get and readt.

Simulation of Parallel Machine

Simulators are used to predict the behavior of systems (machines) in order to save time when designing implementation and to aid in experimentation. Even though simulation alone can not guarantee the success of the design [Mar86], a simulator can be used to understand the behavior of a system, and to provide some details of weakness and strength of the system [John91, VBar90].

One good example for use of simulation is the Chief project [John91] which provides a powerful environment to study parallel systems. Chief provides an integrated set of tools used to create, run, debug and analyze simulations of parallel computer systems. The system has a powerful graphical user interface based on X11 window system. The system has two inputs: the benchmark programs and an architecture specifications. The simulation components are designed using high level language. CARL [Carl90] is a Computer Architecture Research Language used to describe the simulation models of computer architecture in the Chief project. Another example of simulation is the distributed parallel simulation of Hopfield's Neural Network. Barbosa and Lima [VBar90] provide a design and implementation of one class of neural networks introduced by Hopfield. In their paper [VBar90] they describe how to perform the simulation of Hopfield's networks in a generic distributed system for parallel processing and how they implemented the simulation in Occam. Finn and others [AMF91] presented a methodology for modeling and simulation of a multiprocessor architecture. The goal of simulation is to predict early the multiprocessor performance to be used by hardware and software developers. The simulation was based on commercial and custom programs. A custom simulator called TIME_EST was written and used to predict the actual execution time. They used also, a custom program for analysis namely PARAMETERS and STALL . The multiprocessor performance was evaluated using modeling and simulation in (ADAS) an Architecture Design and Assessment System.

Fredrik Dahlgren [Fre91], explored a simulation model that supports a very accurate modeling of multiprocessors with a hierarchical, packet switched interconnection network and private cache. In his report he presented the design and implementation of an MIMD program driven simulator that executes real code. The modeling of the multiprocessor can be very accurate because every action in the processors is simulated, and therefor the architecture dependency problems does not exists.

An efficient method for simulating instruction sets was described in [Chi91]. The method aimed to reduce execution time for instruction set simulators. The method uses compilation approach to map the assembly language of the simulated architecture to the real hardware. This method uses the C in-line macro. Each instruction, to be simulated, is written as a macro and the assembly language is coded as C functions. The paper [Chi91] included a solution for the branching problem that will arise when using the compilation approach.

Monitoring and Debugging Tools

"Debugging is said to be the latest established area in software development" says Keijiro and others [Keij91], that's why most software houses attach a debugger to its compilers. Debugging of sequential programs is based on inserting break points and test the data until the erroneous instruction is found, also it is based on the centralization of control [FB83]. According to Araki and others [Keij91] the most primitive way to debug programs is to insert debugging statements to print the value of data and what statement was executed. On the other hand, the debugging of parallel programs is more difficult because it is difficult to repeat the execution that outputs an error [Keij91].

Also when a debugger is active it may affect the timing of the processors. With the new development in graphical user interface, debuggers need to be implemented using these user interface to be easy to learn and remember. Bovey [JDB87], describes the *ups* source level debugger which use the graphical interface. The debugger was implemented to debug the C and Fortran source code. Bovey also, gives an overview of some design decisions for a debugger in general and graphical interface as well.

Griffin and others [Jam88] have described a debugger for parallel processes called *mtdbx* . The debugging system mtdbx is based on the Sun Microsystems, Inc. debugger *dbx/dbxtool.* It is a source level debugger for C, Fortran, Pascal and Modula 2. Because the system was developed for multiprocessing research at Los Alamos National Laboratory (LANL), in which scientific code are Fortran based, the debugger was used to debug FORTRAN codes on a Sun workstation using a parallel processing simulator, a window and mouse based debugging tool, and real time display routines. The window based system has a master control window that runs the users main program, a window for each active process and a window for task state display, to allow studying the behavior of parallel processes.

VIFOR (Visual Interactive FORtran) [Vac90] is another Fortran based tool. This tool displays the Fortran program in two forms: source code and graphical representation. In the graphical form the program is represented as graph of icons and lines connecting the icons. The graph has two columns one for data icons and the other for main program, subroutines and functions icons. The connecting lines and arrows represents the relations. The data model used in VIFOR is very simple, it has four data classes and three relations. The two main classes are: modules and declarations. The declarations are divided into subclasses: processes and commons (global data). The three relations are: belong to, call and reference.

Another tool is CBUG [Jas85], a C source level debugger designed to debug concurrent processes that uses UNIX system calls fork(), exit() and wait() for concurrent execution. CBUG provides a number of the basic debugging tools such as: single stepping, breakpoints, snapshots and execution tracing. The experimental version of CBUG was implemented using debugging hooks in the source code. The source code is compiled with the debugger. The CBUG tool uses a windows friendly user interface to make the interface easy to understand and memorize.

Durra [Denn89] is a language to describe the tasks to be initiated and executed as concurrent processes for constructing a distributed application running on networks of heterogeneous processors. The Durra application debugger / monitor is used to find bugs in Durra applications, tune performance and control the execution of the application. Durra debugger / monitor had two levels of debugging: the application level and the source level. The application level provides the abstracted Durra view, in which the tasks are treated as black boxes connected together. In the source code level, the Durra monitor / debugger will not debug the source code of different languages on heterogeneous processors, but it will allow the user to use the existing language source level debuggers.

A debugger for MuTEAM Was discussed in [FB83]. MuTEAM is a concurrent language based on CSP. The debugger has two distinct tools: one is a sequential debugger to find the errors in sequential functions and the second is to compare the behavior of the program with a given description of the program behavior. In the sequential debugger, the user may define the value of the messages received by the debugged process.

Some of programming tools are not only used for debugging and monitoring, but also for teaching and demonstration purposes. Vestal is an instructional tool that provides a graphical animation of concurrent programs written in Ada. Vestal (Visual educational system for tasking in Ada language) has two dimensional color animation used for teaching Ada concurrency. The system is based on graphical workstation supporting windows interface. The Ada tasks and other concurrency elements are represented using graphical symbols. A good feature is the use of colors, each task has a color and all symbols related to the task have the same color. One drawback is that the preprocessing of the source program to be animated is done by hand, also some of the Ada concepts does not have an adequate graphical representation.

The work presented in this theses has been influenced by ideas found in the literature described in this chapter. Insted of following one scheme or another, new ideas have been developed and implemented.

CHAPTER III

THE ABSTRACT MACHINE

This chapter is devoted to the description of the abstract machine. The first section describes the high-level design of the abstract machine, and the second section describes the instruction set architecture of the abstract machine and the interpretation scheme.

The Design of The Abstract Machine

Parallel algorithms based on the Linda model need to satisfy specific requirements. Some of the requirements are: a multiprocessor parallel system to efficiently run parallel algorithms, an associated abstract memory to represent the tuple space and a kernel to control and handle the access of tuple space. The abstract machine is designed to satisfy these requirements. The machine model consists of a set of independent identical processors and a tuple space. The model is a multiprocessor architecture in which each processor has its own memory where the instructions and data are stored. The processor architecture is based on Von Neumann model and the organization of a processor is shown in Figure 2. The processor has a control and computing unit that includes three special purpose registers: namely instruction pointer, jump flag and busy flag. The instruction pointer (IP) contains the address of the instruction to be executed. The jump flag (JF) contains the result of compare instruction that determines the action of jump instruction. The busy flag indicates whether the processor is loaded and running or not. Each processor also has an Input/Output channel to accept the user's input and to produce output to the user. The instructions for a processor operates on data stored in the processor memory or operates on tuples located in TS. The conceptual structure of the abstract machine is shown in Figure 3.

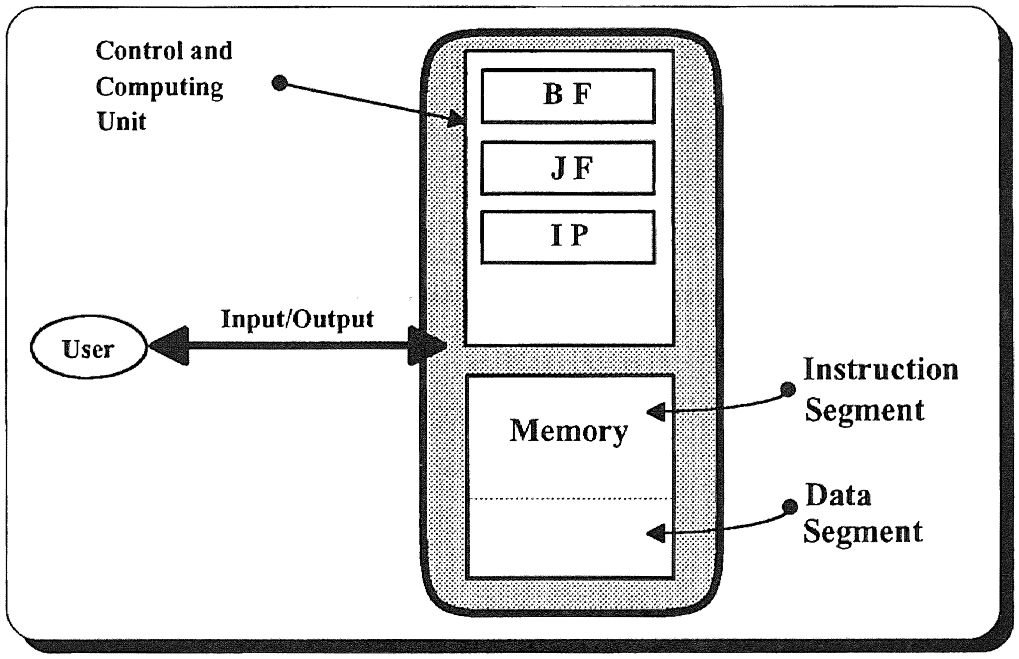


Figure 2. The Processor Structure.

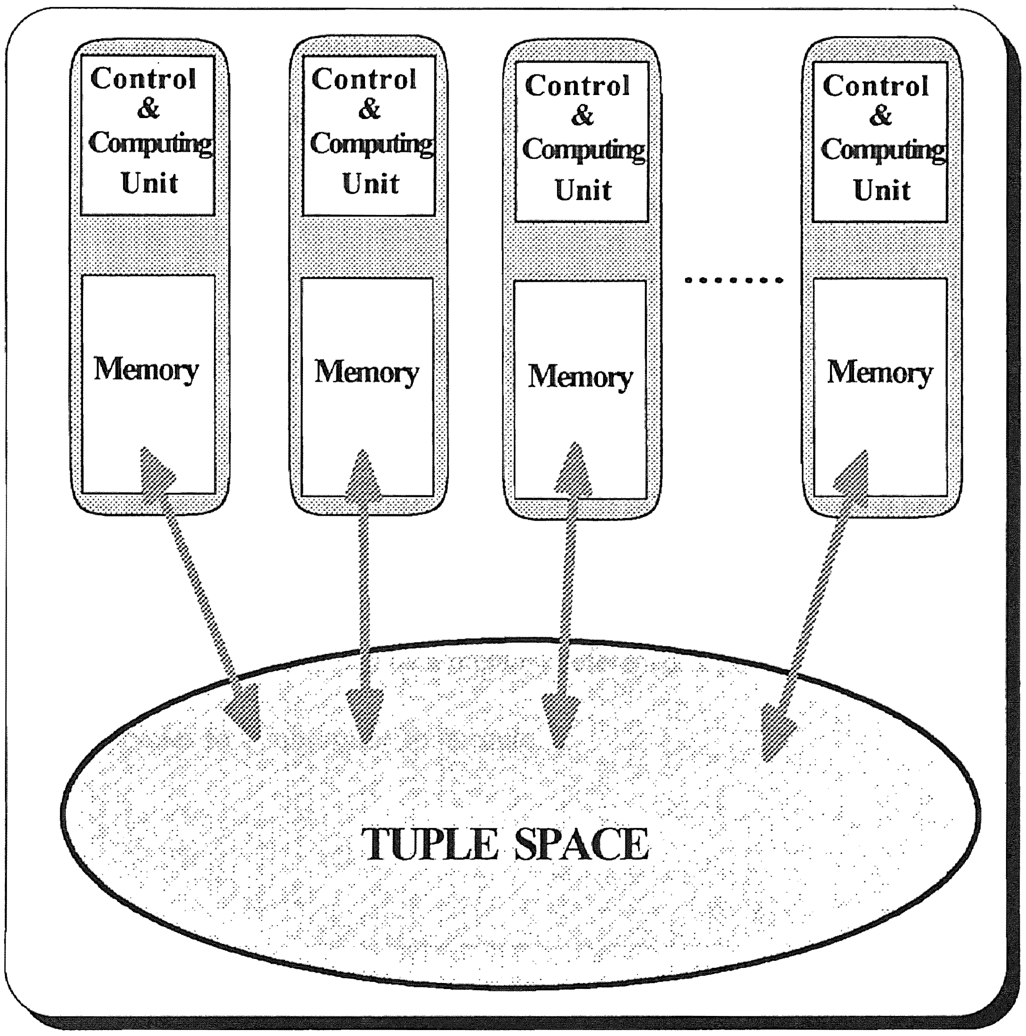


Figure 3. The Architecture of The Abstract Machine.

The Instruction Set and Interpretation of Algorithms

The design of the machine model uses a small set of instructions. An algorithm contains sets of sequences of instructions. The sequences can be executed one at a time or in parallel, but the instructions within a sequence are executed one at a time. Each instruction will go through three phases in its execution cycle: fetch, decode and then execute. In the fetch phase the instruction pointed to by (IP) is fetched from the memory. The operator part of the instruction is decoded, then in the execution phase the referenced memory or tuple is located to perform memory or tuple space update. The (IP) is updated in the execution phase depending on the executed instruction. In Figure 4, the three execution phases are shown. The instruction set of the abstract machine, as defined in this project, is shown in Table 1, Table 2, Table 3, and Table 4. The instruction format in Figure 5, shows the general format where each instruction consists of: op-code and two operands. An operand can be a memory reference, a pointer to tuple, a jump address or a pointer to a subroutine to execute.

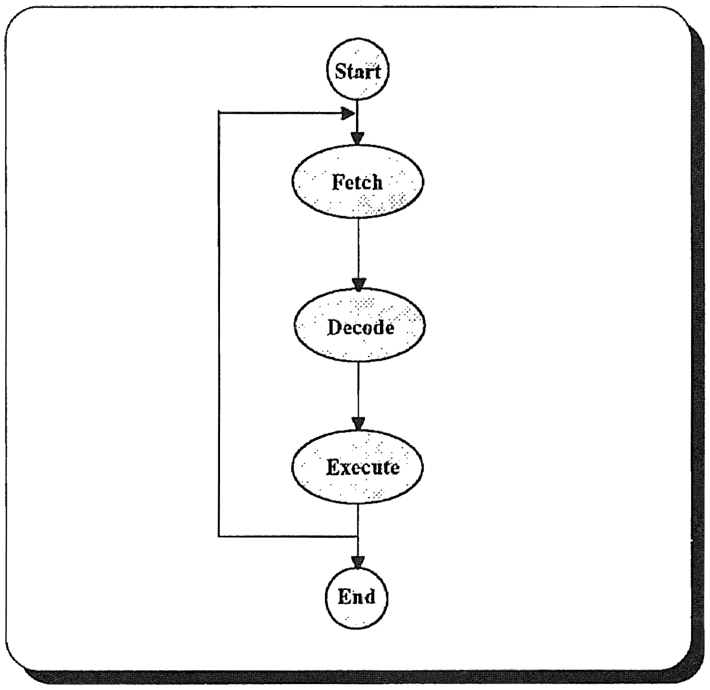


Figure 4. The Three Phases of Instruction Cycle.

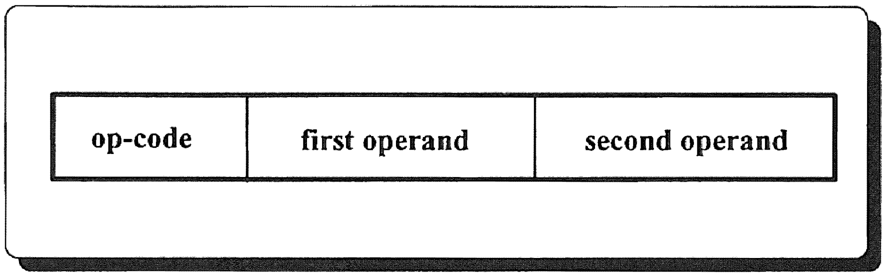


Figure 5. General Instruction Format.

TABLE 1

THE INPUT/OUTPUT INSTRUCTIONS

Instruction	Op-Code	Syntax	Function
RDI	14	RDI A,0	Read input into memory location pointed by first operand. The second operand is always zero.
WRO	15	WRO A,0	Write output from the memory location pointed by the first operand. The second operand is always zero.

TABLE 2

ARITHMETIC INSTRUCTIONS

Instruction	Op-Code	Syntax	Function
ADD	1	ADD A, B	Add the contents of two memory locations pointed by A and B, and place the result in the memory location pointed by A.
SUB	2	SUB A, B	Subtract the contents of the memory location pointed by second operand from the contents of memory location pointed by the first operand and the result is placed in memory location pointed by A.
MUL	3	MUL A, B	Multiply the contents of the two memory locations pointed by A and B and place the result in the memory location pointed by A.
DIV	4	DIV A, B	Divide the content of memory location pointed by the first operand by the none zero contents of memory location pointed by the second operand and result placed in memory location pointed by the first operand.

TABLE 3

THE CONTROL INSTRUCTIONS

Instruction	Op-Code	Syntax	Function
HLT	0	HLT 0,0	Halt the execution. The halt instruction should be the last instruction in instruction sequence.
JMP	5	JMP A,0	Jump unconditionally to the address specified in first operand, second operand is always zero.
JMZ	6	JMZ A,0	Jump to the address specified in first operand if the jump flag is zero.
JGT	7	JGT A,0	Jump to the address specified in first operand if the jump flag is greater than zero.
JLS	8	JLS A,0	Jump to the address specified in first operand if the jump flag is less than zero.
CMP	9	CMP A, B	Compare the two operands and sets the jump flag accordingly. The jump flag value is zero if the two operands are equal, less than zero if the first operand is less than the second operand, and greater than zero if the first operand is greater than the second operand.

TABLE 3 (Continued)

Instruction	Op-Code	Syntax	Function
END	16	END A,0	End of the evaluated subroutine, remove the active tuple in first operand from TS.
RET	17	RET A, B	End the evaluated subroutine, remove the active tuple pointed to by first operand A from TS, and add new data tuple pointed by the second operand B to TS.

TABLE 4

THE TUPLE SPACE OPERATION INSTRUCTIONS

Instruction	Op-Code	Syntax	Function
PUT	10	PUT A,0	Put the tuple pointed to by the first operand A into TS.
GET	11	GET A,0	Get the tuple pointed to by the first operand A from TS.
RDT	12	RDT A,0	Read the tuple pointed to by the first operand A from TS.
EVL	13	EVL A,0	Evaluate the function pointed by the first operand A, and add an active tuple into TS.

The instruction set is divided into four categories: Input/Output instructions, Arithmetic instructions, Control instructions and TS operations. The Input/Output instructions are RDI and WRO. The arithmetic instructions include ADD, SUB, MUL and DIV. The control instructions include HLT, JMP, JGT, JLS, CMP, END and RET . The TS operations include PUT, GET, RDT and EVL. The meanings of the instructions are listed under the column "Function". The processors use the TS as a way for communication and synchronization. There is no direct link between the processors. One processor is loaded with the main function which will cause other processors to be loaded in case of instruction EVL. Once a processor is loaded it continues running independently unless it executes a TS instruction.

CHAPTER IV

THE SYSTEM DESIGN AND IMPLEMENTATION

In this chapter and the following one the design and implementation of the system is described. This chapter contains the simulator design and implementation. The first section is an overview of the global design and describes some design issues of the system. The second section contain an implementation description of the simulator. The implementation of the user interface and the simulator and windows interfacing are described in the following chapter.

The Simulator Design

In this project several decisions were made to satisfy the following design objectives:

- Keep the conceptual view of system components (tuple space, processor).
- Allow the user to see the information and data associated with different elements as needed.
- Allow the user to control the starting and stopping the algorithm execution at any time.

The remainder of this section is a global overview followed by a brief discussion of the design issue.

The overall design of the system in Figure 6, illustrates the major components of the system and the relation between these components. The components are divided into: Control elements and Data sets, and the relations are: direct control, indirect control and data flow. The three major control elements are: *Debug window*, *Simulator kernel* and *Processor*. The user has the control of the debug window; therefore he/she has the control over all the system. The major data sets are: *Tuple space*, *Tuple table*, *Symbol table* and *Source code*. The rest of this section describes the major components and the relations linking these components.

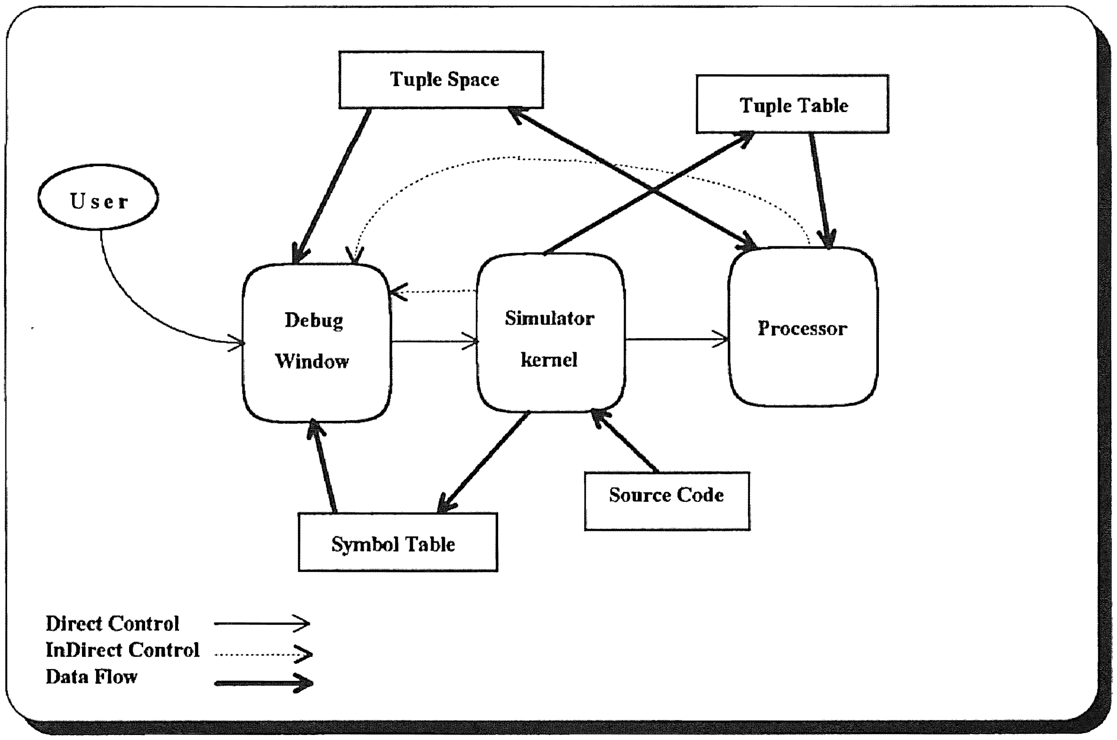


Figure 6. Global View of The System Design.

The Relations

Three types of relations are difined: direct control, indirect control and data flow. If A has a direct control over B, then A can directly call B as function. If A has indirect control over B, then A can not call B as a function but it will activate B using a message send from A to B (more detail regarding this issue presented in the next chapter). If A can access the information and data in B, then there is data flow from B to A.

The Control Elements

A control element is the component that does some computation and activates other control elements. Debug window is responsible for user interface and information display, therefore, it is the only one accepting interactive input from the user. The debug window has access to the tuple space to view its contents and to the symbol table to display the symbolic names of data.

Simulator kernel does most of the actions that are part of the system initialization, thus it reads the source code to initialize the system buffers. Simulator kernel also initializes the symbol table with the symbolic names of the data to be used by the debug window and initializes the tuple table with the format of tuples to be used later by the processor. Simulator kernel need to have some control on the debug window to update information displayed to the user, since debug window is only controlled by the user an indirect control is used in this case.

Processor's main task is executing the code, thus it has access to tuple space to place and remove tuples. Processor has access to tuple table to get the format of tuples to be placed in tuple space or the format of tuple to be read or removed. After execution the user interface need to be updated, indirect control is used to control the debug window.

The Data Sets

A data set is the component which does not involve any computation and does not have control over other components, rather it contains information or data. Tuple space is the most important data set in the system. The information it contains are tuples, and they are stored in a way to allow the access of normal tuple operations (put, get, readt and eval) and the peeking of information by the debug window using friendship.

Tuple table is used to keep the format of the tuple to be placed in or removed from tuple space. When Processor executes a tuple space instruction the instruction contains a pointer to the format of the tuple. As in Figure 7, the format includes: tag, number of elements in tuple, type of tuple and pointer to tuple elements. Symbol table contain: the symbolic representation, the scope and the type of the data. The format of symbol table is shown in Figure 8.

Debug window accesses the symbol table to display the symbolic names of data. The data values are internal to the processor and it has a pointer to its symbol table entry. Source code is accessed by the simulator kernel only and contains the users algorithm with extra information including the contents of symbol table and tuple table. The format of source code file is presented in appendix A.

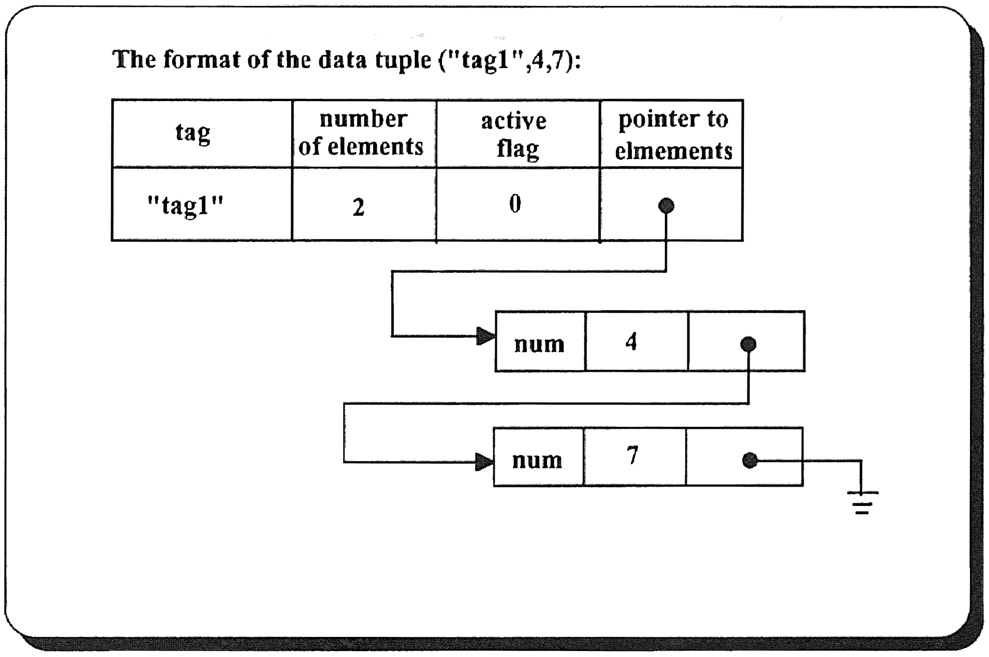
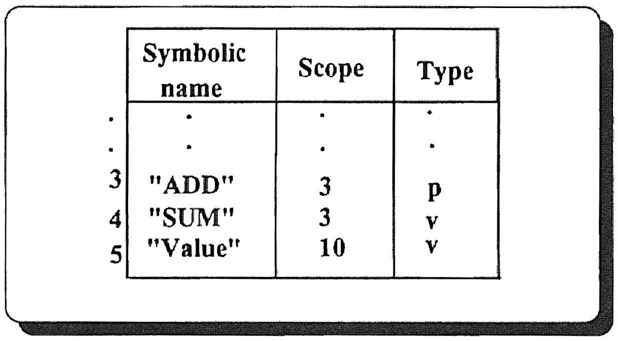


Figure 7. The Tuple Format.

The Simulator Implementation

The simulator for the abstract machine described in the previous chapter is implemented using C++. This language was chosen for its support of object oriented programming and for its efficiency. Moreover, it supports the programming of Microsoft Windows using Borland C++ compiler and its Object Windows Library.



	Symbolic name	Scope	Type
.	.	.	.
.	.	.	.
3	"ADD"	3	p
4	"SUM"	3	v
5	"Value"	10	v

Figure 8. Symbol Table Format.

The implemented simulator consists of three major parts that mutually interact: the processor, the tuple space and the simulator kernel. Each of these parts is defined as a user defined object type (or class in C++ terminology). A class has data members to represent the types and function members to implement operations on the data members. The simulator components are objects defined from the classes. The following sections describe the three main classes: Processor, Tuple Space and Simulator kernel.

The Processor Simulation

For the sake of implementation simplicity, the current implementation of the simulator assumes a four processor machine. As outlined in the previous chapter, each processor has a memory, an instruction pointer, a jump flag, and a busy flag . The busy flag is used by the simulator kernel only and does not affect the execution sequence. On the other hand, TS is not part of the processor. The TS can be accessed only by the designated set of operations. Figure 9, shows the processor class definition.

```
class  Processor {
   char    procname[STR_LENGTH]; // process name
   char    output[STR_LENGTH];   // output string
   Inst    Imemory[IM_SIZE];     // instruction segment
   int     Dmemory[DM_SIZE];  // data segment
   int     symptr[DM_SIZE];   // pointer to symbol table
   int     ip;           // instruction pointer
   int     jf;           // jump flag
   int     busy;         // busy flag

public :
   Processor();           // constructor
   void reset(HWND*);
   void load(Inst inst, int loc, HWND*);
   void exec(int,HWND*);
   void loadd(int n, int p, HWND*); // load data
   void use();
   void free();
   int  used();
   friend  TS;
   friend  TMyWindow;
   friend  TProcWindow;
   friend  TOutWindow;
   friend  TDataWindow;
   };
```

Figure 9. The Processor Class.

The memory is divided into two segments, one segment *Imemory* holds the instructions and the other segment *Dmemory* holds the data. For each element in *Dmemory* there is a pointer to the symbolic name of that element in the symbol table. The symbolic name of data element is used for debugging purposes only. Program is stored in *Imemory* in the form of instructions. Each instruction *Inst,* as shown in Figure 10, has three parts: op-code and two operands. The instructions in the memory are executed one at a time in each execution cycle where the *ip* points to the instruction to be executed. After the execution of an instruction *ip* is set to point to the next instruction, if the instruction is a jump instruction the *ip* is updated depending on the jump flag. The jump flag *jf* is affected only by the compare instruction.

```
class  Inst {
      int     opc;       // op-code
      int     op1;       // first operand
      int     op2;       // second operand

 public:
      void    readopc();
      void    readop1();
      void    readop2();
      friend  Processor;
      friend  TMyWindow;
      friend  TProcWindow;
};
```

Figure 10. The Instruction Class.

The busy flag *busy* is set initially to "unused" for all processors. It is set to "busy" for a processor when it is loaded. The execution of the HLT, END or RET instruction sets the busy flag to "unused". The EVL instruction loads an "unused" processor and marks it as "used". The remaining of this section is devoted to a detailed description of the processor class. The data members of the Processor class are listed in Table 5. The following is an explanation of the member functions of class Processor:

- Processor(): constructor of the Processor class, it initializes *busy* flag to -1.
- reset(HWND*): reset the processor, instruction pointer *ip* is set to 0 and jump flag *jf* is set to 0. The reset function works only if the *busy* flag is 1, otherwise the

function MessageBox(HWND*, LPSTR, LPSTR, WORD) is called to notify the user the type of the error.

- exec(int, HWND*): this function does three tasks, namely fetch, decode and execute. The instruction pointed to by *ip* is fetched from *memory*, decoded and then executed depending on the op-code value *Inst.opc* of the instruction. If the instruction to be executed need data operands, the data are fetched from data segment. If the instruction being executed is a tuple space instruction, the tuple frame is fetched from the tuple table and then the operation is performed. The compare instruction updates the jump flag *jf* and the jump instructions will update the instruction pointer *ip* depending on the contents of the jump flag *jf*. The exec(int, HWND*) function may call functions from other friend classes or other related set of functions. If the instruction is a tuple space operation the following functions may be called: TS::get(int), TS::put(tuple), TS::readt(int) or TS::fetch(tuple). In case of error an error message will be given to the user, for this purpose the function MessageBox(HWND*, LPSTR, LPSTR, WORD) is called to display the error message on the main window. The functions find_func(HWND*, char*) is called to return the index of the process in process table *pt*. At the end of the exec(int, HWND*) function a message is send to the main window to update the active windows in this application by calling SendMessage(HWND*, WORD, WORD, DWORD).
- loadd(int n, int p, HWND*): load the *n* data elements in the data segment of processor *p* and load the pointers to symbols into *symptr* table. If this function is

called and the busy flag is -1, error message is displayed by calling the function MessageBox(HWND*, LPSTR, LPSTR, WORD).

- use(): mark the processor used by setting the busy flag *busy* to 1.
- free(): mark the processor not used by setting the busy flag *busy* to -1.
- used(): return the status of the busy flag *busy*.

TABLE 5

THE DATA MEMBERS OF PROCESSOR CLASS

Data member	Type	Function
ip	int	instruction pointer
jf	int	jump flag
busy	int	busy flag
Imemory	Inst[]	instruction segment
Dmemory	int[]	data segment
symptr	int[]	pointer to symbol
procname	char*	process name
output	char*	output

The TS Class

The TS is not part of any of the processors but it is part of the abstract machine. Keeping the TS separate from processors gives more generality to the abstract machine since the tuple space can be implemented in a shared memory or distributed memory parallel multiprocessor system. The tuple space is accessed using tuple space instructions. Whenever a PUT or EVL instruction is executed a tuple is placed into TS, and a tuple is removed only after execution of GET instruction. Figure 11, shows the specifications of the tuple space class. The *tuples* in tuple space class is a list of tuple type holding the tuples. The tuple type has a *tag* to identify the tuple, a pointer *elm* to the list of tuple elements, a flag *active* to indicate whether the tuple is active or not and a counter *num* to hold the number of tuple elements in the tuple. Each tuple element has three parts: one part *(v)* is to hold the element value, another part *(type)* is to indicate whether this element is a number , a variable or input variable and the third part is a pointer *(next)* to the next element.

The TS class contains the data members shown in Table 6 that represent the tuple space TS, and the member functions to access the TS. In addition to functions that represent Linda operations (PUT,GET and RDT) there are two functions, one to initialize the tuple space and the other to fetch a tuple from the tuple space. The following is an explanation of the member functions of TS class:

- init(): initialize TS by setting the *tsptr* to 0.
- put(tuple t): add the tuple *t* to the tuple space TS.
- fetch(tuple t): return a pointer to the tuple *t*.

- get(int i): return the tuple pointed to by *i* to the caller.
- readt(int i): return a copy of the tuple pointed to by *i*. A new tuple structure is returned.

```
struct  element{
        char     type;     // type of element
        int      v;        // value of element
        element  *next;    // pointer to next element
};

struct  tuple {
        char     tag[str-length]; // string tag
        element  *elm;     // pointer to elements
        char     active;   // active flag
        int      num;      // number of elements in tuple
};

class  TS {
     tuple  tuples[num_tuples];
     int    tsptr;

  public:
     void init();
     void put(tuple t);
     tuple get(int i) ;
     tuple readt(int i) ;
     int  fetch(tuple t);
     friend TTSWindow;
     friend TTSDialog;
};
```

Figure 11. The TS Class.

TABLE 6

DATA MEMBERS OF TS CLASS

Data member	Type	Description
tuples	tuple[]	the tuple space.
tsptr	int	pointer to tuple space.

The Simulator Class

The simulator class is the kernel of the simulation model. The components of the simulator class does not belong either to the Processor class or to the TS class. The data members of the simulator class are listed in Table 7, and Figure 12 shows the specifications of the simulator class. The description of member functions defined in this class follows:

- init(char*, HWND*): open the input file and initialize buffers and tables from input file. This function calls the following functions:

  1) TS::init() to initialize the tuple space TS.

  2) read_tuple_table(int, HWND*) to read the tuple frames into the tuple table.

  3) Processor::use() to mark processor 0 used.

  4) Processor::reset() to reset processor 0.

  5) Processor::loadd(int, int, HWND*) to load the data.

  6) Processor::load(Inst, int, HWND*) to load the instructions.

7) In case of error call MessageBox(HWND*, LPSTR, LPSTR, int) to notify the user. At the end of the function Simulator::init(), call SendMessage(HWND*, WORD, WORD, DWORD) to send a message to the main window to rewrite the simulator's active windows.

- step(HWND*): This function executes one step of each busy processor by calling the function Processor::exec(int,HWND*). If Processor 0 is "unused" call MessageBox(HWND*, LPSTR, LPSTR, WORD) to notify the user. At the beginning of the function step(HWND*) call SendMessage(HWND*, WORD, WORD, DWORD), to send a message to the main window to rewrite the simulator's active windows.
- go(HWND*): This function calls SendMessage(HWND*, WORD, WORD, DWORD) at the beginning to send a message to the main window to rewrite the simulator's active windows, then it calls Processor::exec(int,HWND*) to execute the whole algorithm until the end.

TABLE 7

DATA MEMBERS OF SIMULATOR CLASS

Data member	Type	Description
ptptr	int	process table pointer
buffptr	int	buffer pointer
i	int	private use

TABLE 7 (Continued)

Data member	Type	Description
j	int	private use
clength	int	code length
dlength	int	data length
stlength	int	symbol table length
ttlength	int	tuple table length

```
class   Simulator {
      int     ptptr;         // process table pointer
      int     buffptr;       // input buffer pointer
      int     i,j,           // private use
              clength,       // code length
              dlength,       // data length
              stlength,      // symbol table length
              ttlength;      // tuple table length
   public :
      void  init(char fname, HWND*);
      void  step(HWND*);
      void  go(HWND*);
};
```

Figure 12. The Simulator Class.

This chapter outlined the design issues of the system and the simulator design and implementation including the three major classes Processor, TS and Simulator. The next chapter will spot the light on the design and implementation of debugging windows and the interface between the simulator and the windows.

CHAPTER V

WINDOWS INTERFACE

This simulator project is implemented as a Microsoft Windows application and has been given the name "Linda Tool". A Windows application uses an application window for input and output. The Windows application creates and manages the application window. The application should use the Windows functions to implement the interface with Windows. Because Windows applications are message driven, the application should take care of Windows messages. Figure 13 illustrate the difference between message driven windows programming and sequential programming. The following sections focus on programming for Windows, the object library used, the window and dialog classes defined in this project and the linkage between the simulator and the windowing interface.

Windows Programming

As mentioned in the previous section, programming Windows application is different from sequential procedural programming. The application should take care of the user actions that comes to the application as messages from Windows system. As an example, the click of the left mouse button creates at least two messages one is

WM_LBUTTONDOWN when the left button is pressed and the other is WM_LBUTTONUP when the mouse left button is released. The messages are assigned symbolic names to simplify their use.

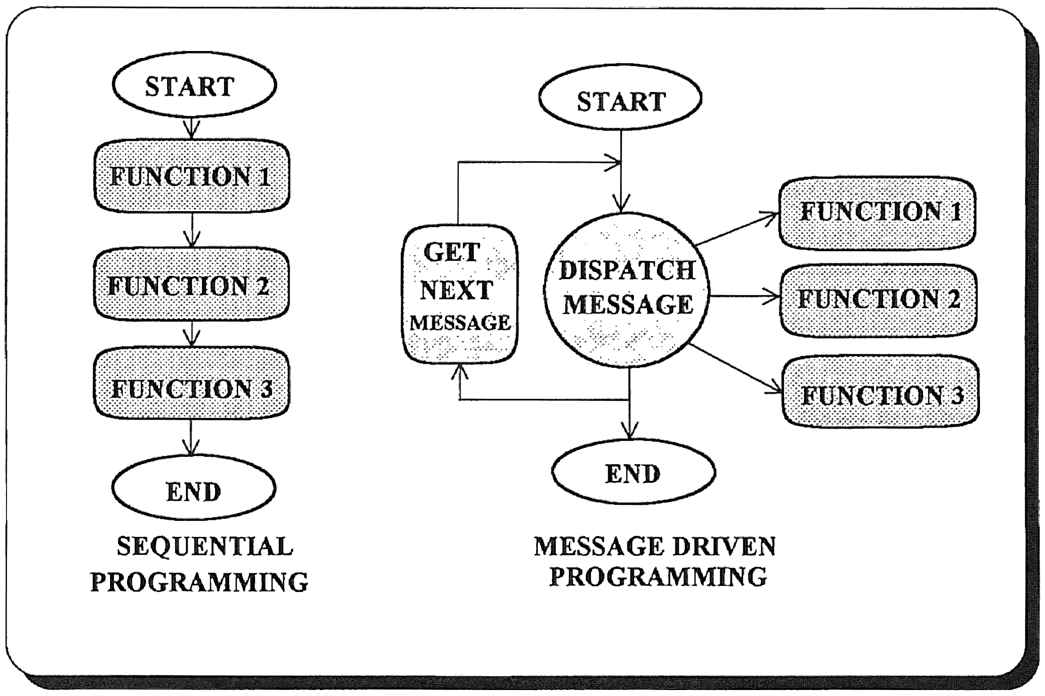


Figure 13. Sequential and Message Driven Programming.

The windows application also should use the same user interface objects provided by Windows. These objects include: icons, windows, menus, dialog boxes, cursors, carets etc. The user will use these objects to interact with the application. The windows application creates and controls user interface objects by calling windows library routines and handling the messages associated with the interface.

Output in windows application is not character oriented, but it is graphics oriented. The lines, ellipses, rectangles and text are all displayed in graphical format. The Windows Graphics Device Interface (GDI) is deigned for device independent graphics, and the windows application uses GDI and it does not need special device drivers to work with different type of devices. Windows application can use for its output any device that has a Windows driver.

In this work Borland C++ and Application frame work are used to program windows application using the object oriented methodology. Object oriented programming makes the application more modular and easy to maintain. The header file *windows.h* should be included in the application source code to access windows run time library and to use Windows data types. Some of the important data types are shown in Figure 14. Object Windows Library (OWL) is an object oriented library that makes object oriented windows programming easier. The next section focuses on OWL and its use in this project.

Windows type	C type
BYTE	unsigned char
DWORD	unsigned long
HANDLE	unsigned int
HDC	unsigned int
HWND	unsigned int
LPSTR	char far *
WORD	unsigned int

Figure 14. Some Windows Data Types.

Programming Using OWL

Borland's OWL is an object oriented windows library that contains various windows interface objects. All windows interface objects in this work are derived from the predefined OWL objects. Using derived objects makes the programming task easier and the program more modular. Moreover, the OWL provides a mechanism to respond to the incoming windows messages using a dynamic dispatch virtual table. It transparently maps the incoming messages to responses. The programmer defines member functions associated with a window message to respond to that message. More information can be found in Object Windows for C++ User's Guide [OWL91] and Borland Languages Open Architecture Hand Book [OAH91]. The OWL classes used in this application are outlined below:

- TApplication class: every Windows application developed using OWL must define an application class. TApplication class initializes and creates the main window in the application.
- TWindow class: it is derived from TWindowsObject class. The TWindow class handles some of the tasks that every Windows application must do including: sizing, painting, moving, etc. The main window is derived from this class as well as other child windows.
- TDialog class: this is the parent class of the dialogs in windows application. The dialogs in an application can be predefined as: TFileDialog, TInputDialog, etc. or can be a custom dialog. The programmer defines the control objects and member functions of custom dialog.

- TFileDialog class: this special dialog class is derived from TDialog. The TFileDialog prompts the user to input a file name to be opened. Besides the input line for file name TFileDialog box contains two list boxes one for file names and the second for directories, one button OK and one button CANCEL.
- TInputDialog class: this class is derived from TDialog and accepts single input line. There are two push buttons in this dialog box namely, OK and CANCEL.

Classes and Functions Used in Linda Tool Application

The Linda Tool application has its own window and dialog classes derived from OWL classes outlined in the previous section. The Linda Tool application used both the OWL functions and the Windows functions. The following are the classes defined in Linda Tool:

1) TMyApp Class: This is the application class which handles the main window initialization and creation, it also queries the system for messages. The TMyApp class is derived from TApplication class as shown in Figure 15, and has one function that redefines InitMainWindow().

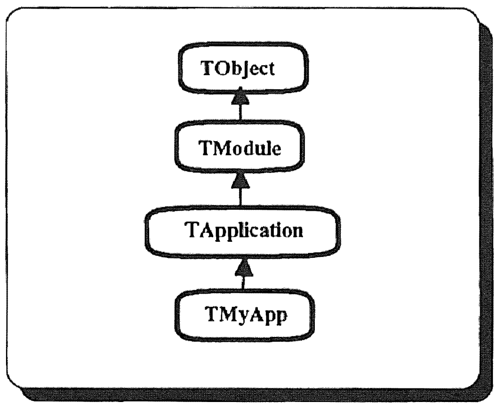


Figure 15. Hierarchy of Application Class.

2) TMyWindow Class: As shown in Figure 16, TMyWindow is derived from TWindow and uses the data members of TWindow class. It also has other data members as flags and pointers to the different child windows and dialogs created by TMyWindow, these data members are listed in Table 8. Besides TWindow member functions the following functions are the major functions newly defined in TMyWindow, and some of them override TWindow functions:

- GetWindowClass(WNDCLASS&): This function calls the TWindow::GetWindowClass(WNDCLASS&) function to get the window class. It also sets the background brush to NULL_BRUSH and loads the icon of the application.CanClose(): This function is called when the user wants to close the window. The function will check if there is an open file. If so it prompts the user with a message box asking either to close the file and exit or return to the application. Depending on the user choice the function CanClose() will return a "true" value to allow closing the window or "false" to prevent closing the window.

- Fopen(RTMessage): This function is called when the user chooses Open from the File menu. It displays a TFileDailog, then it calls the function simulator::init(LPSTR HWND*) to open the file and initialize the system.
- Fclose(RTMessage): Closes the current open file. If no file is open display an error message.
- Habout(RTMessage): Displays a custom dialog containing information about the application.
- Fexit(RTMessage): Exits the system by calling the function CloseWindow().
- Showp(int, RTMessage): Creates a child window, and calls Display(HDC, int) to display the processor information.
- Showts(RTMessage): Creates the tuple space child window.
- GetInput(RTMessage): Displays TInputDialog to get input from the user. This function is a response to SM_GetInput sent by the simulator while executing the RDI instruction.
- Rewrite(RTMessage): This function a response to SM_Rewrite message, and it will update the windows contents. The SM_Rewirte is send from other functions when the contents of displayed window need updating.
- Dstep(RTMessage): This function will respond to the user choice of "Single step" from main menu, and it calls the function simulator::step(HWND*).

- Dgo(RTMessage):This function responds to the user choice of "Go" from "Debug" sub menu, and it calls the function simulator::go(HWND*).
- Danimate(RTMessage): This function responds to the user choice of "Animate" from "Debug" sub menu, and it calls the system timer function to cause WM_TIMER message sent to the window after each delay time period, and sets the Animate flag to 1.
- GetDelay(RTMessage): Displays TInputDialog to read the delay time. If the delay time is outside the defined ranges, it is set to default 1 second.
- DAnimatestep(): This function will respond to WM_TIMER sent by the Windows system timer. If the Animate flag is 1, it executes one step by calling the function simulator::step(HWND*). At the end of algorithm it will cancel the timer and no more WM_TIMER is sent.

TABLE 8

DATA MEMBERS OF MAIN WINDOW

Data member	Type	Usage
PWindow[]	PTProcWindow	Array of pointers to Processor windows.
PWindow_ts	PTTSWindow	Pointer to tuple space window.
PWindow_out	PTOutWindow	Pointer to output window.

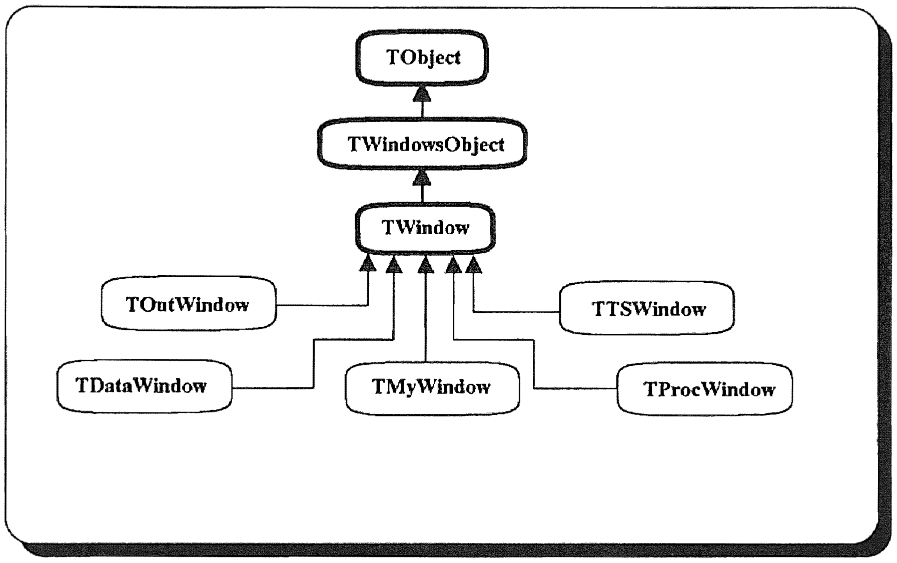


Figure 16. Hierarchy of Window Classes.

3) TDataWindow Class: This class is derived from TWindow as shown in Figure 16. Data window is a child window of processor window and displays the contents of data segment of the parent processor. TDataWindow class has the following two functions:

- Paint(HDC, PAINTSTRUCT&): Repaints the window contents in response to WM_PAINT.
- Display(HDC, int): Displays the data segment of the parent processor.

4) TSWindow Class: The window associated to this class is used to display the tuple space using graphical representation. It is derived from TWindow class as shown in Figure 16, and has the following functions:

- CanClose(): This function overrides the TWindow CanClose() function, it marks the window to be closed as hidden and then allows closing of the window.
- Paint(HDC, PAINTSTRUCT&): This function responds to WM_PAINT. It calls GDI functions to repaint the tuple space window.
- ShTuple(RTMessage): Creates a TTSDialog to display the tuples in textual form. This function respond to WM_LBUTTONUP.
- Rewrite(RTMessage): This function updates the tuple space window. It respond to SM_Rewrite.

5) TProcWindow Class: This window displays the processor information and creates a data window when the user presses the left mouse button. When it is released the data window is closed. TProcWindow class is derived from Twindow class as shown in Figure 16, and has the following functions:

- Paint(HDC, PAINTSTRUCT&): Repaints the window contents in response to the message WM_PAINT.
- Display(HDC, int): This function displays the current status of the processor.
- WMLButtonDown(RTMessage): Creates a data window in response to the message generated by pressing the left button in the processor window.
- WMLButtonUp(RTMessage): Closes data window in response to the message WM_LBUTTONUP (releasing the left button in the processor window).

6) TOutWindow Class: This window displays the output from the processors and Figure 16 shows how it is derived from TWindow class, it has the following functions:

- CanClose(): This function overrides the TWindow CanClose() function, and allow closing of the window after marking it as hidden.
- Paint(HDC, PAINTSTRUCT&): Repaints the contents of the output window. This function is called when the output window or part of it is erased.
- Rewrite(RTMessage): Updates the contents of the output window.

7) TTSDialog Class: This dialog has two push buttons "OK" and "Next". It displays the tuples in text format. TTSDialog is derived from TDialog as shown in Figure 17, and has the following functions:

- Paint(HDC, PAINTSTRUCT&): Repaint the dialog contents.
- TSOK(RTMessage): Closes the dialog in response to pushing the OK button.
- TSNext(RTMessage): Shows next tuple in the tuple space.

8) TAboutDialog Class: As shown in Figure 17, this class is derived from TDialog, and has one function AboutOk(RTMessage) to close the dialog box when the user pushes the OK button. This dialog displays information about the application.

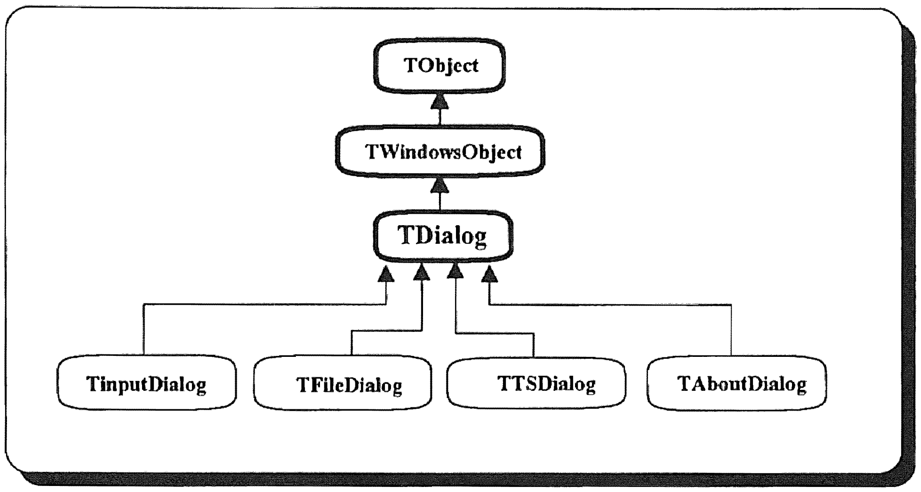


Figure 17. Hierarchy of Dialog Classes.

The Interface with Simulator

The simulator needs to interact with the windows because the windows level is closer to the user than the simulator level. The interaction is either control transfer or data access. The control is transferred in two ways:

1) The windows call the simulator functions init(), step() and go(). This is called direct control.

2) The simulator sends messages to the windows to activate a certain function mainly to update the output of the windows.

Figure 18, illustrates the Simulator-Windows interaction. The simulator can not call the Windows functions directly because Windows is message driven. So it send a message to the Windows. The Windows will call the function that will respond to the message. On the other hand, some windows need to display information and data of the simulator components so it should have one way access to the related classes in the simulator. These classes are Inst, Processor, and TS. The access is gained by using the friend relation in the definition of the simulator classes. The one way access is to read the information only with out updating.

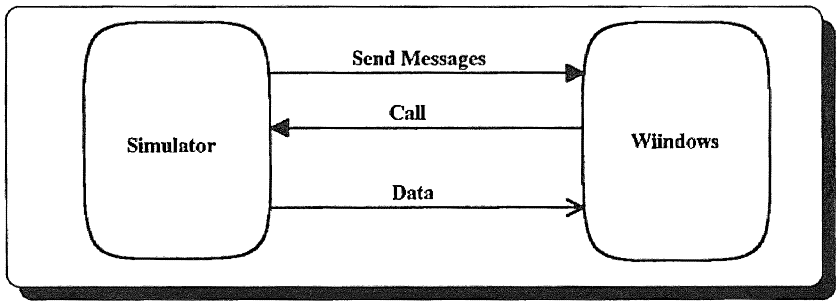


Figure 18. Simulator-Windows Interaction.

In this chapter, the Windows interface is described. A list of classes and methods and their functionality has been described. the user interface is described in the next chapter.

CHAPTER VI

THE USER INTERFACE

The simulation system uses the Microsoft Windows graphical user interface. The user can easily learn how to use the system using mouse point and click. The simulation system is given the name Linda Tool. This chapter describes the main elements of Windows, the Linda Tool main window components and other child windows.

Microsoft Windows Elements

This section give a brief introduction to Microsoft Windows. Figure 19 illustrates the elements of Windows. Each window has the following elements: Window title where the name of the window appears, control menu box on the upper left corner of the window, the minimize button and the maximize button in the upper right corner of the window, the main menu bar under the title, the vertical and horizontal scroll bars on left and bottom of window and the work space in the middle of the window. The window has also, a border which shows the boundary of the window. The user can change the size and position of the window using the mouse. For more information about Windows the reader is referred to Windows users guide [MWU92].

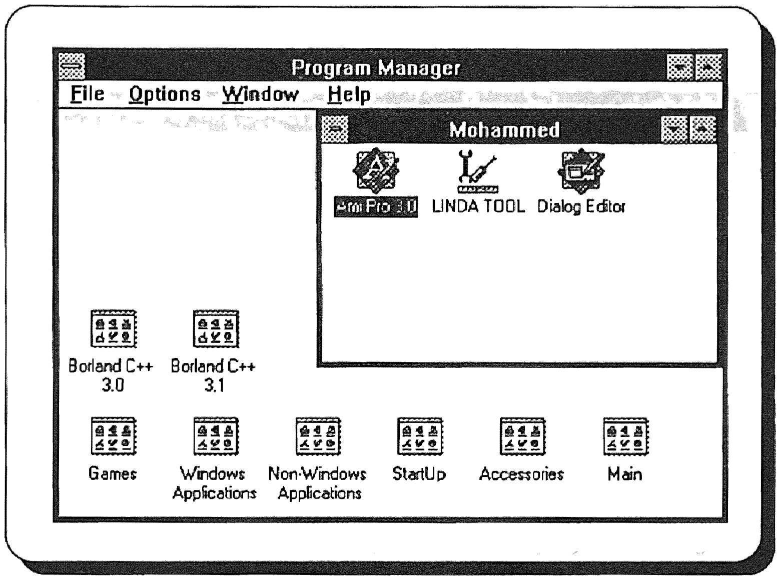


Figure 19. Elements of Windows.

Main and Child Windows Components

The main window of Linda Tool has a similar lay out as the one described in the above section. As shown in Figure 20, the main menu bar has five components: *File*, *Single step*, *Debug*, *Show / hide* and *Help*. The title bar displays the application name "Linda Tool" with the name of the current open file. Figure 21 shows examples of the child windows, the tuple space window and processor(0). Besides the application dialogs that are used to prompt the user for input or output, the application has six child windows to provide the user with information on the current state of the algorithm and the processors. The user can show or hide any child window using the *Show / Hide* option from main menu.

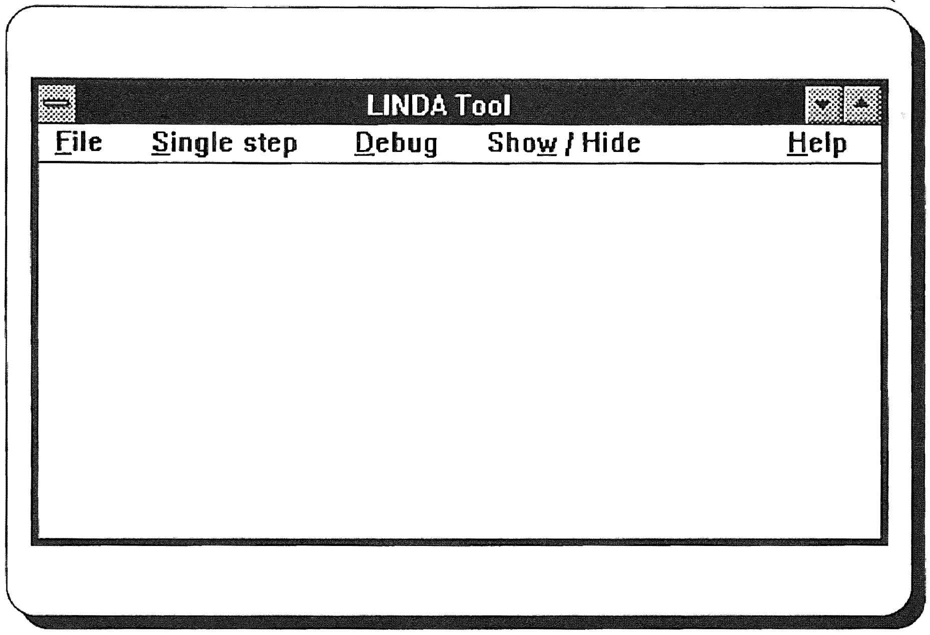


Figure 20. The Application Main Window.

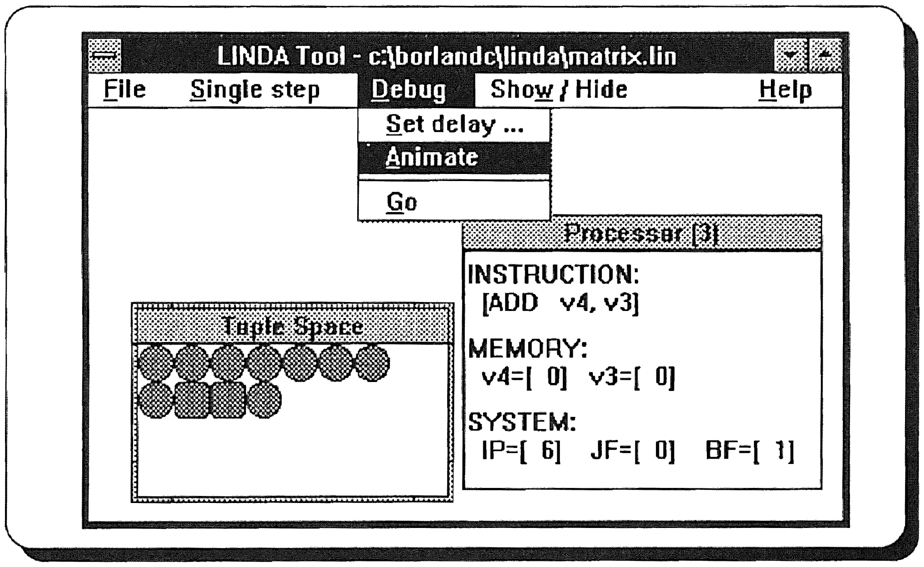
The main menu bar in Figure 22, has five menu items: *File*, *Single step*, *Debug*, *Show / Hide* and *Help*. Figure 21, shows an example of the *Debug* sub menu. The description of each menu item and its sub menu follows:

The "File" Menu Item

The File menu item has a sub menu with three items: Open, Close and Exit. The Open and Close are directly related to file access. The Exit is related to file only if file is open. The three items are described below:

- Open: Selecting the Open item generates an open file dialog. The open file dialog provides the user with list of directories on the system and another list of files on the current directory. The user selects the desired file by double clicking on the file name or one click and push the OK button. When the file is opened the simulator is initialized with the file contents and the name of the file appears in the title bar to indicate that it is the current file. If the user opened another file the old file is closed and the newly opened file become the current file.

- Close: The Close menu item, closes the current file if any. The name of the current file is removed from the title bar. If there is no open file to close the system notify the user with a message.
- Exit: The Exit menu item will close the application and check if an open file exist the system notify the user and ask if he/she wants to close the current file. If the user selects "Yes" both the file and the application will be closed, and if "No" is selected the system returns back to the main window.



INSTRUCTION:
[ADD v4, v3]
MEMORY:
v4=[0] v3=[0]
SYSTEM:
IP=[6] JF=[0] BF=[1]

Figure 21. Linda Tool with Child Windows.

The "Single Step" Menu Item

The Single step option has no sub menu, therefore it works as a push button. Selecting "Single step" will execute the current instruction on all the busy processors and update the child windows. The "Single step" is not part of the debug sub menu to make it easier for the user to single step the algorithm, the user can do one selection "Single step" instead of two: "Debug" and then "Single step". If the user tried to step after the end of the algorithm, an error message will be displayed.

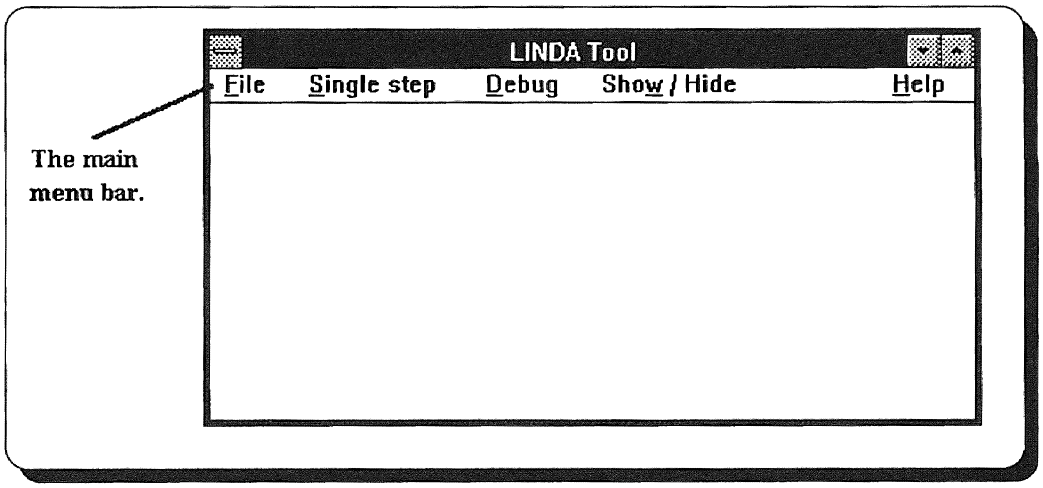


Figure 22. The Main Menu Bar.

The "Debug" Menu Item

The Debug menu item has a sub menu containing: *Set delay*, *Animate* and *Go*. The "Set delay" does not run the algorithm, but the options "Animate" and "Go" will execute the algorithm. The function of each of the items is stated below:

- Set delay: Select the "Set delay" menu item to set a new delay time for algorithm animation. The input time is the time between to steps in seconds. The default time is one second and the user can change this time by entering new time in the "Animate delay time" dialog box. If the new time is less than zero or greater than ten seconds the time is set automatically to one second.
- Animate: Selecting "Animate" will run the algorithm with delay time between two consecutive steps. The delay time is set in the "Set delay" option. If the user selects "Animate" while it is running in animate mode, the animate mode is turned off and the user can continue running the algorithm again using "Single step", "Animate" or "Go".
- Go: Selecting the "Go" option will run the algorithm without any interruption. If the algorithm ends and the user tries to select "Go" again an error message is displayed. The delay time does not affect the "Go" option.

The "Show / Hide" Menu Item

The Show / Hide menu item is used to toggle the display of the child windows. It has a sub menu with six items. The first four to show or hide the processors window, and the other two to show or hide the tuple space and output. Selecting a menu item will display or close the appropriate window.

- Tuple space: Selecting this item will toggle the Tuple space window. The Tuple space window represent the tuples in graphical mode. An ellipse represent a data tuple and a round rectangle represents an active tuple. To show the tuples in text format click the left mouse button in the TS window, a dialog will appear with two buttons *Next* and *OK*. Push "Next" to view the tuples in tuple space one by one. The text color indicates the type of the tuple, black text for data tuples and red text for active tuples. Push the "OK" button to close the dialog.
- Output: The "Output" option toggles the output window On or Off. The output generated by the instruction WRO for each processor is displayed in a separate line in the output window. The output consists of the symbolic name of the data item and the value of that item.
- Processor: Selecting this option will show or hide the processor window. Each processor has menu item indicating the process number (0,1,2 or 3). The information in the processor window include: the current instruction, the contents of data memory which will be affected by the instruction, and the processor flags. Pressing the left mouse button while in processor window will show the "data window" that displays the data symbols and values for the processor.

The "Help" Menu Item

The help sub menu contain two items: *About* and *How to*. The "About" option displays the information about "Linda Tool" including the version number. The about dialog has one push button "OK" to close this dialog. Selecting the "How to" option will open the "How to" help window to provide the user with help on using the system. The user can select a topic and then push the "How to" button to get help on the selected topic. The "Cancel" button will close the help window.

The user interface described in this chapter is window based and the user who has experience with other windows applications should find it easy to use the system. However, users who have no experience with windows can refer to the "Microsoft Windows user's guide" [MWU92].

CHAPTER VII

FUTURE WORK

The study of parallel computing can be broadly divided into two fields: the study of parallel hardware and the study of parallel software. The work presented in this theses is focused on parallel software. The implemented tool is reasonably adequate for monitoring and debugging parallel Linda algorithms; however, it might need some improvements.

The current implementation of the system requires the use of algorithms written in a special format (presented in Appendix A) using the defined instruction set in chapter 3. A possible improvement can be achieved using algorithms written in high level algorithmic language. X-Linda is a proposed high level algorithmic language described in appendices B and C can be used for writing parallel algorithms based on Linda approach. The use of this a language requires a translator to translate algorithms from X-Linda to the format presented in appendix A. The user interface should be modified to accompany the use of high level language. One of these changes requieres adding a new window for high level language representation. Also the translator can be included as part of this tool.

Another improvement is to use trace files to keep a record of the steps in a debugging session. The trace file should contain information that can be used for algorithm analysis. Trace file may contain one line for each simulator step showing the instruction executed by each processor and the contents of related memory locations or tuples.

The current parallel machine simulator is implemented on a sequential machine, thus, the simulated processors are not running actually in parallel. At each simulator step each processor will execute one instruction in the following sequence: Processor 0, Processor 1, Processor 2 and Processor 3. This sequence is always fixed, making the algorithm behavior stable each time it runs on the simulator. A possible enhancement is allowing the user to change the sequence at the beginning of the debugging session. Furthermore, a random number may be used to change the sequence of processors at each simulator step. The current implementation of the system is not concerned with performance issues. Extension of the simulator to provide estimated performance analysis of algorithms is also considered a future work.

CHAPTER VIII

CONCLUSION

This thesis is concerned with software parallelism and tools. In this work, a tool, called "Linda Tool", was designed and implemented on a Personal Computer environment to assist debugging parallel algorithms based on Linda. Linda Tool is useful to persons who develop parallel programs in languages based on the Linda model. It supports algorithm development at a level different from, and independent of programming languages. This tool assists the developers of Linda programs in detecting and locating algorithmic errors in early stages of algorithm development before creating programs in the target language to run it on a computer system.

Linda Tool accepts an algorithm as input. The tool runs the algorithm on a simulated parallel machine. The developer can view the algorithm behavior by examining current tuple space contents and the current status of each processor. Since the communication and synchronization among processes in Linda are done via the tuple space, the user can locate the causes of unexpected behavior (such as indefinite postponement or deadlock) or unexpected results of the algorithm by examining the contents of the tuple space. The user can also examine the contents of variables as instructions are executed in each process to help decide if the process is running in a correct manner. The tool facilitates the user to view which processes are running in parallel at any particular time. The user will be able to use the information provided by the system to locate errors in the algorithm or modify the algorithm to give better structure. It should be mentioned that one of the motivations for designing such a tool is to serve as an instructional tool.

The current implementation of the tool includes a simulator for parallel machines with four processors. The implementation environment is Borland C++ and applications framework. The limitation on number of processors was imposed by the current environment in which the simulator is implemented. However, because the system is modular it can be modified to simulate more than four processors depending on the limitation of the target computer. The Linda Tool software may be modified to work on other computer systems that support a graphical user interface. The time and effort required to implement such a system depends on many factors, one of which is the availability of a system with a user interface similar to Microsoft Windows. The software will be available from the Computer Science Department at Oklahoma State University.

REFERENCES

[All89] Allen Ambler and Margret Burnet, Influence of visual technology on the evolution of language environments, Computer (October 1989), pp.9-22.

[AMF91] A. M. Finn, M. F. Griffin, and W. C. McClurg, Modeling and simulation of an i860-based multiprocessor, Proceedings of the 24th annual simulation symposium, (April 1991), pp.91-97.

[BPW92] Peter Norton and Paul Yao, Borland C++ Programming for Windows, (1992).

[Carl90] Carl J. Beckmann, CARL: An Architecture Simulation Language, Research report, University of Illinois, (December 1990).

[CC90] C. C. Charlton, P. H. Leng, and D. M. Wilkinson, Program monitoring and analysis: software structures and architectural support, Software Practice and experience, (September 1990), pp.859-867.

[Chi91] Christopher Mills, Stanley C. Ahlat and Jim Fowler, Compiled Instruction Set Simulation, Software Practice and experience, (August 1991), pp.877-889.

[Dav87] David Gelernter, Programming for advanced computing, Scientific America,(October 1987), pp.301-308.

[Dav90] David L. Eldredge, John D. McGegor, and Marguerite K. Summers, Applying the object-oriented paradigm to discrete event simulation using the C++ language, Simulation, (February 1990), pp.83-91.

[Denn89] Dennis L. Doubleday, The Durra Application Debugger / Monitor, Technical report, Carnegie Mellon University, (September 1989).

[DWA92] James McCord, Developing Windows Applications with Borland C++3, (1992).

[FB83] F. Baiardi, N. DeFrancesco, E. Matteoli, S. Stefanini, and G. Vaglini, Development of a debugger for a concurrent language, Proceedings of the ACM , (August 1983), pp98-106.

[Fre91] Fredrik Dahlgren, A program-driven simulation model of an MIMD multiprocessor, Proceedings of the 24th annual simulation symposium, (April 1991), pp.40-49.

[Ger91] Gregory Andrews, Concurrent programming, The Benjamin/Cummings, California, (1991).

[Jam88] James H. Giffin, Hrvey J. Wasserman and Lauren P. McGarvan, A Debugger for Parallel Processes, Software Practice and experience, (December 1988), pp.1179-1190.

[Jas85] Jason Gait, A debugger for concurrent programs, Software Practice and experience, (June 1985), pp.539-554.

[JDB87] J. D. Bovey, A Debugger for Graphical Workstation, Software Practice and experience, (September 1987), pp.647-662.

[John91] John Bruner, Hoichi Cheong, Alexander Veidenbaum, and Pen-Chung Yew, CHIEF: parallel simulation environment for parallel systems, Report No. 1050 center of supercomputing research and development, University of Illinois, (April 1991).

[Keij91] Keijiro Araki, Zengo Furukawa, and Jingde Cheng, A general framework for debugging, IEEE software (May 1991), pp.14-20.

[KMa91] K. Mani and C. Kesselman, Parallel programming in 2001, IEEE Software, (November 1991), pp.11-20.

[Law91] Lawrence A. Crowl, Architectural adaptability in parallel programming, Ph.D. thesis University of Rochester, (May 1991).

[Mar86] Margaret St. Pierre, A simulation environment for schema, M.Sc. Thesis MIT (1986).

[MWU92] Microsoft Corporation, Microsoft Windows User's Guide, (1992).

[Nic89] Nicholas Carriero and David Gelernter, Linda in context, Communications of the ACM 32, 4(April 1989), pp.444-458.

[OAH91] Borland International, Borland Languages Open Architecture Handbook, (1991).

[OWL91] Borland international, Object Windows For C++ User's Guide, (1991).

[Suh86] Suhdhir Ahuja, Nicholas Carriero, and David Gelernter, Linda and friends, Computer (August 1986), pp.26-34.

[Vac90] Vaclv Rajlich and Wafa Khorshid, VIFOR : A Tool for software maintenance, Software Practice and experience, (January 1990), pp.67-77.

[VBar90] V. Barbosa and P. Lima, On the distributed parallel simulation of Hopfiled's Neural Network, Software Practice and experience, (October 1990), pp.967-983.

[VLo90] V. Lo, S. Rajopadhy, M. Mohammed, S. Gupta, and B. Nitzberg, LaRCS: A language for describing parallel computation for purpose of mapping, University of Oregon, (1990).

[Wil89] Willie Schatz, Programming is the problem, Datamation (May 1, 1989), pp.57-61.

[Yi91] Yi Zheng and Jim Hague, DMT-a Demonstration Tool, Software Practice and experience, (September 1991), pp.949-961.

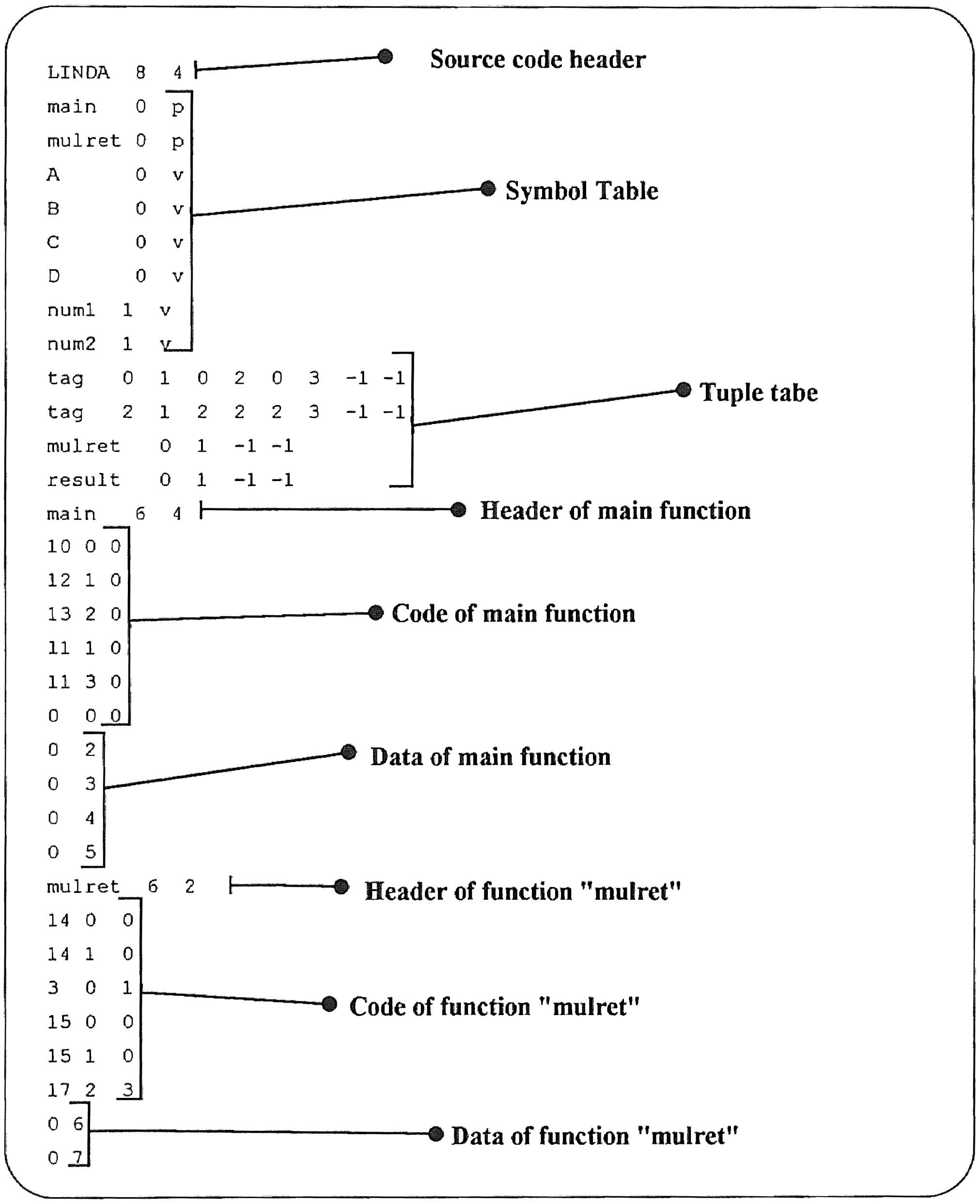
APPENDIXES

APPENDIX A

THE INPUT FILE FORMAT

The Input File Format

The source code input contains a header, symbol table, tuple table, main function and other functions. The header consists of a string "LINDA" followed by the symbol table size and the tuple table size. Also, each function must have a header which contains the name of the function followed by the code segment size and data segment size. The header of the function is followed by the code of the function (code segment) followed by the data used in the function. Each line of the data segment have the initial value for the data element and a pointer to the symbolic name in the symbol table. Figure 23 shows an example of the format.



```
LINDA  8  4
main    0  p
mulret 0  p
A       0  v
B       0  v
C       0  v
D       0  v
num1   1  v
num2   1  v
tag    0  1  0  2  0  3  -1 -1
tag    2  1  2  2  2  3  -1 -1
mulret    0  1  -1 -1
result    0  1  -1 -1
main    6  4
10 0 0
12 1 0
13 2 0
11 1 0
11 3 0
0  0 0
0  2
0  3
0  4
0  5
mulret   6  2
14 0  0
14 1  0
3  0  1
15 0  0
15 1  0
17 2  3
0 6
0 7
```

Figure 23. An Example of Input Source Code.

APPENDIX B

SYNTAX DESCRIPTION OF X-LINDA

The Syntax Description of X-Linda

The following is the syntax description of the Algorithmic language X-Linda.

Note :

[] Means optional.

| Means or.

<> Means defined as LHS.

.. Means range.

```
FunctionDeclaration::=<FunctionHeading>{<FunctionBody>}

FunctionHeading::=function<FunctionName><DataDeclaration>

FunctionBody::=<StatementSequence>

FunctionName::=<Id>

DataDeclaration::=<DataName><DataType>[;<DataDeclaration>]

DataName::=<Id>

DataType::=<Node>|arc

Node::=node|{<DataDeclaration>}

StatementSequence::=<Statement>[;<StatementSequence>]

Statement::=<IfStatement>|

<I/OStatement>|

<AssignmentStatement>|

<WhileStatement>|

<Forstatement>|

<ParallelControlStatement>|
{<StatementSequence>}

IfStatement::=if<Expression>then<Statement>

[else<Statement>]

I/OStatement::=read(<IdList>)|write(<IdList>)

AssignmentStatement::=<Id>'='<Expression>

Whilestatement::=while<Expression><Statement>

ForStatement::=for<Id>'='<Integer>to<Integer><Statement>

ParallelControlStatement::=get(<Tuple>)|

put(<Tuple>)|

readt(<Tuple>)|

eval(<Tuple>)

Expression::=<Id>[<Operator><Id>]|

<Id><Operator><Expression>|

(<Expression>)

Id::=<Letter><Alphanumeric>|<Letter>

Alphanumeric::=<Letter><Alphanumeric>|

<Digit><Alphanumeric>|

<Letter>|

<Digit>

Letter::='A'..'Z'|'a'..'z'

Digit::='0'..'9'

IdList::=<Id>[,<IdList>]

Tuple::=<Operand>[,<Operand>]

Operand::=<Id>|<Number>|<Expression>

Number::=<Integer>|<Real>

Integer::=<Digit>|<Digit><Integer>

Real::=<Digit>[<Integer>]'.'[<Integer>]

[E['+'|'-']<Integer>]

Operator::='>'| '<'| '>='| '<='| '='| '<>'|

'*'| '/'| '+'| '-'
```

APPENDIX C

THE ALGORITHMIC LANGUAGE X-LINDA

The Algorithmic Language X-Linda

X-Linda is an algorithmic language designed to develop algorithms for parallel programs. As mentioned earlier, the model for parallelism adapted for this language is Linda. The following is a description of X-Linda :

Function

A program in X-Linda is a collection of functions capable of running in parallel or sequential according to the application. A function is parallel if it can run simultaneously with other functions. The following is a template of a function:

```
function   f
        ( Declarations )
        {
         ( Statements  )
        }
```

The Parallel Operations

The parallel control is performed using the tuples. A process can put, get or read a tuple. The first operation is "put" an example of usage is shown below:

1. put("tag",Item1,Item2, .. ,ItemN)

This operation inserts the tuple containing the specified tag and items Item1...ItemN in the tuple space (TS). The "tag" is a string to distinguish between tuples, and the data items are values.

2. get("tag",Item1,?Item2, .. ,ItemN)

The get operation gets a tuple from the tuple space (TS) by matching the "tag". The tuple is read and removed from the tuple space.

3. readt("tag",Item1,?Item2, .. ,ItemN)

The readt operation reads a tuple from (TS) by matching the "tag", but the tuple is not removed from the tuple space.

For both get and readt the prefix "?" before the item name as in "?Item2" indicates that the numeric value of this item is input from the tuple space, otherwise it is used to match the tuple.

4. eval("tag",exp1,exp2, .. ,expN)

This operation create a new process tuple. The "tag" is the same as before but the expression "expi" can be a function to be executed. When the execution of eval ends the tuple becomes a data tuple and remain in the TS until a process get it. The operation eval is useful in creating processes that run in parallel.

Declarations

The language X-linda has only two types of data: 1) node which can hold a numeric value, or can be a structure of nodes and arcs, and 2) an arc which is pointer to a node. The following is an example of a linked list declaration :

```
node1 {
    info   node ;
    next   arc -> node1 ;
    }
```

Control Statements

X-Linda has four types of control statements "sequence", "for", "while" and "if". The "for" statement is used for a fixed number of iterations. For example, the statement "read(n)" will be executed 10 times :

```
for I= 1 to 10  read(n)
```

The "while" statement is used for looping until a condition is satisfied. The condition is expressed as an expression. The while loop continues until the value of the expression is zero. The following while statement continues looping until the value of A is greater than value of B :

```
while A <= B {
            A = A + 1 ;
            write(A)
            }
```

The "if" statement is used to make a decision which statement to execute. The "if" statement can be used with or without the "else" option. The following is an example of an "if" statement with the "else" :

```
    if A > B  then   A = A - 1
    else   A = A + 2
```

I/O Statement

Two operations are used to read input and write output to the user. In the following statements a value for the variable A is read and then the same value is output to the user :

```
read(A) ;
write(A) ;
```

VITA

Mohammed Al-abdulkareem

Candidate for the Degree of

Master of Science

Thesis: A DEVELOPMENT ENVIRONMENT FOR PARALLEL ALGORITHMS BASED ON LINDA

Major Field: Computer Science

Biographical:

Personal Data: Born in Riyadh, Saudi Arabia, November 6, 1964, the son of Abdulrahman and Aisha Al-abdulkareem.

Education: Received Bachelor of Science Degree in Computer Science from King Saud University, Riyadh, Saudi Arabia, 1988; completed the requirements for the Master of Science degree at Oklahoma State University in May 1993.

Professional experience: Teaching assistant, Department of Computer Science, King Saud University, July, 1988, to December 1989. A member of Saudi Computer Society and a student member of ACM.