AN OBJECT-ORIENTED PARALLEL SIMULATION OF TR-MACHINE ARCHITECTURE

By

JUAN DUAN

Master of Philosophy

University of Oklahoma

Norman, Oklahoma

1990

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

July, 1994

AN OBJECT-ORIENTED PARALLEL SIMULATION OF TR-MACHINE ARCHITECTURE

Thesis Approved:

[illegible]

Thesis Advisor

[illegible]

Huizhu Lu

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. K. M. George, for his intelligent supervision, constructive guidance, inspiration, encouragement, and instruction through my thesis research work. My sincere appreciation extends to Dr. Hui Zhu Lu, and Dr. Paul Benjamin for serving on my graduate committee. Their scientific guidance, suggestions and support were very helpful throughout my study. Without their support, motivation, and patience, it would have been difficult to complete this work as it is now.

Moreover, I wish to express my sincere gratitude to those who provided suggestions and assistance for this study. I am also thankful to my respected parents and my friends for their love and support. A special thanks goes to my husband and darling daughter, for their love, their understanding, their strong encouragement at times of difficulty, and sacrifices.

Finally, I would like to thank the Department of Computer Science for supporting me during these two years of study.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis presents the design and implementation of an object-oriented simulation of a new scalable parallel processing architecture called TR-machine, which is based on conventional machine model and graph reduction. The simulator for TR-machine is implemented on a Sequent Symmetry S81 to verify structural properties. The simulator is designed such that TR-machine processors are mapped into physical processors. There are 24 processors (on the Sequent Symmetry machine) for the simulation, which are divided into three levels. These levels of processors correspond to the levels of TR-machine balanced tree structure. The top server node and a host are at the top level. In the middle there are three server nodes which are connected to the top-server node. At the leaf level, there are fifteen computation nodes, five nodes are connected to each mid-level server node. There is a bi-directional dedicated bus for communication traffic between processors in the different tree levels. The language used for the simulation is C++. It provides the implementation with encapsulation, data abstraction and inheritance. The simulator is implemented as a tool to verify TR-machine's structural properties and behavior and to experiment with different instructions. The use of object-oriented approach[6] in the TR-machine simulation simplifies the implementation of parallel communication between the simulation objects.

The rest of this thesis is organized as follows: Chapter II will introduce the environment of simulation. Chapter III will introduce the basic concepts contributing to the TR-machine architecture and some definitions related to the TR-machine such as FP, DelFP, BT-interpreter and reduction computation model. Chapter IV will describe the design of the TR-machine such as structure of TR_machine organization and communication architecture. Chapter V will introduce simulation model, Chapter VI will discuss parallel processing scheme, and Chapter VII will provide summary and conclusions.

CHAPTER II

SIMULATION ENVIRONMENT

This section will introduce the simulation platform Sequent S81 machine and its programming library which support the simulation[8].

2.1 SEQUENT SYSTEM

Sequent system is a shared memory multiprosessor system. All processors are identical, but share a single pool of memory to improve resource sharing and communication among processors. Since a single high-speed bus holds all processors, memory modules and i/o controllers, it is easy to add processors, memory, and i/o bandwidth without change in the operating system. The sequent CPUs are general-purpose, 32-bit, microprocessors. The symmetry series includes Sequent S27 and S81. Sequent S81 contains from 2 to 30 processors. The available machine for the simulation is Sequent S81, which contains 24 processors. It can be configured with 8 to 24 Mbytes of memory, which supports 256 Mbytes of virtual address space per processor. Each CPU has 64 Kbytes of cache RAM. Sequent computers run the DYNIX/ptx operating system, which provides most utilities, libraries and system calls. DYNIX/ptx manages the system load among available processors keeping every available processor busy.

2.2 PARALLEL-PROGRAMMING LIBRARY

In our simulation, DYNIX/ptx parallel-programming library is used to execute loops in parallel. The program controls data flow and synchronization by using tools specially designed for data partitioning. The tools provided include system functions and shared variables.

2.2.1 SYSTEM FUNCTIONS

Microtasking programs create multiple independent processes to execute loop iterations in parallel. The microtasking method has the following characters:

. The parallel processes share some data and create their own private copies of the other data.

. The division of the computing load adjusts automatically to the number of available processes.

The tools we used in our project are listed in Figure 1.

System call	Responsibility
m_fork	execute a subprogram in parallel.
m_get_myid	return process identification number
m_get_numprocs	return number of child processes.
m_kill_procs	terminate child processes.
m_lock	lock a lock
m_multi	end single-process code section
m_next	increment global counter
m_park_procs	suspend child process execution
m_real_procs	resume child process execution
m_set_procs	set number of child process
m_single	begin single-process code section
m_sync	check in at barrier
m_unlock	unlock a lock

Figure 1. programming Library Microtasking Routines

2.2.2 SHARED VARIABLES

The parallel_programming library contains a set of routines for dynamic allocation and management of shared memory. In **C++**, **shmalloc** and **shafree** routines are used to allocate and release shared memory for data structures whose size is determined at run time.

2.3 OBJECT-ORIENTED METHODOLOGY

Object-Oriented approach, derived from Simula[6], facilitate encapsulation, data abstraction, polymorphism, and inheritance. Because of its support for sharing code, and interface, object-oriented approach is viewed as a solution to manage the architecture simulation complexity[3].

In object-oriented programming, objects are used as the units of data encapsulation and classes as the units of data abstraction. These classes typically belong to a hierarchy of classes united via the inheritance relationship[6].

Data encapsulation, data abstraction, polymorphism, and inheritance are essential concepts to minimize interdependencies among the simulation modules. Data abstraction provides the ability to group simulation entities according to their common properties. Polymorphism supports the modeling of common behavior among different object types. Inheritance allows code sharing among classes[6].

Thus object-oriented design approach is used for the design of the simulator. The obvious advantages are encapsulation, data abstraction and inheritance associated to the objected oriented programming language. The TR-machine is built from sets of similar objects. Obviously, the object oriented approach simplifies the implementation of nodes and communication between the simulation objects,

CHAPTER III

BACKGROUND OVERVIEW

The design of the TR-machine is based on several concepts. They are Backus' FP, graph reduction, DelFP - a sequential execution architecture for FP, and BT-Server FP Interpreter - a parallel execution scheme for FP. In the following subsections brief descriptions of these concepts and definitions related to the TR-machine are given. The details of those concepts can be found in [1], [4], [5], [9], and [10].

3.1 FP LANGUAGE

Backus' FP system[1] describes the framework of a reduction programming language. The FP system is proposed as an alternative to the conventional programming which is based on the von Neumann model of the computer. There is no notion of a present state, program counter, or storage in FP language. Instead, a program is just a function in the mathematical sense and they map objects to objects. Backus' FP system consists of four basic parts.

(1) Objects - FP has only one data type called objects. An object is either an atom or a list of objects. Examples of objects are numbers, alphabets, and <1,2,3,<A, B, C>>. A

special symbol $\perp$ (called bottom) also belongs to the objects. This object is not considered in the simulation.

(2) <u>Primitive functions</u> - a set of primitive functions that map objects to objects; these functions perform arithmetic, logical, or list-manipulation operations. Examples of primitive functions are +, *, head, tail, eq, and so on.

(3) <u>Functional forms</u> - The functional programming style can be described as the building of complex functions from simpler ones by using functional forms[1]. The functional forms used in the simulation are described below:

• <u>Composition:</u> $f.g{:}x \equiv f(g(x))$. This is the same as composition of functions. During execution, g(x) is evaluated first. The result is the argument for the execution of function f.

• <u>Construction</u>: A set of functions is transformed into a list of functions. The symbol used to represent construction is [ ]. All functions are applied to the same object and can be evaluated (executed) in parallel. Symbolically, $[f_1, f_2,\ldots, f_n]{:}x \equiv < f_1{:}x, f_2{:}x,\ldots, f_n{:}x>$.

• <u>Apply-to-all</u> Apply a function to all elements of a list. This functional form is represented by either **ALPHA** or α. Symbolically,

$(\alpha\ f){:}<x_1, x_2,\ldots, x_m> \equiv <f{:}x_1, f{:}x_2,\ldots, f{:}x_m>$. The function applications can be done in parallel.

• <u>Condition</u> Symbolically $(p \rightarrow f;\ g){:}x \equiv$ if $p{:}x = T$ then do $f{:}x$ else if $p{:}x = F$ then do $g{:}x$ else $\perp$. In the simulation, this functional form has been modified to simple comparisons and branch.

• <u>Constant</u> This is the same as constant function in mathematics. $c : x = c$.

(4) <u>Application</u> - denoted by the symbol " : " represents function application. For example

f:x means apply the function f to the object x. Program execution computes the result.

Examples of programs and their machine language translations will be provided in chapter IV.

3.2 COMPUTATION MODEL

The underlying computation model of the TR-machine architecture can be termed parallel tree reduction. The reduction model can be illustrated by using a set of examples. Each of the FP program examples is represented as a tree and the transformations to the tree at the various levels of computation, and the result are shown.

In the examples shown in the figures 2a - 2d , the nodes labeled ":" represent computation (or application) and the nodes labeled "." represent a list. As the examples illustrate, the computation can be expressed as a tree and evaluation proceeds from lower level to the root in response to a demand from the root. Computation at a level needs to be complete before the next higher level computation can be performed (because FP uses call-by-value semantics.) Each independent computation can be mapped into a processor. The processor mapping is illustrated in the figure 3. using the expression of example from figure 2d. The boxes identified as P1 and P2 represent computations that can be performed in parallel. The result of the computation will be a list which becomes the operand at the next level which is shown in the box labeled P.

The computations that can be performed are called ***active computations.*** As can be seen from figures 2a - 2d, if there is only one active computation, one processor can handle it. If there are several active computations mapped into several processors, we need a mechanism to keep track of the computations and to collect the results into a list.

Therefore, the computations can be classified into two types - "type a" and "type b" as shown in figure 2e. In figure 2e, C, C1, C2,...,Cn represent active computations, S represents the mechanism to keep track of the parallel computations. Every set of active computations can be represented by a tree with at most two levels. This organization suggests an embedding of the computation into a balanced tree. A single active computation is embedded into a leaf. In the case of a two level tree, leaves are mapped into leaves and the root is mapped into an interior node.

The case of functional form condition needs special attention. A sequential approach is adopted as illustrated in figure 4. The predicate ρ is evaluated first and then based on the result, the appropriate function will be evaluated to produce the result.

The TR-machine organization is based on this computation model. Type a computations can be mapped into a processor. Type b computations can be mapped into a set of processors preserving the two level tree structure. The root of the tree will keep track of computations and form the result as a list.

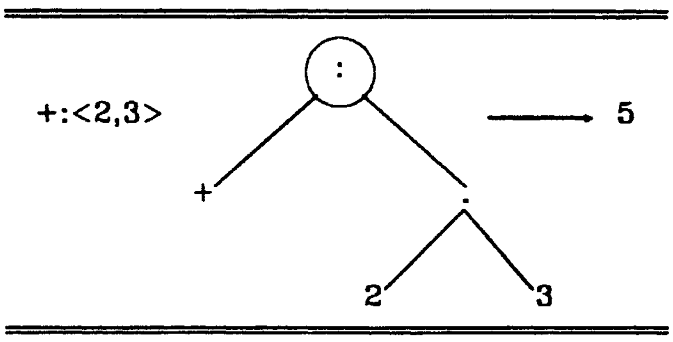

Figure 2 a. Apply Primitive Function.

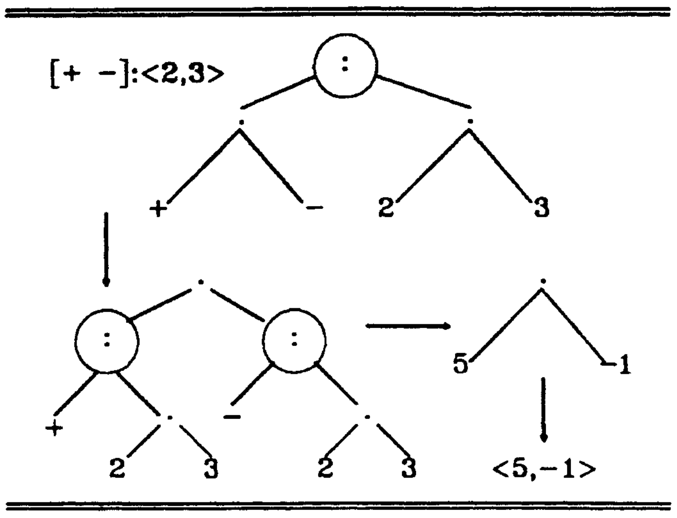

Figure 2 b. Apply a Construction.

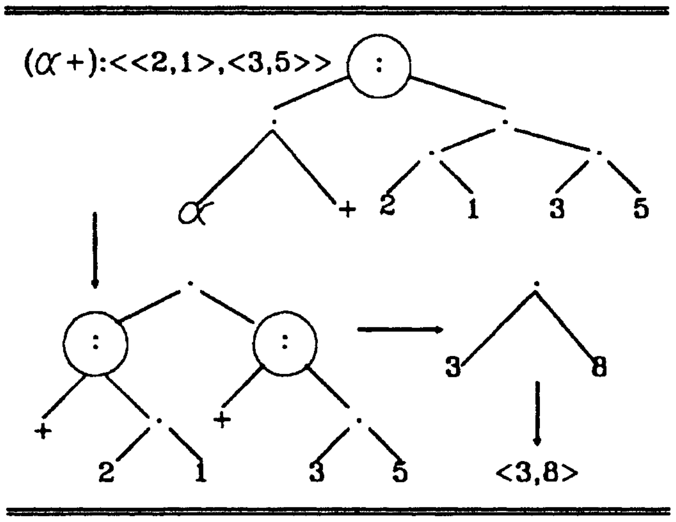

Figure 2 c. Apply ALPHA.

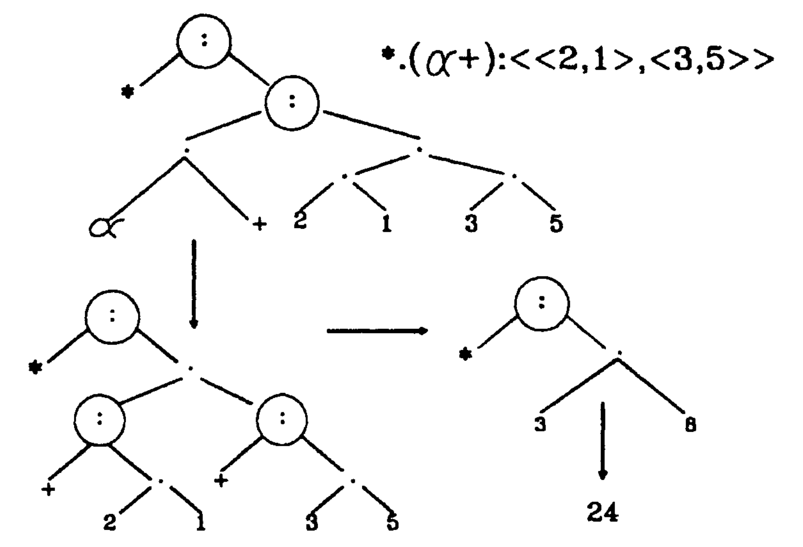

Figure 2 d. Composition of * and (ALPHA +).

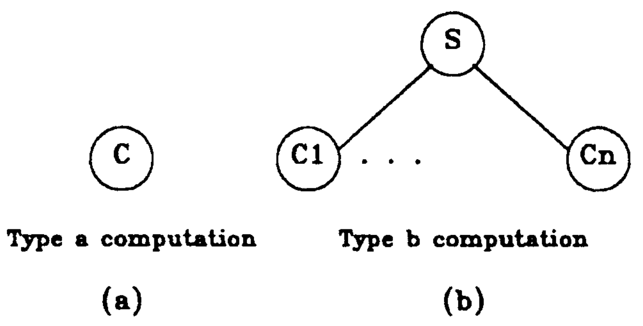

Figure 2 e. Active Computations.

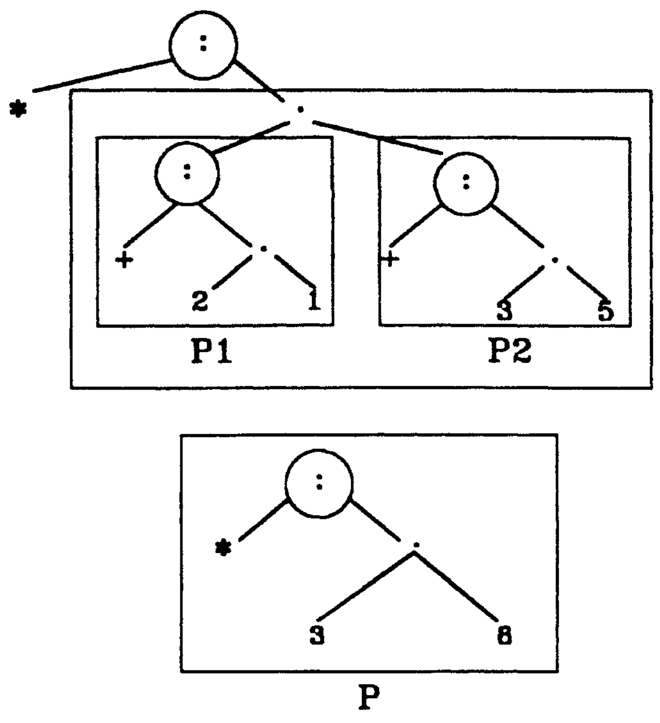

Figure 3. Representation of Active Computations.

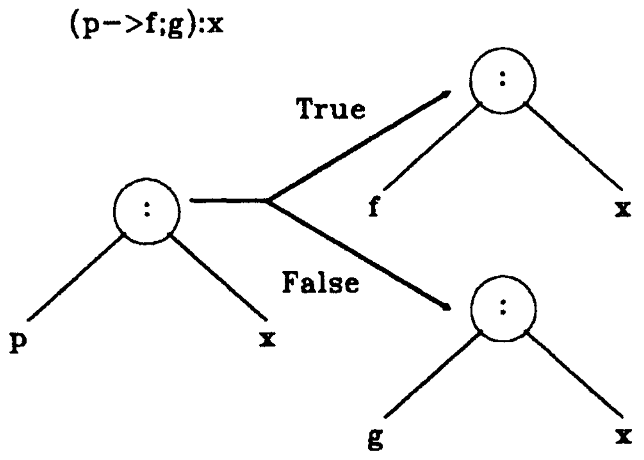

Figure 4. Conditional form.

3.3 DelFP ARCHITECTURE

The architecture DelFP[5] is designed to directly execute FP language in conventional sequential machines. The structure of DelFP is language sensitive and parallels the structure of FP. In FP there are no data addresses, offsets, or variables. Since DelFP is in close correspondence with the source high-level language, the FP language, it absorbs these features.

DelFP operators are functional forms. The operands are primitive or user-defined functions. During the DelFP execution the value of a function is computed and its controls are encoded in implicit format. DelFP can be defined in terms of:

- an instruction set;
- a set of residual control variables describing the interpretive environment at any execution point
- a contour model for retention of activation records and
- a set of format rules to determine the location of data manipulation during each computation step.

The execution of DelFP programs uses the contour model. Functions in a construction are pushed onto a stack and evaluated one by one. A stack of object pointers keep track of the results of the individual computations.

3.4 BT-SERVER FP INTERPRETER ARCHITECTURE

BT-sever FP interpreter[7] describes a parallel implementation scheme for a massively-parallel FP architecture. BT-server FP interpreter's architecture is a massively-parallel FP computer. BT-sever has a balanced tree structure with the leaf nodes being distinct from the internal nodes. The leaf nodes perform computation while the internal nodes perform subtask/node management functions. BT-server uses exclusively distributed memory and massage passing to implement the hardware communication. An FP language is used as the machine language.

CHAPTER IV

TR-MACHINE ARCHITECTURE DESIGN

In the following sections, we briefly discuss the TR-machine instruction set architecture, instruction types, structure of TR-machine organization, communication architecture, and communication between nodes.

4.1 TR-MACHINE INSTRUCTION SET

The machine language of the TR-machine is based on DelFP which is designed to execute FP in conventional sequential processor machines. Because of its close correspondence with the FP language, it is straightforward to develop translators between FP and TR-machine instruction. The following subsections will discuss the instruction architecture such as instruction types, and instruction format.

4.1.1 TR-MACHINE INSTRUCTION TYPES

TR-machine programs are similar to FP programs. The types of instructions are chosen corresponding to different functional forms of FP language[5]. As we will see later, TR-instructions are used the same way as the FP functional forms to build TR-programs. Instructions and data are disjoint. Data in the TR-machine simulation is assumed to be FP objects. An object is an atom or a list of objects, where the atom is a number or character. The instruction types implemented in this research are discussed below (see Figure 5 for opcodes and their encoding):

APPLY (APF) An apply instruction is an application of primitive function. Its operand is a pointer to the function being applied. The result of executing this instruction is to evaluate the operand function with the current object as the operand. The address of the current object will be in a dedicated register (register S).

APPLY LEFT SELECTION (ALS) This instruction selects the n^{th} element from the left in the current object. The operand is an integer n encoded as the operand. The current object is expected to be a list.

APPLY RIGHT SELECTION (ARS) This instruction is same as ALS except that the selection is from the right of the list.

APPLY CONSTANT FUNCTION (ACS) This returns a constant object. Its operand is a pointer to the constant object table containing the value to be returned.

CONDITION (COND) Execute the first operand **P** on the object, If the result is true , the second function **F** will be executed, else the third operand function **G** will be executed.

CONDITION (CONDA) Execute first operand which is address of next instruction to be executed, if control flag is true. The current object does not change.

CONSTRUCTION(BEG...END) There are two instructions "BEG" and "END". The operand of construction instruction can be divided into two parts. One marks the beginning of construction and the other marks the end of construction. The instruction between the two "BEG" and "END" are groups of instructions(functions) that can be executed in parallel. The result of executing this construction instruction is to distribute the computations to different computation nodes(processors). The functions are evaluated in parallel. The "END" instruction marks the end of construction and has no operand.

APPLY TO ALL (ALPHA) The operand of this instruction is a function that can be applied to each element of the current object. The input object is expected to be a list. The operand function is distributed to different nodes for parallel computation with appropriate components of the list. The semantics of construction and apply-to-all are different from DelFP instructions because they initiate parallel computation.

JUMP(JMP) Operand of this instruction is a pointer to a local program to be executed with the current input. This instruction is included in the architecture in order to implement recursion and the while functional form in the FP system. This choice is made so as to keep the instruction set simple.

OPCODES	
APF	0
ACS	1
ALS	2
ALR	3
COND	4
ALPHA	5
BEG	6
END	7
JMP	8
CONDA	9

Figure 5. Opcode matching table

Among the above instructions **'BEG'** and **'ALPHA'** contribute to parallel computation.

4.1.2 THE INSTRUCTION SET FORMAT

The instruction set structure of the TR-machine is represented by a packet (the idea of packet is originally introduced in [2]), this packet is a record divided into several fields as shown figure 6.

```
C-closure | tags | opcode | operand
```

Figure 6. The Instruction Format

The fields opcode and operand have the usual meaning. In the simulator, opcodes represent functional forms. Operands are either addresses or pointers to functions. The C-closure field is used to identify the components of a construction efficiently. It specifies the construction to which the function belongs, (specified by the beginning and end addresses of the construction). If C-closure of an instruction matches the operand field of construction, then it is a member function of that construction(see Figure 9). The tag field is used to specify the position of an instruction relative to the functional forms composition and construction. The tag field may contain one or more of the following tags given in figure 7.

Tag	Meaning
N	End of Construction
B	Sequential Begin
E	Sequential end
C	Construction Begin
I	Inside Construction
S	Inside Sequence
U	Unary block

Figure 7. Tags and meanings

Figure 8 illustrates the correspondence between FP functional forms and TR-instructions. The TR-instructions are listed in the first column. The second column gives a FP function application, and the third column lists the equivalent TR-program. The fourth column gives the result. In the TR-instructions, the primitive function symbols themselves are used to represent operands for the sake of clarity.

Instruction type	FP-Instruction	TR-Instruction	Result
APPLY	+:<1 3>	-- APF +	4
ALS	2L:<1 2 3>	-- ALS	2
ARS	3R:<1 2 3>	-- ARS 3	1
ACS	:5	-- ACS	5
COND	(eq*;-):<2 1>	-- COND eq * -	1
CONDA	N/A	-- Control flag = t, go -- to address	address
ALPHA	+ :<<1 2> <3 4>>	-- ALPHA +	<3 7>
BEG	[+ - *]:<3 2>	C- BEG p q I- APF * I- APF - I- APF + N- END	<5 1 6>

Figure 8. FP-TR instructions mapping.

4.2 TR-MACHINE ORGANIZATION

The organization of the TR-machine is adopted from the BT-Server FP interpreter [7]. It is organized as a balanced tree structure. Each node of the tree is a processor. The nodes of the tree belong to two distinct classes. The leaf nodes form the class of computing nodes (or C-nodes) and the internal nodes form the class of sever nodes (or S-nodes). The class of server nodes serve as a communication and book-keeping network. The C-nodes perform actual computation. The root of the tree is called the top server node and it is connected to a host. The structure of the tree used in the simulation is shown in Figure 10.

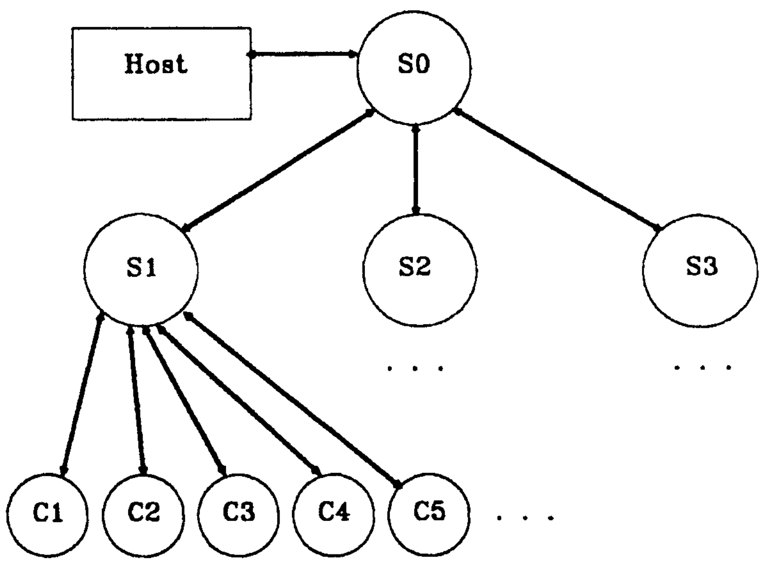

Figure 10. TR-machine Structure.

4.2.1 MAJOR COMPONENTS

The host is the connection between the top server node (the root of the tree) and the user. The function of the host is to pass information back and forth between the user and the TR-machine. Actually it is responsible to convert FP language into TR-machine language and receive the final result from server node.

S-node responsibilities also include keeping track of function requests arriving from the host node, sending subtasks to a free child computing node, handling subtasks overflow, collecting the subtasks' results of a parallel task, and returning the final result to the host.

The computing nodes (leaf nodes or C-nodes) receive and store intermediate subtasks from their parent S-node, execute requested subtasks, send the results back to their parent S-node, and generate new requests for subtasks of a parallel task. The C-node is considered as a microprocessor with its own CPU and local memory. Inter-node communication is accomplished by using dedicated buses[3], and [4].

4.2.2 TREE NODES' FUNCTIONS

The host node stores the instruction set in the instruction memory, objects (data) in the data memory, and sends request package, and reports final results. In order to avoid the bottle-neck problem as a result of the communication of several C-nodes trying to use the same object in-site in the global data memory (in a parallel computation), the instruction set is designed to pass a copy of the appropriate object to be made available to the C-nodes. This design is expected to have better performance.

Top sever node and child server nodes use a list of request packages, a list of mark packages, and a list of join packages for communication. These packages may be stored in local memory. The server nodes have status registers of its children nodes. The status registers indicate whether the C-nodes are free. The top server node also keeps track of the id of its children server nodes whenever it sends massages (see Figure 11). It also has a control flag to keep track of which child node is working for which level and which subtask is being executed. Type a computation involves only one C-node. But, type b computation involves several C-nodes. So The root of the computation tree (refer to Figure 2e) will be associated to an S-node which will have the responsibility to collect the results of the subtasks.

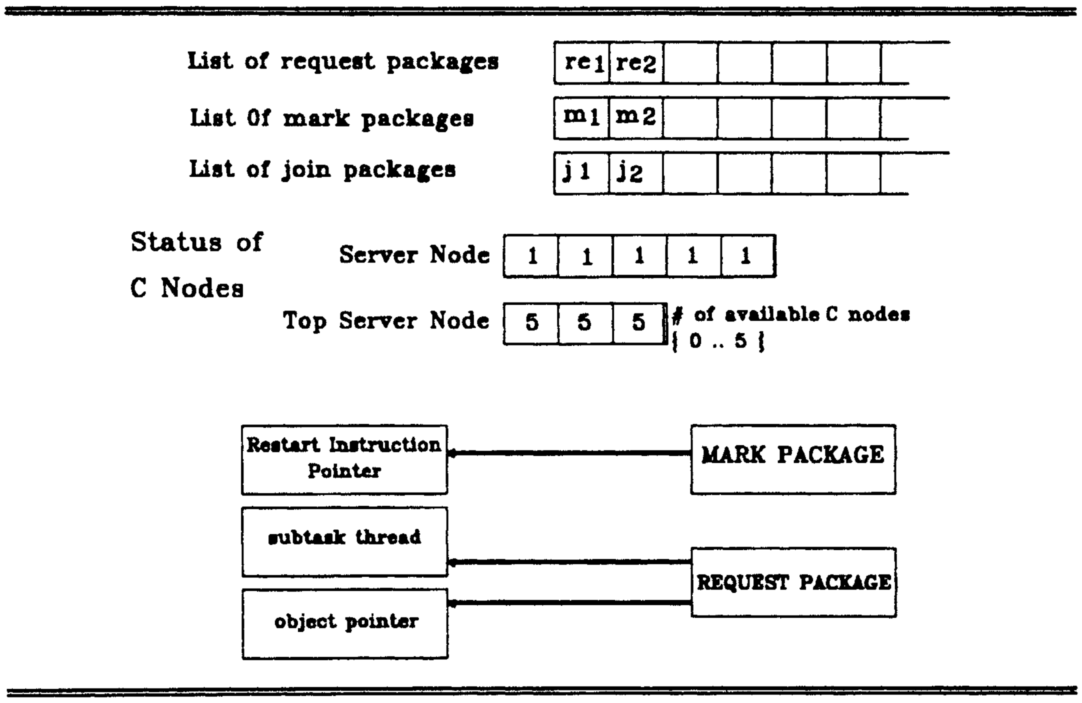

Figure 11. Information stored in Top S-node and Interior S-nodes.

The C-nodes are responsible for computation. They are microprocessors. The components of a C-node include a primitive function table, primitive function units, source-pointer, D-pointer, instruction pointer, control flag, list of request packages, join package, mark package, and local memory to store the object and intermediate results during performing the computation. Figure 12 illustrates the primary components of a C-node.

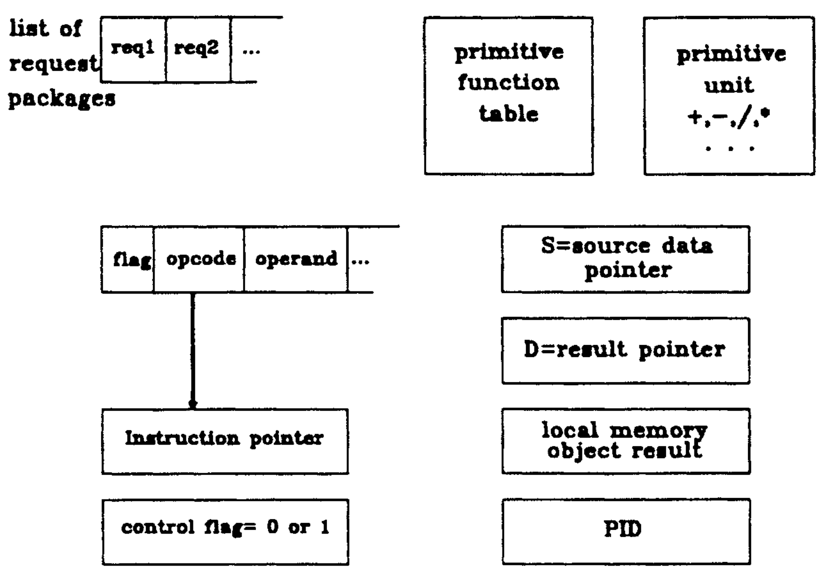

Figure 12. C-node Organization.

4.3 COMMUNICATION ARCHITECTURE

The basic idea of the TR-machine is that: programs are executed part of the time sequentially and part of the time in parallel. As long as parallel computation is not required, computation can be limited to one processor. When execution reach the point which need parallelism, the programs and corresponding data are sent to available computing nodes. These sub-tasks are routed to different computing nodes through a S-node which keeps track of computation. The S-node collects the results from C-nodes, and constructs a list which will be the argument to the next function to be executed. S-node also keeps track of available children C-nodes.

There is no intra-level communication between nodes. Inter-level communication is accomplished by synchronized message passing. The simulation in this thesis assumes the top server node has three child server nodes, and each child server node has five C-nodes (refer to Figure 10). There are three types of information buses, also called packages, that are used to provide communication among nodes. Each package identifies a different event in the computation process and carries appropriate values.

4.3.1 TYPES OF PACKAGES

The communication packages used are called request package, join package, and mark package. Next subsections will give brief description of request package, join package and mark package. The features of the packages are shown in Figure 13.

4.3.1.1 REQUEST PACKAGE

Request package is a message used to initiate computation. Request package will carry Job id (Jid), source data which contains the objects (data to perform the task from the data memory location), instructions, address of data or result for the next instruction in C-node local memory, pointer to the address of the first instruction of this subtask in the global memory (program begin or pbeg), the address of the last instruction of this subtask in the global memory (program end or pend) and process id (Pid).

4.3.1.2 MARK PACKAGE

A mark package contains necessary information to form lists from results of subtasks. Mark package will carry previous job id (ex_Jid), Jid, number of expected results, list of locations for expected results, program counter for the parallel instruction, and previous beginning and end of the task to be forked to parallel subtasks.

4.3.1.3 JOIN PACKAGE

Join package is used to send results of subtasks. Join package will carry Jid which is equivalent to the current request package's Jid, expected result from C-node, and the location of the result relative to the expected results in the mark package.

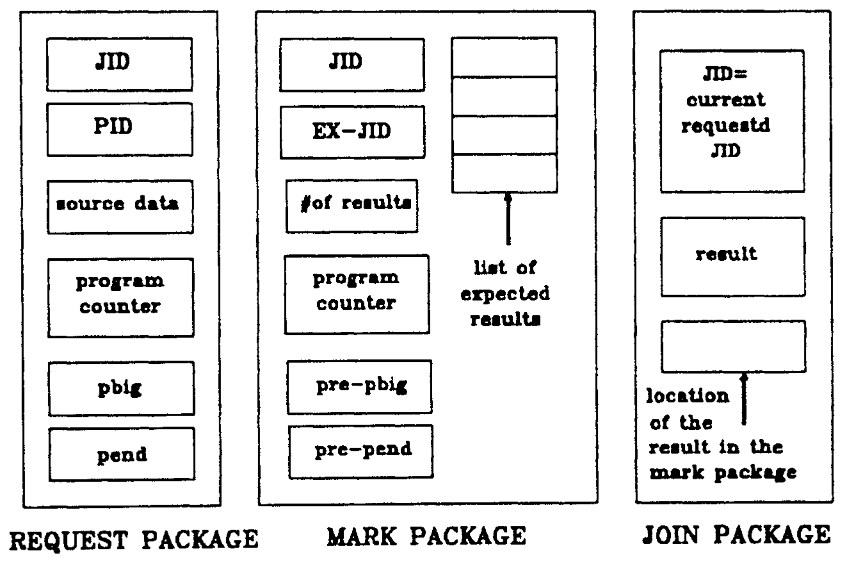

Figure 13. Communication Packages.

CHAPTER V

SIMULATION MODEL

This chapter will outline the simulation. Internode communication, simulation architecture and instruction execution scheme are described. The simulation model in pseudo code format is given in appendix A.

5.1 COMMUNICATION BETWEEN NODES

The conceptual communication scheme between nodes is shown in figure 14. Once the host node gets a task (instruction set or sets and objects), a new request package with JID (job identifier) is created. The top server node will pass the request package from host to the leftmost free child S-node, say S1, with its id '0', and S1 in turn will send the request package to leftmost available C-node of its children with its id '1', say C1. The C-node C1, which has id '1' and parent id '1', will check whether the task is sequential or parallel according to the information the request package provided.

If the instruction is 'APF' for example, which is apply primitive function, then C1 will finish computation sequentially, put the result into a join package, and ship it back to its parent S1, where it will not find a mark package with the same id; so it will continue up to the root S-node and then to the host node.

If the request package contains instruction set that can run in parallel, then a mark package will be created for this task and a new request package will be created for each parallel subtasks. The mark package will keep a record for the expected number of results. C1 node will keep the first request package to compute, and send the other request packages and mark package to S1. When C1 completes execution of its request package, it will send a join package with the result to its parent S1. S1 will receive the remaining request packages and it will send them to other free child C-nodes. If no child is free at that time, S1 will send the remaining request packages to the root S-node to try to send them to other C-nodes under other S-nodes. At the same time S1 will send the mark package to the root S-node.

The S-node S1 will start to receive the join packages from C1 and other C-nodes with the expected results. These join packages will put their results in the mark package which has the same id. When all expected results arrive in the mark package, a new request package will be created with the ex_id, ex_pbeg, and ex_pend (stored in the mark package). S1 will send this request package to C1 to continue the main task execution.

For example, when an alpha-instruction or BEG instruction is encountered, several sub-computations are possible. One subtask is done by the C-node currently executing the program and other subtasks will be sent to the parent sever node to be distributed to available computing nodes. The address of the instruction following the scope of alpha or END instruction will also be sent to the parent S-node. If a S-node does not have any free computing node under it, the subtask will be forwarded upwards. When parallel execution is completed the results will be collected by the server node and computation will be started from the instruction following the scope.

In the simulator we were limited by the available machine which has 24 processors (with maximum 20 processors to use in any single program). Simulation machine was built as a tree structured architecture consisting of the host, one root server node, three server children nodes, and fifteen computation nodes as the tree leaves in addition to one processor for the main simulation driver function. Each node is considered as an object.

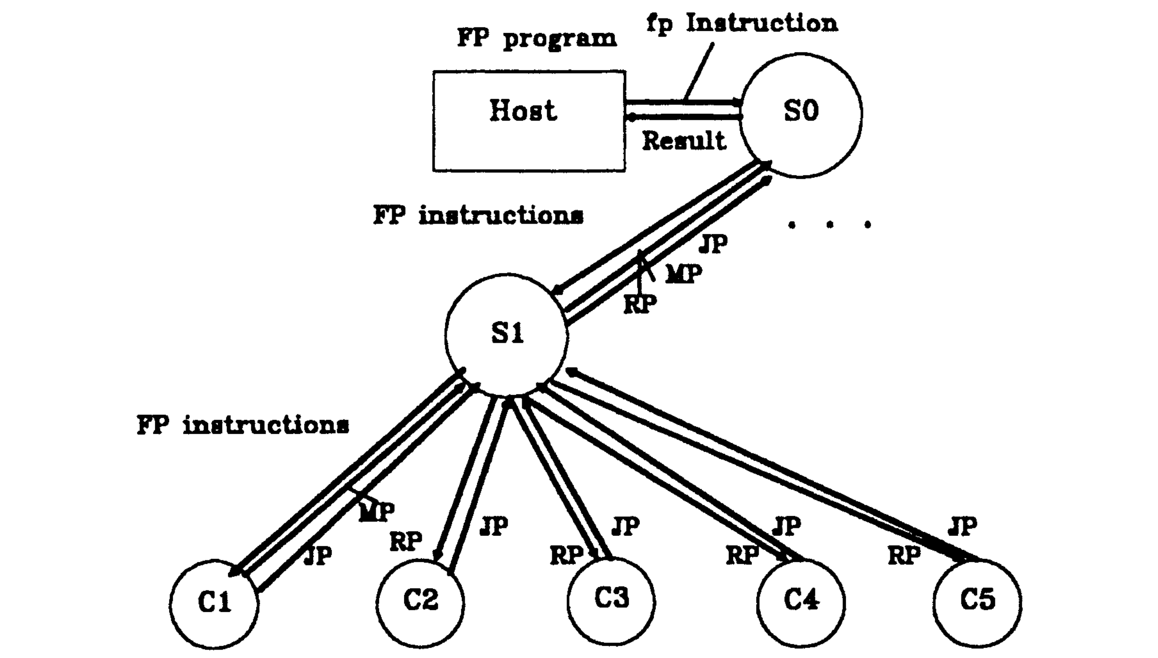

Figure 14. Communication Between Nodes.

5.2 OBJECT-ORIENTED SIMULATION ARCHITECTURE

C++ programming language supports the efficiency of simulation modules. Simulation objects are modeled by C++ classes. Object-oriented methodology has been chosen as the basic paradigm for the simulation, because it processes the necessary characteristics to achieve facilitating sequential and parallel processing environment, increasing the simulation design maintainability, extendabiltity and reuseability.

In the simulation design, request package, mark package, and join package as well as the data list are constructed as objects that interact with one another by sending messages. They may be executed in parallel. In the shared-memory multiprocessor environment the object messages are mapped to shared variables which are network channels among nodes.

Package class contains three subclasses which are request package, mark package and join package, all these three different package classes inherit the features of the package class. In the simulation design, there are host, C-node, and S-node classes which inherit from node class, each of the classes contains subclasses, such as data object class, request package class, mark package class, and join package class. C-node class describes 15 parallel computation instances of C-node or objects. Since any feature of node class inherited can be renamed or redefined in object-oriented methodology, C-node and S-node are inherited from node class, each of the two classes redefines the gettop, recv_req, send_mark, send_join, and send_req methods. Object-oriented method provides the simulation useful constructs for representation of simulation entities. Both design and implementation of the simulator is done in the object-oriented framework.

The implemented simulator consists of the following classes (detailed description of classes are provided in appendix B):

REQUEST CLASS matches request package,

JOIN PACK CLASS matches the join package and

MARK PACK CLASS matches mark package.

HOST CLASS represents the host node. It is responsible for loading instruction sets and data into memory and receiving and displaying final result from join package.

C-node CLASS matches C-node. It gets data object from request package sent by the parent S-node, executes the instruction sets, calculates primitive functions, determines which instruction sets are sequential or parallel, creates and sends join package upwards, creates and sends mark package up, creates and sends request package up.

S-node CLASS implements S-node. It creates list of request packages, mark packages and join packages, It receives join package, puts result in mark package, receives mark package, sends request package, sends join package, and sends mark package up.

5.3 INSTRUCTION EXECUTION

The instruction execution cycle consists of two major steps, namely decode and execute. The actions performed are described below:

CASE (a) opcode=APF: check the operand to find the primitive function code and apply the primitive function on the object in the local memory and return the result in the local memory, using the primitive functions unit table.

CASE (b) opcode=ALS or ARS: execute them by selecting the nth element from left (or right), where n is the operand, from the current object.

CASE (c) opcode=ACS: return the constant value that the operand points to.

CASE (d) opcode=COND: execute the first operand function **P** on the object, if the result is true the second operand function **F** will be executed, else the third operand function **G** will be executed. All the operand functions are executed depending on their type, primitive, ALS, ALR, ACS, ALPHA, or Construction.

CASE (e) opcode=CONDA: execute the first operand user-defined function P, which is the address of the next instruction set, if control flag is true.

CASE (f) opcode=BEG: check the tag to find instructions that can run in parallel until we arrive at the instruction END. Create a mark package with ex_Jid = the current Jid, and Jid = Pid. Create a new request package for each of these subtasks all with Jid = the current Pid, send the mark package to the parent S-node, keep the left most request package to be executed in this C-node, and send the other request packages to the S-node.

CASE (f) opcode=ALPHA: check number of items in its object in the local memory to find the number of new request packages needed, create a mark package with ex_Jid=the current Jid, and Jid=Pid and create a new request package for each of these subtasks all with Jid= the current Pid, send the mark package to the parent S-node and keep the right most request package to be executed in this C-node, and send the other request packages to the S-node.

CHAPTER VI

PARALLEL PROCESSING SCHEME

One of the important characteristics of the simulator is parallelism. The simulator models parallelism in the TR-machine using the system provided features. Using DYNIX/pts function **m_fork**, 20 processes are created, one for each node with its own id **(pid)**. Every process has its parent id **(parent id)** so as to send message to its parent process. A set of two dimensional array (Sharedvalue) is designed as bus to send and receive the three packages back and forth through different nodes (see figure 15). In figure 15, PI represents processor's own id, while PPI represents the parent's processor id. There are four flag values in sharedvalue that are used to identify the different messages[3].

If **sharedvalue.flag** = **0**, request package sending or receiving,

if **sharedvalue.flag** = **1**, mark package sending or receiving,

if **sharedvalue.flag** = **2**, join package sending or receiving and

if **sharedvalue.flag** = **-1**, no active package.

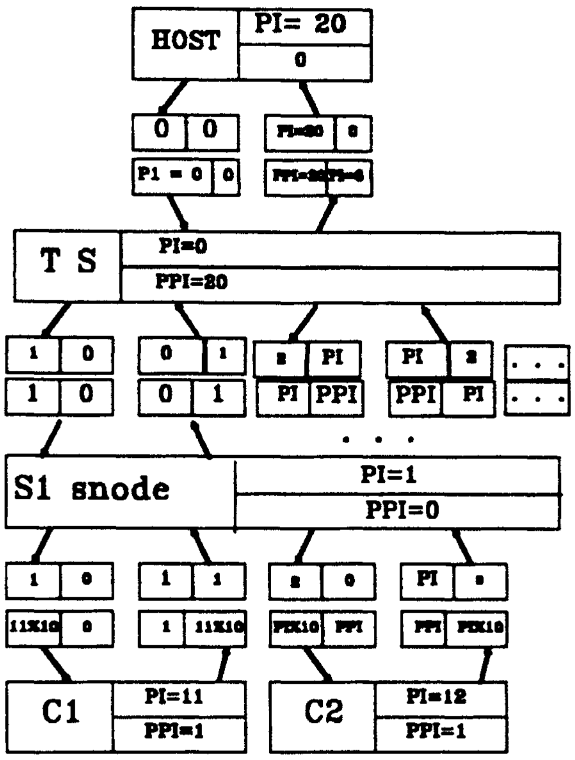

Figure 15. Using shared variables for communication

Figure 15 shows that 'HOST' represents the host node processor with id '20' and with no parent id. TS means Top S-node processor with its id '0' and parent id '20'. Top S-node is connected with three S-node processors with their ids '1', '2', or '3' and their parent is Top S-node with id '0'. S1 means first S-node with its id '1'. Each S-node is connected to five C-nodes. C-nodes have parent ids '1' for S1, '2' for S2, and '3' for S3. The C-nodes which are children of S1 have their own ids '11' for C1, '12' for C2, '13' for

C3, '14' for C4, and '15' for C5. C-nodes children of S2 have their own ids '21' for C6, '22' for C7, '23' for C8, '24' for C9, and '25' for C10, etc.

The first step of communication and message passing in parallel among nodes is to identify the processor's id 'PI' and its parent processor id 'PPI', to decide which sharedvalue should be used to send or receive packages. For example, S-node S1 with its id PI = "1" and parent node id "0", uses sharedvalue[1][0] to receive packages from its parent (top S-node), and uses sharedvalue[0][1] to send packages to its parent (top S-node). On the other hand, S1 has five child C-nodes with id PI=11 to 15. If S1 tries to communicate with its C-nodes, sharedvalue[n][1] is used to send packages to its nth child C-node, and sharedvalue[1][n] is used to receive packages from its nth child C-node. C1 has its id PI=11, and its parent S1 has id PPI=1. C1 uses sharedvalue[11%10][0] to receive packages from its parent, and uses sharedvalue[1][11%10] to send packages to its parent. '%' method is used to match the C-node child location related to its parent. For example, C1 is the first child for S1, C1 has id PI=11, so its PI=11%10=1. It uses shardvalue[1][1]. C2 is the second child for S1, C2's id 12, so its PI=12%10=2. The shardvalue[1][2] is used by C2. If Cn is the nth child for S1, Cn's id is 10+n; so its PI=(10+n)%10=n, and shardvalue[1][n] is used.

CHAPTER VII

SUMMARY AND CONCLUSION

An object-oriented simulation of a multiprocessor computer architecture TR-machine has been presented, which is based on conventional machine model and graph reduction. The simulator has been implemented on Sequent Symmetry S81 running DYNIX/ptx operating system which provides microtasking environment to support parallel simulation to achieve high performance. The object-oriented language C++ has been used to implement the simulation. From our experience of simulation, we noticed that the TR-machine is built from sets of similar objects, therefore, object-oriented approach matches the structure of simulation very well. It simplifies the implementation of nodes and parallel communication between the simulation objects and makes the communication between node much easier. Parallel programming has supported simulation to achieve high degree of performance in terms of validation, as observed by others, object-oriented approach seems to be a good solution to manage the architecture simulation complexity.

Object-oriented technique adopted to design the simulation, helped to develop a flexible simulation model which supports its changeability and reuseability. The user of the simulation can easily implement different formats of instruction sets, instruction types and receive expected results. We have used the simulator to try different instruction types and formats. If these variety of changes were implemented by using conventional method, it would be very time consuming and may cause design changes. Besides, the classes and objects provided the facility to manage the time clocks on different nodes and maintain the output correctly. Several example programs are run on the simulator to verify correctness. A sample set of programs is provided in appendix C. Results from a performance study is shown in Figure 16. Program size is measured as instructions modulo parallelism. For example, if two instructions are executed in parallel, they are counted as one. The CPU time does not include communication cost. Figure 16 shows good performance. Development of a performance model that includes communication cost is considered for future work.

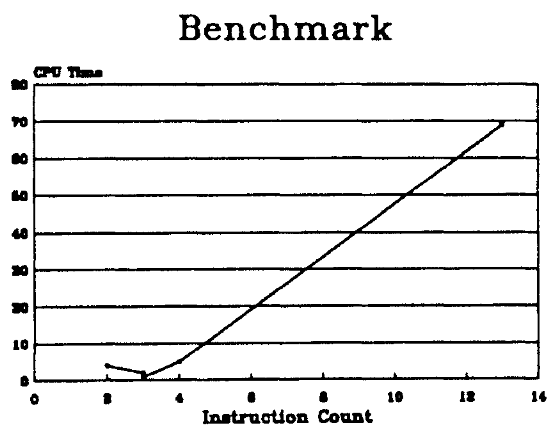

Figure 16. programs size and execution time

REFERENCES

[1] Backus, J. "Can Programming be Liberated from the von Neuman style? A Functional Style and its Algebra of Programs," CACM, August, 1978, pp.613-641.

[2] Cripps, M.D., Darlington J., Field A.J., Harrison, P.G., and Reeve, M.J., "The Design and Implementation of ALICE: a parallel Graph Reduction Machine", Proceeding of the Workshop on Graph Reduction, 1987, pp. 300-322.

[3] Duan J., George, K. M., and Lu H. Z., "Object-Oriented Simulation of Multiprocessing Architecture", to be published, proceeding of the Summer Computer Simulation (SCSC) Conference , San Diego, July, 1964.

[4] George, K. M., and Duan, J., "TR-machine Architecture", to be published, proceedings of the Massively Parallel Computing Systems (MPCS) Conference, Ischia, Italy, May 1994.

[5] Huynh, T., Hailpern, B., Hoevel, L.W. " An execution Architecture for FP", IBM J. RES DEVELOP, VOL. 30 NO.6 November 1986, pp. 609-616.

[6] Kirkerud, B., "Object-Oriented Programming with Simula", Addison-Welsey Publishing Company, Inc., Reading, MA, 1989.

[7] Ong, Teng, E., George, K.M., Teague, Keith, A. "BT-SERVER FP Interpreter", The Fifth Distributed memory Computing Conference, April, 1990, pp.1147-1151.

[8] Rochkind, M.J. "Guide to Parallel Programming", Prentice-Hall, 1985.

[9] Steven R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", IEEE Transaction Computer, Vol C-23, Number 12, December 1984, pp.1050-1071.

[10] Tsanakas P., Alendridis N., and Parakonstantinou G., "An FP-based Design

Methodology for Problem-oriented Architecture", The Computer Journal, vol. 32, no. 5 1989, pp. 453-460.

APPENDICES

APPENDIX A

SIMULATION MODEL

SIMULATION DRIVER

```
begin
        initial shared variables for communication between node;
        initial list of request package;
        initial  list of mark package;
        initial list of join package;
        m_set_procs (number of processors = 20)
        //create  20 parallel processors for simulation
        m_fork (simulation module, data memory, instruction memory)
        m_killprocs(processors);
end;
```

SIMULATION MODULE

```
simulate 20 processors to match TR-machine tree structure
begin
      m_get_myid( get process id );
      switch( process id)
            case HOST:
                  create host(host_id,no parent node)
                  while TRUE do
                  m_lock()
                        if_new_program_arrives
                              load_into_instruction memory(program
                               instructions)
                              load_into_datamemory(program objects)
                              create_request_package(new program)
                              send_request_package()
                        else
                              check_receive_join_package(program result)
                  m_unlock();
                  endwhile


            case TOP_S-node:
                  create top_S-node(top_S-node_id,host id)
                  // set up status of available C-nodes  under each child S-node
```

```
        while TRUE do
        m_lock()
              if receive_request_package(new_program_from_host)
              try_send_request_package(to_available_child_S-node)

                    check_receive_mark_package(from_child_S-node)
                     if receive_join_package(from_child_S-node)
                          if find_mark_package(join_package)
                                put_result_in(mark_package)
                          else try_send_join_package(to_host)
                     if receive_request_package(from_child_S-node)
                          try send_request_package
                          to_available_child_S-node
                    else keep_in(request_package)
         m_unlock()
  endwhile.
```

case MID_S-node:

```
          create mid_S-node(mid_S-node_id,parent_S-node id)
             // set up status of available C-node  under this mid_S-node
    while TRUE do
        m_lock()
               if receive_request_package(from_parent_S-node)
               try_send_request_package(to_available_child_C-node)
               if receive_request_package(from_child_C-node)
               try_send_request_package(to_available_child_S-node)
             else if part_of parallel
                   try_send_mark_package(to_parent_S-node)
                   try_send_request_package(to_parent_S-node)
                   check_receive_mark_package(from_child_C-node)
                if receive_join_package(from_child_S-node)
                   if find_mark_package(join_package)
                         put_result_in(mark_package)
                   else try_send_join_package(to_parent_S-node)
           m_unlock()
        endwhile.
```

case C-node:

```
        create C-node(C-node_id,parent_S-node_id)
         while TRUE do
              m_lock()
                 if receive_request_package(from_parent_S-node)
                        if parallel_instruction
                        create_mark_package()
                               send_mark_package()
                               for(parallel_elements-1)
                        create_request_package()
                        send_request_package()
                        create_request_package(last_instruction)
                        do_computation(last_request_package)
                    else do_computation
                        create_join_package(result)
                        send_join_package()
        m_unlock()
    endwhile.
```

APPENDIX B

CLASSES OF THE SIMULATION

The implememted simulator consists of following classes which build the architecture of the simulation:

```
// instruction set structure
structure of  instruction {
      int closure[2]; //to identify the components of a construction;
      char flag[2];  //to specify the position of an instruction related to
                   // the functional forms composition and construction;
      int opcode;      //such as auf, apf, als, ars, cond, insert etc;
      int operand[3];  //either addresses pointer to functions or primitive functions.
      };
// class of object (input data)
class Object {
private:
              void traverse(Object* Ob,Object Obviously);// function for search the
                                                         // objects
public:
              Object() {list=NULL;} // class of object
                 int count;     //count number of objects or list
              int type;       //datatype could be char or integer or list of list
                                  // list of char , list of integer
               char atom;   //object as character
               int atomi;    //object as integer
               Object* list; // list of objects
              //void operator=(Object Obviously);
              void clear(); // clear the memory
              ~Object() {
                     if(list!=NULL) clear();
                     }
            }; //end of Object class
shared_t Object datamem[MAX]; //global memory of objects
shared_t instruction insmem[MAX]; // global memory of instruction
class package {
public:
   long Jid; // job id
   void operator=(package pk);
};
```

```
class reqpack:public package{  //class of request package
public:
    Object s;                  // objects
    int pc;                    // program count
    int pend;                  // program end
    int pbeg;                  // program begin
    int cond;                  // check flag for condition od parallel
    int mylocation;            // program job location
    long pid;                  // process id
    void operator=(reqpack pk);  // function to be operated
};
 class joinp: public package{      // class of join package
  public:
      Object result;           // result of computation
      int location;            // location of result in mark package
      void operator=(joinp pk);
     };
class markpack: public package{ // class of mark package
public:
     Object result;            // result of one job
     long ex_jid;              // provirus job id
     int numresult;            // number of results to be expected
     int progcount;            // program count
     int prevbeg;              // previous job begin
     int prend;                //previous  job end
     void operator=(markpack pk);
  };

class nodes{
public:
    reqpack req;               // define request package
    markpack mark1;            // define mark package
    joinp join1;               //define join package
    int flag;                  // control flag
    int sendnodeid;            // send node id
    nodes() { flag=-1; sendnodeid=-1; // class of node
         req.s.list= new Object[2];     // create the new request package
         mark1.result.list= new Object[2]; //create a new mark package
         join1.result.list=new Object[2]; // create a new join package
         }
~nodes()                            // clear the memory
    };
shared_t nodes  sharedvalue[36][4]; // define two dimensional shared value
shared_t Object localdata[10];      // define local data memory
shared_t int mid[10];               // define nodes id
shared_t int loc[10];               //define shared location
#endif
```

Host.h class:

```
//* This part of program is related to host.c, which is class of host .c
//* in host.h contains .h liberay and simulat5.h , it defines isdigit,
//* isalpha , isascii and shed pointer shmalloc; Host class contains following: //
// intialization of instruction counter number host, processor id and top S-node id;
// req1 for C-node.c;
// newlist for C-node.c, and functions in host.c which are load_inst; show_inst;
 // receicv_join   into_men; LOAD_DATA; show_data;

class host{                    // class of host node
 public:
   int instruct_no,count;    // instruction count number
   int id,parentid;             // processor id and its source processor id
   host(int i,int j){          // int i= processor id, int j=parent or child id
    id=i; parentid=j;instruct_no=0;count=0;old=0;
                                          }
   void load_inst();            //load instruction set function
   void show_inst();           //display the instruction set loaded
   void recv_join(int );       //reveive join package function
   void into_mem(char*, int, int, int );
   void load_data(Object*,int*);  // load data objects to data memory;
   int old;
   void show_data(Object);       //display data objects loaded already;
   void data2mem(char* ) ;
   reqpack* req1;               // used for host.c request package
   Object* newlist(void);
     };
extern shared_t int fg;    // fg used to define
```

C-node.h class:

```
 // This part program uses class to declare all the structure of c-node,
//  and all the package <request, join, marked> transfer between
//  all the s-node and c-node. Also there are 24 the primitive function
//  is used in the c-node.
class C-node
{
 public:                //variables used in C-node.c are defined in public
  int location; // define node location
  int value,my_flag; //define value and flag
  int id,parentid,ffg, g; //define node id, parent node id and control flag
  void get_object(); // function to be used to get objects
  void computation(int);          //for calculate different primitive functions
  shared_t static markpack mark1[15];//mark package is used in C-node5.c
  joinp* join1;                 //join package is used i C-node5.c
  shared_t static reqpack req1[10]; //request package is used to C-node5.c
  shared_t static Object temp;      //object temp variable is used to C-node.c
  shared_t static reqpack tempreq[15];//temprequest is used in C-node5.c
  C-node(int i, int j){           //initial integers, objects location functions used in C-
node
        ffg=0;
           g=0;
        id=i;
        value=0;
        my_flag=0;
        parentid=j;
        location=0;
        join1=new joinp;
        join1->result.list=new Object[2];
        temp.list=(Object*) shmalloc(sizeof(Object)*2);

        for(int rr=0;rr<MAX;rr++)a  //initial the list of objects
        temp.list[rr].list=(Object*) shmalloc(sizeof(Object)*2);
        for(rr=0;rr<15;rr++)           initial the list of results
        mark1[rr].result.list=(Object*) shmalloc(sizeof(Object)*2);
        }

void EQ(int);                  //EQ is used to calculate primitive function 'eq';
void NUL(int);                 //NUL is used to compute primitive function'nul'
void REVERSE(int);           //REVERSE is used to computer reverse function;
void LENGTH(int);            //LENGTH is used to computer length of objects
void ADDITION(int);  //ADDITION is used to computer adding two objects
void SUBTRACT(int);  //SUBTRACT is used to subtract two objects
void MULTIP(int);           //MULTIP is used to computer mutiple two objects
void DIV(int);                  //DIV is used to computer divide two objects
void TRANS(int);             //TRANS is used tranform two objects
void AND(int);                //AND is used to define T or F condition
void OR(int);                   //OR is used to define T or F condition
void NOT(int);                 //NOT is used to define T or F condition
void ATOM(int);             //ATOM is used to define object is atom or not
void excutefp();              //excutefp is used to execute primitive functions
void Ars();                       //ARS is used to select left object of the list
void Als();                       //ALS is used to select right object of the list
void Apf();                      //apf is used to decide weather use primitive function or not
void Cond();                   //Cond is used to define two situations
void APNDL(int i);        //APNDL is used to append a object to left of list
void APNDR(int i);        //APNDR is used to append an object to right of list
void ROTL(int i);         //ROTL is used to rotate the left object of list
void ROTR(int i);         //ROTR is used to rotate the right object of list
void DISTL(int i);       //DISTL is used to distribute left object of list
void DISTR(int i);       //DISTR is used to distribute right object of list
void MID(int i);        // MID is used to copy the object of list
void HD(int i);        //HD is used to chose the first object of the list
void TL(int i);        //TL is used to chose the last object of list
void TLR(int i);       //TLR is used to chose right last object of the list
void send_join();      // send_join is used to send join package up
void send_mark(int);   // send_mark is used to send mark package up
void send_req(int );   //send_req is used to send request package up
void Acs();            // Acs is used to select constant as variable
void gettop();         //gettop is used to get request package from top node
int ctrfla;            //ctrfla is used to decide
};
extern shared_t int fg;  // shared value flag
```

S-node.h class:

```
struct  rlist {                  // structure of request package link list
        reqpack  req;            // define request package
        rlist*   next ;          // next of request package
};
struct mlist {                   // structure of mark package link list
        markpack  mk;            // define mark package
        mlist*  next ;           // next of list
};
class list {             // class of list
      rlist*  r;         // link list of request package
      mlist*  m;                 // link list of mark package
 public:
    shared_t static mlist* temp;  //define temporary mark package
    list() { r=NULL; m=NULL;  // initial request  package and mark package
        mlist* temp=(mlist*) shmalloc(sizeof(mlist)); // add to front of list
        }
    ~list() { if(m!=NULL) delete m; if(r!=NULL) delete r; } // clear the memory

    void radd(reqpack tm) { //add new req pack in the list
//add new req pack in the list
// function is to add a new request package
// to the link list, if the link list is empty
// put the request package in the head of list
        rlist* rtemp=new rlist;
        rtemp->req=tm;
        if (r==NULL) {rtemp->next=NULL; r=rtemp;}
        else{
        rtemp->next=r;
        r=rtemp;
        }
}

    int rdel(long pid) {
```

```
// function is to search pid = jid in request
// package in the link list, and delete it
// from the link list
                rlist* prev=r;
                rlist* rtemp=r;
                while((rtemp!=NULL)&&(rtemp->req.Jid!=pid)) {
                        prev=rtemp;
                        rtemp=rtemp->next;
                        }

                if(rtemp==NULL) return -1;
                if (prev==rtemp) {r=NULL;}
                else prev->next=rtemp->next;
                delete rtemp;
                return(0);
                }
   reqpack* rfind(long pid) {
// function is to search pid = jid in request
// package in the link list, if found, return
// the request package pointer
                rlist* rtemp=r;
                while((rtemp!=NULL)&&(rtemp->req.Jid!=pid))
                        rtemp=rtemp->next;
                return(&(rtemp->req));
                }

   void rpr_list()
// this function prints out jid of request
// package of link list
   {
   rlist* rtemp = r;
   while(rtemp!=NULL){
       cout<< rtemp->req.Jid;
       rtemp=rtemp->next;
       }
   cout<<"\n";
   }
  void madd(markpack mk);  //
   int mdel(long p)
  markpack* mfind(long pid);
  void mpr_list();
};

class S-node {   //class of S-node
                //initial all integers functions variable used to S-node
public:
     list mylist;              //define link list
```

```
    joinp    join;              // define join package
    reqpack rq;                 // define request package
    int      stat_node[5];      //define 5 location of C-node
    int      restart_p;         // for restart flag
    int      sub_thread;
    int      sz;                // size of C-node
    Object   ob_p;              //define objects list
    int      id;                // define node id
    long my_flag;               // job flag
    int      parentid;          // parent id
    shared_t static markpack snew_mark[4];       // define mark package
      shared_t static markpack* tt;
    S-node(int i,int j);
    void put_result(int,joinp*);      //send result function
    void gettop();                    // get package from top node function
    int  check_node();                       // check status node function
    void recv_join(int fl);           // receive join package function
    void recv_req(int fl);            // receive request package function
    void recv_mark(int fl);           // receive mark package function
    void send_join(long);             //send join package function
    void send_req(long Pid,int fl);   //send request package function
    void send_mark(long Pid);         //send mark package function
};

class TS-node :public S-node{        //top S-node class
public:
  TS-node(int i,int p);                     // top S-node function
  };

extern shared_t int fg;              //shared variable for checking flag
```

APPENDIX C

EXAMPLE PROGRAMS

FP: [tl, hd]. o*,

TR-machine program:

```
                    0 B 0 0 ALPHA * -1 -1
                    0 0 E C BEG 1 1 4
                    1 4 U I APF hd -1 -1
                    1 4 U I APF tl -1 -1
                    0 0 N 0 END -1 -1 -1 -1
input data:          < < 15 15 > < 14 14 > > -1
output:              <<196,>,225>
time clock:          2
```

(Factorial):
FP: fact = eq.[sl2,0] _> sl1;
[*.[sl1, fact], -. [sl2, 1]]

TR-machine program:

```
0 0 B C BEG 2 0 7
0 7 B C BEG 1 1 4
1 4 U I ACS 0 -1 -1
1 4 U I ALS 2 -1 -1
0 0 N 0 END -1 -1 -1
0 7 E I COND eq -1 -1
0 7 U I APF id -1 -1
0 0 N 0 END -1 -1 -1
0 0 S 0 CONDA 19 -1 -1
0 0 S 0 ALS 1 -1 -1
0 0 0 C BEG 2 10 17
1 0 17 B C BEG 1 11 14
11 14 U I ACS 1 -1 -1
11 14 U I ALS 2 -1 -1
0 0 N 0 END -1 -1 -1
10 17 E I APF - -1 -1
10 17 U I APF * -1 -1
0 0 N 0 END -1 -1 -1
0 0 S 0 JMP 0 -1 -1
0 0 S 0 ALS 1 -1 -1
0 0 E 0 ALS 1 -1 -1 -1
```

```
input:              < 1 9 > -1
output:             362880
time clock:         69


FP:  trans. reverse
TR-machine program:
                    0 0 B 0 APF reverse -1 -1
                    0 0 E 0 APF trans -1 -1 -1
input:              < < 5 6 > < 7 8 > >  -1
output:             < 7,5> <8, 6> >
time clock:         4


FP: or.[null, eq]
TR-machine program:
                    0 0 B C BEG 1 0 3
                    0 3 U I APF eq -1 -1
                    0 3 U I APF NUL -1 -1
                    0 0 N 0 END -1 -1 -1
                    0 0 E 0 APF or -1 -1 -1
input:              < 4 4 >  -1
output:             1
time clock:         2

FP: [tl, hd, o*]
TR-machine program
                    0 0 B C BEG 1 0 4
                    0 4 U I ALPHA * -1 -1
                    0 4 U I APF hd -1 -1
                    0 4 U I APF tl -1 -1
                    0 0 N 0 END -1 -1 -1 -1
input:              < < 15 15 > < 14 14 > > -1
output:             < < < 14, 14> < 15,15> > < 225, 196>>
time clock:         1


FP: *.  o+. trans
TR-machine program:
                    0 0 B 0 APF trans -1 -1
                    0 0 S 0 ALPHA + -1 -1
                    0 0 E 0 APF * -1 -1 -1
input:              < < 4 3 > < 5 6 > > -1
output:             81
time clock:         4

FP:[sl2, -.[1l, 1]]
```

VITA

Juan Duan

Candidate for the Degree of

Master of Science

Thesis: AN OBJECT-ORIENTED PARALLEL SIMULATION OF TR-MACHINE ARCHITECTURE

Major Field: Computer Science

Biographical:

Education: Received Master of Science Degree in Philosophy from Graduate School of Chinese Academy of social Science, Beijing, China 1987; Received Master of HR in Human Relations, University of Oklahoma, Dec,1990; Completed the requirements for the Master of Science degree at Oklahoma State University in July 1994.

Experience: Teaching Assistant, Computer Science Department, from 1993 to present. Data Analyst, Entomology Department, from 1992 to present Oklahoma State University; Teaching and research Assistant, Philosophy and Human Relations Department, 1987 to 1990 the University of Oklahoma.

Professional Membership: Member of honor society of PHI KAPPA PHI at OSU Member of ACM and IEEE-CS. Dean's list, the Graduate School of the Chinese Academy of Social Science. Member of the Chinese Association of Sociology, Social Psychology and Philosophy.