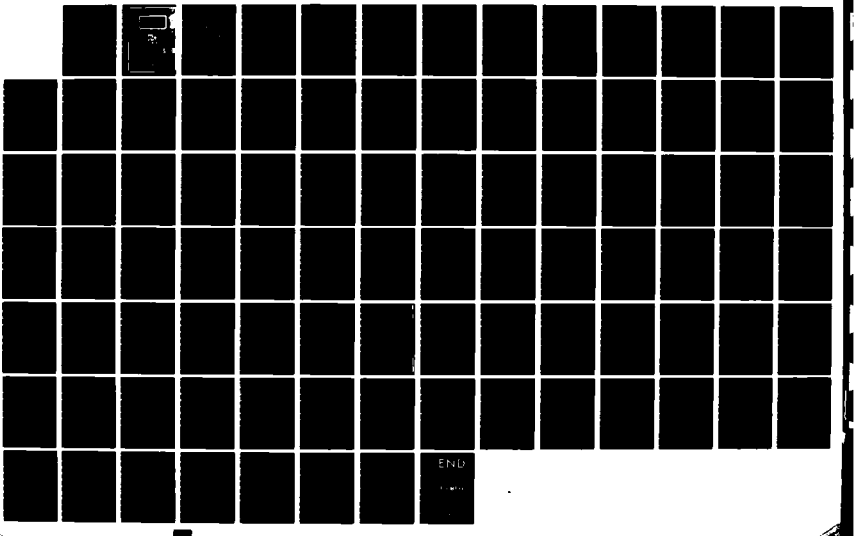
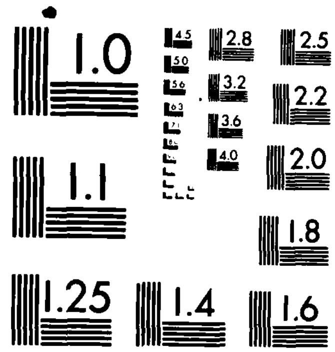
AD-A162 234 ASSOCIATIVE NETWORKS ON A MASSIVELY PARALLEL COMPUTER (U) DUKE UNIV DURHAM NC DEPT OF COMPUTER SCIENCE G JACKOWAY OCT 85 AFOSR-TR-85-1050 AFOSR-83-0205 1/1

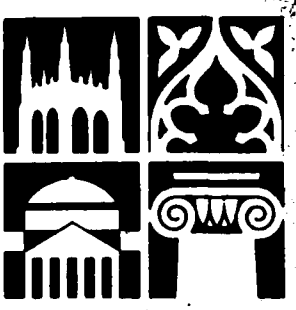
UNCLASSIFIED F/G 9/2 NL





ASSOCIATIVE NETWORKS ON A MASSIVELY PARALLEL COMPUTER

GARY JACKOWAY



DEPARTMENT
OF
COMPUTER SCIENCE

DUKE UNIVERSITY

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) UNCLASSIFIED

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-85-1050	2. GOVT ACCESSION NO.: AD-A162 234
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Associative Networks on a Massively Parallel Computer	5. TYPE OF REPORT & PERIOD COVERED: Technical paper
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Gary Jackoway	8. CONTRACT OR GRANT NUMBER(s): AFOSR-83-0205
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, Duke University, Durham, N.C. 27706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A7
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: October 1985
	13. NUMBER OF PAGES: 85
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research, Air Force System Command, Bolling AFB, Washington, D.C. 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Associative nets, semantic nets, parallel computation, knowledge structures

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A generalization of semantic networks, called an associative network (Findler [1979]), is mapped onto a massively parallel processor which is currently under development. The results show:

- The time required to process a query is dependent strictly on the pattern of the query, not on the size of the classes being processed. A system built using this knowledge representation will give consistent semantic processing performance.

(continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (continued)

- The order of processing a query does not affect the speed. Thus there is no need for heuristics and monitors to determine the most efficient way to process a query.

- Although we do not receive anywhere near an n-fold speedup by using n processors, we still receive significant performance benefits over a single processor.

- The associative network may be used not just as a semantic network, for example, it also allows some problems involving numerical minimizations to be solved efficiently.

The primary result of this work is that a large number of simple processors, each responsible for a small piece of information, can work in unison to answer queries significantly faster than a single, highly complex processor can.

This paper has the following structure. The first chapter introduces associative networks by first describing semantic networks and then formalizing associative networks. The second chapter describes the parallel processor upon which we will implement associative networks. Chapter three describes this implementation in detail. In chapter four, examples are given and expected timing results are compared to a sequential method. Chapter five considers numerical calculations. Finally, chapter six discusses limitations and possible extensions of this method.

2

ASSOCIATIVE NETWORKS ON A MASSIVELY PARALLEL COMPUTER

GARY JACKOWAY

Associative Networks on a Massively Parallel Computer

Gary Jackoway

Department of Computer Science

Date: _____

Approved:

_____
Alan W. Biermann, Supervisor

_____

_____

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Arts in the department of Computer Science in the Graduate School of Duke University

1984

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to express my thanks to Hewlett-Packard Company, whose fellowship program made this work possible.

I also thank Professors Biermann and Wagner for their many hours of discussions and their strong belief in striving for excellence. Professors Biermann and Wagner were supported by the Air Force Office of Scientific Research, Air Force Systems Command under USAF grant 81-0221, thus giving them the time to help me.

Sherri Tomboulian has worked in tandem with me on these problems, and her endless enthusiasm has helped me keep going.

I would also like to thank my wife, Ingrid, for her support and for her help in humanizing my work.

Finally I would like to thank Carly, our dog who has laid patiently at my feet through long nights in front of the computer.

INTRODUCTION

Many natural language projects in the past fifteen years have used semantic networks as their underlying knowledge representation (Brachman [1979], Hendrix [1979]). In a separate realm, recent breakthroughs in very-large scale integration (VLSI) have lead to designs for machines with vast numbers of processors (Schwartz [1980], Wagner [1983]). In this paper we will marry these two technologies. A generalization of semantic networks, called an associative network (Findler [1979]), will be mapped onto a massively parallel processor which is currently under development. The results shall show:

- The time required to process a query is dependent strictly on the pattern of the query, not on the size of the classes being processed. A system built using this knowledge representation will give consistent semantic processing performance.
- The order of processing a query does not affect the speed. Thus there is no need for heuristics and monitors to determine the most efficient way to process a query.
- Although we do not receive anywhere near an n-fold speedup by using n processors, we still receive significant performance benefits over a single processor.
- The associative network may be used not just as a semantic network; for example, it also allows some problems involving numerical minimizations to be solved efficiently.

The primary result of this work is that a large number of simple processors, each responsible for a small piece of information, can work in unison to answer queries significantly faster than a single, highly complex processor can.

This paper has the following structure. The first chapter introduces associative networks by first describing semantic networks and then formalizing associative networks. The second chapter describes the parallel processor upon which we will implement associative networks. Chapter three describes this implementation in detail. In chapter four, examples are given and expected timing results are compared to a sequential method. Chapter five considers numerical calculations. Finally, chapter six discusses limitations and possible extensions of this method.

CHAPTER 1

Associative Networks

Associative networks are a generalization of semantic networks (Findler [1979]). Semantic networks have been a common method for knowledge representation in natural language systems (Brachman [1979], Hendrix [1979]). Associative networks may be used for virtually any application whose knowledge base may be represented in network form.

1.1. Semantic Networks and Natural Language

Figure 1.1 shows a part of a semantic network for a "family" data base. Nodes are labeled with the names of individuals, classes of individuals,[1] and attributes of individuals (such as "tan"). An arc is labeled with the *relationship* that holds between the nodes it connects. Most arcs are directed: "job" points from "John" to "engineer". (Note that, in the text, words in double quotes are labels of arcs and nodes.) But other arcs, such as "married" are undirected; one may represent an undirected arc by two arcs both labeled the same, one pointing in each direction. Queries are answered by *traversing* the network. "What is Jane's spouse's job?" may be answered by traversing the "married" arc from "Jane" to "John" and then the "job" arc

[1]There are epistemological problems with mixing the use of individuals and classes of individuals. Even in this simple data base we can envision problems if we try to represent the fact that there are "lots of dogs" by merely attaching such a fact to the dog node. Rather than getting involved in these problems, it is sufficient to remember that we are building an *underlying representation* upon which systems of various kinds may be built. Brachman [1979] gives a good summary of the issues surrounding the representation used in this paper. We will use it because it is intuitive, not because we believe it to be theoretically best.

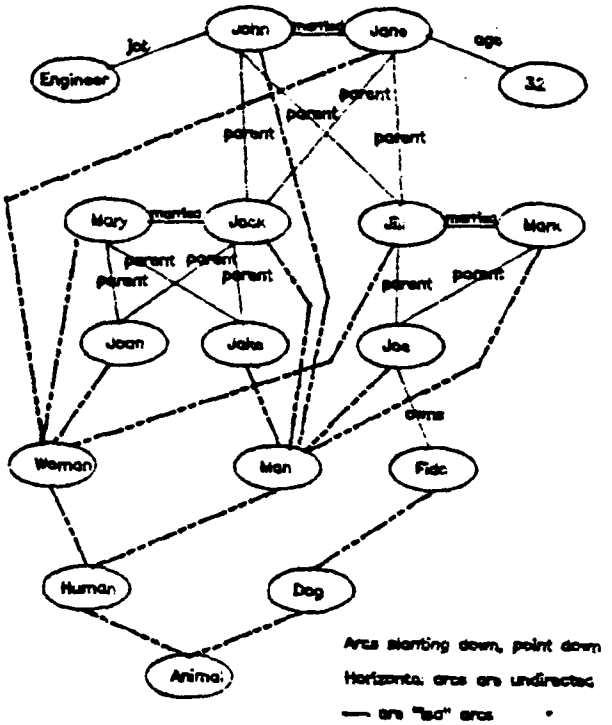


Figure 1.1 A small semantic network

to the answer, "engineer".

Computational complexity rises quickly as the queries become more involved and larger groups of nodes must be processed. Consider a semantic network for a company having ten thousand employees, several pieces of information stored concerning each employee, and a hierarchy of bosses leading from a dock worker to the president of the company. A query such as "which female employees have a boss whose boss is female" would require accessing tens of thousands of records and following at least as

many links.

The possibility of employing a parallel computer to improve the query processing performance appears fruitful since queries may be processed by working on sets of nodes at a time. Consider, for example, "Who are the grandchildren of John?" We can think of gathering together the nodes which are the children of John, giving each of them to a separate processor, and then having these processors simultaneously find the grandchildren. We will see in chapter three that our implementation actually goes a bit farther than this in building a parallel associative network.

1.2. Associative Networks

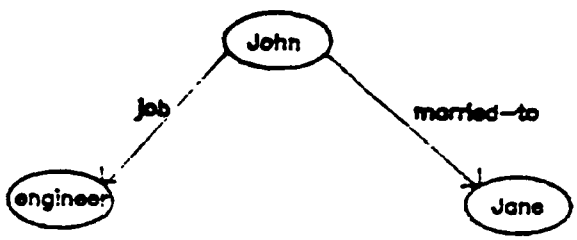
A semantic network is loosely defined as a network whose arcs and nodes are labeled with words. This dependence on words to give meaning to the network is both a positive and negative trait (Brachman [1979]). The definition we give of associative networks eliminates words and meanings altogether in favor of numbers. It is assumed to be a function of a higher level of the system (e. g. a natural language front end) to assign meanings to these numbers.

In the forward to his book, *Associative Networks: Representation and Use of Knowledge by Computers*, Findler [1979] describes the difference between semantic networks and associative networks as follows:

> The main theme of this book is *associative networks*. I have deliberately avoided the term *semantic networks*,... to indicate that the former are more general in objectives and, possibly, in structure than the latter. Semantic networks aim, I believe, at furnishing a representation for linguistic utterances to capture underlying relations of words and to produce the information contained in text... Indirectly, they also reveal the intricacies and use of language... Associative networks, on the other hand, could, but need not, be language-independent... Associative networks are constructed to serve

as the *knowledge base* of programs that exhibit some operational aspects of understanding

An associative network can be defined formally as a set of *nodes* and a set of *arcs*. A node is simply a positive integer. An arc has three parts: the from-node, the relation and the to-node. The *from-node* is the number of the node from which the arc emanates. The *to-node* is the number of the node to which the arc points. The *relation* is the object that labels the arc; for now we will assume that it is an integer. Given the following piece of a semantic network:



and the following dictionary:

name	number	node or arc
engineer	1	node
jane	2	node
job	3	arc
john	4	node
married-to	5	arc

the associative network is:

nodes: {1,2,4}

arcs: {(4,5,2),(4,3,1)}

Within this simple framework we can represent semantic networks as well as other types of information.

1.3. Associative Networks: A Numerical Example

As an example of an associative network which involves numerical processing, suppose one wishes to use a computer to find optimal trips between cities in the United States. *Optimal* might mean ***fastest*** or ***shortest***. An associative network may be used to build such a data base. Consider an associative network where the nodes are cities and the relation of an arc is a triple which has three parts: a highway number, a distance in miles, and a travel time in minutes. (American Automobile Association maps include estimated times for each major route.) This associative network will allow us to answer queries such as:

- What is the fastest route from Baltimore to Richmond?
- What is the shortest route from Raleigh to Washington DC.
- How long does it take to get from Boston to each city?
- Does the shortest route from Philadelphia to St. Louis use highway 40?

Section 5.4 discusses this data base in greater depth.

CHAPTER 2

The Boolean Vector Machine

2.1. Massive Parallelism

In years past, "parallel processing" referred to a few processors working simultaneously on a relatively short vector of numbers to speed up numerical calculations. Recent advances in VLSI technology allow us to consider building computers which consist of thousands or even millions of processors, and lead to new ways of thinking about computing (Mago [1980], Snyder [1982]).

To achieve this *massive* parallelism, one must use simple processors. One method employs a single controller responsible for decoding the machine instruction. Once decoded, the instruction is broadcast to all processors at the same time, and each processor executes this instruction on its local data. This machine organization is called Single Instruction Multiple Data (SIMD) (Flynn [1972]). The processors which make up such a system are generally called Processing Elements (PEs).

Having a million PEs each of which can only operate on its isolated data is a grave limitation in functionality. Instead an *interconnection network* needs to be designed which allows processors to share information. Many interconnection patterns have been suggested; A good overview is presented in Feng [1981] which describes mesh, shuffle-exchange, flip, and Benes as well as other interconnection alternatives.

With the combination of large numbers of processors and a powerful interconnection network, significant time improvements can be achieved (Schwartz [1980]).

2.2. Organization of the Boolean Vector Machine

The Boolean Vector Machine (BVM) is one such massively parallel computer (Wagner [1981], Wagner [1983]). The design of the BVM calls for one million PEs[1] interconnected using cube connected cycles (as shown below). The next section describes a typical PE and following that the interconnection network will be described.

2.2.1. A Typical Processor of the BVM

A processor in the BVM is very simple. Each PE has about 200 bits of local memory. The only calculation a PE may make is a boolean function of three variables. (Addition, for example, must be handled bit-wise.) Of the three input variables, two must be from a PE's local memory while the third may be from one of three PEs to which the PE is connected (see the next section).

[1] A prototype BVM with 2048 PEs is currently under construction.

2.2.2. The Interconnection Network of the BVM

The interconnection network used on the BVM is called *cube connected cycles* (Preparata [1981]). It is an outgrowth of the *binary n-cube*. In a binary n-cube, the PEs are numbered from 0 to N-1 for some $N = 2^n$. Each processor may access the n PEs which differ in exactly one of the n bit positions of the processor number. For example, with N=8, PE 0 has a lateral connection to PEs 1, 2 and 4; PE 5 is connected to PEs 4, 7 and 1. Unfortunately, a binary n-cube is too expensive to build. Each PE in a one million PE machine would need twenty interconnections. Instead the BVM uses *cube connected cycles* (CCC).

To understand the CCC interconnection network, consider each PE to be defined uniquely by a pair of integers <cycle-number, element-number>. Cycle-number ranges from 0 to 2^K-1, while element-number ranges from 0 to K-1, for some fixed K. The PE numbered <i,j> is said to be in the ith cycle and to be in position j within the cycle. There are $N = K \times 2^K$ processors in the machine. For a machine with one million processors, we have K=16.

The PEs can be seen as residing in an array where each cycle is a row. The interconnections within a row are cyclical: PE <i,j> is connected to a *successor* PE number <i,(j+1) mod K> and a *predecessor* PE number <i,(j-1) mod K>.

Unlike a mesh-connected system, where each PE would also have connections "north" and "south" (Barnes [1968]), the CCC interconnection uses a single connection to the *lateral* PE. The lateral to a given PE <i,j> is that PE in cycle position j of the cycle whose cycle number differs from i in the jth bit position. Examples: PE <0,0> is connected to PE <1,0>; PE <0,1> is connected to PE <2,1>; and PE <15,3> is connected to PE <7,3>. Figure

2.1 shows a 64 PE BVM with all of its interconnections. The advantage of this interconnection pattern is that every PE is no more than 2×K connections away from any other PE, instead of 2^K as it would be if only mesh connections were supported. Each PE has exactly three connections, no matter what size machine is being built. Note that the CCC has the same interconnection pattern between *cycles* as the binary n-cube has between *PEs*. This equivalence allows the BVM to emulate algorithms for the binary

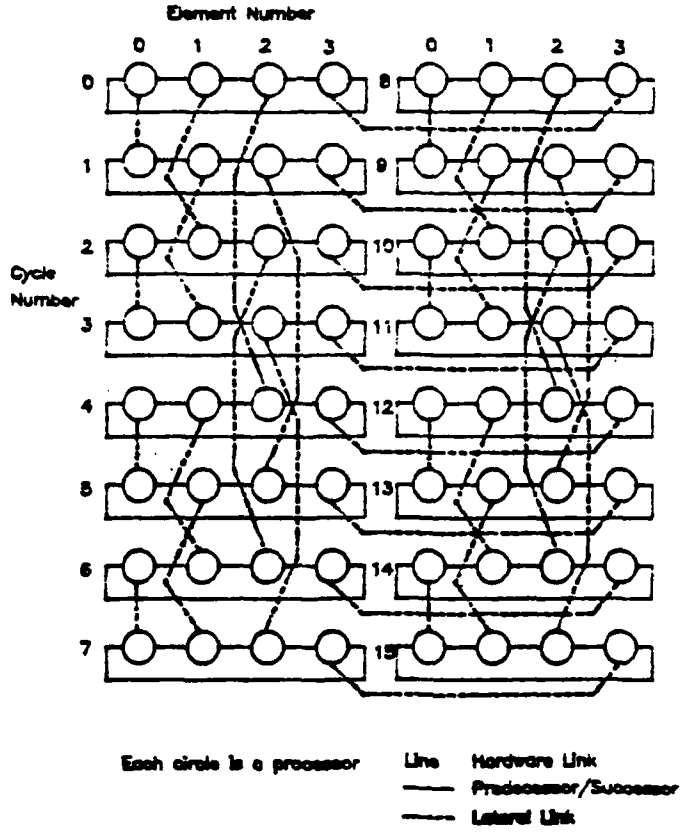


Figure 2.1 A BVM with N=64 (K=4)

n-cube efficiently (as shown in appendix A).

2.2.3. Programming Aspects of the BVM

For our purposes, one can think of programming the BVM as writing a program which is executed simultaneously, statement by statement, by each processor. (We will ignore input/output issues, granting that I/O is a serious question which deserves further study.) We will assume that all ancillary procedures are available for such standard operations as addition, multiplication, minimum and maximum. In our timing results, of course, we will take into account the bit-wise nature of the BVM. *IF..THEN..ELSE* constructs may be used, but, given the parallelism, both the THEN and ELSE clauses must be executed serially.

CHAPTER 3

Implementation of an Associative Network on the BVM

Fahlman [1978] gives an in-depth description of how a semantic network could use specialized massively parallel hardware to achieve dramatic performance improvements. Fahlman [1982] submits a hardware design for such a machine. This paper attempts a different approach. Instead of building a machine explicitly for processing networks, we will adapt the BVM, a general purpose, massively parallel computer.

3.1. Some Options

So far we have seen that associative networks provide a framework for problem-solving. We have described the BVM, a computer with one million processors. Now, how can we implement the former on the latter?

A first attempt might be to give to each processor one node of the associative network. This fails for several reasons. First, the number of arcs emitting from and pointing to a given node is not fixed. Given the limited size of a processor's local memory, there is no guarantee that all of the information about a node will fit in a single processor. Second, how does one traverse the network? Since each processor has only three links, one cannot use hardware links for arcs.

Another possibility is to give a single arc to each processor. This solves the previous problem of limited space. Each processor maintains its from-

node and to-node numbers as well as its relation. Once again, however, it is not at all clear how queries may be processed. Again consider the query "Who are John's grandchildren?" To start, it is easy to find "John's children". Those processors with "John" as the from-node and parent as the relation have a child as the to-node. At this point, however, one would have to sequentially step through each child to find their children. This defeats the purpose of introducing parallelism. (Tomboulian [1984] shows a possible way around this problem.)

3.2. The Implementation Choice

Our implementation of an associative network on the BVM splits information down to the lowest level. Instead of having one processor per node or one processor per arc, we will have one processor *per from-node* and to-node in each arc. Thus, a processor only maintains its node number and relation. The two processors which make up an arc are connected in hardware by their lateral link (described in section 2.2.3). Thus "job of John is engineer" consists of two processors: one is a "John" processor with relation "job"; the other is an "engineer" processor with relation "inverse job". (We think of a processor as belonging to the node it contains information about; thus we say a "John" processor, or a processor with *type* "John".) There will be many "John" processors and many "Jane" processors, etc. Further, they will be scattered throughout the machine due to the lateral connection requirement.

Let us look again at the query "Who are John's grandchildren?" Using our new layout, it is easy to turn on a child processor in each of John's

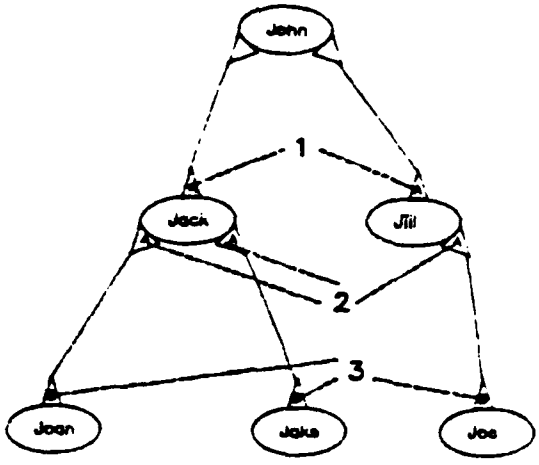
children. (We use *turn on* a processor to mean *set a bit to true* in the processor.) If all of the children processors were on, it would be simple to find the grandchildren. The missing step, then, is an algorithm which turns on all the processors of a particular type if any of that type are already on. We call this algorithm *share* since it shares information among processors of the same type.

3.3. The Share Algorithm

Share uses the CCC interconnection scheme of the BVM to simultaneously bring together a piece of information from all processors of each type. Figure 3.1 shows snapshots of a small associative network as it answers "Who are John's grandchildren?" The sequence of steps needed to find John's grandchildren are listed. Note that the triangles in the figure represent each of the processors and only the bits of information used in the calculation are listed.

If we wish to handle "What is owned by John's grandchildren?" we use what we have done, appending another share and another "turn on" step which we will call a *match* step. These match-share pairs allow a wide range of queries to be processed (as shown in chapter 4). In section 4.1.2 we will formalize the match statements.

The share algorithm works by simultaneously bringing information together from all processors of each type. In step two of the figure above, one "Jack" and one "Jill" processor have bit A on; no other processors have bit A on. When we use algorithm share on bit A, the system simultaneously determines that all "Jack" processors and all "Jill" processors are to have bit



(1) Turn on bit A in processors with relation "inverse parent-of" and whose lateral is of type "John".

(2) Apply the share algorithm to bit A.

(3) Turn on bit B in processors with relation "inverse parent-of" and whose lateral has bit A on.

Figure 3.1: Calculating "John's grandchildren" (the triangles are processors)

A on, while "John", "Joan", "Jake" and "Joe" processors are to have bit A off. The algorithm works in two steps. First we *concentrate* information about each node. In this case, we OR together all bit A values for each node as they are concentrated. The second step returns the concentrated value to each PE of each type. Since the problem above involves sending a single bit of information, operation OR is the natural choice. Share may also be used on integers, in which case one must determine what operation to perform. The most useful operations are ADD, MIN and MAX.

Figure 3.2 illustrates the steps necessary to calculate share with operation ADD. In the initial position, each of the processors assigned to a node have initial values. In step one, these values are added, the result being placed in the center of the node to indicate that, at this point, there is exactly one value per node. Step two copies this value into each processor assigned to that node. The share algorithm is developed in full along with proofs concerning its optimality in appendix A. The next section describes expected run times for share and other operations.

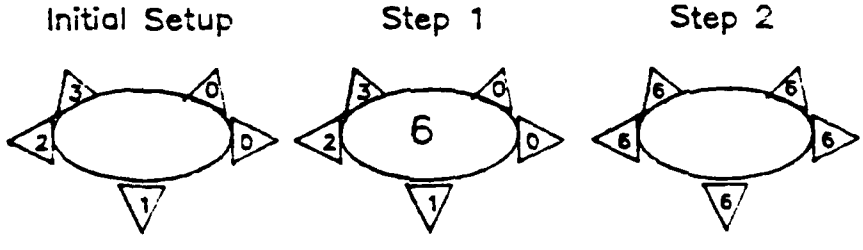


Figure 3.2 The steps in share with operation ADD

3.4. Timing Results for Selected Operations

As proved in appendix A, the share algorithm takes $24\log_2(N)$ BVM operations per bit in the object to be shared. Since the BVM has a cycle time of 250ns (Wagner [1984]), and $\log_2(N)$ is 20 for the one million PE machine, the share algorithm will take 120μs, per bit.

Since the BVM is a bit-oriented machine, the size of integers affects the results we will achieve. Throughout we will use 20 bit integers. This unusual choice is made for several reasons. First, 20 bits is sufficient to range to one million; the nodes may be represented uniquely in 20 bits for otherwise the network would not fit in the machine. Second, 20 bits is a natural number for this machine since it equals $\log_2(N)$, which appears in many standard timing results. Thus if an algorithm takes $\log_2^3(N)$ steps per bit, it will take $\log_2^4(N)$ steps per word. Finally, on a machine with a 250ns cycle time, we will get results which are multiples of 5μs per word, which are particularly easy to manipulate.

The operations to be performed on words are numerical comparisons (less than, equal, greater than), minimum and maximum, and addition. Each of these operations will take 5μs if only one address calculation is necessary. This is the case if the input and output variables are at the same position in the processor, but the values may be in a processor connected by a hardware link. Thus doubling a number in place may be accomplished in 5μs by adding it to itself (x←x+x). Also two numbers may be added in 5μs if they are in the same place in connected processors (x←x+connected x); this is the case in the share algorithm. Other forms of numerical operations will take 10 and 15μs for two and three address computations respectively.

CHAPTER 4

Implementing Data Bases

In this chapter we will describe two data bases and formally define how queries may be processed. A simple language will be developed first which allows us to "program" queries for the BVM.

4.1. A Query Programming Language

To control the flow of information through the network requires some formal specification of what the network is to do at each step. This section describes a language that allows us to describe how a query may be resolved. The language below is procedural in nature: the system executes a series of statements, each of which performs some manipulation upon the network as a whole. It should be remembered that one would generally build a natural language processing system, such as Thompson's POL system (Thompson [1981]), whose input would be user queries and whose output would be a series of statements in this language.

4.1.1. The BVM, as the Language Sees it

From the language's vantage, the system consists of a set of PE's, and each PE contains: a set of *mark bits* which are boolean toggles and can be set or read; a set of *storage words* which may be set and manipulated using arithmetic; a *type field*, which is an integer whose value defines the node in

the associative network to which this processor belongs; and finally a *relation field*, which is an integer whose value defines the relation arc in the network to which this processor is connected.

A processor (PE) may access its own local information and the information stored in the PE on the other end of its relation arc, the *associated* processor. (On the BVM the associated PE is connected to this PE by its lateral link.) If a relation arc is directed, the sign of the relation field determines the direction of the arc: positive, away from this node; negative, toward this node. Consider the following example data base which contains the two pieces of information "John loves Mary" and "The job of John is engineer" with the following numerical assignments:

name	number	node or arc
engineer	1	node
job	2	arc
john	3	node
loves	4	arc
mary	5	node

and the following processor assignments:

information	from-PE → to-PE
John loves Mary	A → B
Job of John is engineer	C → D

The information in each processor would be as follows:

PE	Type field	Relation Field
A	3	4
B	5	-4
C	3	2
D	1	-2

4.1.2. The Language Definition

Returning to our language, we shall assume a standard pseudo-language similar to Pascal, limit the set of variables, and add a few specialized procedures. Then we will be able to write "programs" which are executed on every PE within the system.

The variables we allow are: M0-M9 for ten mark bits; S0-S4 for five storage words; TYP for the type field; and REL for the relation field. (The choice of ten mark bits and five storage words is arbitrary.) Further, any of these variables may be preceded by the letter "A" to refer to information in the associated PE. Now, the language will allow assignments to the (local) variables using the ← symbol. Mark bits are assigned boolean expressions, which may include integer comparators such as "less than" (<), as well as the boolean operators such as "and" (&). Storage words are assigned integer expressions involving operators such as "plus" (+) and "min" (MIN). The storage word assignment expressions may be preceded by an IF clause specifying a mark bit such that only those storage words are assigned if the appropriate mark bit is on. (A WHILE construct is also available, but requires a specialized function as its predicate. See the next section.) The following is a list of sample statements:

```
M0 ← M1 & (AM5 ∨ AM2);
M4 ← (NOT AM1) ∨ (S0 < AS2);
S4 ← 0;
S1 ← AS3;
S3 ← S1 + (AS2 − 3);
IF M2 THEN S2 ← S3;
```

4.1.3. Specialized Procedures and Functions

The additions necessary to use this language for querying are implemented as built-in procedures and functions. The share algorithm is executed by a SHARE procedure which takes as a parameter the bit or word upon which the algorithm is to be executed. If SHARE is executed on a word, the operation to be applied to the values as they are brought together must be specified. The following are sample uses of the SHARE procedure.

```
SHARE(M2);
SHARE(S0,ADD);
SHARE(S2,MIN);
```

An important function is ANY. Passed a bit in the BVM, ANY sets the mark bit which is its parameter to true if any processor has this bit on, and false if none does. ANY can also be used as a function which returns the true or false value; this value may be used with the control structure IF or WHILE. We will see that this function is critical when traversing hierarchies.

A PRINT statement is provide as well. This procedure executes the concentrate portion of SHARE, just to the point where the value for each node has been determined. Instead of then spreading the concentrated values among processors, PRINT displays the value each node has. When PRINT is called with a mark bit, the names of those nodes which have the bit on are printed out. (This is the extent to which we will handle I/O.)

Other functions will be introduced as they are needed.

4.1.4. Network-Specific Constants

An associative network is built of numbers, but for our purposes we care little what number is assigned to each node. We don't care whether "John" is mapped to 32 or 30172, as long as this value is unique to "John". To make our programs readable and flexible, we will assume that constants are defined for each node and arc label in the data base. The constant will be the label preceded by a pound sign (#). Thus if we wish to set M0 to true in all processors with node "John" and relation "parent-of", we say:

M0 ← (TYP = #John) & (REL = #parent-of);

Our language is now independent of the exact numerical assignments chosen.

4.2. The Family Data Base

The "family" data base describes a fictitious family and their relationships with other people, animals and objects. This simple data base will allow us to explore the breadth of queries which may be answered and to deal with some of the difficulties that arise.

Although the data base is small, a diagram of the whole network is unwieldy. Instead the data base is described using lists of relations. Figure 4.1 describes the entire data base. Each relation is named, followed by the list of pairs which have that relationship. From the line

job((john,engineer),(jane,doctor),...)

we discern that John's job is engineer, Jane's job is doctor, etc. Figure 1.1 displays a portion of the data base as a network.

```
isa((john,man),(jack,man),(mark,man),(joe,man),(jake,man),(pete,man))
isa((jane,woman),(mary,woman),(jill,woman),(joan,woman),(pat,woman))
isa((man,male),(woman,female),(man,human),(woman,human))
isa((human,animal))

job((john,engineer),(mary,doctor),(jack,welder),(mark,doctor))
boss((john,pete),(jack,pat))
{ salary is in thousands of dollars }
salary((john,30),(joseph,40),(manfred,100),(mary,45))
salary((pete,40),(pat,45))

parent((john,jack),(jane,jack),(john,jill),(jane,jill))
parent((mary,joan),(jack,joan),(mary,jake),(jack,jake))
parent((jill,joe),(mark,joe))
married((john,jane),(jack,mary),(jill,mark))

own((john,porshe),(mary,corvette),(mark,mercedes))
own((jake,fido),(jake,fluffy),(jill,poopsy))
isa((porshe,car),(corvette,car),(mercedes,car))
isa((fido,dog),(poopsy,dog),(fluffy,cat))
isa((dog,pet),(cat,pet))
isa((fido,male),(poopsy,female),(fluffy,female))
color((porshe,blue),(mercedes,blue),(corvette,tan))
color((poopsy,tan),(fido,tan),(fluffy,grey))
color((jake,white),(jill,tan),(mark,tan),(jack,white))
```

Figure 4.1 The family data base

What sort of questions can be answered using this data base? We may ask questions concerning familial relations, jobs and salaries, possessions and colors. The list of queries below will be used to demonstrate some of the features of the system.

- What is John's job?
- Who are John's children?
- Who is the son of John?
- Who are the parents of a tan dog owner?
- What cars are tan?
- Who are Joe's ancestors?

4.2.1. What is John's job?

To answer this simple query requires marking those names which are the values of relation "job" applied to "John". In network terminology, we need to return those nodes to which "John" has a job arc. In our language we can achieve this as follows:

```
M0 ← (REL = -#job) & (ATYP = #john);
Print(M0);
```

Note the use of -#job, since the result is the node *toward which* the relation arc points.

4.2.2. Who are John's children?

This query is processed analogously to the previous one except that we expect several responses.

```
M0 ← (REL = -#parent) & (ATYP = #john);
Print(M0);
```

4.2.3. Who is the son of John?

Since "son" means "male child", we will first find all children of John, and then ferret out those who are not male.

```
M0 ← (REL = -#parent) & (ATYP = #john);
Share(M0);
{We now have all processors for children of John turned on.}
M1 ← M0 & (REL = #isa) & (ATYP = #male);
Print(M1);
```

4.2.4. Who are the parents of a tan dog owner?

Note that this query is ambiguous. In the program listing below, "(tan dog) owner" is on the left "tan (dog owner)" is on the right. Code common to both meanings is on the left. It would be up to the natural language front end to determine which query is meant, or to run both and respond with both results.

```
{First set M0 to dogs}
M0 ← (REL = #isa) & (ATYP = #dog);
Share(M0);
```

{Find tan dogs}
M1 ← M0 & (REL = #color)
& (ATYP = #tan);
Share(M1);
{Find owners of tan dogs}
M2 ← AM1 & (REL =#owns);
 	
Share(M2);
	{Find dog owners}
	M1 ← AM0 & (REL = #owns);
	
	Share(M1);
	{Find dog owners who are tan}
	M2 ← M1 & (REL = #color)
	& (ATYP = #tan);
	Share(M2);

```
{Find parents of nodes with M2}
M3 ← AM2 & (REL = #parent);
Print(M2);
```

4.2.5. What cars are tan?

Although humans automatically know that "tan" is a color and thus only color arcs are of interest in solving this query, no such generalization is made by our query system. Instead, a search will be performed for any cars which have a connection to "tan". The parallel nature of the search is completely exploited by our machinery. It takes no longer to respond to "What cars are tan?" than to the less elliptical "What cars have color tan?" The following blind search solves "What cars are tan?":

4.2.4. Who are the parents of a tan dog owner?

Note that this query is ambiguous. In the program listing below, "(tan dog) owner" is on the left "tan (dog owner)" is on the right. Code common to both meanings is on the left. It would be up to the natural language front end to determine which query is meant, or to run both and respond with both results.

```
{First set M0 to dogs}
M0 ← (REL = #isa) & (ATYP = #dog);
Share(M0);
```

{Find tan dogs}
M1 ← M0 & (REL = #color)
& (ATYP = #tan);
Share(M1);
{Find owners of tan dogs}
M2 ← AM1 & (REL =#owns);
 	
Share(M2);
	{Find dog owners}
	M1 ← AM0 & (REL = #owns);
	
	Share(M1);
	{Find dog owners who are tan}
	M2 ← M1 & (REL = #color)
	& (ATYP = #tan);
	Share(M2);

```
{Find parents of nodes with M2}
M3 ← AM2 & (REL = #parent);
Print(M2);
```

4.2.5. Which man is married to Jane?

This query is as simple to implement as the ones before it, but it serves to point out one of the powerful features of the parallel implementation. A sequential processor evaluating this query from left to right might search thousands of men, selecting only the one with a "married" arc to "Jane". A sequential processor evaluating from right to left, on the other hand, starts from Jane and checks through the elements in the (singleton) set of nodes with a "married" arc connection to Jane, finding which of the nodes has an "isa" arc to "man". The processing time will be dramatically affected by the order of processing. The parallel implementation, however, takes the exact

```
M0 ← (REL = #isa) & (ATYP = #car);
SHARE(M0);
M1 ← M0 & (ATYP = #tan);
PRINT(M1);
```

This search is blind in that no mention is made of what arc is found between "tan" and "car". In this case only relation "color" was possible, but consider the query "Which *New York salesmen* drive corvettes?" This elliptical phrase may refer to salesmen whose sales district includes New York, whose home is in New York or perhaps whose favorite city is New York. Thus a method is desired which returns not just the name but also the arc used to make the connection. The response might be:

```
New York (sales district) salesmen
   Joe
   Carl
New York (home) salesmen
   Pete
New York (favorite city) salesmen
   Ed
```

A mechanism of this form is available in Thompson's POL system (Thompson [1981]). For our system to reply in this form, the following program works:

```
M0 ← (REL = #job) & (ATYP = #salesman);
SHARE(M0);
S0 ← 0;
{ S0 will remain 0 in salesmen nodes not connected to New York nodes}
M1 ← M0 & (ATYP = #New-York);
IF M1 THEN S0 ← REL;
PRINT(S0);
```

To handle this situation in a more complete manner, sorting should be used. This extension is described in section 6.1.

same time in either case as demonstrated in the side-by-side comparison below:

```
{ left to right analysis }
{find men}
M0 ← (REL = #isa)
     & (ATYP = #man);
SHARE(M0);
{now find those married to Jane}
M1 ← M0 & (REL = #married)
        & (ATYP = #Jane);
PRINT(M1);
```

```
{ right to left analysis }
{find those married to Jane}
M0 ← (REL = #married) & (ATYP = #Jane);

SHARE(M0);
{now find those which are men}
M1 ← M0 & (REL = #isa)
        & (ATYP = #man);
PRINT(M1);
```

The parallel method is independent of processing order and the size of intermediate sets; this is one of the most significant features of the system and guarantees consistent performance.

4.2.6. What cars are tan?

Although humans automatically know that "tan" is a color and thus only color arcs are of interest in solving this query, no such generalization is made by our query system. Instead, a search will be performed for any cars which have a connection to "tan". The parallel nature of the search is completely exploited by our machinery. It takes no longer to respond to "What cars are tan?" than to the less elliptical "What cars have color tan?" The following blind search solves "What cars are tan?":

```
M0 ← (REL = #isa) & (ATYP = #car);
SHARE(M0);
M1 ← M0 & (ATYP = #tan);
PRINT(M1);
```

This search is blind in that no mention is made of what arc is found between "tan" and "car". In this case only relation "color" was possible, but consider the query "Which *New York salesmen* drive corvettes?" This elliptical phrase may refer to salesmen whose sales district includes New York, whose home is

4.2.6. Who are Joe's ancestors?

Although we could proceed, much as we have so far, and build a program which will solve this query, it is far better to consider some related queries. For instance, consider "Who are the mercedes owner's ancestors?" How many times do we need to follow the parent arcs up the family tree? We don't know until we know who the mercedes owner is. But it is clear that no matter who that person is, we want to follow the parent arcs until there are no more. In many instances we need to process trees of information like "ancestors". We say that the "parent" arcs form a *hierarchy*. There is also an "isa" hierarchy which has four *levels*, "John" → "man" → "human" → "animal" for example. The next section will introduce a framework in which queries involving hierarchies may be handled gracefully.

There are many other queries of equal and higher complexity that may be asked of the family data base. In other sections of this paper we will revisit this data base and explore it further.

4.3. Animal Kingdom Data Base: A Hierarchy

There are over 700,000 species of animals (Newman [1967]). The animal kingdom is divided into seven phyla, and these phyla are divided into classes, classes into orders, and orders into families. Families are divided into genuses and finally genuses into species. At each level of this enormous hierarchy various features or *properties* distinguish each group. For instance, the animal kingdom is differentiated from the plant kingdom by

power of locomotion, nonphotosynthetic metabolism and other properties. Animals of the order perissodactyla (which includes horses) have an odd number of toes, while artiodactyla (which includes deer) have an even number (Lane [1984]).

If we wish to design a data base of zoological facts, we could list for each species all of its attributes. This is most inefficient, however, since we would be ignoring the power of the hierarchy and data would be repeated many times. Instead, let us attach information as high up the tree as possible. Since "animals move" we will say this once, at the top level, instead of attaching this piece of information to each species (or, worse yet, each animal). We can take this concept one step further and include generalizations which are "almost always" true. We might include "birds fly" in our data base. Then it will be necessary to *cancel* this information where it is incorrect ("penguins and ostriches don't fly").

Figure 4.2 shows a part of an associative network for this information. Our query processor will accept questions such as "What animals fly and lay eggs?" We need to build machinery so that parrots will be listed, penguins will not be, and perhaps flying fish will be.

4.3.1. Marking a Hierarchy

Let us start with a slightly simpler problem. How can we mark all of the species under a given node in the hierarchy? This method will handle queries such as "What species of animals are vertebrates?" Assume that the hierarchy is built using "isa" arcs up the tree: "carnivore 'isa' mammal." Also assume that relation "level" exists which determines what level in the

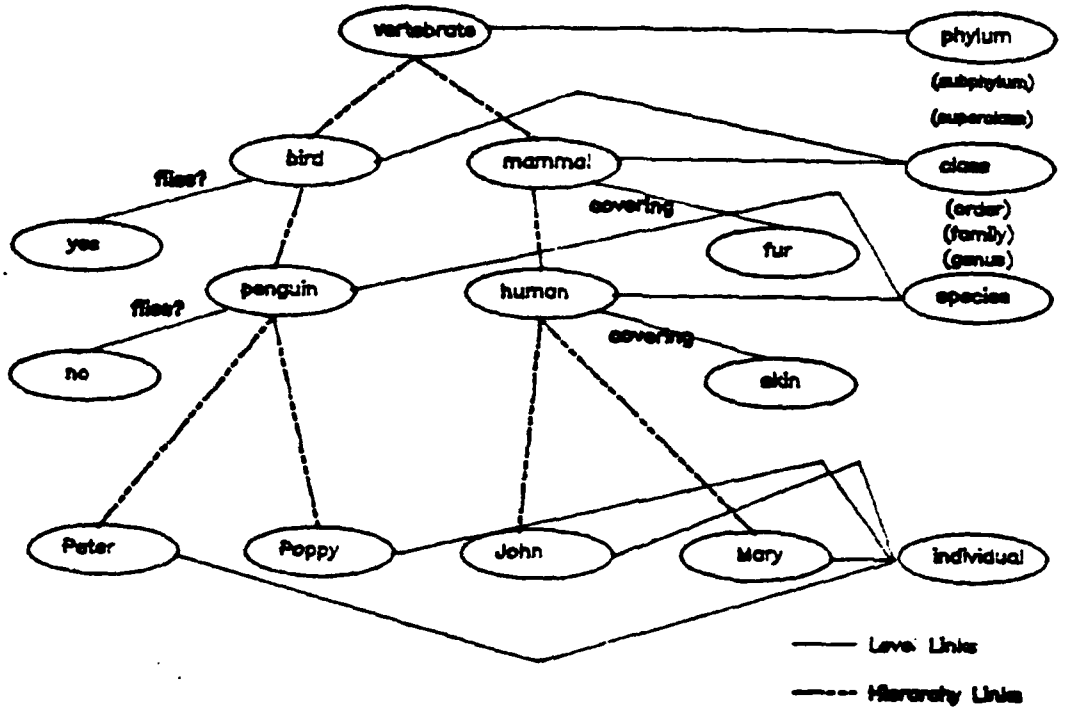


Figure 4.2 Part of the Animal Kingdom Associative Network

hierarchy this node is at: "the 'level' of carnivore is order." Given this information, we write the following program:

```
{M0 will be true in all nodes under vertebrate}
M0 ← TYP = #vertebrate;
{M1 will hold the frontier of the tree we are marking}
M1 ← M0;
WHILE ANY(M1) DO
    {M2 is true for all new nodes to be marked}
    M2 ← AM1 & (REL = #isa);
    SHARE(M2);
    M0 ← M0 ∨ M2;
    {These nodes become the new frontier of the tree}
    M1 ← M2;
END;
{Now set M3 to just those nodes in the subtree which are species}
M3 ← M0 & (REL = #level) & (ATYP = #species);
Print(M3);
```

Using the very same technique, we can now answer the query "ancestors of Joe" from the family data base (section 4.2.5). The changes necessary are: the starting node M0 should be set to TYP = #Joe; M2 should be assigned REL = #parent-of instead of REL = #isa; at the end we will set M3 if M0 and (TYP ≠ #Joe), since Joe is generally not considered to be an ancestor of himself.

We can formalize the marking of a tree by defining a procedure MARKALL. This procedure accepts two parameters, a mark bit and a relation number. On input the mark bit will be true for the starting values in the tree ("vertebrate" or "Joe"); on output the mark bit will be true in the entire subtree built of the relation number. The relation number thus tells which hierarchy we are trying to use and in which direction. The parent-of hierarchy may be used either "up" for ancestors or "down" for descendants. We may rewrite our program for "What species of animals are vertebrates" as:

```
M0 ← TYP = #vertebrate;
MARKALL(M0,#isa);
M1 ← M0 & (REL = #level) & (ATYP = #species);
Print(M1);
```

A program for "ancestors of Joe" is now:

```
M0 ← TYP = #Joe;
MARKALL(M0,#parent-of);
M1 ← M0 & (NOT (TYP = #Joe));
Print(M1);
```

4.3.2. Property Inheritance

As an extension of MARKALL, the INHERIT procedure passes a property down a tree as opposed to just marking a tree. INHERIT is called by:

INHERIT(WORD,RELNAME);

The code for INHERIT is slightly more complicated than MARKALL since it is important that we block information from moving down the tree if a value is already assigned to that subtree (Fahlman [1981]). For example, a relation "fly" might have values "Yes" and "No" and we want "birds fly yes" but "penguins fly no". If we simply used MARKALL to start from "birds fly yes", penguins would inherit the trait from birds. (In a sense we have a three-valued logic. Some nodes are marked "Yes", some "No" but most are marked "whatever I inherit from above.") Assuming parameter WORD is a word set to the starting values and RELNAME is some relation, we can implement INHERIT as follows:

```
{We will use MAX on our shares, though in general}
{there will be only one non-zero value}
SHARE(WORD,MAX);
{S0 holds the frontier of the tree}
S0 ← WORD;
WHILE (MAX(S0) > 0) DO
    {There is some value in the frontier}
    {M0 marks nodes to be added to the frontier}
    M0 ← (AS0 > 0) & (REL = RELNAME) & (WORD = 0);
    {If M0 is on, transfer the value of the connected node}
    S1 ← 0;
    IF M0 THEN S1 ← AS0;
    SHARE(S1,MAX);
    {M1 marks nodes which have obtained WORD values}
    IF M1 THEN WORD ← S1;
    S0 ← S1; {The new frontier}
END;
```

With MARKALL and INHERIT in place we can handle queries such as:

What animals fly and lay eggs?
Which vertebrates have tails?
Are there any birds that swim?
What is the order of penguins?
To what category do both ostriches and horses belong?

4.3.3. Timing Results for MARKALL and INHERIT

An instruction count for MARKALL and INHERIT is dependent on the number of levels in the hierarchy. MARKALL will take approximately 135μs per level of the hierarchy, most of that time being the cost of the SHARE statement. INHERIT will take approximately 2.5*ms* per level, most of that time for the expensive integer SHARE statement. If one knows that there are less than 256 different properties of this type (for example, colors) and also that these properties have been assigned numbers such that the final eight bits are different for each property, then INHERIT may be used on eight bit quantities and the execution time is only 1ms.

4.4. Timing Comparisons with a Sequential Method

In order to determine how successful the parallel implementation is, a sequential method needs to be developed for comparison. One feature of any method for comparison is that it should be possible to determine general timing results easily. The method described below has this feature and is also reasonably efficient.

An associative network is based on a set of arcs which describe all of the interrelations in the network. An arc has three parts: a from-node, a relation and a to-node. We can thus think of an associative network as an

arc table with three columns. To implement the queries efficiently, we want fast access to all of the arcs which have a particular value in a particular column. One method is to build a sorted binary tree for each column. A leaf in this tree is a linked list of pointers to those elements of the table with a given value in that column. Figure 4.3 shows an arc table and a sample tree necessary for access to the to-node column.

Let us assume that our sequential machine can execute a basic operation (numerical or boolean comparison, pointer dereference or assignment) in 250ns, the same cycle time as the BVM. Given information about the data base, we can determine the time to execute a query using

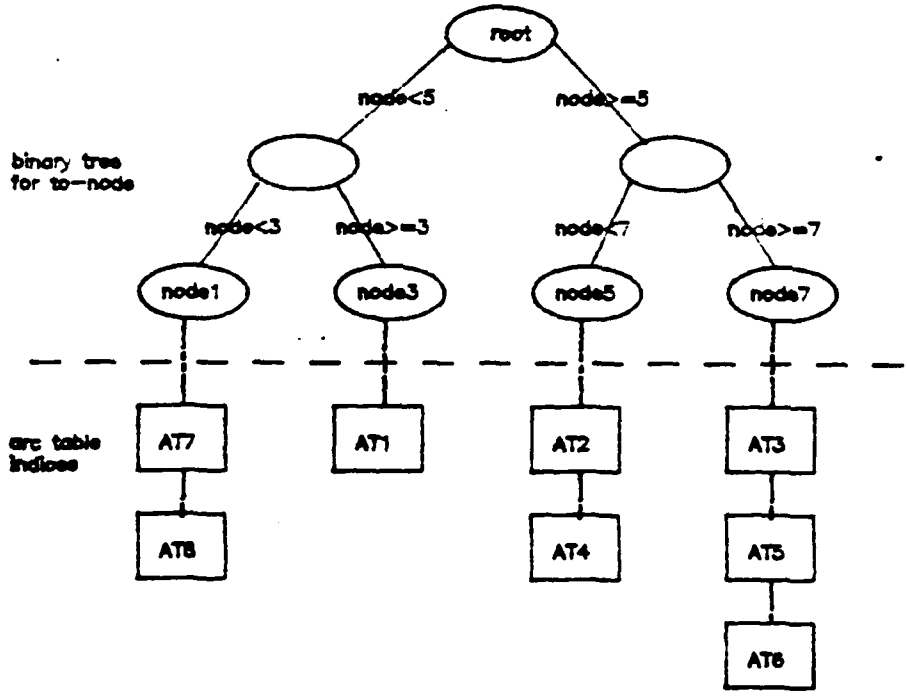


Figure 4.3 Data Structures for Sequential Method

- 100,000 animals in 2000 species
- 40% birds, 30% mammals, 30% others
- 500,000 arcs, 125,000 nodes

Now let us look at a sample query. Consider "Which birds have red wings?" For the sequential method, the final step will take most of the time, wherein 40,000 birds are input and a test will be made which requires three comparisons (relation = "wing-color", to-node = "red" or inherited color = "red"). Even though we have used the hierarchy, at the last stage all birds must be checked since there may be cancellations and exceptions (perhaps a bluebird flew into some red paint!). Using the formula above with n_i = 40,000, b = 4 (500,000 / 125,000), c = 3, and n_o = 1000 (an approximate), the timing result for this final step is 630ms.

The parallel implementation must inherit wing color to each bird. Assuming there are four levels of hierarchy between "bird" and each specific bird, the total time to execute the *INHERIT* statement, according to section 4.3.3, will be 10ms, a sixty-fold speed-up over the sequential method. If we use the method suggested in that section and assume no more than 256 colors, the run time will be only 4 ms, a 150-fold speed-up. Finally, if wing-color is only stored at the individual bird level, we can use MARKALL to turn on the birds and then only need a single integer command to find those birds with red wings. Timing for this method is 550μs, which is a thousand-fold speed-up over the sequential method.

CHAPTER 5

Numerical Data

In Thompson's POL natural language system, there are "attributes" and "number attributes" (Thompson [1982]). These correspond to arcs that point to nodes and arcs that point to numbers respectively. POL has distinct syntax and semantics for these two cases. There are a number of reasons to make distinctions between numeric and non-numeric data.

(1) Attributes may be chained, one after another, whereas number attributes may not. Compare "the boss of the boss of John" and "the salary of the salary of John."

(2) Number attributes may have comparators applied to them whereas attributes may not. An example is the query "Which employees of John have salaries *greater than* John's."

(3) Number attributes may have statistical and numerical functions applied to them whereas attributes may not. Consider "What is the *average* salary of female engineers."

(4) Attributes, when unqualified, refer to their range and can be used as such in a query. For instance "bosses" means "bosses of anyone" and the query "Which bosses are overweight?" is understandable. "Salaries" might be thought of in the same light (as a group of numbers, in this case), but this only leads to sensible queries when a statistical function is applied: "What is the largest salary?"

In our system, we will find that new methods are necessary for numerical calculations. In this chapter we will explore these methods.

5.1. Representing Number Attributes

In our stated method for implementing associative networks on the BVM, we would require a processor to be allocated to "$20,000" if there is a fact "John's salary is $20,000." The rationale for separating "John" and "$20,000" into separate PEs was so that the SHARE algorithm would work on each. It is inconceivable, however, that a SHARE is needed for "$20,000" or any other number. Instead we will allocate a single PE to this fact and add a VAL field in the processors to hold the value to which this arc points. Now a query like "Who has $10,000 as salary?" may be evaluated by:

```
M0 ← (REL = #salary) & (VAL = $10,000);
Print(M0);
```

Compare this to the non-numeric query of the same form, "Who has John as parent?" which is solved in section 4.2.2. In this query the value "John" is stored in ATYP, the type field in the associated processor. For numeric queries, we store this number directly in the processor. Thus, numeric facts require only a single processor.

5.2. Numerical Procedures

It was noted earlier that procedure SHARE could be executed on words using any of the operations MIN, MAX or ADD. These yield much of the necessary numerical power. MIN, MAX and ADD may also be used as procedures over the entire machine. In this case they are similar to ANY, but work on words. Thus MAX(S1) finds the maximum value of S1 in any PE and places that value in each PE's S1 word. Further, just like ANY, MAX can be called as a function which returns this value to be used in an IF or WHILE

statement or to be PRINTed. The timing result for MIN and MAX is 20μs. ADD takes 30μs due to the carry propagation. SHARE with MIN or MAX takes 2.4ms, with ADD 3.6ms.

A peculiar feature of the system is that MAX(S1) takes significantly less time than SHARE(S1,MAX), even though the number of maximums is the same. MAX is homogenous (all values may be MAXed in any order), while SHARE with MAX requires that numbers be MAXed with specific others. On parallel processors, homogenous processes tend to be faster.

5.3. Numerical Queries

In a previous example we saw a simple numerical comparison. Another query of importance is "Whose salary is greater than John's?" We can use MAX to replicate "John's salary" into all of the PEs. The following program implements the query.

```
M0 ← (REL = #salary) & (TYP = #John);
S0 ← 0;
IF M0 THEN S0 ← VAL;
{Exactly one processor has a non-zero value for S0}
MAX(S0);
{Now all S0's have that value}
M1 ← (REL = #salary) & (VAL > S0);
PRINT(M1);
```

MIN may be used to answer "Who has the smallest salary?" Note the care necessary in the following program to make sure only people with salaries are considered. There is assumed to be some value which stands for infinity (∞).

```
M0 ← (REL = #salary);
S0 ← ∞;
IF M0 THEN S0 ← VAL;
{S0 has non-infinite values only in REL = salary processors}
MIN(S0);
{Now S0 has the smallest salary}
M1 ← M0 & (VAL = S0);
PRINT(M1);
```

A variety of queries require that the system to be able to count. Consider the query "How many employees does each boss have?" SHARE with operation ADD will suffice. We will have 1's in only those processors which have a "boss" relation pointing toward them. (If "boss of John is Paul" holds, some "Paul" processor will have value 1.) Adding them will give us the correct count for each boss.

```
M0 ← (REL = -#boss);
S0 ← 0;
IF M0 THEN S0 ← 1;
SHARE(S0,ADD);
PRINT(S0);
```

To handle "Which boss has the most employees?" it is necessary to add the following lines:

```
S1 ← S0;
MAX(S1);
M1 ← (S0 = S1);
PRINT(M1);
```

A variety of more complex numerical queries can be imagined. Most of them involve the basic ingredients already described, with perhaps new procedures for other basic operations.

5.4. A Map Data Base

In this section we pursue the numerical data base from section 1.3. The primary arcs in the system are "highway" arcs which have three parts: a highway number, a distance in miles, and a travel time in minutes. Figure 5.1 shows the representation of part of a time-distance map as an associative network. Note that along with "highway" arcs, there are "state" arcs which give the state associated with each city. Other information such as population, rest stops and road condition could also be included.

This data base can handle queries such as "What cities are in Maryland?" and "What cities on highway 40 have more than 500,000 people?" but our primary interest in this data base is to calculate trip information. A typical query might be "How far is it from Baltimore to each city in Virginia?" Much as INHERIT works, we will solve this by starting with selected information and then emanating information from the starting points. The frontier of this growing tree will be those cities which have a new minimum value. Using SHARE with operation MIN, the routine will give to a city which is on more than one route its minimum value. We call this routine MINIMIZE for obvious reasons. To make the program for MINIMIZE easier to read, we will use variable names which make sense and then "declare" them to be words or bits. We will also take a few liberties with the language, recognizing this procedure as a sketch of the solution. A leading "A" is still used to specify the associated processor. The three parameters are: MINWORD, a word which is already set to the initial values and will be returned set to the final values; ENABLE, a precalculated bit which is on in all processors whose arcs the procedure is allowed to use; CALCWORD, the word containing the increment used when traversing arcs, will usually be set to either the

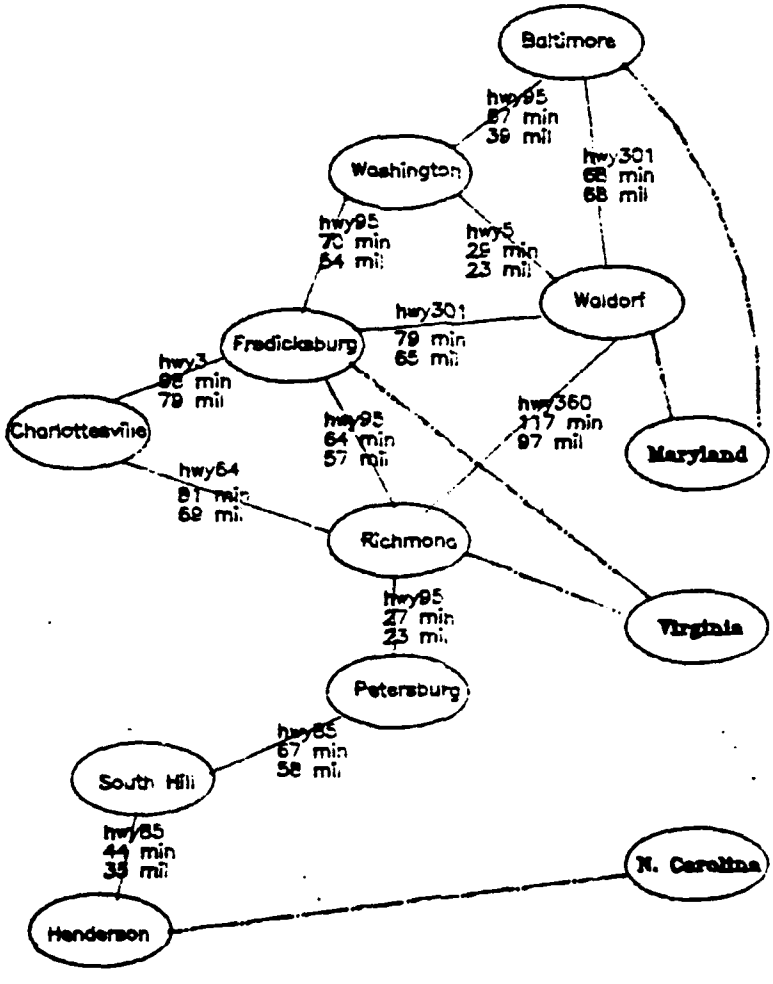


Figure 5.1 Representation of a time-distance map as an associative network

Distance word or the Time word.

```
MINIMIZE(Minword,Enable,Calcword);
WORD: Minword, Calcword, Currval, Newval;
BIT: New, Enable;
BEGIN
   Currval ← ∞;
   Newval ← ∞;
   IF (Minword ≠ ∞) THEN Newval ← Minword;
   New ← (Newval < Currval);
   WHILE (ANY(New)) DO
      IF New THEN Currval ← Newval;
      IF (New & AEnable) THEN Newval ← ACurrval + ACalcword;
      SHARE(Newval, MIN);
      New ← (Newval < Currval);
   END;
   Minword ← Currval;
END;
```

We can use this procedure to calculate the distance from Baltimore to each city in Virginia as follows:

```
M0 ← REL = #highway; { The enable bit }
{S0 will be set to zero in Baltimore and ∞ elsewhere}
S0 ← ∞;
M1 ← M0 & (TYP = #Baltimore);
IF M1 THEN S0 ← 0;
MINIMIZE(S0,M0,Distance);
{We now have the distance to each city, so we need to isolate Virginia}
M2 ← (REL = #state) & (ATYP = #Virginia);
S1 ← 0;
IF M2 THEN S1 ← S0;
Print(S0);
```

5.5. Timing Considerations for the Map Data Base

To estimate the run time of this problem, we will make the following assumptions. There are 5500 cities in the United States whose population is greater than five thousand (Lane [1984]), and we will assume that 4500 smaller cities which lie on junctions of highways will be added. These 10,000 cities will be assumed to lay in a 100×100 grid, each city with highway connections to the eight surrounding cities.

For the parallel analysis we need an estimate of the number of arcs in the longest minimum path. We will use 200, as this allows connecting diagonal cities by going straight horizontally and then straight vertically. The main loop of MINIMIZE will be executed 200 times and the cost of one execution is about 2.6ms (due primarily to the integer SHARE). Total execution time will be 0.52 seconds.

Recall from section 4.4 that we define A to be the total number of arcs, b to be the average branching factor, and N to be the number of nodes. In our case, A = 80,000, b = 8 and N = 10,000. One good sequential method for this problem is to build a list of nodes sorted by minimum distance and expand the minimum distance node that has not already been expanded. (This method is better than Dijkstra's algorithm (Aho [1974]) when $A \log_2(N) < N^2$, in this case 1 million < 100 million.) The current node is visited and its b arcs followed, altering the position in the sorted list of those nodes with new minimum values. The expected run time is:

$$N(t_g + b(t_p + (t_r + t_l) \times \log_2(N)))$$

where t_g is the time to get a new node's arcs, t_p is the time to process each arc, t_r is the reinsert cost for a node and t_l is the time to look for the new position for a node in the sorted list. We estimate the following values: t_g = 2, t_p = 5, t_p = 3 and t_r = 4. Under these assumptions, the sequential method will take 7.7 million instructions which is around 2 seconds of processing time.

The parallel method is only four times as fast as the sequential method. Why don't we receive better improvement? First, less than one sixth of the machine is being used. Second, integer operations are 20 times slower on the BVM. Third, the SHARE operation takes milliseconds per use. Finally,

the parallel machine gains the most when the branching factor is high, when graphs are "busby". In the given example the branching factor of eight significantly limited the available parallelism.

Consider the following more favorable problem. Suppose we have a completely connected graph of 700 nodes, thus 490,000 arcs. Making the same assumptions as above the results are: the parallel method runs in 75ms; the sequential method runs in 9 seconds. Now the speedup is 100-fold. (For this case, however, Dijkstra's algorithm will run in around 1 second but we still have an 12-fold increase.)

It is conceded that the constants used throughout this section are rough. Even though the BVM has an integer speed one twentieth that of the sequential machine to which it is compared, the BVM still obtains performance improvements over the sequential method. It is apparent, however, that the precise form and size of the problem causes significant changes in the improvement level.

graphs are "bushy". In the given example the branching factor of eight significantly limited the available parallelism.

Consider the following more favorable problem. Suppose we have a completely connected graph of 700 nodes, thus 490,000 arcs. Making the same assumptions as above the results are: the parallel method runs in 75ms; the sequential method runs in 9 seconds. Now the speedup is 100-fold.

For this problem, however, Dijkstra's algorithm will run in around 1 second but we still have an 12-fold increase. Alternative methods will also improve the parallel timing result. Consider, for example, the shortest path method used in Aho [1974]; a connection matrix is manipulated to determine shortest paths. A parallel version of this method does not rely on Share and should improve our results since it takes advantage of greater parallelism.

It is conceded that the constants used throughout this section are rough. Even though the BVM has an integer speed one twentieth that of the sequential machine to which it is compared, the BVM still obtains performance improvements over the sequential method. It is apparent, however, that the precise form of the problem and its size in comparison to the number of processors available to solve the problem cause significant changes in the level of improvement achieved by a parallel processor.

CHAPTER 6

Extensions And Limitations

In this chapter we will explore the boundaries of our system. We shall look at tasks which are very difficult using this framework. Our results will show that the method is general enough to handle most of the problems. We will also investigate extensions which will allow representation of more complex objects.

6.1. Tabular Output

The response to a variety of queries is a table of results. Consider the query "What is the name, age, height and sex of each engineer?" which might produce a table like the one below:

Engineers

name	age	height	sex
John	31	72	Male
Jill	28	64	Female
Peter	47	74	Male
Kathy	24	69	Female

Our system can implement this query by PRINTing one column of information at a time. The following program is a possibility.

```
M1 ← (REL = #job) & (ATYP = #engineer);
SHARE(M1);
PRINT(M1); {Prints the names of engineers}
M2 ← M1 & (REL = #age);
{ The age processor for each engineer has M2 on }
S0 ← 0;
IF M2 THEN S0 ← VAL;
```

```
PRINT(S0); {Prints the ages}
M2 ← M1 & (REL = #height);
{ The height processor for each engineer has M2 on }
S0 ← 0;
IF M2 THEN S0 ← VAL;
PRINT(S0); {Prints the heights}
M2 ← M1 & (REL = #sex);
{ The processor connected to the sex of each engineer has M2 on }
S0 ← 0;
IF M2 THEN S0 ← ATYP; {sex is an attribute, not number attribute}
PRINT(S0); {Prints the sexes}
```

Another type of tabular output is generated in response to "List the children of each child of John." The table might look like:

child of John	child of child of John
Jack	Joan
	Jake
Jill	Joe

At first it appears that we may have to use a sequential technique. But there is an alternative. If we have a SORT routine available we can use it to sort *pairs* of names. If we sort by column one value, all of the information will be grouped appropriately. A program like the following will work:

```
M0 ← (REL = -#parent-of);
S0 ← 0;
IF M0 THEN S0 ← ATYP;
SORT(S0);
PRINTSORT(S0,VAL);[1]
```

Were there more than two levels in the clause we would no longer be able to answer the query in parallel. To find "each son of each daughter of each son of John" requires using some sequential step. In relational database terms, our system cannot compute a cross-product of relations without resorting to sequential processing. Our system can, however, handle

[1]The standard PRINT procedure will not work since it assumes a single value per node. PRINTSORT, we shall say, prints sorted information and will allow us to print as many columns as desired (in this example, two).

```
PRINT(S0); {Prints the ages}
M2 ← M1 & (REL = #height);
{ The height processor for each engineer has M2 on }
S0 ← 0;
IF M2 THEN S0 ← VAL;
PRINT(S0); {Prints the heights}
M2 ← M1 & (REL = #sex);
{ The processor connected to the sex of each engineer has M2 on }
S0 ← 0;
IF M2 THEN S0 ← ATYP; {sex is an attribute, not number attribute}
PRINT(S0); {Prints the sexes}
```

Another type of tabular output is generated in response to "List the children of each child of John." The table might look like:

child of John	child of child of John
Jack	Joan
	Jake
Jill	Joe

At first it appears that we may have to use a sequential technique. But there is an alternative. If we have a SORT routine available we can use it to sort *pairs* of names. If we sort by column one value, all of the information will be grouped appropriately. A program like the following will work:

```
M0 ← (REL = -#parent-of);
S0 ← 0;
IF M0 THEN S0 ← ATYP;
SORT(S0);
PRINTSORT(S0,VAL);[1]
```

More than two columns might be necessary in a query such as "Who are the contact people for the largest customer of each salesman in each region of the USA?" The result might be:

[1] The standard PRINT procedure will not work since it assumes a single value per node. PRINTSORT, we shall say, prints sorted information and will allow us to print as many columns as desired (in this example, two).

all of the other primitives of the relational algebra (Date [1982], Ullman [1982]). The only reason we were able to handle the previous query is due to its simplicity. Of course sequential methods would work (stepping through each son of John) but in the cross-product case, and only in this case, the required processing time is a function of the size of the class being processed.

6.2. Molecules: Complex Object Representation

Consider a fact such as "The Maru carries coal from Portland to Tokyo." In our current scheme, it is necessary to divide this fact into three separate pieces of information: "cargo of Maru is coal"; "departure-point of Maru is Portland"; and "destination of Maru is Tokyo." This division is not acceptable, however, since we have lost the *context* of the individual pieces of information. Consider the following example. "John hit Joe in the back" and "Pete hit Joe in the stomach" yield the four pieces "John hit Joe", "Joe was hit in the back", "Pete hit Joe" and "Joe was hit in the stomach." Now it is not at all clear that the correct response to "Did Pete hit Joe in the back?" is "No." Nilsson [1980] gives a method of changing any n-ary relation into a set of binary relations without loss. There are several reasons we will choose another method.

- Nilsson's method adds a central node to which each piece of information is attached. This method adds an unnecessary node type for every complex fact in the system.
- For both theoretical and efficiency reasons we would like pieces of information which are tightly coupled to be equally tightly coupled in the representation. This minimizes the "semantic distance" of the information (Sowa [1984]).
- There is another method which uses the BVM efficiently and precisely represents the facts.

region	salesman	customer	contact people
South	Joe	Southern Bell	Carol
			Paul
	Fred	RTI	Pete
West	George	Bank of America	Ellen
			Ralph

One can extend the sorting technique to sort n-tuples of the necessary length at a speed cost linear in the size of the n-tuple. Queries of this sort are computing a cross-product of relations in the machine, and this could easily overflow the available number of processors. In the example above, the word "largest" served to reduce dramatically the number of n-tuples being considered. If we use this sorting technique we might need to monitor the size of intermediate structures to avoid overflows. Ridding ourselves of such monitors, however, is one of the positive features of the system in all other situations. Thus we have reached one of the limits of the system.

6.2. Molecules: Complex Object Representation

Consider a fact such as "The Maru carries coal from Portland to Tokyo." In our current scheme, it is necessary to divide this fact into three separate pieces of information: "cargo of Maru is coal"; "departure-point of Maru is Portland"; and "destination of Maru is Tokyo." This division is not acceptable, however, since we have lost the *context* of the individual pieces of information. Consider the following example. "John hit Joe in the back" and "Pete hit Joe in the stomach" yield the four pieces "John hit Joe", "Joe was hit in the back", "Pete hit Joe" and "Joe was hit in the stomach." Now it is not at all clear that the correct response to "Did Pete hit Joe in the back?" is "No." Nilsson [1980] gives a method of changing any n-ary relation into a set of

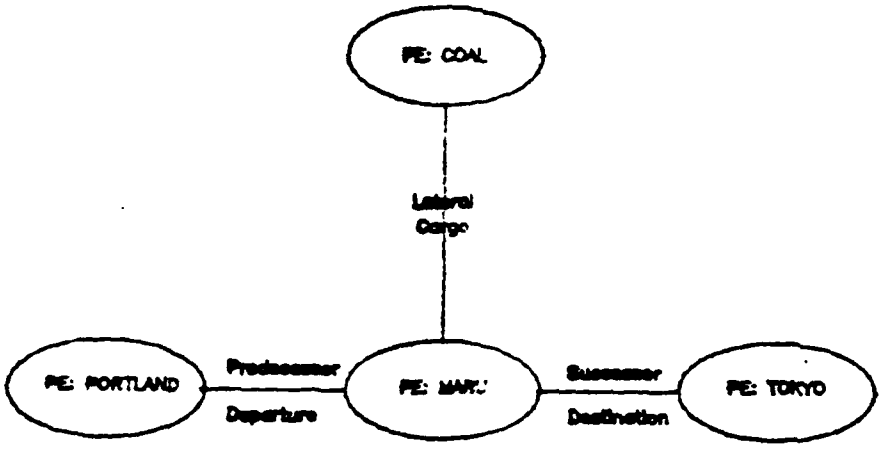
6.3. Some Practical Questions

Let us deal with some practical issues. First, how expensive is this implementation? How much does a BVM cost compared to a sequential machine? Being so simple, each PE in the BVM takes "no more silicon area than some 512 bits of memory" (Wagner [1983]). Since each PE is given around 200 bits of local memory, the BVM takes the silicon area of around 700 million bits instead of 200 million bits. Thus the cost of the BVM is no more than three times the cost of memory alone for a sequential machine with an equivalent amount of memory.

What sort of general improvement does one expect over a sequential machine? We have looked at timing comparisons in section 4.4. What we have found can be summarized as follows. One can think of processing a query as processing one set of nodes after another until a final set is found. If there are k sets of nodes, the average number of nodes in a set is n, and b new nodes must be checked from a given node, the sequential machine will take time proportional to k×b×n, whereas the parallel method will take time proportional to k. The constant for the parallel method is quite large, around 120 microseconds for selection operations (boolean) and around 2.5 milliseconds for inheritance operations (integer). For large enough queries, the parallel method will win; further, the parallel method gives results which are independent on the number of nodes to be processed.

A final question is how transportable is the method described in this paper? Will it only work on the BVM? The limitation of this system is the ability to implement the SHARE algorithm on a given machine. The only requirement for implementation of this algorithm is that the interconnection pattern be one of those which is equivalent to the binary

The method we suggest is called *molecules* because we will use hardware links to chain together all of the pieces of information which make up a fact. Again consider "The Maru carries coal from Portland to Tokyo." We may choose to connect "coal" to "Maru" in the standard way: a processor is assigned to "Maru" and its lateral is assigned to "coal", with the link labeled "cargo". Now let's assign to the processor which is the predecessor of the "Maru" processor the value "Portland" and label the predecessor arc "departure-point". Further, we assign to the successor neighbor of the "Maru" processor the value "Tokyo" and label the successor arc "destination". We thus have



and all the links in the above diagram are in hardware. Now a query such as "who carries coal to Tokyo" requires *no* SHARE operations.

Although we shall leave molecules as a possible extension, one point is worth mentioning. If "destination" is found via a successor link in this molecule, it should always be found via the successor link of every molecule (and single connection) in which it is used. Otherwise all "match" timings will increase by a factor of three, as the system will have to search the three arcs for each processor.

n-cube, in that one can emulate the other with only a constant delay. Besides for the mesh, most interconnection patterns are equivalent to the binary n-cube; these include the important shuffle-exchange, Benes and omega patterns (Feng [1981]). The method described in this paper then, although designed for the BVM, may be used on a number of other possible machine configurations.

8.4. Conclusion

This paper has described a parallel implementation of associative networks. The BVM, a parallel computer with one million processors, is used to obtain speedups over a "good" sequential method of up to three orders of magnitude. The timing results are "smooth", depending only on the complexity of the query, not the size of the classes being processed. The knowledge representation is elegant in its division of information to the most atomic level. Important concepts from the field of semantic networks, such as property inheritance and cancellation, have been shown to be successfully handled by the system. This paper has described in detail how to build an associative network based querying system which embodies the principle "forget about trying to avoid or minimize the deductive search, and simply *do* it, employing a rather extreme form of parallelism to get the job done quickly" (Fahlman [1978]).

APPENDIX A

The SHARE Algorithm

To implement the share algorithm discussed in the text requires developing a mapping which concentrates the information for each node of the associative network into a single processor then returns the concentrated values to each original processor. The mapping for concentrating may be used in reverse to disseminate the concentrated values, thus the proofs below develop only the concentrating mapping. The proofs determine constraints on the time complexity necessary to reorder the information according to this mapping. The binary n-cube interconnection will be used in the proofs and then a constant-time method will be described which emulates the binary n-cube on the BVM.

A-1 Definitions

N equals 2^n is the number of PEs in the parallel computer. The PEs are numbered 0,1,... N-1. Each PE is assumed to hold a single *value* of interest.

A *bit i relationship* is an interconnection of PEs which pairs up PEs differing only in the ith bit of their PE number. The "larger" PE is the one with the bit on (it has the larger PE number); the "smaller" PE is the one with the bit off. We *apply* a bit i relationship by selecting for each pair of PEs one of the following four *operations:*

STAY: each PE keeps its original value

CROSS: the PEs swap values

ORUP: the larger PE is assigned the value of the OR of the two values; the smaller PE has undefined value.

ORDOWN: the smaller PE is assigned the value of the OR of the two values; the larger PE has undefined value.

The operations which may be applied to the PEs can be pictured as switches (figure A1). The OR operations are used on Boolean values, which are our primary interest. (The operations ADD or MAX/MIN might be considered for integer or floating point values. The actual operation chosen does not affect the analysis.)

A general *mapping* assigns to each input PE an output PE. The mapping is said to be *applied* or *implemented* when the values are moved from their input PE to the assigned output PE. If more than one input PE has the same output PE, the values of these PEs will be ORed together before they reach the output PE.

A *permutation* is a one-to-one mapping: each input PE is mapped to a unique output PE. For implementing a permutation, operations ORUP and

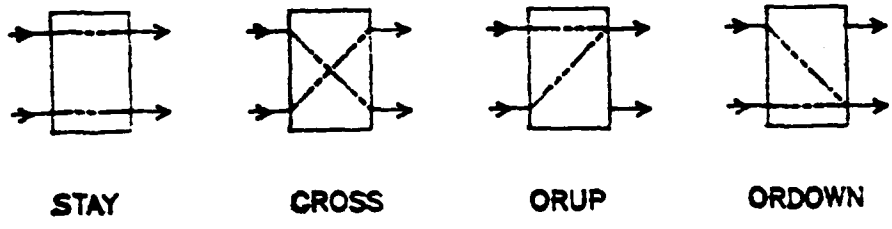


Figure A1: The four operations, pictured as switches

ORDOWN are not necessary.

A *binary n-cube* is a parallel computer consisting of a set of N PEs. Each PE has n direct (hardware) connections to those PEs with which it shares a bit i relationship, for each i less than n. In a cube with n=3 (N=8), PE 5 (binary 101) has connections to PE 4 (100), PE 7 (111) and PE 1 (001). PE 2 has connections to PEs 3, 0 and 6.

A *state* of the binary n-cube is a list of values assigned to the PEs at a given moment.

A *sweep* is a set of n states in which going from state j to state j+1 is achieved by applying a bit i relationship, for some i. Each value of i from 0 to n-1 appears once within a sweep.

Ascend is a sweep with the bit i relationships applied in order: 0,1,...n-1.

Descend is a sweep with the bit i relationships applied in reverse order: n-1, n-2,... 0.

A-2 Theorems for Permuations and Mappings

Theorem 1: A single sweep is not sufficient to implement all permutations.

Proof: There are N! permutations of N items. Since STAY and CROSS are the only operations which may be applied to each pair in a bit i relationship of a permutation, each bit i relationship of a sweep allows $2^{N/2}$ possible switch settings. Further, there are n states in a sweep. Thus, a sweep can only describe $2^{n \times N/2}$ combinations. But this equals $N^{N/2}$ since n is $\log_2(N)$. In other words, there are only $\sqrt{N^N}$ possible permutations that the sweep can

describe. N!, however, has asymptotic value $(N/e)^N$ and is larger than $\sqrt{N^N}$ for all N>2. Thus, a single sweep may not implement all general permutations.

Theorem 2: An ascend followed by a descend is not sufficient to implement all mappings.

Proof: This proof develops a mapping which cannot be implemented for the case N=8. The mapping that provides the counter-example is

Input PE:	0	1	2	3	4	5	6	7
Output PE:	0	1	0	2	1	2	3	3

This means that PE 0 sends its value to itself, so does PE 1, but PE 2 sends its value to PE 0 as well. Thus, at some time in the process, the values from PE 0 and PE 2 must be ORed together. Figure A2 shows the switches for an ascend-descend pair which we claim cannot implement this permutation. Note that between the ascend and descend there would be two bit n-1 relationships. Since these compare the same pieces of information we will remove one of the two redundant steps. The state of the system over time is described by its initial values and the settings of the switches in each phase from left to right. To choose a particular mapping, one needs to assign to each switch one of the four operations. Our claim is that no such assignment will work.

From figure A2, after the first vertical bank of switches, one of PEs 0 and 1 will have sent its value to the high tier, and one to the low tier. Similarly with PEs 2 and 3, and PEs 4 and 5. Either all of the PEs trying to

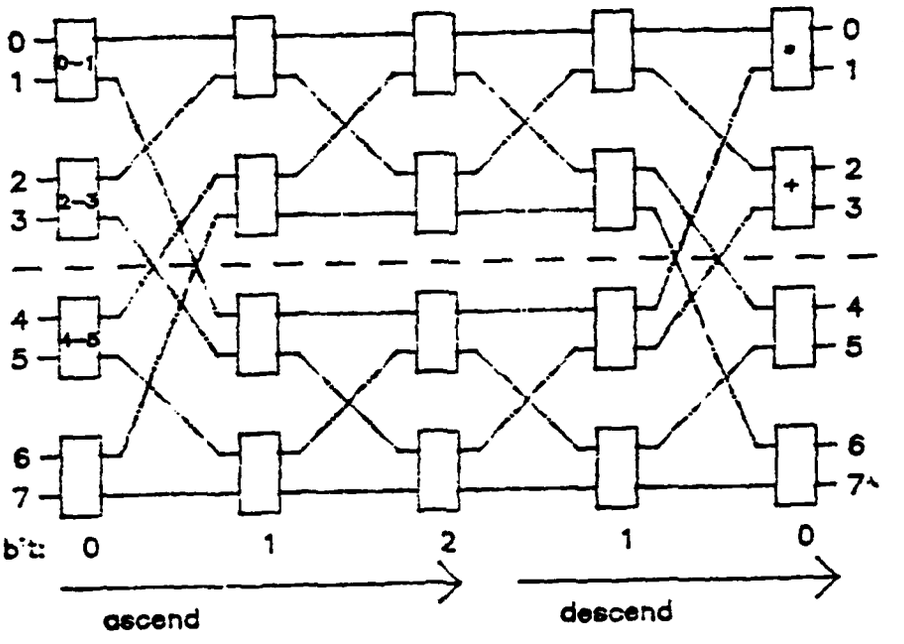


Figure A2: The counter-example proving Theorem 2.
*If PEs 0 and 2 are to send values to PE 0, switches 0-1 and 2-3 must be the same. If PEs 3 and 5 are to send values to PE 2, switches 2-3 and 4-5 must be the same. If PEs 1 and 4 are to send values to PE 1, switches 0-1 and 4-5 must be **different**. This yields a contradiction.*

reach PE 0 at the end must be in the same tier just before the last switch, or the switch marked * would have to be an ORUP. This switch must not be an ORUP, however, since then the output to PE 1 would be undefined. Thus, all of the 0's (the values heading for PE 0) must be sent to the same tier. Since, after the first bank of switches, only the next to last switch allows values to cross tiers, the switches labeled 0-1 and 2-3 must either both be STAY or both be CROSS. Now what can the setting of switch 4-5 be? A similar argument as the one used for the switch marked * yields that the switch marked + must not be ORUP, so the 2's must all be in the same tier. Switch 4-5 must be set the same as switch 2-3 and thus switch 0-1, for otherwise

the 2's would be in different tiers. Finally, look at the situation for the 1's. Obviously one of the 1's will be in the high tier and one in the low tier if the three switches are set the same. Thus, the switch marked * would have to be an ORDOWN to get the 1's to PE 1, and we know this cannot be. Therefore no setting of the switches will allow the mapping to be generated and the theorem is proved.

Theorem 3: A mapping of the share type may be accomplished in an ascend followed by a descend followed by a final ascend. Further, the first ascend and descend implement a permutation and thus require only the STAY and CROSS operations.

Proof: In the Share algorithm we only care that the data gets concentrated, not into which PE it is concentrated. Thus, we will arbitrarily assign the first m PEs the task of holding the concentrated data for the m nodes of the associative network.

The two lemmas below are sufficient to prove this theorem. The first says that an ascend-descend pair implements a general permutation. The second shows that, from a particular permutation of the input, a single ascend can concentrate the data in the manner described above.

Lemma A: An ascend-descend pair may implement a general permutation.

Proof of lemma A: This lemma is well known in the literature. The original proof was by Waksman [1968]. Schwartz [1980] gives a proof which does not use any discussion of "switches" or "connections" but instead builds up the

solution using mappings. Lev [1981] gives a method which allows the control bits to be computed on a parallel processor.

Lemma B: Consider a mapping in which the first n_0 PEs all have PE 0 as their target (output), the next n_1 have PE 1 as their target, etc. and each n_i is greater than zero. Assume that there is a Boolean value associated with each PE and the Booleans are to be ORed together if they have the same target. Then a single ascend can implement this mapping using the four operations STAY, CROSS, ORUP and ORDOWN.

Example:

PE:	0	1	2	3	4	5	6	7
Target:	0	0	0	1	1	2	2	3
Value:	T	F	F	F	F	F	T	F

↓

PE:	0	1	2	3	4	5	6	7
Value:	T	F	T	F	-	-	-	-

Proof of Lemma B: This proof is by induction on the number of initial bits in the target which agree with the current PE. The claim is that after stage i each target can be placed in a PE whose PE number agrees with that target in the first i bits. Clearly if this is true the theorem is proved, since after stage n-1 the PE number will be exactly the target.

For stage 0, we need to move just those values whose target disagrees with the current PE number in the bit 0 position. What could stop us from being able to move these values? Suppose PE 2k and PE 2k+1, for example, both have odd targets. Then each will wish to place its value in PE 2k+1. Now if these two targets are the same, we merely use an OR operation. If the two targets are not the same we have a collision. But note that this can only

occur if both targets are odd and different. Thus, the targets differ by at least two. The initial position, however, requires that PE 2k and PE 2k+1 hold targets that are the same or differ by exactly one. Thus this is a contradiction and we can always move the values so that, after stage 0, each target agrees with its PE number in bit 0.

Now suppose that all targets agree with their respective PE numbers in all bits from 0 to i-1 and we are at stage i. We claim that after stage i we can guarantee that all targets will agree in the first i bit positions. Once again consider what it means to have a collision. We will have two targets which need to agree in the first i bit positions but differ in some higher position (for otherwise the targets are the same and we would OR them). The targets for these two PEs differ by at least 2^{i+1}. Two targets compared at stage i started within $2^{i+1}-1$ of each other by the very design of ascend. (For instance, at bit 1, PE 0 may be compared with the original value from PE 2 or PE 3 but never PE 4.) Once again we have a contradiction, since there is at least one target for each of the first k PEs. So there is no way that two targets starting within $2^{i+1}-1$ of each other can differ by at least 2^{i+1}.

The proof of lemma A in Waksman [1968] and our proof of lemma B are both constructive. We may now assign to each node of the associative network a unique number. The many PEs which make up that node will have this number as their target. We then use lemma A to order those PEs as necessary for input to lemma B. Lemma B then guarantees that we can concentrate the data into the first k PEs of the system. Having completed this construction it is worth noting that it is reversible. We can start from the newly concentrated information and, following the same paths in reverse, return the ORed bits to their initial location. Thus the entire

process takes six sweeps to complete.

Further, since we use the same path back as we used forward, no extra control information is necessary. The amount needed for the forward direction is 2n bits per PE. In the permutation, ascend and descend each use one bit per switch. Since there are n switches per sweep but only half as many switches as PEs, the ascend-descend system requires only n bits per PE. In the concentrate sweep each switch is "four-way" and thus requires two bits. Thus we need another n bits per PE.

In $6\log_2(N)$ logical steps using only $2\log_2(N)$ bits per processor the share algorithm may be implemented on the binary n-cube.

A-3 Emulating the binary n-cube on the BVM

Ascend and descend may be implemented on the BVM at a cost of a constant time factor above that necessary on the binary n-cube. Preparata [1981] describes this implementation in detail; we will sketch that method.

A BVM contains 2^k cycles of k PEs each, so we have the same number of PEs as a binary n-cube with $n=k+2^k$. We define $K=2^k$ so ascend will have k+K phases. The first k phases are *low sheaf* reorderings, that is reorderings within a cycle. The last K phases are *high sheaf* operations that bring pairs together across the lateral connection.

Low Sheaf Phases

Preparata [1981] gives an algorithm which, from a position where the PEs have bit i adjacency, a sequence of $2^{i+1}-1$ steps yields bit i+1 adjacency. For cycle length K equal to eight, the algorithm gives:

```
bit 0 adjacency:  0---1   2---3  4---5   6---7
                  0  1 <- 2  3  4  5 <- 6  7
bit 1 adjacency:  0---2   1---3  4---6   5---7
                  0  2   1  3 <- 4  6   5  7
                  0  2   1 <- 4  3 <- 6  5  7
                  0  2 <- 4  1 <- 6  3 <- 5  7
bit 2 adjacency:  0---4   2---6  1---5   3---7
```

The arrows in the above diagram denote where swapping takes place. Instead of using this algorithm each time Ascend is called, the algorithm is executed once at startup and *shuffle bits* are stored in each PE which tell it what operation to perform at each step. The number of shuffle bits necessary turns out to be 2×(K−k). For k=3, the 2048 PE machine, 10 shuffle bits are necessary; for k=4, 24 bits would be necessary. A small price to pay to avoid a relatively complex calculation on each step of the Ascend algorithm.

High Sheaf Phases

Each cycle of the BVM has K lateral connections to exactly those cycles that differ in one of the K high-bit positions. Thus the following simple-minded algorithm serves to implement the high sheaf phases of Ascend:

```
FOR i := 0 TO K-1 DO BEGIN
  FOR j := 0 TO K-1 DO BEGIN
    apply the necessary operation between the
      ith PE in each cycle and its lateral.
    shift the values within a cycle to the right,
      cyclically.
  END;
END;
```

Variable i determines the current high sheaf phase, which is equivalent to the element number of the PE within a cycle that is active. This method is easy to implement but obviously yields less than 1/K of the binary n-cube's performance, since only a single PE within a cycle is executing the

operation each time through the inner loop.

Pipelined High Sheaf

For the cost of a more complex implementation, we can dramatically improve the high sheaf performance by using a pipelined approach. We will allow a value to proceed through the phases as it cycles right from the zeroth PE in the cycle. The following pseudo-code is sufficient:

```
FOR i := 0 TO K-1 DO BEGIN
  { fill pipeline }
  apply the necessary operation between the
      0 through ith PE in each cycle and its lateral.
  shift the values within a cycle to the right,
      cyclically.
END;
FOR i := 0 TO K-1 DO BEGIN
  { empty pipeline }
  apply the necessary operation between the
      i+1th through Kth PE in each cycle and
      its lateral.
  shift the values within a cycle to the right,
      cyclically.
END;
```

Now the PEs are kept busy half the time. The values loop exactly twice around the cycle instead of the K times of the previous algorithm.

This method seems like a clear winner except that, at any moment, each value in the pipeline is at a different phase of "ascent". Since the operation depends on the current Ascend phase, this method requires that the control bits which determine the operation to perform be reordered so that the operations will occur at the correct time.

An ascend on the BVM will take four times as long as it would on a binary n-cube with the same number of PEs. One factor of two comes from the pipelining, which requires twice as many operations to be performed. The other factor arises due to moving the data around the machine. Given

the simple operations being used during ascend, each movement step costs the same as executing an operation. Thus, for a factor of four in speed we may use the BVM, which is easy to layout in VLSI, instead of the binary n-cube, which is not.

APPENDIX B

Simulation of Parallel Associative Networks

The results described in this paper have been verified by building a simulator of such a system. In this appendix we will describe the computer project. The first section gives a brief account of the BVM simulator used. Section two discusses the program which calculates control bits for the SHARE algorithm. The third section describes the SHARE program. In the fourth section we look at a program which converts the formal description of a data base (figure 4.1) into input for the data base simulator. Finally the data base simulator is described and sample executions shown.

B-1 A High-Level Simulator for the BVM

In Jackoway [1984] a high-level simulator for the BVM, SAL, is detailed. This section will give a summary of its properties. SAL allows the user to write BVM programs in standard Pascal. The user writes a Pascal procedure which, when passed to SAL, is called with the memory of each PE in turn. To avoid referencing data in ways not possible on the BVM, the procedure is sent four parameters by SAL: a pointer to the local memory of the current PE and read-only pointers to the memories of the three PEs to which the current PE is connected. Jackoway [1984] describes rules which eliminate usage which would not be possible on the BVM. Basically these rules restrict the procedures to two or three lines; each procedure does some fundamental task such as copying a word or ANDing two bits. The user

writes a bank of such procedures and then writes his program, which merely sends these procedures to SAL in the appropriate order.

The user may place a template over each PE's memory. This template allows the user to use Pascal integers as well as booleans in the user's programs. Thus one might decide to treat the first sixteen bits as booleans and the next 60 bits as three twenty-bit integers. The flexibility allowed by SAL and the capabilities available from Pascal make algorithm development easier. Since SAL is not an assembler-level simulator, precise timing results cannot be obtained. Instead the user must be satisfied with a count on the number of executions of each user procedure.

B-2 Control Bit Calculation

The control bit program takes a mapping as input and outputs the control bits necessary for the BVM to implement a mapping. The method used is the two step process suggested in the proof of Theorem 3 in appendix A. The first step follows Waksman's algorithm to compute a permutation which sorts the information (Waksman [1968]). The second step uses lemma B to determine the final ascend control bits. Thus to handle the following mapping:

```
Input PE:   0 1 2 3 4 5 6 7
Output PE:  0 1 0 3 1 2 0 1
```

the first step will use the following permutation

```
Input PE:   0 1 2 3 4 5 6 7
Output PE:  0 3 1 7 4 6 2 5
```

to bring the data to a position where the following mapping needs to be determined:

Input PE:	0	1	2	3	4	5	6	7
Output PE:	0	0	0	1	1	1	2	3

Now, lemma B may be applied.

The sorting step is implemented using the method described in Waksman [1968]. The control bits are equivalenced to switches in the following manner. The two processors involved in a bit i relationship have a single control bit which, if on means they swap values (switch CROSS), if off means they keep their original values (switch STAY). ORUP and ORDOWN are never used during a permutation. Since the permutation is implemented as an ascend-descend pair, each pair of processors will meet twice, once during ascend and once during descend. The smaller-numbered processor maintains the ascend control bit, the larger-numbered processor, the descend control bit.

The second step is achieved using Lemma B, but runs in reverse. The control bits for the final stage are determined first. This final stage is the only one that allows values to cross from the lower half of the permutation to the higher half and vice versa. Thus, we can determine the necessary switch settings by noting which half the values must have been on before the final stage. Consider the mapping we have been using throughout, and compare the final state to the initial state.

PE:	0	1	2	3	4	5	6	7
Target:	0	0	0	1	1	2	2	3

It is clear that 0 will only come from the 0-3 half of the target, 1 will come from both halves, and 2 and 3 from the 4-7 half. Thus, at the final stage, 0

will need operation STAY, 1 will need operation ORUP (up to the first half), and 2 and 3 will need CROSS. Now we can determine the next-to-final stage by considering the following two subproblems.

PE:	0	1	2	3	and	4	5	6	7
Next-to-last:	0	1	-	-		-	1	2	3
Target:	0	0	0	1		1	2	2	3

This process may be continued until all switches have been determined. It takes two control bits to define a switch; one bit is stored in each of the two processors connected by a bit i relationship. In the bit 0 relationship, for instance PEs 0 and 1 are connected. The four possible switches are defined as follows:

	STAY	CROSS	ORUP	ORDOWN
PE 0:	0	1	0	1
PE 1:	0	1	1	0

Figure B1 shows a sample execution of this program.

B-3 Implementation of the SHARE Algorithm

To implement the SHARE algorithm, I started by designing and implementing Ascend and Descend (A/D). A non-pipelined version of A/D exists as well as the pipelined version described in (Preparata [1981]). The advantage of the non-pipelined version comes in debugging algorithms. Using the pipelined version, at any point in time each piece of data within a cycle is at a different phase of ascent or descent. Finding bugs is nearly impossible under this condition. Thus, higher level algorithms may be tested using the non-pipelined version and then converted to the faster pipelined version.

```
Mapping to generate:
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
   0   6   7   0   0   0   6   4   7   6   2   1   2   6   7   1

To Decide Switches:  (This is the permutation for Waksman's algorithm)
   0   9  13   1   2   3  10   8  14  11   6   4   7  12  15   5

(Results of Waksman's algorithm)

Switch # 0  STAY  STAY  STAY  STAY  STAY  STAY  STAY
Switch # 1  STAY  CROSS CROSS CROSS CROSS STAY  STAY
Switch # 2  STAY  CROSS STAY  CROSS CROSS STAY  STAY
Switch # 3  CROSS CROSS CROSS STAY  CROSS CROSS CROSS
Switch # 4  CROSS STAY  STAY  CROSS CROSS CROSS STAY
Switch # 5  CROSS STAY  CROSS STAY  STAY  STAY  CROSS
Switch # 6  STAY  STAY  STAY  STAY  STAY  STAY  CROSS
Switch # 7  STAY  CROSS CROSS CROSS CROSS CROSS CROSS

To DOBIT: (Lemma B algorithm)
   0   0   0   0   1   1   2   2   4   6   6   6   6   7   7   7
   0   1   2   3   4   5   6   7  -1  -1  -1  -1  -1  -1  -1  -1

(Results of lemma B algorithm, the numbers are the lower processor of the bit i relationship)

 0:ORUP     0:ORUP     0:STAY     0:STAY
 2:ORUP     1:STAY     1:CROSS    1:STAY
 4:ORDOWN   4:STAY     2:CROSS    2:STAY
 6:ORUP     5:STAY     3:STAY     3:ORDOWN
 8:CROSS    8:ORDOWN   8:CROSS    4:CROSS
10:ORUP     9:STAY     9:STAY     5:ORDOWN
12:STAY    12:CROSS   10:ORDOWN   6:CROSS
14:ORDOWN  13:ORDOWN  11:STAY     7:CROSS
```

Figure B1: Sample run of control bit calculation program

Having routines Ascend and Descend in place, the next step involved

introducing the control bits computed by the program described in the previous section. It is not sufficient to place the control bits in the processor to which they are assigned by this program. In the low sheaf and high sheaf, the data is shifted around and so must be the control bits. Although one could predetermine where each control bit will be needed, a simpler method is used. The control bits are sent around the cycle along the same path that the data will flow during a SHARE and each bit is "dropped" in the processor where it will be needed.

Due to the cost of simulating a 2048 PE machine on a single processor, only the concentrate half of SHARE has been implemented. This is sufficient since the second half involves using the same control bits in reverse.

B.4: The Data Base Simulator

Since SHARE is so expensive to simulate, in the data base simulator an inexpensive version of SHARE is used. (This version violates BVM constraints, but it is the only routine used in the data base simulator that has this trait.) SHARE may also be used to print out the concentrated values, thus it doubles for PRINT.

A MATCH procedure is available which handles most boolean assignment statements which depend on TYP and REL values. The parameters to MATCH are: a TYP field value, a REL field value, an ATYP field value, an associated mark bit number, and the mark bit number of the bit to set. Any of the field values may be set to "ANY" (0 is used); the associated mark bit may be set to "NONE" (again 0 is used). The MATCH routine sets the mark bit in those processors whose fields match the field values and whose associated processor has the associated mark bit on. MATCH can be said to compute

the following boolean function:

(TYP = TYP field value) & (REL = REL field value) &

(ATYP = ATYP field value) & (associated mark bit on)

Other procedures are available for boolean operations, integer operations, copying integers, and an IF routine. As well as these simple procedures, the procedures described in the text have been written. These routines include INHERIT, MARKALL and MINIMIZE. The following pages show annotated runs of the data base simulator.

Having routines Ascend and Descend in place, the next step involved introducing the control bits computed by the program described in the previous section. It is not sufficient to place the control bits in the processor to which they are assigned by this program. In the low sheaf and high sheaf, the data is shifted around and so must be the control bits. Although one could predetermine where each control bit will be needed, a simpler method is used. The control bits are sent around the cycle along the same path that the data will flow during a SHARE and each bit is "dropped" in the processor where it will be needed.

Due to the cost of simulating a 2048 PE machine on a single processor, only the concentrate half of SHARE has been implemented. This is sufficient since the second half involves using the same control bits in reverse.

B-4 Data Base Entry Program

The data base entry program accepts a data base description of the form shown in figure 4.1. Each line begins with the name of a relation and contains the names of pairs of nodes which are to be connected by that relation. The program alphabetizes all of the words to determine the numerical value assigned to each node and relation. Once the numerical assignments are made, the PE assignments are made. Each PE is given the number of the node to which it belongs and the relation in which it is used. These assignments are determined so that the pair of PEs representing an arc are connected by a lateral link. The numerical and PE assignments are saved for use by the control bit calculation program and the data base

```
[A printout of the TYP values for the first 32 processors
 follows.  The first processor has value 1B in hexadecimal
 which is 24 in decimal.  This is a "John" processor.]

state of the machine:
  001E  0011  001B  0017  0012  0023  0013  001D
  001A  0014  0015  0021  001A  002B  001A  002B
  000F  001A  001E  001D  0011  001E  001E  0011
  0007  0025  001B  0011  001B  001D  0023  0021

[We will calculate "grandchildren of John"
 First, MATCH is called to set Word 3 to True in all processors
 which have INVERSE PARENT link to a JOHN node.
 All of the information which is not in brackets [ ] was produced
 by the program.  Information in curly brackets {{{ }}} is a
 subroutine call.
 Note that the names below were printed by the program which looks up
 the values in the dictionary built by the data base program.
 The number in parenthesis after a name is the value assigned to that
 type.
 Note also that in this implementation, all things are stored
 in words, booleans have value 1 for TRUE, 0 for FALSE.]

 {{{ Word[3]  <- REL= INVERSE PARENT(-32    &  ATYP= JOHN(24)   }}}
 {{{ SHARE Word[3] }}}
[children of John:]
TYPES ON:        JACK(17:1)      JILL(20:1)

[Grandchildren are found by turning on those nodes which have
  an INVERSE PARENT link to a node with word 3 on]
 {{{ Word[4]  <- REL= INVERSE PARENT(-32)   &   AWord[3]=True  }}}
 {{{ SHARE Word[4] }}}
[grandchildren of John:]
TYPES ON:        JAKE(1B:1)      JOAN(21:1)      JOE(23:1)
```

```
[Now we proceed to the example "Who are parents of tan dog owners?"]
 {{{ Word[3]  <-  REL= ISA(16)   &  ATYP= DOG(9)   }}}
 {{{ SHARE Word[3] }}}
[dogs:]
TYPES ON:        FIDO(12:1)    POOPSY(36:1)
 {{{ Word[4]  <-  REL=  COLOR(6)   &  ATYP= TAN(39)   }}}
 {{{ SHARE Word[4] }}}
[tan things:]
TYPES ON:    CORVETTE(7:1)      FIDO(12:1)      JILL(20:1)
                MARK(27:1)

[First we will determine (tan dog) owners]
 {{{ Word[5] <- Word[3] & Word[4] }}}
[Word 5 is set true in tan dogs]
 {{{ Word[6]  <-  REL=  OWN(31)   &   AWord[5]=T  }}}
 {{{ SHARE Word[6] }}}
[Jake is the (tan dog) owner]
TYPES ON:        JAKE(18:1)
 {{{ Word[7]  <-  REL=  PARENT(32)   &   AWord[6]=T  }}}
 {{{ SHARE Word[7] }}}
[Jake's parents:]
TYPES ON:        JACK(17:1)      MARY(29:1)

[Now we will determine tan (dog owners)]
 {{{ Word[5]  <-  REL=  OWN(31)   &   AWord[3]=T  }}}
 {{{ SHARE Word[5] }}}
[dog owners:]
TYPES ON:        JAKE(18:1)      JILL(20:1)
 {{{ Word[6] <- Word[4] & Word[5] }}}
[Word 6 is on only in Jill nodes, since color of Jake is White]
 {{{ Word[7]  <-  REL=  PARENT(32)   &   AWord[6]=T  }}}
 {{{ SHARE Word[7] }}}
[Jill's parents:]
TYPES ON:        JANE(19:1)      JOHN(24:1)

[A summary of the run follows.
We have executed 8 MATCH instructions, 2 Booleans (both &'s)
and 8 Binary shares (we used operation OR between values).
By the values which are tabulated, this run would take
only one millisecond on the BVM.]
## @matches: 8 @bools: 2 @ints: 0 @bin shares: 8 @int shares: 0 @steps: 0  ##
```

REFERENCES

(1) Barnes, G. H., et. al., The ILLIAC IV Computer, *IEEE Trans. on Comp.*, Vol. C-17, No. 8, pp. 746-757, Aug. 1968.

(2) Batcher, K. E., Sorting Networks and Their Applications in *1968 Spring Joint Computer Conf., AFIPS Conf. Proc.*, Vol. 32, pp. 307-314, 1968.

(3) Brachman, R. J., On the Epistemological Status of Semantic Networks, in *Associative Networks: Representations and Use of Knowledge by Computers*, ed. Findler, N. V., Academic Press, New York, 1979.

(4) Date, C. J., *An Introduction to Database Systems*, third ed., Addison-Wesley, Reading, MA, 1982.

(5) Fahlman, S. E., *NETL: A System for Representing and Using Real-World Knowledge*, The MIT Press, Cambridge, Mass, 1979.

(6) Fahlman, S. E., Design Sketch for a Million Element NETL Machine, *Proc. AAAI Conf.*, pp. 249-252, 1980.

(7) Fahlman, S. E., Touretzky, D. S., van Roggan, W., Cancellation in a Parallel Semantic Network, *Proc. of the Seventh International Joint Conf. on AI*, Vol. 1, pp. 257-263, Aug. 1981.

(8) Feng, T., A Survey of Interconnection Networks *IEEE Computer*, Vol. 14, No. 12, pp. 12-27, Dec. 1981.

(9) Findler, N. V., ed., *Associative Networks: Representations and Use of Knowledge by Computers*, Academic Press, New York, 1979.

(10) Flyn, M. J., Some Computer Organizations and Their Effectiveness, *IEEE Trans. Computers*, Vol. C-21, No. 9, pp. 948-960, Sept. 1972.

(11) Hendrix, G. G., Encoding Knowledge in Partitioned Networks, in *Associative Networks: Representations and Use of Knowledge by Computers*, ed. Findler, N. V., Academic Press, New York, 1979.

(12) Jackoway, G., SAL: A Simulator for Algorithms, (not published), Term Paper, May 1984.

(13) Lane, H. U., ed., *The World Almanac and Book of Facts 1984*, Newspaper Enterprise Assoc., New York, 1984.

(14) Lev, G. F., Pippenger, N., Valiant, L. G., A Fast Parallel Algorithm for Routing in Permuatation Networks, *IEEE Trans. Computers*, Vol. C-30, No. 2, pp. 93-100, Feb. 1981.

(15) Mago, G. A., A Cellular Computer Architecture for Functional Programming, *IEEE Spring COMPCON*, pp. 179-185, 1980.

(16) Nassimi, D., and Sahni, S., Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network, *JACM*, Vol. 29, No. 3, pp. 642-667, July 1982.

(17) Newman, J. R., ed., *The Harper Encyclopedia of Science*, Harper and Row, New York, 1967.

(18) Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga Pub. Co., Palo Alto, 1980.

(19) Preparata, F. P., and Vuillemin, J., The Cube Connected Cycles: A Versatile Network for Parallel Computation, *Comm. ACM*, Vol. 24, No. 5, pp. 300-309, May 1981.

(20) Schwartz, J. T., Ultracomputers, *ACM Trans. Programming Languages and Systems*, Vol. 2, No. 4, pp. 484-521, Oct. 1980.

(21) Snyder, L., Introduction to the Configurable, Highly Parallel Computer, *IEEE Computer*, Vol. 15, No. 1, pp. 47-56, Jan. 1982.

(22) Sowa, John. F., *Conceptual Structures: Information Processing in Mind and Machine* Addison-Wesley, Reading, MA, 1984.

(23) Thompson, B. H. and Thompson, F. B., Shifting to a Higher Gear in a Natural Language System, *National Computer Conference*, pp. 657-662, 1981.

(24) Thompson, F. B., Personal Communication, June 1982.

(25) Thompson, C. D., Generalized Connection Networks for Parallel Processor Intercommunication. *IEEE Trans. Computers*, Vol. C-27, No. 12, pp. 1119-11125, Dec. 1978.

(26) Tomboulian, S. J., A Parallel Implementation of Associative Memory, (not published),Term Paper, May 1984.

(27) Ullman, J. D., *Principles of Database Systems*, second ed., Computer Science Press, Rockville, MD, 1982.

(28) Wagner, R. A., A Programmer's View of the Boolean Vector Machine, Model-2, Tech. Report, *CS-1981-8, Dept. of Computer* Science, Duke University, Oct. 1981.

(29) Wagner, R. A., The Boolean Vector Machine (BVM), *IEEE Conf. Proc' 10th Annual International Symp of Comp Arch.*, pp. 59-66, June 1983.

(30) Wagner, R. A., Personal Communication, June 1984.

(31) Waksman, A., A Permutation Network, *JACM*, Vol. 9, No. 1, pp. 159-163, Jan. 1968.

END

FILMED

1-86

DTIC