NEW!

IMPROVED!

Programming the Connection Machine

by

David Park Christman

Submitted to the Department of Electrical Engineering and Computer Science
on January 1, 1983 in partial fulfillment of the
requirements for the Degree of 3.1^0 Kelvin in
Electrical Engineering and Computer Science

DRAFT

Abstract

This thesis examines the issues and techniques of programming the Connection Machine, a highly parallel computer. The Connection Machine consists a large number (on the order of 1 million) of processors connected by an independent message passing network. The programming of this machine is much different from that of serial machines in that a computation is distributed among many small processors. A primary concern of parallel programming is to break a computation into several parts so that the individual parts can be run in parallel. The connection machine offers a paradigm for breaking down a computation: Each processor represents a node in some *sofiware* graph (e.g. a semantic network, a circuit, a tree, or an array). A key concept of the Connection Machine is that the graph is independent of the interconnection topology of the message passing network.

The first section of this thesis describes the Connection Machine architecture, programing examples, and potential applications. The second section describes basic algorithms for for computation on the Connection Machine. A few sample applications for the Connection Machine are discussed in the third section.

Reduced Rice!

All new Figures!

Thesis Supervisor: Gerald Jay Sussman

Title: Associate Professor of Electrical Engineering and Computer Science

Programming the Connection Machine

by

David Park Christman

Submitted in partial fulfillment
of the requirements for the
degree of

3.1^{0} Kelvin

at the

Massachusetts Institute of Technology

January 1, 1983

Signature of Author_____
Department of Electrical Engineering and Computer Science
January 1, 1983

Certified by_____
Gerald Sussman
Thesis Supervisor

Accepted by_____
Chairman, Department Committee

Programming the Connection Machine

by

David Park Christman

Submitted to the Department of Electrical Engineering and Computer Science
on January 1, 1983 in partial fulfillment of the
requirements for the Degree of 3.1^0 Kelvin in
Electrical Engineering and Computer Science

Abstract

This thesis examines the issues and techniques of programming the Connection Machine, a highly parallel computer. The Connection Machine consists a large number (on the order of 1 million) of processors connected by an independent message passing network. The programming of this machine is much different from that of serial machines in that a computation is distributed among many small processors. A primary concern of parallel programming is to break a computation into several parts so that the individual parts can be run in parallel. The connection machine offers a paradigm for breaking down a computation: Each processor represents a node in some *software* graph (e.g. a semantic network, a circuit, a tree, or an array). A key concept of the Connection Machine is that the graph is independent of the interconnection topology of the message passing network.

The first section of this thesis describes the Connection Machine architecture, programing examples, and potential applications. The second section describes basic algorithms for for computation on the Connection Machine. A few sample applications for the Connection Machine are discussed in the third section.

Thesis Supervisor: Gerald Jay Sussman

Title: Associate Professor of Electrical Engineering and Computer Science

CONTENTS

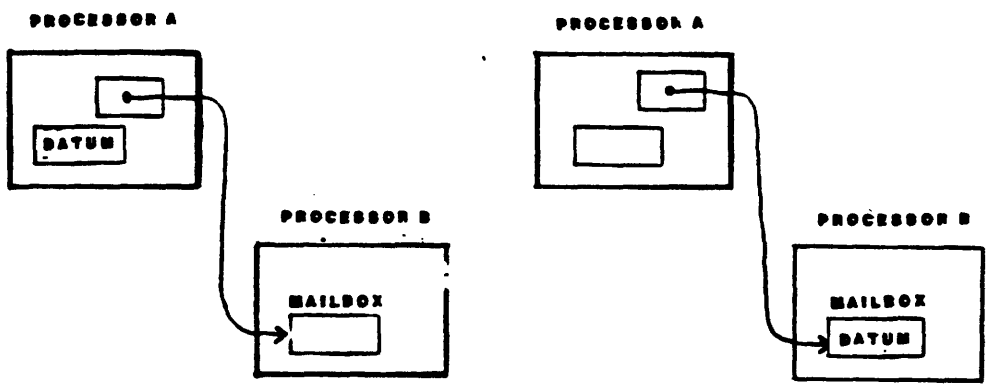
FIGURES

1. Introduction

Now that it has become feasible to build large parallel computer architectures it should be possible to take advantage of parallelism by applying large numbers of processors to a problem. Unfortunately, writing programs for parallel Machines has turned out to be very difficult. In fact, it is not even clear how to build parallel architectures that are useful for any general class of parallel algorithms or applications. There are two basic difficulties: 1) expressing the parallelism of a computation, and 2) exploiting that parallelism on a parallel architecture. Traditional programming languages for serial Machines do not incorporate any way to express parallelism in a computation. It may be possible to write a compiler that finds parallelism in a programs written for serial Machines but this possibility seems limited. A new methodology that is more natural for programming parallel Machines is needed. This thesis will develop a methodology for programming the Connection Machine (CM), a highly parallel computer. This methodology is meant to exploit the specific architecture of the Connection Machine and may have only limited usefulness on other architectures.

The Connection Machine consists of a large collection of simple processors connected by a communication network. Each processor has a unique address in the communication network. Each processor also has a small amount of local memory and a simple ALU for operating on its local memory. Local memory can store data, including the addresses of other processors. If processorA has the address of processor-B then processorA can send a message containing a finite amount of data to processorB using the communication network. (See figure <sending mail>.) Graphs of arbitrary topology can be built using a processor for each vertex. The processor representing each vertex contains the addresses of the other processors representing the vertices to which it is connected. These pointers form the arcs of a directed graph. If two processors have each other's addresses then the arc is bidirectional; this is called a *Connection*. The topology of the *software* graph is independent of the topology of the communication network that interconnects the processors. Since addresses are data, addresses can be sent in messages. This is a very important feature of the Connection Machine: the *software* graph can be manipulated by passing processor addresses in messages.

Fig. 1. Sending Mail



The datum in processor-A is sent to the mailbox in processor-B. The address of the mailbox in Processor-B is stored in Processor-A.

The programming methodology presented in this thesis is fairly simple: the entire computation is represented by a software graph in the Connection Machine and a program that controls the individual processors in the graph. The Connection Machine provides two basic forms of parallelism:

1) Each processor can operate on its local memory concurrently with every other processor.

2) Messages are delivered by the communication network in parallel.

Messages sent from any number of vertices along an arc can be delivered concurrently. The graph abstraction limits the number of cells that can send a given cell a message. Local communication within the graph avoids communication bottlenecks, where one processor receives a large number of messages at once. The major part of this thesis is concerned with techniques for using this methodology to solve interesting problems.

1.1 Thesis Outline

Chapter 2) Concepts

This chapter discuses the important concepts of the architecture of the Connection Machine and programming the Connection Machine. This chapter should be read.

Chapter 3) Notation

This chapter introduces a notation for programming the Connection Machine. The main purpose of this cahpter is to give examples of simple progrms for the Connection Machine. It is not particularly important to understand the details of this chapter.

Chapter 4) N-cube Algorithms

Many algorithms can be performed very quickly using any regular highly interconnected communication topology. This chapter describes some algorithms that we have found to be useful and their implementation using a boolean N-cube connection topology. The particular implementation of these algorithms should be transparent to most programmers.

Chapter 5) Tree Algorithms

Binary trees are an important grahpical abstraction for parallel processing. This chapter describes algorithms for manipulating binary trees on the Connection Machine.

Chapter 6) Application: GA1

This chapter explains how the Connection Machine can be used to explore a search space in parallel. GA1, an expert system that analysis DNA molecule structure, is used as an example.

Chapter 7) Application: Combinators

This chapter describes the implementation of a graph reduction interpreter on the Connection Machine. A graph language is introduced that is interpreted by reductions performed on the graph.

Chapter 8) Application: Relational Data Base

This chapter illustrates an application that takes advantage of the particular connectivity of the communication network for communication.

Chapter 9) Conclusions

This chapter summarises the ideas of programming the Connections Machine.

2. Concepts

The purpose of this chapter is to present some examples of programming the Connection Machine which will serve as a framework for the details presented in later chapters. The first section describes the architecture of the Connection Machine. The second section discusses some programming examples. The third section outlines several applications that could be run on the Connection Machine.

2.1 The Connection Machine Architecture

This section outlines the major parts of the Connection Machine. A forthcomming paper should describe the details of the architecture.

The Connection Machine has 3 main parts:

1) 1 million processors, each with a small amount of local memory

2) a communication network that connects the processors

3) a controlling computer

The communication network is a batch packet switching network that delivers messages between processors. The controlling computer broadcasts a single instruction stream which all of the processors execute. Each part will be discussed in detail below.

2.1.1 The Processors

The processors themselves are very simple; each has about 300 bits of memory and a 1 bit ALU. There are also 16 1 bit flags which perform special functions. (See figure <CM processor>.) The power of the the Connection Machine is in the number of processors, not the speed of any single processor. Processors are very simple (32 will fit on a chip) so that millions can be fabricated. Each processor has a unique address. A processor can store the address of another processor in its memory. Graph vertices are represented by processors; an arc between processor A and processor B is represented by processor A containing the address of processor B.

The ALU operates on 2 bits from the registers and one of the flags and produces two 1-bit results. The first result is optionally written back into one of the operand bits. The second is written into a flag. An *instruction* specifies:

which two bits from the registers will be operated on

which flag will be operated on

which operation the ALU will perform

whether the first result should be written to one of the operands

which flag to write the second result to

There are two special flags: Global and COND.[1] These two flags can be read or written normally. Execution of the instruction stream is conditionalized on the COND flag. If the COND flag of a processor is set that processor is *active* It is possible to set the COND flag in every processor since once a processor is deactivated it cannot activate itself. Special hardware is used to OR every Global flag from each processor in the machine and provide the result to the controlling computer. This mechanism is used to determine if any processors are in a particular state.

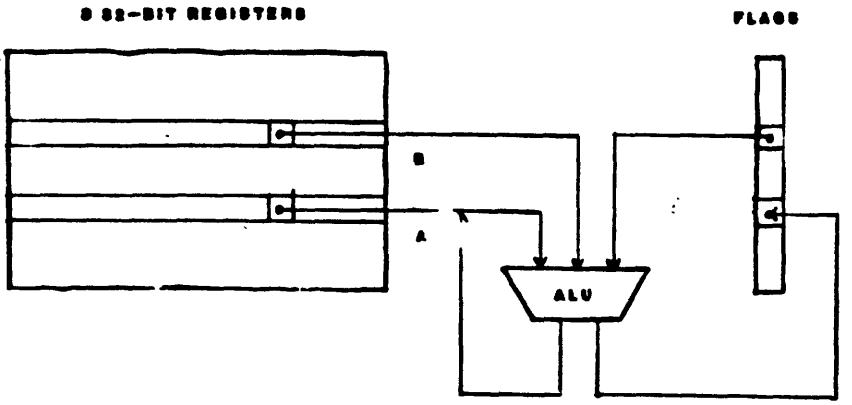
2.1.2 Communication

There are two separate communication networks on the connection machine. The *communication network* is a highly connected network used for global communication. Special hardware is used at each vertex of this network. The *NEWS network* is a 2-d toroidal grid of all the processors. The NEWS network is used for local communication and is also useful for diagnostics since it is much simpler than the communication network.

Communication Network

1. The description of the COND flag is a somewhat simplified version of actual conditional mechanism implemented on the Connection Machine.

Fig. 2. CM Processor



Architecture of the Connection Machine processor. There are 8 32-bit registers, 1 ALU, and 8 flags.

The communication network is a independently addressable batch packet switching network. *Independently addressable* means that messages can be independently addressed to any processor. *Batch* means that a set of messages are delivered concurrently in a batch, or a Delivery Cycle. It should be noted that processors do not compute during a Delivery Cycle. *Packet switching* means that messages have a fixed size. Messages are delivered by passing them back and forth between nodes, or *routers*, in the communication network. A router is a special piece of hardware that routes messages through the communication network. A single router is connected to some small number of processors. A single processor is connected to one router.

The communication network acts as a mailman: picking up addressed messages from processors with messages to send, delivering messages to the processors at the indicated addresses. This is an important abstraction; we do not wish to deal with the particular topology of the communication network when writing programs. It is only important to understand the functionality of the communication network.

There are some considerations that must be taken into account when designing the communication network to fit the above abstraction efficiently. The network should be homogeneous since messages can potentially be sent from any part of the network. A homogeneous network *looks the same* from any cell within the network. A cube is a homogeneous network because the topology looks the same from every corner of the cube; a tree is not a homogeneous network. To efficiently route messages around the network the degree of connectivity should be as high as possible. Generally speaking the higher the degree of connectivity of the communication network the higher the throughput the network. Of course, there are practical limits to the degree of connectivity for large numbers of verticies.

In the prototype Connection Machine currently under construction the topology of the communication network is a 15 dimensional hypercube (or 15-cube) with a router at each vertex (or corner). Each router is connected to 32 processors. An N-cube is an N dimensional cube; each vertex of the cube has a single neighbor in each direction. There are 2^N corners in a boolean N-cube and each vertex is connected to N other corners, one in each dimension. The distance between two verticies is the minimum number of arcs traversed to get from one to the other The maximum distance between verticies is N; potentially one step in each dimension. Each vertex of an N-cube has a unique N bit address relative to a single arbitrarly chosen vertex of the N-cube. Each bit (B_n: nth bit) of the address represents a dimension (D_n: nth dimension). The neighbor of cornerX in dimension D_n has the same address as cornerX except that bit B_n is toggled. The addresses of neighboring corners only differ by one bit. Figure <4 dimensional N-cube> exhibits an example of addressing in a 4-cube.

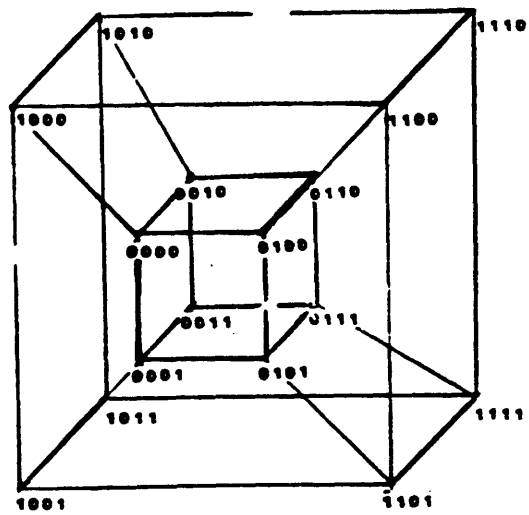
Each processor can only store a small number of messages because each processor has only a small amount memory. There are two bad effects of a single cell receiving a large number of messages:

1) Only a small number of them can be stored

2) The router that is connected to that processor becomes very congested because it has to deal with all of those messages.

A single processor should never receive a large number of messages. A simple way to achieve this is to limit the number of processors that have the address of a single processor. If a processor always has enough memory to store a message from every processor that has its address then there will never be a problem.

Fig. 3. 4 Dimensional cube



NEWS Network

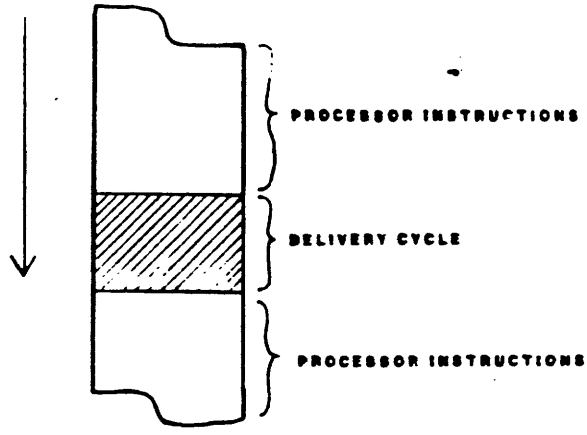
Processors are also connected to one another in a 2-d toroidal grid called the NEWS Network. The NEWS network is not independantly addressable; data can be sent from processors their neighbors in one of the 4 directions (North, East, West, or South). The NEWS network does not require special routing hardware since the sender and receiver are well defined and connected by a wire. The overhead of routing is not required so local communication using the NEWS Network is quite fast although it is quite resticted. The NEWS network is also useful for diagnostics since it is much simpler than the communication network.

2.1.3 Controlling Computer

The third part of the CM is the controlling computer (or CC). The Connection Machine has a single instruction stream which is controlled by the CC. Each processor is connected to the global instruction bus and interprets the single instruction stream; thus, each processor is doing exactly the same thing. At the lowest level a Connection Machine program is one long stream of instructions. (See figure <instruction stream>) During Delivery Cycles the instruction steam is used to control processors communicating with their router Progams have the form: *COMPUTATION delivery-cycle*

COMPUTATION delivery-cycle etc.

Fig. 4. Instruction Stream



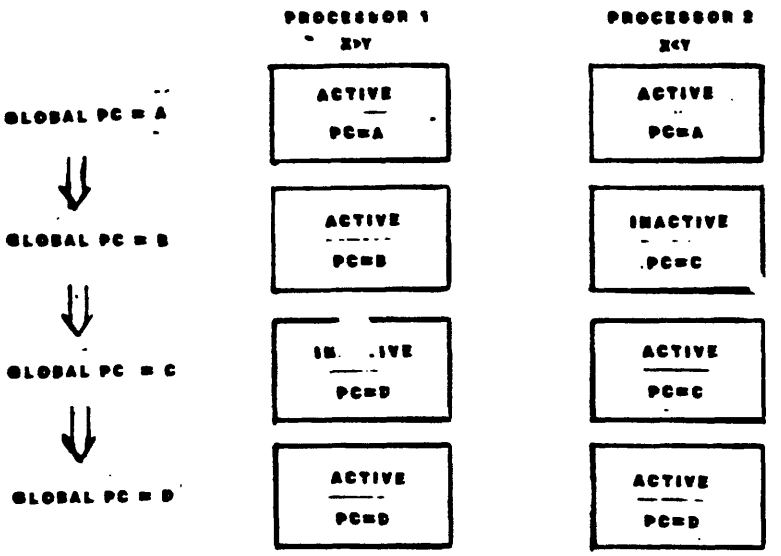
The single instruction stream of the Connection Machine contols the processors. The processors can either be manipulating stored data (processor instructions) or communicating with the communication network (delivery cycle).

Conditional Execution

It is useful to have processors do different things, depending on the data contained in the memory. This is accomplished by each processor conditionally executing the instruction stream using its special COND flag. A processor only executes the instruction stream if its COND flag is set. A processor is de-activated by clearing the COND flag. The CC has the ability to set all COND flags; effectively turning all processors on. More complex control structures can be built using this simple mechanism. Consider the following program:

```
A: If x>y then JUMP B else JUMP C
B: <action B> JUMP D
C: <action C> JUMP D
D: END
```

Fig. 5. Conditional Execution



This figure illustrates how the single instruction stream can control 2 different processors depending on their internal state.

This program is sent one instruction at a time to every processor. The CC does not execute any jumps because there may be some processors that need to execute *action B* and some processors that need to execute *action C*. The global-PC is the current instruction being executed in the linear instruction stream.

The objective is to have each cell perform either action-B or action-C depending on the outcome of the comparison $x>y$. Action-B and action-C can be arbitrarily complex, perhaps even containing conditionals themselves. One method of achieving this control structure would be to have a local-PC on each processor. If an active processor interprets a jump instruction it sets its local-PC to the new value and deactivates itself. After every instruction block the CC sends out the value of the global-PC to all active and inactive processors. The processor is reactivated when local-PC = global-PC. Active processors continue to execute instructions until deactivated. Figure <conditional execution> shows an example of two processors activating and deactivating while running the above program.

The GLOBAL flag

GLOBAL flag of every processor. It is often useful to know if all processors are in a particular state (for example, if any processors are active.) The CC can use this value to control a conditional jump within the program. GLOBAL is most often used as the end-test of an iteration. Each processor may require a variable number of iterations through the same code to terminate. Each processor uses the global bit to indicate that the computation has NOT terminated. The CC checks the value of the globally OR-ed GLOBAL flag after each iteration. If any processor has not terminated (the GLOBAL flag of that processor is set) then the CC would broadcast the body of the iteration again.

2.1.4 Summary of Connection Machine Architecture

The Connection Machine is a very fine grain parallel computer. There are 1 million processors; each processor has 300 bits of local memory. Communication between processors is accomplished by an independent communication network which delivers independanly addressed messages. Processors store the address of other processors in the network forming a software graph. The Connection Machine is a single instruction stream computer. This instruction stream is controlled by a Controlling Computer. To implement conditional control structure there are two special flags: COND which controls conditional execution, and GLOBAL, which is globally OR-ed with the GLOBAL flags of all other processors. The result of OR-ing GLOBAL flags is used by the Controlling Computer to control the instruction stream.

2.2 Programming Examples

There are two basic paradigms of computation using graphs on the Connection Machine:

1) Concurrently passing data within the graph performing computations in parallel on the data.

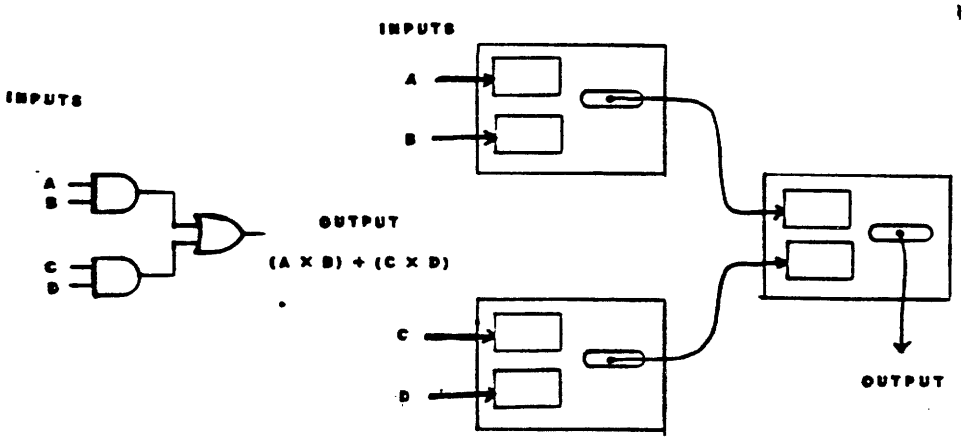
2) Concurrently modifying the graph by passing addresses.

Here are two simple examples to illustrate the two types of computation.

Example 1: Passing data in a graph: Constraint Propagation

A combinational logic circuit is represented as a graph in which the logic gates are the vertices. The wire Connections between the output of gate-1 and the input of gate-2 are represented by the processor that represents gate-1 containing the address of the processor that represents gate-2. When the output of gate-1 changes the new output value is sent to gate-2. The output can be calculated in O(depth of circuit) time.

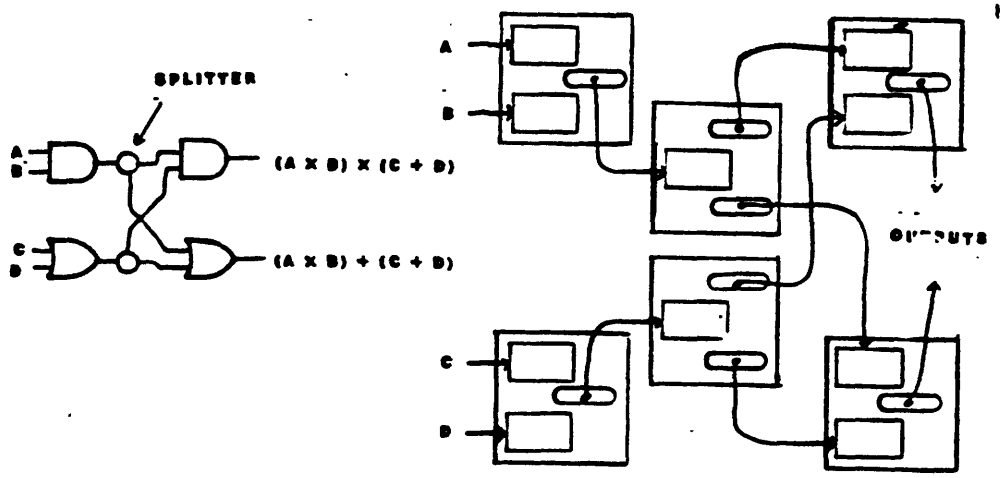
Fig. 6. Constraint Propagation



The logic circuit shown on the left is represented by the graph shown on the right. Rectangles within the processor boxes represent mailboxes for receiving mail. The Ovals represent address of other processors.

Consider the more complex circuit in figure <fan out> below. Because the output of any gate can be the input of any number of other gates and each processor can only store a finite number of addresses (because it has a finite amount of memory) we need to introduce two more processors called fan-out processors that take one value in and send it to two other processors. These fan-out processors can be arranged in a tree so that one output can be the input to an arbitrary number of logic gates.

Fig. 7. Fan Out



Fan out cells (or splitters) are used to connect one cell to two others. Fan out trees built of fan out cells can connect a single cell to an arbitrary number of cells.

The output of such a combinational logic circuit can be computed in time proportional to the number of levels in the circuit. Values for the inputs are passed to the first level of gates which calculate the appropriate function of the inputs and pass the results to the second level of gates. This is procedure is iterated until the final output is calculated. The important point in this example is that the computation is accomplished by local message passing in the graph, which is done in parallel. The computation performed at each node is also done in parallel but the time required for this is small compared to the time required for communication.

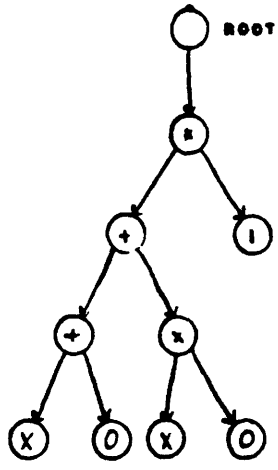
Example 2: Modifying the network: An algebraic simplifier

An algebraic expression can be represented as a tree. Simplifications of the expression can be performed by making local modifications to the tree. Each reducible part of the network can be modified in parallel. For this example the branches of the tree are the binary operators *plus* and *times* {+ *}. A binary operator has a left branch and a right branch. A branch can be a value or another algebraic expression represented as a tree. Values are either a variable {x} or {1 0}. A root vertex is connected to the top of the expression tree. As an example, the algebraic expression:

(* (+ (* x 0) (+ x 0))

is shown in figure <example expression>.

Fig. 8. Example Expression



Graphic representation of the expression: (* (+ (* x 0) (+ x 0)) 1)

Reductions can be carried out by using the following rules:

```
(+ x 0) => x
(+ 1 0) => 1
(+ 0 0) => 0
(* x 0) => 0
(* 1 0) => 0
(* 0 0) => 0
(* 1 1) => 1
(* x 1) => x
```

To reduce, each operator and value sends a message to its parent telling the parent its type. The parent (which is an operator or the ROOT) then decides if a reduction is possible. If a reduction is possible then the parent sends the reduced expression (one of its branches in this case) to its parent, which replaces its branch with the new value and sends its address to the new branch to complete the Connection (that is, make it bidirectional). For simplicity assume that reductions are done in cycles: all operators that can be reduced are reduced in a cycle. When one cycle is complete another cycle begins

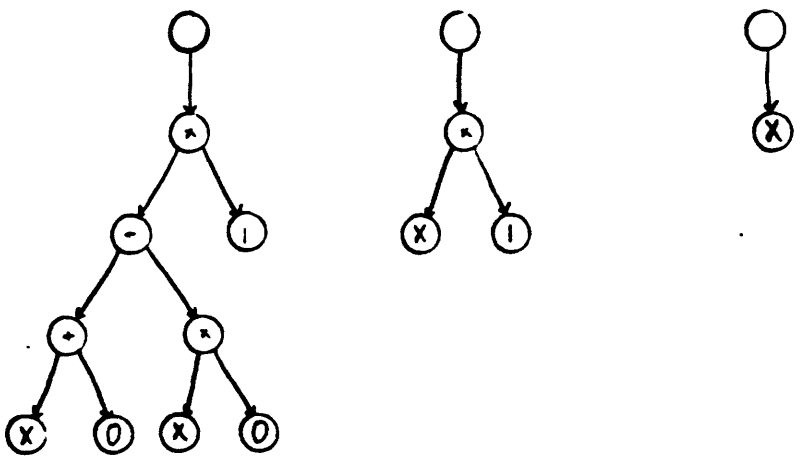
until no further reductions can be performed.

The program for the operator *times* {*} might look like this:

```
Send <message-type: TYPE, value: *> to parent;
When <message-type: TYPE received from both children> do
  BEGIN
  If <my left child or my right child is 0> then become a 0;
  else If <my left child and my right child are 1> then become a 1;
  else If <my left child is a 1> then
    send <message-type: REPLACE, value right-child> to parent
  else If <my right child is a 1> then
    send <message-type: REPLACE, value left-child> to parent;
  END
When message-type: REPLACE received from left child> do
  BEGIN
  Set left child to be the value of the message
  Send <message-type: UPDATE-PARENT, value: self> to left child
  END
When message-type: REPLACE received from right child> do
  BEGIN
  Set right child to be the value of the message
  Send <message-type: UPDATE-PARENT, value: self> to right child
  END
When message-type: UPDATE-PARENT received do
  BEGIN
  Set parent to be the value of the message
  END
```

If these rules are applied to the example expression 4 reductions are performed in 2 reduction cycles; 3 during the first, and 1 during the second. This transformation is illustrated in the figure <Two Reduction Cycles>.

Fig. 9. Two Reduction Cycles



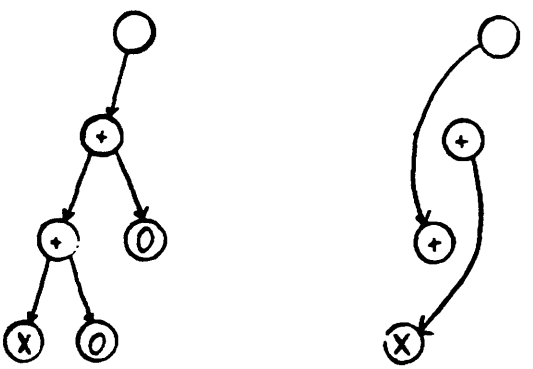
Two reduction cycles are applied to the graph on the left. Three operations are reduced on the first cycle; one operation is reduced on the second cycle.

There is a synchronization problem with this scheme. Consider the expression:

```
(+ (+ x 0) 0)
```

The program given above will fail because 2 reductions are operating on the same part of the network at during the same reduction cycle. (See the figure <Incorrect Reduction>.) The problem is that more than one reduction can overlap the same vertices in the graph; this is a fundamental problem for many graph manipulation computations.

Fig. 10. Incorrect Reduction



Example of an error using the simple reduction algorithm.

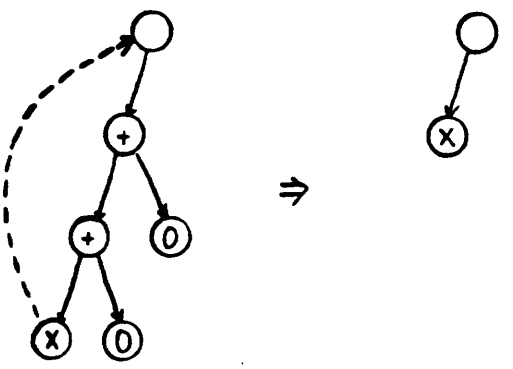
To avoid this synchronization problem allow the value of a reduction to be the value of a reduction. If a branch determines that it can reduce then it checks to see if the branch with which it will replace itself (one of its children) is also reducing. If so then the parent branch must wait until it receives the value of its reducing child before it can send the new value to its parent. Notice that there is some synchronization required to perform the reductions so that the tree remains consistent. Consider the following algorithm:

step 1: branches decide if they can perform a reduction and which branch to replace (local) check if replacement branch is also reducing: if so then wait until new value is attained before sending replacement value up the tree.

ONLY GO ON TO STEP 2 WHEN EVERYONE IS DONE WITH STEP 1

step 2: send new values up the tree waiting when necessary.

Fig. 11. Correct Reduction



The new reduction algorithm produces the correct reduction.

The resulting code would be:

```
STEP 1:
set wait-for-right-child false
set wait-for-left-child false
set replace-left-child false
set replace-right-child) false

Leaf Nodes:
Send <message-type: TYPE, value: *> to parent;

Branch Nodes:
When <message-type: TYPE> is received from both children do
  BEGIN
  If <my left child or my right child is 0> then become a   leaf node;
  else If <my left child and my right child are 1> then become a 1 leaf node;
  else If <my left child is a 1> then
    set replace-with-right-child true;
  else If <my right child is a 1> then
    set replace-with-left-child true;
  If <replace-with-left-child or replace-with-right-child> then
    send <message-type: child-reducing> to parent;
  END
When <message-type: child-reducing> received from left-child do
  If replace-with-left-child then set wait-for-left-child true;
When <message-type: child-reducing> received from right-child do
  If replace-with-right-child then set wait-for-right-child true;

STEP 2:
If replace-with-right-child and (not wait-for-right-child) then
  send <message-type: REPLACE, value right-child> to parent;
If replace-with-left-child and (not wait-for-left-child) then
  send <message-type: REPLACE, value left-child> to parent;

LOOP-UNTIL <no messages are sent in the network>
BEGIN
When <message-type: REPLACE> received from left-child DO
  IF wait-for-left-child THEN
    Send <message-type: UPDATE-PARENT, value: self> to left-child
  ELSE
    BEGIN
    Set left-child to value of message;
    Send <message-type: UPDATE-PARENT, value: self> to left-child
    END
When <message-type: REPLACE> received from right child DO
  IF wait-for-right-child then
   send <message-type: REPLACE, value: value of message> to parent;
  ELSE
    BEGIN
    Set right-child to value of message;
    Send <message-type: UPDATE-PARENT, value: self> to right-child
    END
When message-type: UPDATE-PARENT received do
  BEGIN
  Set parent to be the value of the message
  END
END
```

This program, written in the notation introduced in the next chapter, will appear in an appendix.

2.3 Applications

The Connection Machine was originally designed to process semantic networks. [Hillis81] The architecture is general enough to be useful, though perhaps not optimal, for a larger class of applications. One goal of this thesis is to begin to define this larger class of applications. The next section discusses semantic networks. The following section classifies several types of applications that could be efficiently implemented on the Connection Machine such as digital circuit simulation and data flow computations.

2.3.1 Semantic Networks

A semantic network[1] is a directed graph in which the vertices are nodes and arcs are relations between nodes. Consider the example in figure <semantic network>. This structure states that Apple-3 is a Apple; an Apple is a Fruit; and Fruit tastes sweet. Apple-3 will inherit the fact that it tastes sweet. The semantic network will be represented on the Connection Machine as a software structure by representing nodes as processors. Semantic Networks allow a node to have an arbitrarily large number of relations. Unfortunately, CM processors only have a small amount of memory and cannot store the address of all the processors they are liked to by a relation. The same method that was used in the circuit example to solve the problem of multiple outputs can be used to deal with multiple relations in a semantic network. A node will become the root of two binary trees, the fan-in and fan-out trees. The branches of the fan-in and fan-out tree hold links to the fan-in and fan-out trees of related nodes. The leaves of these two trees are called LINK nodes. Each LINK in a fan-out tree will also be in the fan-in tree of the node to which the relation points. Link nodes store the type of relation. There are four types of processors in this scheme:

---

1. This thesis will only deal with a simple model of semantic networks. See "What's in a Link" by Woods, "NETL" by Fahlman, and "Epistemology Status of Semantic Networks" by Brachman for more information on semantic networks.

1) Nodes; represent the nodes of the network.

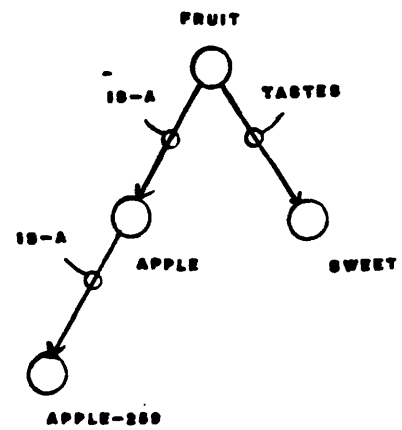
2) fan-in cells; one branch in a binary tree which stores links TO a node.

3) fan-out cells; one branch of a binary tree which stores links TO other nodes.

4) Links; connect two nodes via the fan-out tree of one to the fan-in tree of the other.

For example, there are many kinds of fruit; therefore, there will be many nodes related to the Fruit node. An example semantic network is shown in figure <semantic network>. Figure <CM Graph of Semantic Network> shows how this part of a semantic network would be represented on the CM.

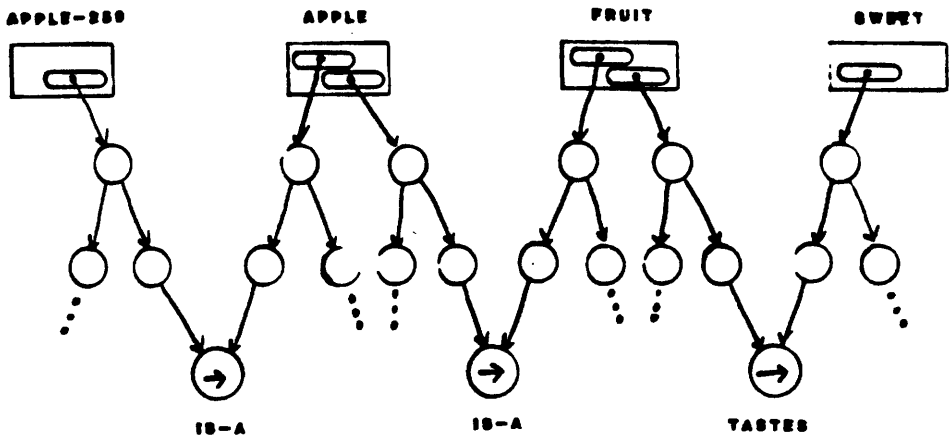
Fig. 12. Semantic Network



Apple-259 is a Apple. Apple is a Fruit. Fruit tastes Sweet.

There are several operations that are important to perform quickly on large semantic networks that are very slow on serial Machines, and could be efficiently implemented on a parallel machine. Here are two examples:

Fig. 13. CM Graph of Semantic Network



Graphical representation of a semantic network using fan-in and fan-out trees to hold multiple relations.

Example 1: Simple Queries

This class of problems involves a simple search of the graph. Property inheritance is such a problem. Given the semantic network above a user might ask the question "Does Apple-259 taste sweet?". APPLE-259 does not have an explicit TASTE relation; it inherits it from APPLE which inherits it from FRUIT. APPLE-259 could inherit this relation from more than one sources. A serial computer would have to search each possibility sequentially. The Connection Machine explores each possibility in parallel.

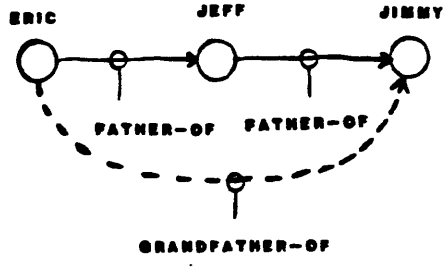
How fast is the Connection Machine versus a serial computer? For a simple calculation, model a query as a simple tree search on a balanced binary tree with N leaves. Communication amoung processors on CM is roughly 100 times slower than memory access on a serial computer. For simple data operations involving no communication, processing on CM is just as fast as a serial Machine. The serial Machine will have to traverse the entire tree which will take (2N)*(communication time + processing time) where N is the number of leaves in the tree. The CM can perform a parallel breadth first search will will take (log N)*(100*communication time + processing time). The CM is a factor of 2N/(log N) faster in processing time because each level of the tree can be processed in parallel. The more significant

comparison is the communication time. Since communication on the CM is slower than memory accesses on a serial Machine N must be rather large (N > 500) for CM to be significantly faster.

Example 2: Adding New Relations

Another important operation is adding new relations to the semantic network in parallel. In the example below part of a family tree is represented using only the father-of link. The goal is to add a paternal-grand-father relation wherever possible in the family tree. New structure must be added for every instance of paternal-grand-father. It is relatively easy to add a signal relation but there may be thousands to add throughout the network. The Connection Machine can add the new relations in parallel.

Fig. 14. Adding New Relations



The grandfather relation is added to a father's father.

2.3.2 Classes of Applications

There are several classes of applications that could potentially take advantage of the parallel architecture of the Connection Machine. Some of theses classes that have been identified will be discussed below.

Semantic Networks: Semantic network operations use the CM to concurrently manipulate a large data structure. The semantic network operations discussed above use the parallel communication abilities of the connection machine to traverse the entire graph in parallel. The ability to modify the network by passing addresses in parallel is also useful. Operations such as set intersection that could take advantage of associative memory can take advantage of parallel processing. In fact, without the communication network the Connection Machine is just a hairy associative memory.

Constraint Propagation: The CM can also be used to process constraint networks. The constraint network is represented as a software graph. Values are propagated in parallel along the arcs of the network. The digital gate example given earlier in this chapter is an example of a constrant propogation network. Another potentially useful application of constraint propocation is switch level simulation of VLSI circuits. Current VLSI chips can contain as many as 500,000 elements. Simulating large systems is very expensive on serial machines because only one element can be considered at a time. The Connection Machine can propagate signals through the network in parallel. **Systolic Algorithms:** A systolic array performs a parallel opereration by passing data through a network of connected processors. Each processor performs some simple operation on the data as it is passed through. Systolic arrays rely on regular grids of interconnected processors to process data. The algorithm is tied to the topology of the communication network. An example of NxN array multiplication in O(N) time using a hexagonally mesh connected network is given in [Mead and Conway80 pg 276-280]. The Connection Machine can simulate a systolic array by either 1) projecting the interConnection topology of the systolic array onto the CM communication topology, or 2) building a software structure that models the topology of the systolic array. In either case the Connection Machine can simulate the systolic array within a constant factor of speed. The Connection Machine could be used as an efficient simulation tool for systolic array designers. If the application did not warrant the cost of building special purpose hardware (the systolic array) the Connection Machine would still be much faster than a serial computer.

Generate and Test: Generate and test is a method for exploring a search space; points in the search space are generated and tested for success. Generate and test applications can take advantage of the Connection Machine by generating the search space in parallel and testing generated possibilities in parallel. The search space is often a tree which can be generated breadth first one level at a time. Testing of generated structures is also done in parallel. The implementation of GA1, an expert system that infers the structure of DNA molecules, will be examined in a letter chapter.

Graph Reduction Evaluation: Computations can be represented as graphs. An operater is a branch of the graph and its operands are the children of the branch. Evaluation is done by reducing the graph by replacing an application of an operater to its operands with the result. The algebraic simplifier that was described earlier in the introduction is a simple example of this. Turner [?] describes an implementation of SKI combinators which translates lambda calculus expressions into a graph which can be evaluated by performing simple local reductions on the graph. The implementation of SKI combinators on the Connection Machine will be discussed in a latter chapter. In graph reduction evaluation the data and the program are represented as data structures in the Connection Machine. The CM instruction stream acts as the interpreter for the program represented as a software structure.

Data Flow: Data flow languages represent a program as a fixed graph. Evaluation is performed by passing streams of messages through the graph. For example: the procedure

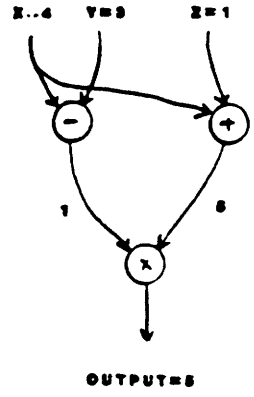
```
(defun foo (x y z) (* (- x y) (+ z x)))
```

can be represented as a graph shown in figure [data flow].

Exploiting the communication network topology: Even though the underlying philosophy of the Connection Machine is to use the communication network abstractly, any regular topology can be exploited. For example, highly connected topologies can be used as sorting Machines [Kung]. A sorting Machine can remove duplicate elements in a set by sorting all of the elements and eliminating all but one of each element type. This is the Projection operation in the Relational Algebra described in [Date]. A latter chapter will examine using the CM for processing Relational Data Bases.

At a lower lever of abstreaction there are certain operations which are useful for manipulating graph structures which can be accomplished mush more efficiently by using the underlying topology of the network. For example: locating free cells to build new structure. Operations that rely on the

Fig. 15. Data Flow



Data Flow graph of (defun foo (x y z) (* (- x y) (+ z x)))

---

communication topology can be formulated as atomic operations; the programmer is not concerned with the particular implementation. If the underlying topology changes only the atomic operations need be reimplemented. It is useful to form hybrid systems in this way.

3. Notation: MP

This chapter describes a simple notation for writing programs for the Connection Machine. It is included for the interested reader; it is not necessary understanding the rest of the thesis. A more abstract language for programmimg the Connection Machine is described in [Bawden83].

The MP language is a assembly language for the CM. MP Expressions are easily reduced into the machine instructions of the single instruction stream which all processor interpret. The major features of MP are named variables, expression evaluation, conditionals, and special features for handling mail.

3.1 Variables

MP has a type system similar to PASCAL. Because there are only a small number of bits availible to each processor the number of bits allocated for each variable is limited. It is possible do declare types as sets or as scalars. Here is an example:

```
;;;Type declarations
DCL-SET-TYPE bit: {0 1}
DCL-SCALAR-TYPE random-set: {0 .. 17}
DCL-SET-TYPE another-random-set: {red yellow orange green}
DCL-SCALAR-TYPE register: {0 .. 2e32-1}

;;;variable declarations
VAR foo: bit
VAR bar: another-random-set
```

Variables can be assigned and tested for equality. Scalars can be compared to other scalars using greater-than and less-than. The results of tests can only be used in conditionals which will be described next. Here is an example:

```
(if (= bar 'red)
    (progn
     (set foo 0)
     (set bar 'blue))
    (toggle foo))
```

3.2 Conditionals

```
(if <conditional> <then-clause> <else-clause>)
```

IF has the same semantics (for a single processor) as IF in any serial language. IF offers a nice abstraction for handling conditional execution using the single instruction stream. An IF expression expands to code that turns on the appropriate processors (depending on the value of the conditional) to evaluate the appropriate clauses of the expression. Expressions can be grouped together to form a clause by `(progn <exp1> ... <expN>)`. Consider this example.

```
;first level conditional
(if (= bar 'red)
    ;first level then
    (progn
     (set foo 0)
     ;second level conditional
     (if (> number 3)
         ;second level then-clause
         (set foo 1)))
    ;first level else
    (toggle foo))
```

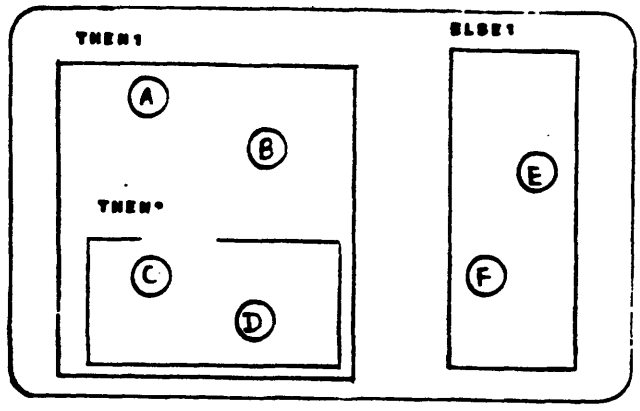
Assume all processors are interpreting the instruction steam. All processors perform the test (= bar 'red). Those processors for which the result is true execute the <then-clause>; the rest evaluate the <else-clause>. While evaluating the first clause there is another conditional. Only those processors that are evaluating the first level <then-clause> will evaluate the second conditional. Only those processors for which both the first level conditional and second level conditional are true will evaluate the second level conditional. Notice that at each level of conditional a subset of the previously *active* processors will become active to evaluate the next level of the conditional. This is called subset selection. For a graphical interpretation of what is happening see figure [graphic-int].

3.3 NEWS communication

Values can be passed along the 2 dimensional toroidal NEWS communication network.

```
(get <NEWS-FLAG> <source-var> <destination-var>)
```

Fig. 16. Graphical Interpretation of IF




```
;first level conditional
(if (= bar 'red)
    ;first level then
    (progn
     (set foo 0)
     ;second level conditional
     (if (> number 3)
         ;second level then-clause
         (set foo 1)))
    ;first level else
    (toggle foo))
```

All processors {A B C D E F} are initially interpreting the instruction stream. The first conditional (= bar 'red) is true for {A B C D}. Those processors remain active. The first level <then-clasue> is evaluated. The second conditional (> number 3) is true for the subset {C D} of {A B C D}. {C D} remain active. The second level <then-clause> is evalued. After the first level <then-clause> has completed evaluation the subset {A B C D} are deactivated and the subset {E F} are activated. The first level <else-clause> is evaluted. All processors are reactived.

The effect of this command is to set <destination-var> in a cell to the value of <source-var> in the cell neighboring it in the direction indicated by <NEWS-flag>. Valid directions are {N E W S} corresponding to North, South, East, and West.

3.4 Example: Conway's Life

Conway's Life is a popular animated graphics demonstration. The state next state of a pixel is determined by the state of its 8 neighbors. Pixels have 2 states: ON and OFF. If 3 neighbors are ON then the next state of that pixel is ON. If there are fewer than 2 or more than 3 neighbors that are ON then the next state of that pixel is OFF. Otherwise, the state of that pixel remains unchanged.

```
Conway's Life:
VAR count: {0 .. 8}
VAR temp, state: {0 1}

;;;initialization
(set count 0)

;;;for each neighbor get it's state and conditionally increment count
;;;(get <NEWS-flag> <source-var> <destination-var>)
;;;Diagonal neighbors require 2 steps (ex: get NW neighbor by going
;;;west, then north)
(get N state temp)
(if (= temp 1) (increment count))
(get E state temp)
(if (= temp 1) (increment count))
(get W state temp)
(if (= temp 1) (increment count))
(get S state temp)
(if (= temp 1) (increment count))
(get N state temp)
(get W temp temp)
(if (= temp 1) (increment count))
(get N state temp)
(get E temp temp)
(if (= temp 1) (increment count))
(get S state temp)
(get W temp temp)
(if (= temp 1) (increment count))
(get S state temp)
(get E temp temp)
(if (= temp 1) (increment count))

;;;conditionally update
(if (= count 3)
    (set state 1)
    (if (not (= count 2))
        (set state 0)))
```

This program expands into about 100 micro instructions.

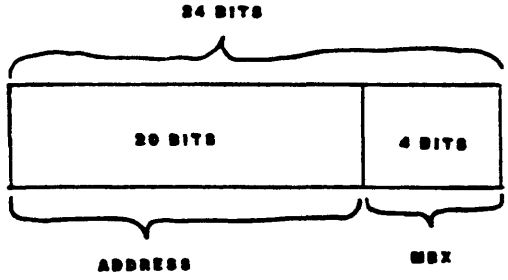
3.5 Mail

There are several types and variables for handling mail and pointers. A *Pointer* is a composite data type that contains the address of another processor and a mailbox within that processor.

```
DCL-TYPE MBX {LEFT-CHILD RIGHT-CHILD PARENT}
```

```
DCL-TYPE ADDRESS (0 .. 2e20-1)
;;;Pointers are composite data types
DCL-TYPE POINTER composite (MBX ADDRESS)

(get-mbx <pointer> <var>)
(get-address <pointer> <var>)
(set-mbx <poiner> <var>)
(set-address <pointer> var>)
```

Fig. 17. Pointer Type



The command set-mbx sets the mbx part of the pointer to the value of <var>. The command get-mbx sets the variable <var> to the value of the mbx part of the pointer. The commands get-address and set-address are analogous.

For every symbol *<quux>* declared to be a MBX a boolean *<quux>-mail* is also declared. This boolean is set by the communication network when a message is delivered to that mail box. In the example code above there would be three booleans (*LEFT-CHILD-MAIL RIGHT-CHILD-MAIL PARENT-MAIL*) declared.

Sending mail is done by invoking a Grand Delivery Cycle. This is done using the command:

```
(send (var1 var2 var3 var4) pointer)
```

When a message arrives in a MBX the boolean *<quux>-mail* is set indicating that mail has arrived in that

MBX. A MBX is abstract buffer that holds the parts of the message. Data is extracted from the mailbox by the command:

```
(get-msg <mbx> <index>)
```

Get-msg will get the indicated message. It is used either as the value of an assignment or as the argument to a predicate.

There is another important abstraction that is used for sending messages. S · the time it takes to execute a CM program is usually dominated by communication time it is useful to share GDCs.

```
(set-up-send (var1 .. varN) pointer)
```

Set-up-send will mark the cell and move the values of the variables into an output buffer where they will be sent. Only one message can be sent to a pointer in this way since there is only one output buffer per pointer. Buffered messages are all sent at once by send-buffered-messages.

```
(send-buffered-messages)
```

Send-buffered-messages sends all buffered messages.

3.6 Iteration

The iteration branching mechanism is implemented by branching conditionaly on the GLOBAL flag. This is the only way to look at the result of ORing all GLOBAL flags together in MP.

```
(while <global-exp>
       Body)
```

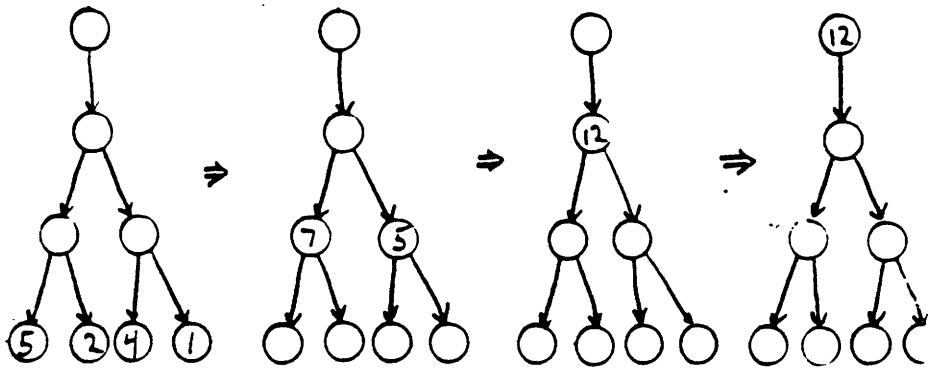
<global-exp> is an expression that is computed at all active cells, the result of which is put in the GLOBAL flag. The body is executed until <global-exp> is false for all active cells.

3.7 Example: Tree Addition

To show how these commands are used here is a simple program that computes the sum of values stored in the leaves of a binary tree.

```
DCL-TYPE MBX {LEFT-CHILD RIGHT-CHILD PARENT}
DCL NODE-TYPE {fan-in leaf TOP}
DCL LEFT-CHILD POINTER
DCL RIGHT-CHILD POINTER
DCL PARENT POINTER
DCL ACCUM NUMBER

(define fan-in-add
  ;;;initialize accum
  (if (= NODE-TYPE 'fan-in)
      (set accum 0))
  ;;;leaves send to parent
  (if (= NODE-TYPE 'leaf)
      (send accum parent))
  ;;;iteration loop
  (while (or (= left-child-mail true)
             (= right-child-mail true)) ;while there is mail
  (if (= NODE-TYPE 'TOP)
      (progn
       (if (= left-child-mail true)
           (add accum (get-msg left-child-mbx 1)))
  (if (= NODE-TYPE 'fan-in)
      (progn
       ;;;add mail from left-child to accum
       (if (= left-child-mail true)
           (add accum (get-msg left-child-mbx 1)))  ;accum <- accum + (gm lc 1)
       ;;;add mail from right-child to accum
       (if (= right-child-mail true)
           (add accum (get-msg right-child-mbx 1)))
       ;;;set up send to parent
       (if (or left-child-mail right-child-mail)
           (set-up-send (accum) parent))
       (set left-child-mail false)
       (set right-child-mail false)))
  ;;;**other code for other processors**
  (send-buffered-messages))
```

Fig. 18. Adding Leaves of a Tree



4. Algorithms for N-cubes

This chapter presents several useful algorithms for a parallel computer that has a boolean N-cube communication topology. These algorithms perform operations that would be inefficient to implement at the level of abstraction where the programmer does not care about the topology of communication network of the specific machine he is programming. A programmer would view these algorithms as primitive operations; like CONS in LISP. Hopefully, these algorithms could be adapted to run efficiently on any parallel machine with a highly interconnected communication network.

> Example 1: A programmer would like to write a CM program in which cells in a data structure build more structure in parallel. This requires that new *free cells* be located to form the new structure. It turns out to be very efficient to do a global computation that calculates the address of a free cell for each cell that wants to cons.
>
> Example 2: There are two sets called A and B. The goal is to form a new set C that is the cartesian product of sets A and B. A primitive is supplied for performing this computation. Primitives are also supplied to access elements from a set one at a time.

The general idea of many of the algorithms in this chapter is to acomplish the computation by a regular patern of passing messages. This tends to utilize the communication much more efficiently than a random pater of passing messages. For example, a delivery cycle where the distance between the sender and recipient is only one step in the N-cube would be much faster than if the distance between sender and recipient was 2 or more.

4.1 Mapping Notation

Many algorithms in this section operate on the absolute address of a cell. In a boolean n-cube the corners are defined by an n bit address. Each corner has n neighbors, one in each dimension. Each bit in the address corresponds to one dimension. The address of a cell's neighbor in the Mth direction is that cell's address (SELF) with the Mth bit toggled. I use a special notation for dealing with sets of addresses and mappings between sets. x represents either a 1 or a 0. The mapping between two sets (ex: x1 sends a message to x0) is defined by each member of the first set mapping to and address in the second set such

that for each dimension:

> 1) if there is an x in the Mth position in the first set and an x in the Mth position in the second set, addresses in the first set with either a 0 or 1 in the Mth position would map to the address in the second set with the same value (x1 -> x0: 11 maps to 10, and 01 maps to 00).
>
> 2) if there is an x in the Mth position in the first set and a 1 (or 0) in the Mth position in the second set, addresses in the first set with either a 0 or 1 in the Mth position would map to the address in the second set with 1 (or 0) in the Mth position (x1 -> 00: 01 maps to 00, and 01 maps to 00).

Example: 1xxx send message to 01xx. 1xxx defines a set of 2^3 cells. 01xx defines a set of 2^2 cells. Each cell of the second set will receive a message from 2 cells in the first set:

```
1000, 1100 -> 0100
1001, 1101 -> 0101
1010, 1110 -> 0110
1011, 1111 -> 0111
```

This notation is useful for describing sets and message passing patterns.

4.2 Dimension Projection

Dimension projection is a way of imposing a spanning tree onto a boolean n-cube using the arcs between corners of the cube as arcs between branches of the tree. These trees are called *calculated trees* because the parents and children of a branch are calculated as a function of the address of the branch. The advantage of calculated trees is that tree operations can be accomplished very quickly because arcs between branches are real communication paths. The calculated trees of Dimension Projection span all processors in the n-cube.

4.2.1 Folding Tree

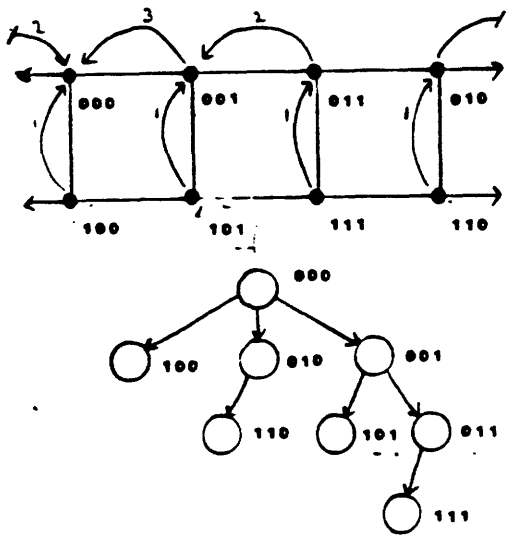
One calculated spanning tree is called the *Folding Tree*. Each cell in a boolean n-cube has n bits of address. In the folding tree the address of a cell's parent is calculated by toggling the first non-zero bit in that cells address. The number of leading zeroes in a processor's address defines the level and number of children of that processor. This definition produces a tree that has a non-uniform branching factor. All children are nearest neighbors in the boolean n-cube. Therefore each child is in a different dimension.

The children of 0001xxx would be:

```
{1001xxx 0101xxx 0011xxx}
```

If dimensions are handled one at a time each branch will receive a maximum one message from its child in that particular dimension. Figure <Folding Dimension Projection> shows a folding tree imposed on a 3-cube.

Fig. 19. Folding Dimension Projection



The Folding Tree spanning a 3-cube. Step 1: 1xx -> 0xx; Step 2: 01x -> 00x; Step 3: 001 -> 000.

Calculated trees are often used for collecting data from the branches and leaves. Example: Each processor stores a number in a variable called ACCUM. The goal is to compute the sum of every ACCUM. The sum is computed by sending data up the tree from the leaves to the higher branches of the tree. The root of a folding tree imposed on an N-cube will have N children. If dimensions are handled one at a time each branch can receive a maximum of one message. When a processor receives a message it adds that value to ACCUM. Iterate through all dimensions starting from the dimension corresponding to the most significant bit (most significant dimension). Each successive iteration deals with the the next most significant dimension. The computation is complete in N iterations. This calculated tree is called the folding tree because on the first iteration half of the cells send a message to the other half; on each successive iteration half of the cells that just received messages send a message to the other half of the cells that just received messages. The final effect is that the cell with address 0000... will contain the sum of every ACCUM.

```
ACTIVE:=TRUE

Iterate: DIM = Start with most significant bit of address.
               on each iteration assign DIM to next most
               significant bit.
  ;;;STEP 1:
  IF ACTIVE=TRUE and NTH-BIT(SELF) = 1 then
     Send ACCUM to Toggle(SELF DIM)
     ACTIVE:=FALSE
  ;;;STEP 2: after mail is delivered
  IF message is received THEN
     ACCUM:= ACCUM + <datum just received>
```

4.2.2 Binary tree

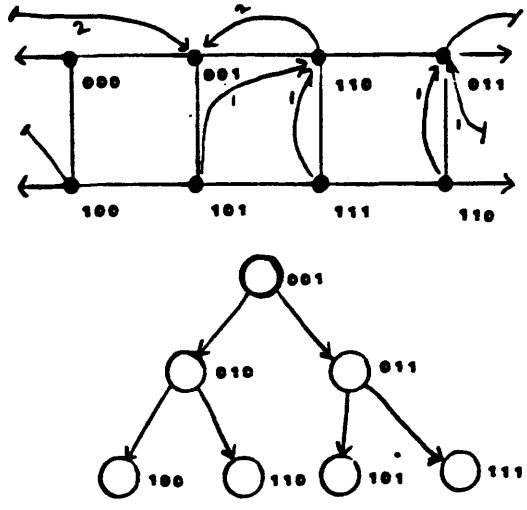
It is often useful to impose a binary tree on the N-cube. One advantage is that information can be pipelined up the tree because each branch only has two children. It is impossible to impose a binary tree on a N-cube using only nearest neighbors. This section describes an algorithm for calculating parents such that the distance from a branch to one of its children is 1 edge of the N-cube and the distance to the other is 2 edges of the N-cube.

The parent of a cell is calculated by toggling the first non-zero bit in its own address and setting the next least significant bit to 1. Successive levels of the tree, starting from the leaves (1xxx) to the root (0001) look like this:

```
1xxx -> 01xx
01xx -> 001x
001x -> 0001
```

Figure <Binary Dimension Projection> shows a binary tree imposed on a 3-cube.

Fig. 20. Binary Dimension Projection



This figure shows a binary tree projected onto a 3-cube. Step 1: 1xx -> 01x; Step 2: 01x -> 001.

4.3 Enumeration

Enumeration, the assignment of unique number from 0 to M-1 to M marked cells, is the basis of many important algorithms. Abstractly, enumeration can be viewed as ordering a disjoint set of cells.

Enumeration is done by a process called subcube induction.[1] Subcube induction works by combining two b-cubes with certain properties into one (b+1)-cube that also maintains these properties. Cells that are to be enumerated are marked. Assume that each b-cube has the following two properties:

> Every element knows how many marked cells are in this b-cube (call this NUMBER-MARKED)
>
> Marked cells are enumerated uniquely from 0 to NUMBER-MARKED - 1. (call this ID)

Assume that there is a one-to-one mapping between elements in two b-cubes.

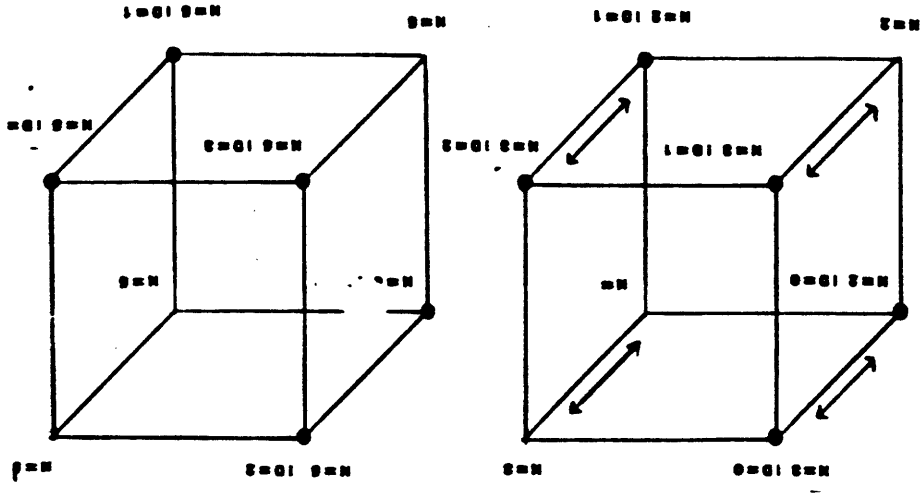
The goal is to combine two b-cubes into one (b+1)-cube maintaining the properties described above. Each element in both b-cubes send their NUMBER-MARKED to the congruent element in the other b-cube. Each element receives a message from the congruent element in the other cube (call it OTHER-NUMBER-MARKED). Each element sets NUMBER-MARKED to the sum of NUMBER-MARKED and OTHER-NUMBER-MARKED. NUMBER-MARKED is now the total number of marked cells in both b-cubes. Within only one of the b-cubes all marked cells set ID to the sum of OTHER-NUMBER-MARKED and ID. Marked cells are now uniquely enumerated from 0 to NUMBER-MARKED - 1. Both properties are maintained in the (b+1)-cube. Figure <enumeration> shows two 2-cubes combined into one 3-cube.

Now we shall show how this process of combining two enumerated b-cubes to form a (b+1)-cube can be applied to enumerating an N-cube. Initially there are 2^N 0-cubes. A 0-cube is just a single cell. In a 0-cube if the cell is marked then NUMBER-MARKED is 1 and ID is 0; if the cell is not marked NUMBER-MARKED 0 and ID is undefined. 0-cubes are paired and combined into 2^{N-1} 1-cubes. 1-cubes are paired and combined into 2^{N-2} 2-cubes. This process is iterated until there is 1 N-cube.

---

1. Invented by Alan Bawden in the context of the Connection Machine.

Fig. 21. Enumeration



Two enumerated 2-cubes combine to form one enumerated 3-cube.

When this is done all marked cells will be uniquely enumerated. This requires N iterations of pairing and combining.

A single combination will run very quickly if the mapping between the two combining cubes is along communication lines. Observe that a set of B bits of the address bits defines a B-cube that is embedded in the N-cube assuming the remainder of the bits are fixed. For example, in an 7-cube:

```
xxx0000
xxx1000
```

defines 2 3-cubes embedded in the 7-cube. There is one-to-one mapping along arcs of the 7-cube between the two 3-cubes (this should be fairly obvious). To perform the enumeration on an 7-cube would require 8 iterations of pairing and combination. Communication will be between the cells as paired below. The leading Xs represent the b-cubes; the trailing Xs represent the number of b-cubes being combined. There will be 2*128 messages sent each iteration.

```
0xxxxxx  x0xxxxx  xx0xxxx  xxx0xxx  xxxx0xx  xxxxx0x  xxxxxx0
1xxxxxx  x1xxxxx  xx1xxxx  xxx1xxx  xxxx1xx  xxxxx1x  xxxxxx1
  128      64       32        16        8        4        2
0-cubes  1-cubes  2-cubes  3-cubes  4-cubes  5-cubes  6-cubes
```

4.4 Consing

Dynamically building structure in parallel is an improtant capability of the Connection Machine. Cells, being viewed as active processes, must be able to "cons" free cells quickly and in parallel. Problem Statement:

> There are two sets: A set of marked cells want to find the address of a free cell and a set of marked free cells. Assume that the set of free cells is larger than the set of cells that wants to cons. The goal is have each cell that wants to cons receive a unique address of a free cell.

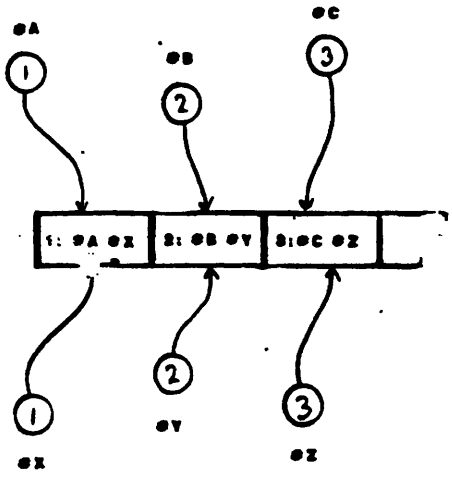
Historically this has been one of the more interesting problems that the CM group has tried to solve.

The algorithm presented here consists of two parts:

> 1) Uniquely enumerate cells that want to cons. then enumerate free cells. The time required for this operation is roughly 20 delivery cycles per enumeration. Enumeration was described in the last section.
>
> 2) Use the ID (enumeration number) of the cells in both sets as the address of an intermediate cell. Cells in both sets send a return address to this intermediate cell. Intermediate cells will have to have 2 mailboxes free to handle these two messages. Intermediate cells send the return address of the free cell TO the return address of the cell that wants to cons. When complete each cell that wants to cons has the address of a unique free cell. This takes 2 delivery cycles. See figure <consing>.

There are two refinements which can be made to this algorithm. First, the intermediate cells should be spread throughout the communication network as much as possible because messages coming into intermediate cells will be serialized. This is easily avoided by having one intermediate cell per chip. If more are required then there could be 2 intermediate cells per chip, etc. The second refinement is to enumerate all free cells initially creating a kind of a free list. After a consing cycle the total number of consed cells is known globally (because of the enumeration of the cells that want to cons). All free cells decrement this number from their ID number. This is analogous

Fig. 22. Consing



---

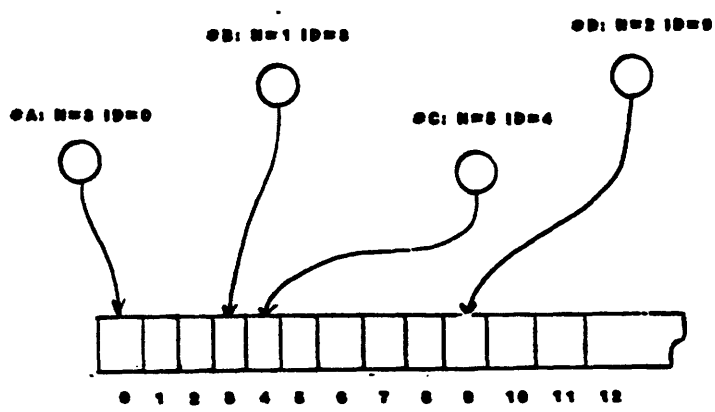
to the concept of a *free list*. The difference is that
it is accessed in parallel by address arithmetic.

It is often useful to allocate more than one free cell at a time.
Marked cells may want the address of some independent number of free cells.
Some might want to cons 3, others 7, etc.
Call this number DELTA.
Goal: Enumerate the
cells that want to cons so that the next enumerated cell from a
given enumerated cell will be ID+DELTA.
For example:

```
cell A: delta=3 id=0
cell B: delta=1 id=3
cell C: delta=5 id=4
cell D: delta=2 id=9
```

Once this is done cells that want to cons will point to the first cell in a block of DELTA intermediate cells. Free cells can be collected by accessing the contiguously addressed intermediate cells. See figure <consing blocks>. This is easily accomplished by modifying the initial conditions of the enumeration

Fig. 23. Consing Blocks of Free Cells



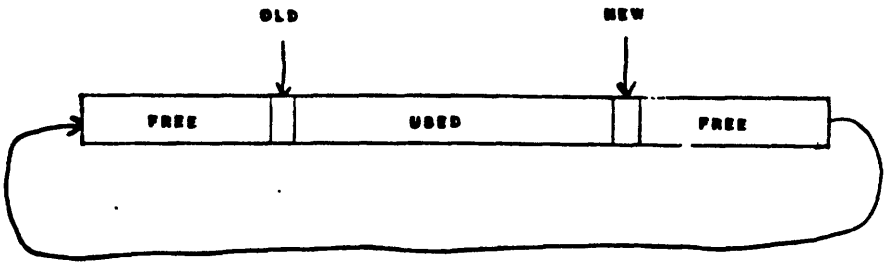
algorithm. Using the enumeration algorithm presented in the last section just count yourself DELTA times when setting up the 0-cubes. NUMBER-MARKED = DELTA initially for the 0-cubes.

4.4.1 Free List Consing

Another modification of this algorithm would be to directly calculate the address of free cells instead of using intermediate cells. This can be done by organizing free cells into a linearly contiguous region. A method for doing this is described in the section on grey code transformations. The address of the first free cell is a globally known number: NEW. Enumeration is done as usual. Instead of going through the intermediate cells the address of a free cell is directly calculated. When the consing is complete NEW is incremented by the total number of cons cells allocated in the consing cycle. This is the next free cell in the list.

Define the list to be a linear ordering of all cells in the machine which wraps around from the end to the beginning. Non-free cells are located between a pointer called OLD and NEW. If Non-free cells can be reclaimed from the cells directly ahead of OLD then NEW can wrap around allocating new cells until it reaches OLD without ever having to perform garbage collection. If NEW ever hits OLD then garbage collection is required.

Fig. 24. Free List Consing



Gargabe collection for this scheme requires that active structure that is distributed through the linear array be compacted to a contiguous region at the beginning of the linear array. The rest of the linear array will be free cells and free list consing can continue. This operation is accomplished in three steps:

> 1) Enumerate cells that are part of active structure. Active cells are numbered from 0 to M. Each cell's ID will be its new address at the beginning of the linear array.
>
> 2) Pointers within the active structure must be updated with the new addresses. Since each cell knows its new address this operation is easy.
>
> 3) Once address have been updated each cell moves to its new address. Moving data to a cell that is itself moving data to another cell is no problem if there is a small amount of temporary storage available at each processor. New data just replaces the old data.

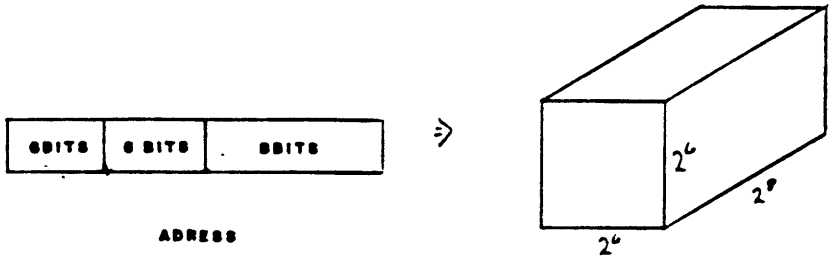
The time required to do Garbage collection is independent of the amount of data to be moved and logarithmicly proportional to the size of the N-cube (enumeration) if one assumes that delivery cycle time is constant. Enumeration takes 20 delivery cycles which is logarithmicly proportional to size of the N-cube. Updating connections (bidirectional pointers) can be done in constant time because each cell

can only have a small number of connections. Moving cells requires time proportional to the amount of data contained in each cell.

4.5 Grey code transformations

This section describes how to find a Hamiltonian path[1] through an N-cube, or a subcube of the N-cube. The N bits can be subdivided into S sets (Si bits in each) which will define an S dimensional space with 2^{Si} elements in each dimension. For example, the 20 bits in the address of each processor in the 20-cube could be divided into 3 sets: S1 S2 S3. S1 would be 6 bits; S2 would be 6 bits; and S3 would be the remaining 8 bits. This would define a $2^6 x 2^6 x 2^8$ 3-dimensional space embedded within the N-cube. See figure <3-d space projected onto N-cube>.

Fig. 25. 3-d space projected onto N-cube



The address space of the machine is divided into 3 sections which define a 64x64x256 3 dimensional space.

---

1. Visit every member is set exactly once.

A Grey coding is a numbering where the binary representation of each number is only different from its predecessor by 1 bit. Such a numbering will define a Hamiltonian path through an N-cube. An algorithm is presented for converting boolean numbers to grey coded numbers and converting grey coded numbers to boolean numbers.

```
(defun number-to-grey (number)
  (do ((i bits-in-pointer (1- i))
       (result number))
      ((= i 0) result)
    (if (= (nth-bit i number) 1)
   (setq result (toggle-bit (1- i) result)))))

(defun grey-to-number (number)
  (do ((i bits-in-pointer (1- i))
       (result number)
       (first-1-p nil))
      ((= i -1) result)
    (cond (first-1-p
      (cond ((= 1 (nth-bit (1+ i) result))
             (setq result (toggle-bit i result)))))
     ((not first-1-p)
      (cond ((= (nth-bit i number) 1)
             (setq first-1-p t)))))))
```

Example: 5-cube

N	N base 2	Grey Coded N
0	00000	00000
1	00001	00001
2	00010	00011
3	00011	00010
4	00100	00110
5	00101	00111
6	00110	00101
7	00111	00100
8	01000	01100
9	01001	01101
10	01010	01111
11	01011	01110
12	01100	01010
13	01101	01011
14	01110	01001
15	01111	01000
16	10000	11000
17	10001	11001
18	10010	11011
19	10011	11010
20	10100	11110
21	10101	11111
22	10110	11101
23	10111	11100
24	11000	10100
25	11001	10101
26	11010	10111
27	11011	10110
28	11100	10010
29	11101	10011
30	11110	10001
31	11111	10000

4.6 Projection of a tree onto a linear sequence

This section describes projecting a tree onto a linear sequence. This operation is useful for accessing linear sequences of cells in log time instead of linear time. For example: There are 100 linear blocks of 1000 cells each. A unique datum in the first cell of each block is to be copied to every element of the respective linear blocks. It would be advantageous if a tree could be superimposed on the linear blocks.

This turns out to be very easy by combining the ideas of dimension projection and grey coding. The first step is to define the position of each cell in the block relative to the first cell by the folding tree dimension projection algorithm. Once this is done the either dimension projection algorithm can be used by using the position of the cell in the block as its address offset by the address of the first cell in the block.

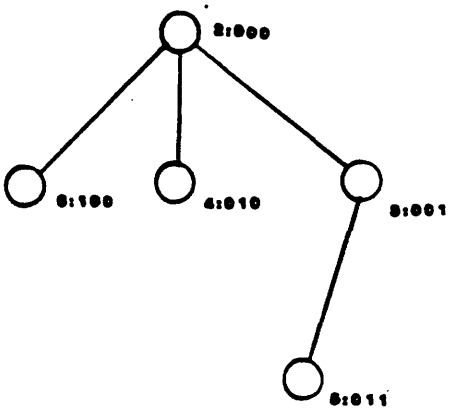
The goal is to have each cell in the block know its position in the block (from which it can calculate the address of the first cell). The block is a set of contiguous cells ordered by a grey code numbering. The first cell knows the number of cells in the block. We will number the cells in the block by using the folding tree dimension projection algorithm to calculate children. The calculation of each child is done by using the offset from the first element as the address and then grey code adding the address of the first element. This is best illustrated by example shown in figure <linear projection>. In this example there is a 5 element block starting at address 011, 2 in grey code numbering. The first child of 011 calculated using the folding tree dimension projection algorithm (the rule is 000 -> 001) would be 010 (011 grey+ 001 = 010, or 2 + 1 = 3). During the second step 000 (index:000) and 010 (index:001) would calculate children using the rule 00x -> 01x. The child of 000 is 110 (011 grey+ 011 = 110, or 2 + 2 = 4). The child of 010 is 111 (011 grey+ 010 = 111, or 2 + 3 = 5). The next step uses rule 0xx -> 1xx. The child of 000 would be 101. All other cells calculate children that are outside of the block.

```
A  B    C  D
0  000
1  001
2  011  0  000
3  010  1  001
4  110  2  010
5  111  3  011
6  101  4  100
7  100

A=index
B=address, sequence is defined by grey code numbering
C=block offset
D=tree folding dimension projection address (same as C)


Folding Tree Rule (use E)
1xx -> 0xx
01x -> 00x
001 -> 000
```

4.7 Cartesian Product

The Cartesian Product calculation can be done by using the ideas of enumeration and linear projection. Given two sets A and B the cartesian product of these two sets is the set of pairs of each possible combinations of 1 element from A and 1 element from B. The cartesian product will have |A|*|B| elements.

Fig. 26. Linear Projection



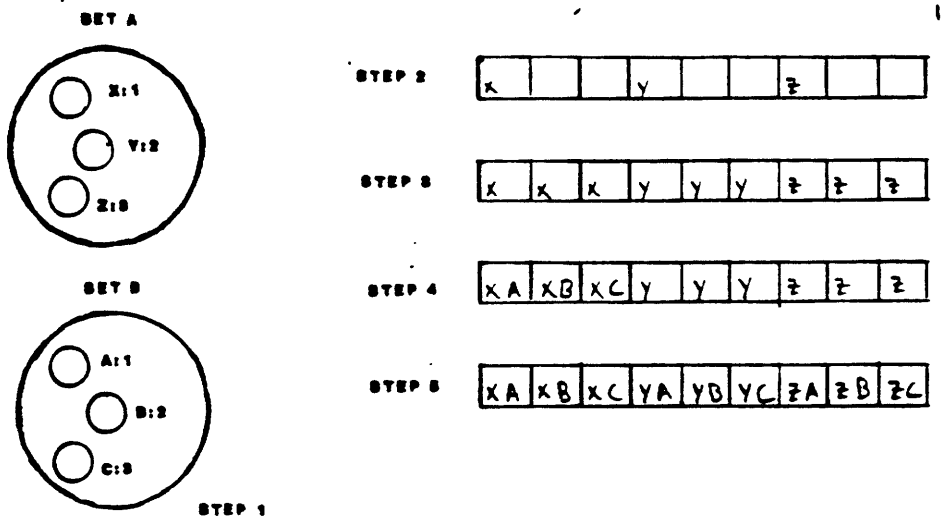
```
Step 1: Enumerate set A and set B.

Step 2: Each element of A is sent to ID*B.  (ID is the enumeration)

Step 3: Each element from A in the linear block replicates itself B times
        into the next B elements from where it started.  This is done by
        linear projection.

Step 4: Each element of B is sent to ID.  This takes one delivery cycle.

Step 5: Each of these elements replicates
        itself A times.  Each successive element is offset by |B|.
```

Step 1 takes 2*logN PDC for 2 enumeration. Step 2 takes 1 GDC. Step 3 takes 2*log|B| PDC. Step 4 takes 1 GDC. Step 5 takes log|A| GDC. Obviously it is better to call the smaller set A because replication does not follow a nice pattern.

4.8 Sifting

Even though locality is not important in our model of the Connection Machine it is the case that cells that are closer together can communicate faster than cells that are separated by large distances. Sifting is a global algorithm for moving cells around the communication network in such a way that cells are closer together. Pointers TO a cell must be updated when that cell moves. An optimization makes it possible to move several times before updating pointers.

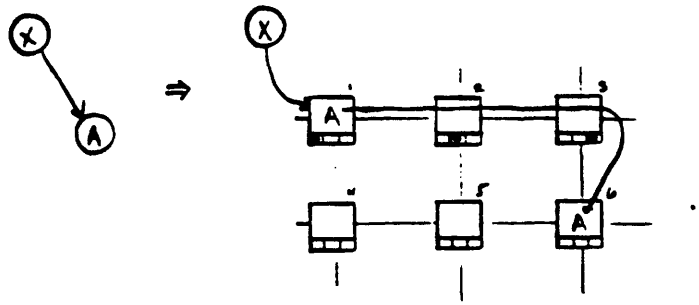
Fig. 27. Cartesian Product



The basic idea is to pair processors and compare the pointers stored by the cells on those two processors. If trading positions is mutually beneficial then the cells trade places. Pairing is done by choosing a dimension and comparing processors along that dimension are. When cells trade places the processors remember when (what iteration) the trade took place. Several iterations can be made without updating pointers TO the moving cells. After several iterations each cell sends a message to the processor where the cell it pointed to used to live. This message then traces the trail left by the cell. When the cell is found then a message containing the new address is sent back to the origin of the message. The procedure would be done for each pointer TO a cell.

An example is shown in figure <Sift>. Cell X points to cell A which lives in processor 1. Cell A moves from processor 1 through 2 and 3 to processor 6. Cell X sends a message to processor 1 which knows that the cell that used to live there moved to processor 2 on the first iteration of the SIFT. The trail is followed to processor 2 which knows that the cell that lived there after the first iteration moved to processor 3. The trail is eventually followed to processor 6 where cell A now lives. A message is sent back to X with the new address.

Fig. 28. Sift



4.9 Arbitration

Given a set of active cells Arbitration selects a single element. This is useful for accessing elements in a set one at a time.

```
Step 1: All cells in the set are activated.

Step 2: Iterate through all bits of the address.
        For each bit: If there are any active cells whose address
        is a 1 in this particular dimension then those cells stay
        activated and all others are deactivated.  Existence of active
        cells is determined by using the GLOBAL bit.  In the case
        where all cells are turned off just back up one step.
```

When this algorithm is complete the element of the set with the highest address is active. This computation requires O(N) I-cycles. An example is shown below. Initially all cells are active. The goal is to deactivate all but one.

```
Step0  Step1   Step2   Step3   Step4   Step5
01101
10011  10011
11001  11001   11001   11001
11011  11011   11011   11011   11011   11011
10111  10111
```

4.10 Sorting; no, not again

Using the CM as a sorting machine can be very useful. For example, to remove duplicate elements from a set: Sort the set and only keep first of duplicate elements. Sorting on highly connected networks has been described in [Kung][1] will not be described here.

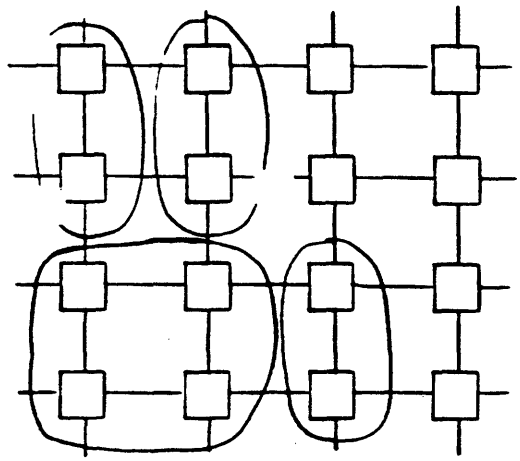
4.11 Macro Cells

Abstractly, it would be desirable if cells were not strictly limited to be contained on a single processor. To state it another way, granularity [2] should only loosely be defined by processor size. Cells will still have to be fairly small to run efficiently on the machine but cells should scale up gracefully. Large cells can be made from smaller cells by connecting them together to form a conglomerate structure. Unfortunately this requires that large cells communicate by using delivery cycles which is fairly inefficient. It would be better if large cells be contained in contiguous memory so that communication would be done over real communication paths. NEWS flags are used to group cells together to form *macro* cells. A macro cell lives on 2 or more contiguous processors. NEWS communication can be used because communication is well defined and the overhead of general message passing is not needed. Macro cells are easily grouped into 2 dimensional areas. Mail to the cell could be delivered to any of the processors that comprise the cell. Abstractly, mailboxes could be located on any of the processors in the cell. This should all be transparent to the programmer.

---

1. Kung uses a parallel version of a Batcher merge sort.
2. Granularity is the amount of memory required for a cell.

Fig. 29. Macro Cells



Single cells are grouped together to from larger Macro cells.

5. Algorithms for Binary Trees

This chapter deals with algorithms for manipulating binary trees as data structures on the CM. Trees are a useful structure for parallel machines because they relate a root to N leaves through logN levels using N-1 branches. Many interesting things can be done by using regular message passing patterns within trees.

There are two types of trees discussed in this thesis:

> **Calculated trees**: The address of the parent and two children of a branch are a function of the address of the branch. Note that the topology of a calculated binary tree cannot change. Calculated trees are usually projected onto some other topology so that it can be treated as a tree. An example of a calculated binary tree is the spanning binary tree used in Dimension Projection described in Chapter "N-cube Algorithms".
>
> **Explicit trees**: The address of the parent and children of a branch are stored explicitly by the branch. The advantage of explicit trees is that they can be manipulated quite easily.

Algorithms described in this chapter that treat a binary tree as static structure can be used on either calculated or explicit trees. For example, the collection algorithm can be run on either a calculated tree or an explicit tree. Algorithms that modify the structure of the tree (eg. tree balancing) can only be used on explicit trees because the structure of a calculated tree can't be changed without modifying the function that calculates the addresses of parents and children.

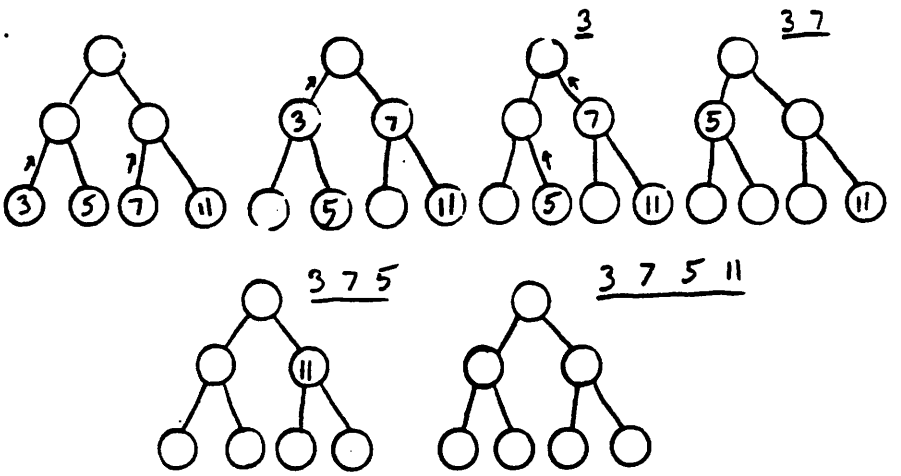
5.1 Passing data in trees

The most basic operation on a tree is passing data between the root and the leaves. Sending data from the root to the leaves is called broadcasting because a single datum is send from the root to many leaves. Sending data from the leaves is called serialization because many data from the leaves are sent to the root which receives them serially.

5.1.1 Serialization

Problem Statement: Each of a subset of the leaves of a binary tree contains a datum. The goal is to take the data out of the tree at the root one at a time to form a serial stream. This is to be accomplished by sending the data through the branches of the tree towards the root. Note that the cells that make up the tree (the fan cells) have a fixed amount of memory to buffer data. Assume each cell has enough memory to buffer one message (excluding memory used for receiving mail). If a cell is buffering a message we will say that it is *full*; if it is not buffering a message we will say that it is *empty*.

Fig. 30. Serialization



M = number of leaves
log M = depth of tree
N = number of leaves containing data

This operation runs in O(M) time on a serial machine because it has to traverse the entire tree to identify leaves that contain data. This operation can be done in O(max[log M, N]) on the CM. Although there is no dramatic decrease in running time between running this operation on the CM and a serial machine, it is useful to see how this operation is performed with decentralized control on the CM. This section

outlines an algorithm for tree serialization and some useful extensions.

```
Algorithm for linear serialization:

Initially leaves with data are marked.

Repeat steps 1 through 3 until there are no data is left in the tree.

  Step 1: Each full cell (either leaf cells or fan-in cells) sends the datum it
          is buffering to its parent.

  Step 2: Each cell cell in the tree can receive 0, 1, or 2
          messages.  Every cell always has enough room to receive a
          message from each of its children.
          If the cell received no messages it does nothing.
          If the cell received 1 message and it is empty then the new datum
          is put in the buffer.  If a empty cell
          receives 2 messages it puts one in the buffer.
          If a cell put a new message in its
          buffer it sends a "confirm" message to the child that sent it.

  Step 3: Each cell that receives a confirm message sets itself to the
          empty state.

VAR value: number
VAR datum-present: {yes no}
VAR confirm: {yes no}
VAR right-child-mail, left-child-mail: {yes no}
VAR right-child, left-child, parent: connection ;;;also declares mbx
VAR right-mail, left-mail, parent-mail: {yes no}
VAR right-mbx, left-mbx, parent-mbx: MBX
VAR aux: pointer
VAR cell-type: {node fan leaf}

;;;assume leaves of the tree are marked (= datum-present yes)
(until (not (global (eq DATUM-PRESENT 'yes)))
      ;;;STEP1
      ;;;DATUM-PRESENT at the root means data at the root
      (if (and (eq cell-type 'node)
               (eq DATUM-PRESENT 'yes))
          (progn
            ;;;do whatever you want with the value
            (set DATUM-PRESENT 'no)))
      (if (and (or (eq cell-type 'fan)
                   (eq cell-type 'leaf))
               (eq DATUM-PRESENT 'yes))
     (send value parent))
      ;;;STEP2
      (if (and (= left-mail 'yes)
               (= datum-present 'no))
          (progn
            (set value left-mbx)
            (set datum-present 'yes)
            (set aux Left)
            (set confirm 'yes))
          (if (and (= right-mail 'yes)
                   (= datum-present 'no))
              (progn
                (set value right-mbx)
                (set datum-present 'yes)
                (set aux Right)
                (set confirm 'yes))))
```

```
(set left-mail 'no)
(set right-mail 'no)
;;;if a datum was taken, confirm to the sender
(if (eq confirm 'yes)
    (send NULL aux))
(set confirm 'no)
;;;STEP3
(if (= parent-mail 'yes)
    (set datum-present 'no))
(set parent-mail 'no))
```

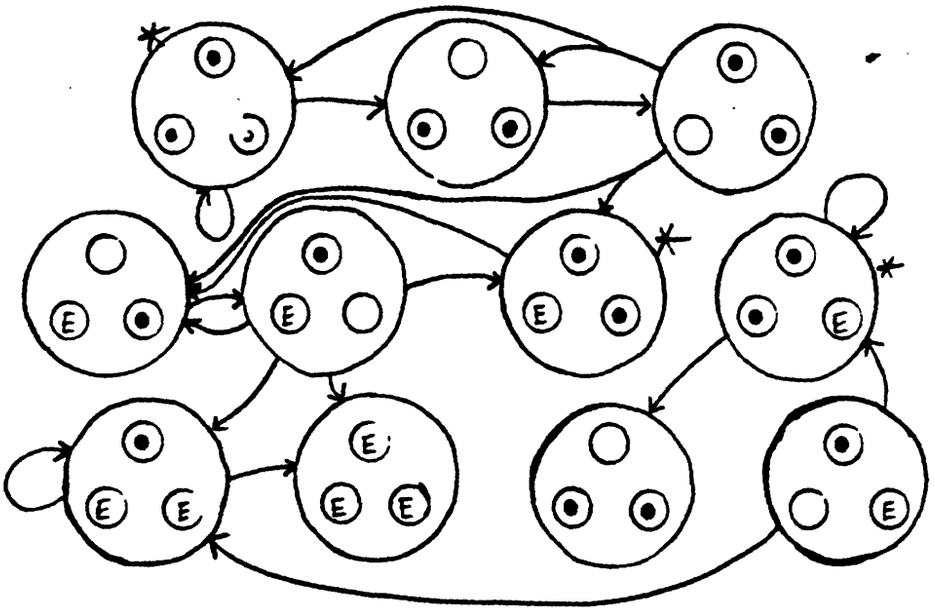
Assuming that the tree is balanced this algorithm runs in time O(d + n) where n is the number of data to be serialized and d is the number of levels in the tree. It takes C(d) time for the first data to reach the root. It then takes O(n) to extract the n data.

PROOF: I shall prove that n data contained in a set of n connected set of branches including the root can be extracted in O(n). In a connected set of branches containing the root the parent of each branch is also in the set. Assume that the data can be moved into such a set in O(d) time. Each fan-in cell in a tree is the root of a "cannonical binary subtree". A canonical binary subtree is a root cell with two children (left and right). Each canonical binary subtree is in the start position. <see figure: tree states> We will try to prove that if the root of any subtree is empty for more than 2 "cycles" (steps 1 through 3 twice) then there are no data in the either of its children or their children. Assume this is the case for the leaves of the canonical binary subtree. We will try to prove it for the root of the subtree. From the start position all possible transitions from the start state to the end state are drawn in figure b. There is no way to change states in such a way that the root is empty for more than 2 cycles. (Assume this for the leaves of the canonical subtree.) By induction a single datum can be extracted from the root of each canonical subtree 2 cycles after the last datum was extracted until the canonical subtree is empty. This is also the case for the root of the tree.

Fairness

Instead of serializing a single datum from each leaf say an infinite stream of data is being fed in at each leaf. We would like our algorithm to have the property that data from each leaf will eventually get to the root. The previous algorithm fails in this requirement because it always chooses data from the left branch. Fairness can be accomplished by each fan-in cell remembering which branch it chose the last time it had to choose between a datum from left-child and a datum from right child. The next time the

Fig. 31. Tree States



fan-in cell has a choice it will take the datum from the other child. This mechanism works because a datum can never be blocked indefinitely. If it is blocked once it will be selected on the next opportunity to move up the tree.

Sorting

A useful extension to serialization is extracting the data in sorted order. This is accomplished in 2 steps: 1) Data are initially sorted into a heap; 2) The choice between the data from the two children at each branch of the tree is based on a comparison. Assume that the comparison operation is greater-than and we want to form a stream from smallest to largest.

```
N=Number of data
D=Depth of tree

Step 1:
Divide the fan-in cells into two sets
(odd and even) based on their depth in the tree.  The root is even (level 0).

STEP 2:  Forming a Heap
Apply the following steps to the tree  until no data are exchanged:

  Step 2-1-even: All odd cells send datum to the even cell above.

  Step 2-2-even: Even cells take the minimum
                 of the the data from their children and the
                 datum they are buffering.  If the smallest datum is
                 from a child, the old datum is replaced
                 by the smallest datum in the buffer.  The old
                 datum is sent to the child that sent the
                 smallest datum.

  Step 2-3-even: Odd cells that received data replace the old datum
                 (now buffered above) with the new datum.

  Step 2-4-odd: All even cells send datum to the odd cell above.

  Step 2-5-odd: Odd cells take the minimum of the the data from
                their children and the datum they are buffering.  If the
                smallest datum is from a child, the old datum is replaced
                by the smallest datum in the buffer.  The old
                datum is sent to the child that sent the smallest datum.

  Step 2-5-odd: Even cells that received data replace the old datum (now
                buffered above) with the new datum.

It takes O(D) to form the heap.

STEP 3: Removing Data
  Once the data are in a heap the next step is take them out in sorted order.
  This is done by taking on element out of the top of the tree after
  running steps 2-1-even through 2-6-odd.
  Notice that after an iteration empty cells, or "bubbles", will always
  be on an even level.  This is important because
  2 adjacent bubbles will allow a datum to go up to the next level of
  the tree without being compared to the datum being stored at its sibling.
  This algorithm runs in O(N) time.
```

This algorithm is significantly faster than heap sort on a serial machine because the heap need not be totally reset after removing an element from the top. Running time on a serial machine is O(N * D) versus O(N + D) on CM.

5.1.2 Broadcasting

Sending a datum from the root to the leaves is called Broadcasting. A single datum can be replicated 2^D times in O(D) time. Algorithm:

```
Step 1:  If you receive mail from Parent send it to Left-Child and Right-Child.
```

Fig. 32. Serialization Sorting

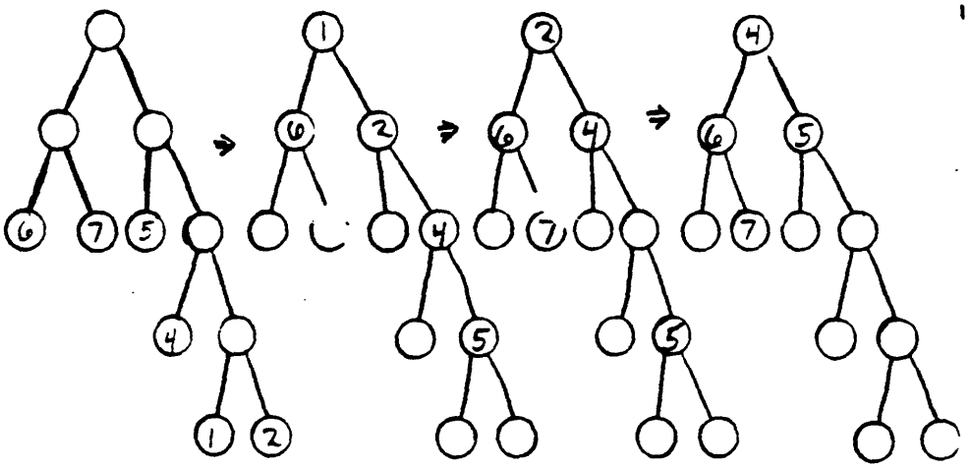
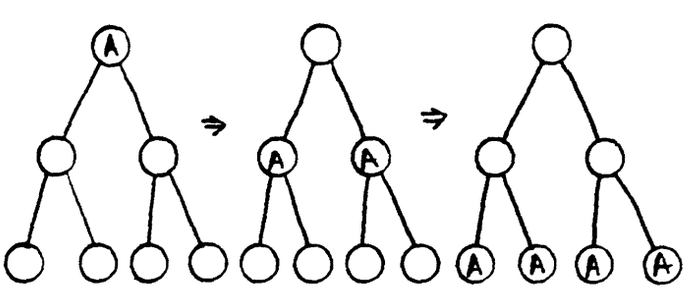


Fig. 33. Broadcasting



5.2 Adding Leaves

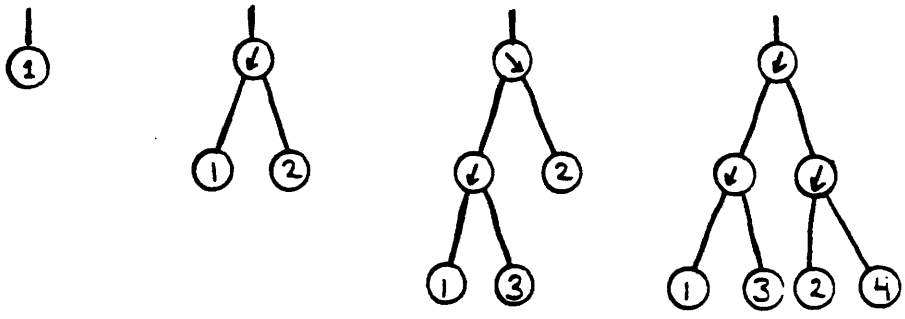
It is important that trees be balanced because the efficiency of many algorithms is proportional to the depth of the tree.

```
Definition of a Balanced Tree:  A tree where the number of leaves
below the left side of a branch is within 1 of the number of leaves
below the right side of the branch.
```

It is useful if tree modifications maintain a balanced tree. This section describes an algorithm for adding single element to a balanced tree resulting in a balanced tree.[1]

The address of the new leaf starts at the root of the tree. This address is passed down the branches of the tree until it reaches the fringe where it is added to the tree by adding a new branch. To maintain a balanced tree each branch remembers which branch the last new element was sent down. The next new element is sent down the other branch. Since new elements alternate between the left and right side of each branch is obvious that this maintains a balanced tree. See figure <Adding Leaves>.

Fig. 34. Adding Leaves



---

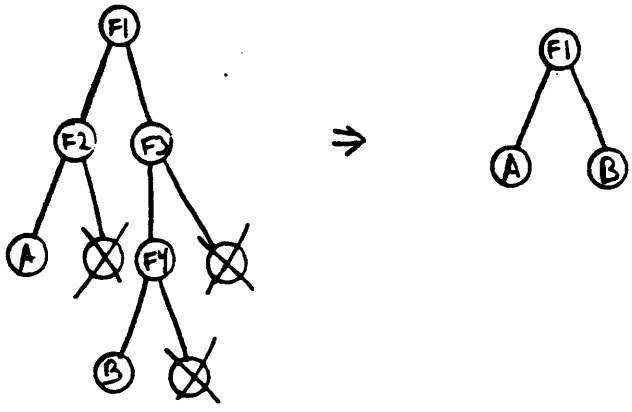
1. This algorithm is described in [Hillis] and [Browning].

5.3 Deleting Elements

The algorithm given in this section deletes a subset of leaves from a binary tree. The resulting tree in not necessarily balanced. See figure <Deleting Leaves>.

```
Step 1: Deleted leaves send an "empty" message to Parent.

Repeate Step 2 until no messages are sent:
Step 2:
If a fan cell receives and "empty" message from one
of its sides (left or right) and has not received an "empty" message
form the other side it sends a "replace" message with the address
of the other side.
If a fan cell receives an "empty" message from one side
and has received and "empty" message from the other side then
that branch sends an "empty" message to Parent.
If a fan cell receives a "replace" message from one its sides
it will replace that side with the address contained in the
message.

Step 3: Each fan cell sends its address to each side that has been replaced.
The cells that receive these messages replace Parent with the new address.
This makes the links between branches bidirectional.
```

Fig. 35. Deleting Leaves

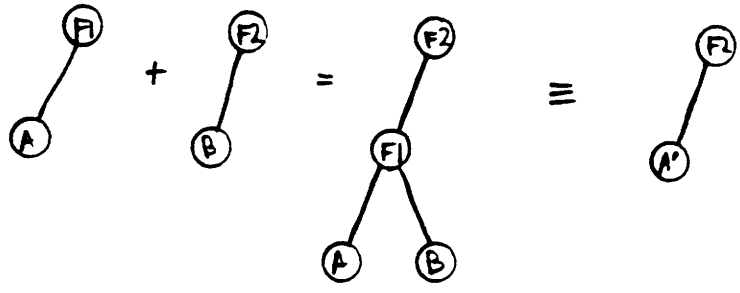


5.4 Collection

The goal of collection is to create a new tree from the subset of the leaves of another tree called the *master tree*. The resulting tree is not necessarily balanced. This algorithm is particularly useful for collecting a tree of marked cells that are not connected in any way by using the spanning binary tree introduced in Chapter "N-cube Algorithms".

As in all parallel algorithms, we would like to distribute the computation as much as possible and keep the total amount of communication low. The goal is to form subtrees in the leaves of the master tree and pass them up the branches of the master tree merging them together. The formation of the new tree with N elements will require N-1 new cons cells. It would be convenient if these new cells could be consed at the same time because it more efficient to cons many cells at once. A tree with a single element is created from the new cell. The left side of this tree leaf of the master tree that created the new cell; the right side is null. Two of these trees can be merged together to form a tree whose left side is a tree that contains the left sides of the original tree and whose right side is null. This tree can be merged with others of this form. See figure <collection>.

Fig. 36. Collection

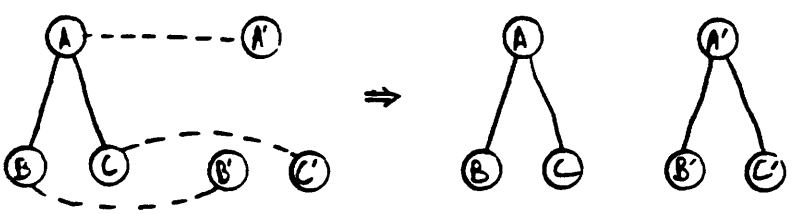


```
STEP1: form subtrees at the leaves that want to be collected
  Each leaf that wants to cons gets a new
  cell (see cons alg in n-cube section).
  It is only necessary to form a uni-directional
  tree while merging.  The uni-directional
  tree can be made into a bidirectional tree in
  one step when the main iteration
  collection step is complete.  At the end of this step each leaf points to the
  root of a canonical subtree.

STEP2: iterate:merge trees and send result to parent.
  This merge can be done in 3 DCs.  The nice thing about this step is that the
  merging can be done concurrently with passing the subtrees
  up the master tree.
```

5.5 Copying

Copying a tree or a graph can be easily done in constant time (assuming a constant number of connections per cell) once the structure is marked so the parts know they are copying themselves. First, each cell makes a copy of itself. Each cell then passes the address of the new cell to all cells it is connected to. These cells pass the address to the copies of themselves so the new graph will have the same interconectivity as the original network. See figure <copying>.

Fig. 37. Copying

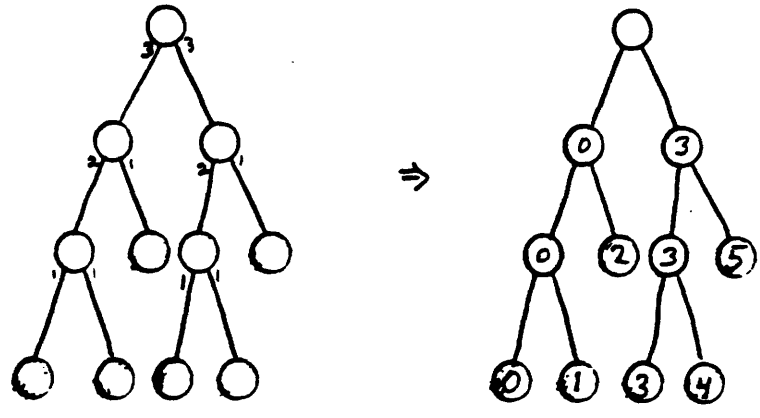


```
Step 1: mark the tree (or network)

Step 2: Each cell that is coping Conses a copy of itself.

Step 3: Send address of new cell to all cell you are connected to.

Step 4: Send received address to new cells and form new graph structure.
```

5.6 Enumeration

Enumeration is useful for establishing priority and calculating hashing functions. The algorithm presented this will enumerate the leaves of a tree in O(D) time where D is the depth of the tree. See figure <enumeration>.

Fig. 38. Enumeration



```
STEP1: Each branch counts the leaves to below in on its left side and
its right side.  These numbers are left-children and right-children.

STEP2: The root of the tree sends the number 0 to its left side and
the number left-children to its right side.  Semantically this means
the left subtree numbers its leaves from 0 to left-children - 1 and the
right subtree numbers its leaves from left-children to left-children +
right-children.  Each branch receives a number N from Parent.  The
branch sends N to its left side and N + left-children to its right side.
The number that the leaf receives will be its enumeration.
```

5.7 Balance

It is often easy to build unbalanced trees, but most algorithms work much faster if trees are balanced. The algorithm presented in this section uses the binary tree projected on the boolean N-cube as a template for the balanced tree. The leaves of the tree that is to be balanced calculate where they fit into the tree and send their address to the appro · ate branch which will be the new Parent.

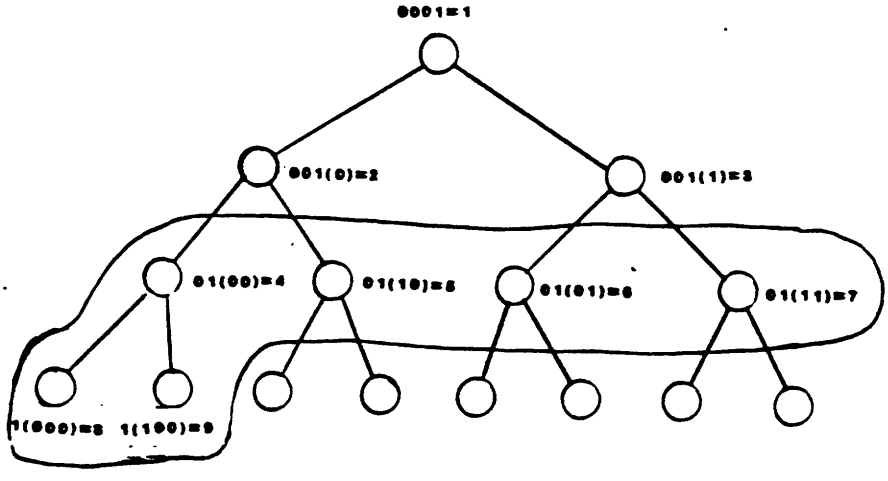
The first step is to enumerate the N leaves of the tree 0 to N-1. The root also broadcasts the total number of leaves to each leaf. Assume that cells 1 through N-1 are to be used for the template of the tree. We will use a slight modification of the algorithm for projecting a binary tree onto an N-cube to calculate parent of each cell. Here is the algorithm repeated:

```
1xxx => 01xx => 001x => 0001
left is most significant
```

To make the this algorithm work for a linear sequence of address reverse the low order M-1 bits on the Mth level of the tree. See figure <Balanced tree>. The bits to be reversed are in parenthesis. Each leaf calculates the address of its new Parent by reversing the low order M-1 bits (depending on its level) and applying the algorithm for calculating its Parent given above. Call this number New-Parent

---

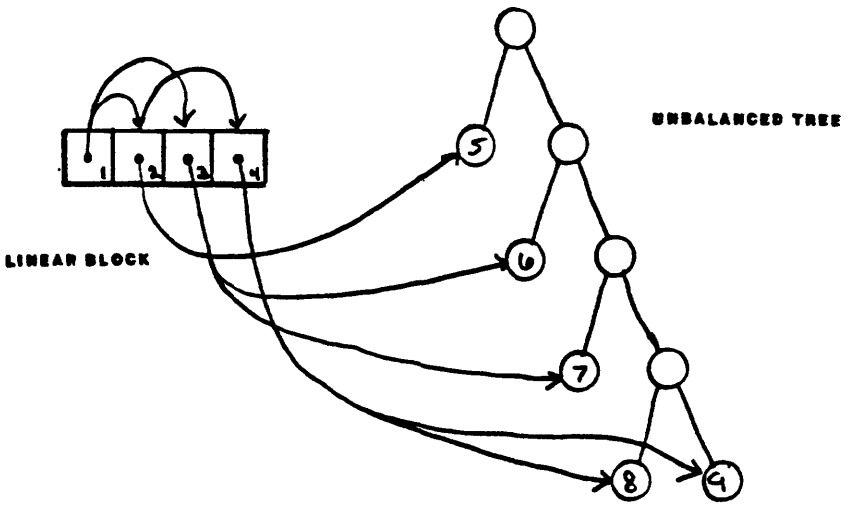
Fig. 39. Tree Balancing



The root of the tree will cons N-1 new cells which will be the branches of the balanced tree. The new cells are in a linear block of addresses from Q to Q+N-1. The new cells calculate their parent within this linear block using the algorithm above. The address Q is broadcast to all leaves of the tree. The actual address of the new parent will be Q + New-Parent.

Figure <Tree Balance Example> shows an example of balancing a tree with 5 leaves. The first step is to enumerate the leaves of the tree from 5 to 9 (N + 0 to 4). The root of the tree conses 4 new cells in a linear block that will be used as fan cells. The first cell is at address Q. This address is broadcast to the unbalanced tree so that each cell can calculate the address of its parent. Each cell in the linear block also calculates its parent.

Fig. 40. Tree Balance Example



---

6. Application: GA1 on the connection machine

GA1[1] is an expert system which infers the structure of DNA molecules from data about their segmentation by enzymes. Genetisists use GA1 implemented on a serial computer. Unfortunately, the practical scale of problems that can be solved by GA1 on a serial computer is limited by computational complexity (rather than memory limitations). GA1 explores a search space of possible solutions. This chapter examines the feasibility of implementing GA1 on the Connection Machine by exploring this search space in parallel.

Because genetisists want to find all solutions, GA1 uses an exhaustive generator to propose all possible structures which are then tested for correctness. This approach is sometimes called generate-and-test. Since the solution space is very large (ie $>>10^8$ for small problems) GA1 relies on early pruning to reduce the number of structures that are considered. The space is generated incrementally by filling in partial descriptions of the DNA structures. The generator defines the search space by incrementally building up partial descriptions. The partial descriptions form a tree where each successive level is a more complete description. The leaves of the tree are complete descriptions. Each partial description represents a class in the solution space, or, a branch in the generation tree where the leaves of the class are represented by the common branch. When a partial description is pruned, the entire class it represents is also pruned. The key point is that there is enough information so that partial descriptions can be eliminated with incomplete description. As the tree is generated level by level "pruning rules" are used to eliminate impossible branches of the tree therefore saving the cost of generating the pruned branch's offspring. The use of pruning rules drastically reduces the solutions space that needs to be searched.

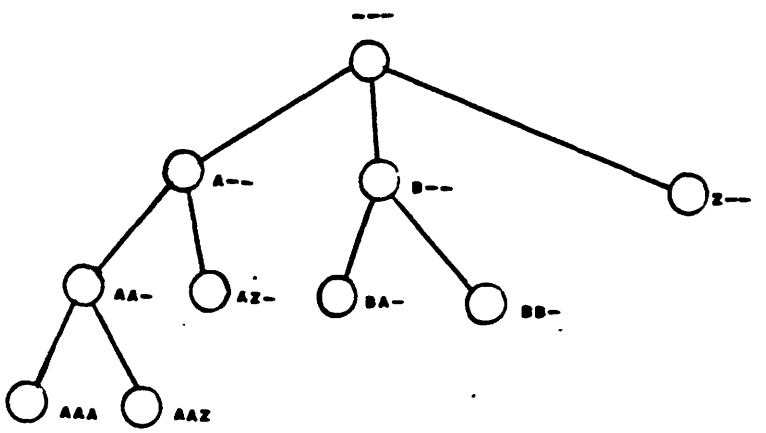
---

1. [Stefik]

6.1 Generate and Test Three Letter Words

For example, let the search space be the set of 3 letter "words". The generator builds up partial descriptions by placing letters one at a time into a template which has 3 slots for 3 letters. For each of the 3 slots there are 26 possibilities, one for each letter of the alphabet. The branching factor of the tree is 26 and the tree has 26 leaves. All of the leaves need not be generated though. If it is known that there are no 3 letter words where the first two letters are the same and not vowels then branches of t' tree that match "**_" can be pruned. This simple prune will save 26^2 leaves from being generated.

Fig. 41. Word Generation Example



The factored search space for complex problems is still too large for feasible computation. There are two major parts of the computation:

1) time required to generate new branches;

2) time required to run pruning rules on a generated partial description.

The time complexity of the serial version is proportional to the total number of nodes that are generated and evaluated. Generating and evaluating the tree in parallel would be more efficient. Pruning rules can

still be used when searching the tree in parallel to prune impossible partial descriptions so memory requirements for the parallel machine versus the serial machine would be within a constant factor. The time complexity of a parallel approach is proportional to the depth of the tree times the log of the branching factor assuming there are always enough processors available to store the partial descriptions of the tree. Since the search tree produced by the GA1 generator tends to be bushy (high branching factor) the parallel solution is theoretically faster.

The space complexity of the serial approach depends on the search strategy. If a breadth first search is used where the levels of the tree are generated one level at a time the space complexity of the serial approach for each level of the generated search space will be proportional the valid partial descriptions at that level. Since the parallel approach is essentially a parallel breadth first search the space complexity of the parallel approach at each level of the generated search space is also proportional to the number of valid partial descriptions at that level.

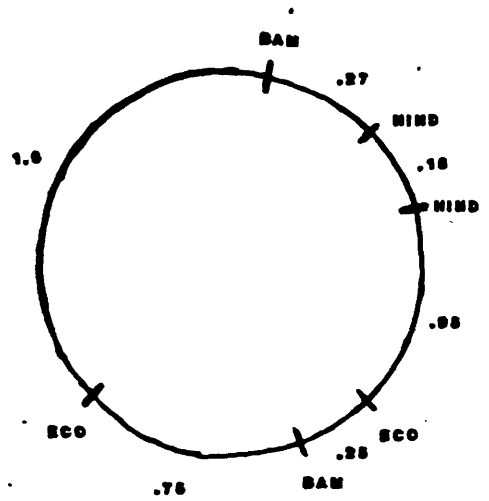
```
G = time to generate 1 new partial description
E = evaluation time
L = levels in the tree
N = branching factor
T = total cells generated after pruning

Serial: GET
Parallel: (EL)(GlogN)

For GA1:
T >> LlogN
G > E
L = 10 -> 30
N = 10 -> 50
T = 10**3 -> 10**8
```

6.2 Description of Segmentation Problems: segments and sites

The goal is to infer the structure of a circular DNA molecule from experimental data.

The structure of a DNA molecule is defined as an ordered set of enzyme recognition sites on the circular strand. Each solution is a sequence of segments separated by sites that is consistent with the experimental data. Segments are measured in arbitrary units. For example: figure <Circular DNA strand>. depicts a circular DNA molecule with 6 sites and 6 segments

Fig. 42. Circular DNA strand



Simple example of a circular DNA strand cut by three enzymes.

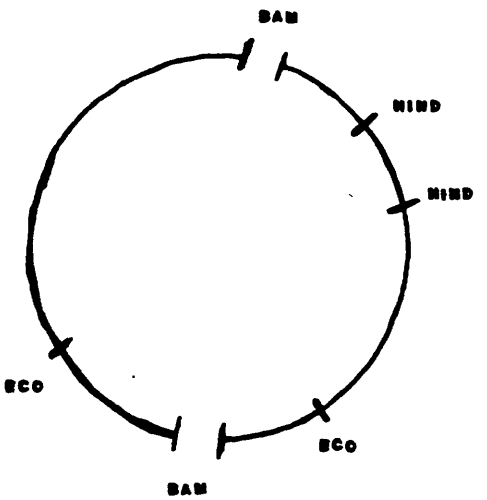
6.2.1 Experimental Data

An enzyme recognition site is a point where a particular enzyme cuts the circular DNA strand. The BAM enzyme would cut the ring into two pieces at the places labeled BAM. Using the enzyme BAM to cut the segment would result in 2 segments with size 2.35 and 1.65. Experiments are carried out using one or more enzymes to cut the strand at all of the recognition sites cut by those enzymes. The size of the resulting segments can then be measured. For the purposes of this discussion assume that the data is error free.

6.2.2 A Template for the Solution

A template is a data structure with slots for each site and segment of the physical structure. Once the template is defined the sites and segments for filling it in must be determined. The generator produces descriptions by placing these sites and segments into the template. Abstractly the problem can be viewed as a slotted table top and a set of blocks that fit into the slots.

Fig. 43. Bam Cuts

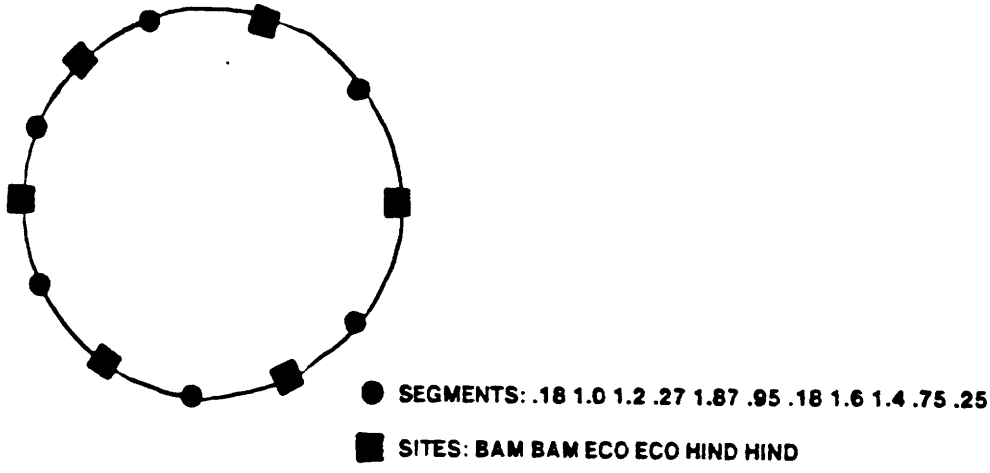


The first goal in calculating the template is to determine the number of sites and segments in the circular structure. This is done by counting the segments in all of the 1-enzyme digests. The number of segments resulting from 1-enzyme digests determine the number of that particular enzyme recognition site in the solution. For example: The 1-enzyme digest using Bam resulted in 2 segments; therefore we know there are 2 Bam recognition sites. The sum of the individual sites is the total number of sites. The number of segments is equal to the number of sites.

The next step is to find the set of segments that will be used to fill the template. The size of the segments between the sites can be determined from the 2-enzyme digests. All 2-enzyme digests will include all of the segments between 2 sites. There are (N(N-1))/2 2-enzyme digests for N enzymes. All segments between any two adjacent sites will be produced by one of these digests.

In the example problem 6 digests would be performed. The table below contains the segment sizes produced by using the indicated enzyme or enzymes.

Fig. 44. Blocks and Slots



```
1-enzyme digests:
Hind III:       3.82     .18
Bam:            2.35    1.65
Eco RI:            3.0     1.0

2-enzyme digests:
Hind III & Bam:         2.35 1.2  .27  .18
Hind III & Eco RI:      1.87 1.0  .95  .18
Bam & Eco RI:           1.6  1.4  .75  .25
```

The segments sizes for filling in the template are a subset of the segments in the 2-enzyme digests with some duplicates. In the example problem that set would be:

```
{.18 1.0 1.2 .27 1.87 .95 .18 1.6 1.4 .75 .25}
```

The goal is to take this information and induce the structure of the DNA molecule.

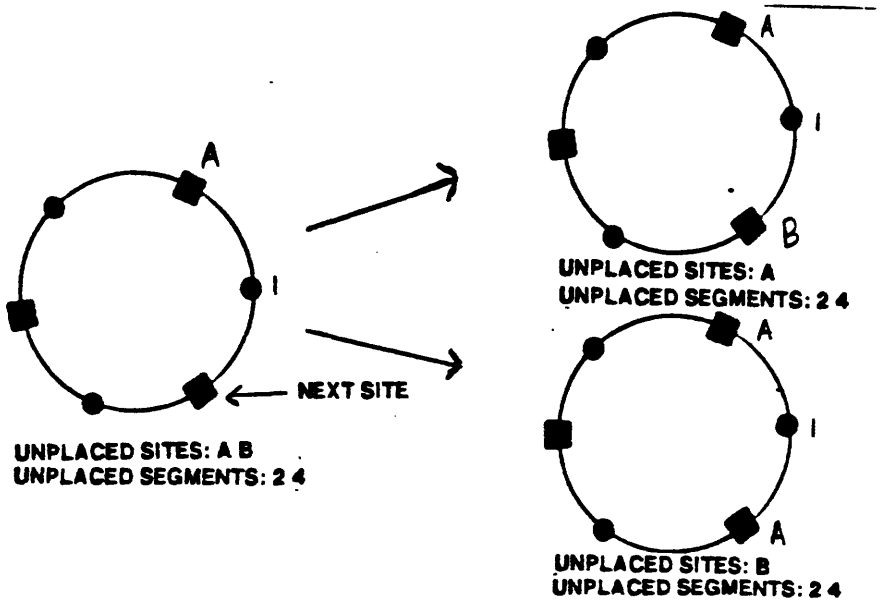
6.3 Generate and test

The strategy for finding solutions is the same for the parallel and serial approach: generating a search tree and pruning losers. The considerations for making the search fast vary considerably. This section is a discussion of the generate and test strategy independent of the target machine.

6.3.1 The Generator

The generator is a procedure that produces the offspring of a branch in the search tree. A branch of the search tree is a partial description of a final structure. A partial description is a template and a set of sites and segments, some placed in the template, some unplaced. For the GA1 problem the template is a sequence of alternating slots for sites and segments. At level N of the tree the generator generates the data structure for new offspring of each partial description at level N-1. The generator then copies the data of the parent to offspring and places one of the unplaced sites or segments (segment if level is even, site if level is odd) at each of the new partial descriptions. The branching factor at each level is the number of unplaced sites or segments. Figure <example generation> shows a branch (a partial description) of a template with 3 sites and 3 segments. One site and one segment are placed. The generation places an unplaced site. There are two unplaced sites so the branching factor will be 2.

Fig. 45. Example Generation



After the new partial descriptions have been constructed the pruning rules are applied to them, pruning inconsistent descriptions.

6.3.2 Pruning rules

Once partial descriptions are generated they are evaluated to determine if they are consistent with the experimental data. This section will discuss 2 pruning rules and used to eliminate inconsistent structure. For the complete set of pruning rules see appendix <pruning rules>.

```
Rule P10:  If a segment is about to be placed which would increase
the mass of the current structure to be greater than the expected
molecular weight and there are more sites to be placed, then this
branch of the generation may be pruned.
```

In the previous example: It is known that the total size of the molecule is 7. Segments 4, 3, and 2 are placed in the three segment slots the total size would be 9. This branch may be pruned because the summation of placed segments is larger that the known size of the molecule.

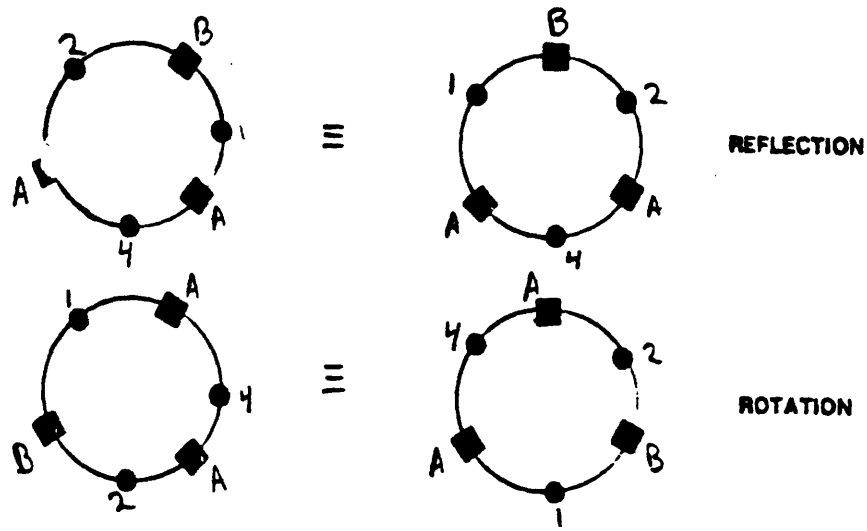
```
Definition P13: Allowable inter-site segments.  For recognition sites
E1 and E2, a segment is said to be allowable between E1 and E2 when
it appears in the appropriate digests.  Specifically, if E1 is distinct
from E2, the segment must appear in the 2-enzyme complete digest involving
E1 and E2.  Otherwise it must appear in the 2-enzyme complete digest for E1.

Rule P14: If a site E1 is about to be placed and there is another site E2
preceding it in the description (and there is no site equal to E1 or E2
between them) and the sum of the intermediate segments is not an allowable
segment for E1 and E2, then this branch of the generation may be pruned.
```

Using the data in example <example generation>: a site A is placed, then segment 1 is placed, then a site A is placed. The segment 1 does not appear in the 1-enzyme digest using enzyme A. This branch may be pruned because a segment of size 1 cannot be the only segment between two A sites.

6.3.3 Canonicalization rules

In the systematic generation of descriptions multiple partial descriptions are generated that represent the same physical structure corresponding to reflections and rotations. Generating these redundant descriptions is wasteful and unnecessary. Canonicalization rules prune reflection and rotated descriptions early in the generation. These rules are applied at the same time as pruning rules. See appendix <pruning rules> for a list of these rules for circular structures.

Fig. 46. Equivelent Structures



6.3.4 Generator Loop

The processes of exploring the search tree is by expanding one level of the tree and then pruning inconsistent partial descriptions. The search is complete when the template of each leaf of the tree is full (ie. a site is in every site slot and a segment is in every segment slot.)

```
Generator Loop:
1) alternately place site or segment in all active partial descriptions
until template is full. (place a segment first)

2) apply pruning rules to illuminate inconsistent partial descriptions
```

6.4 Implementation Considerations for Serial and Parallel Search

The parallel and serial approaches differ in their use of time and memory. The running time of the serial approach is bounded by the number of nodes that have to be generated (and therefore evaluated) before pruning. The number of potential final descriptions (leaves of the search tree) is:

```
N = total segments to be placed
M = segments slots in template
E = number of enzyme slots in template
Ti = for each enzyme i, Ti is the number of sites of that type

(N!/(N-M)!)(E!/PI(Ti!))(1/(M+E)

For the example problem with 6 sites and 6 segments this number is:
N = 11
M = 6
E = 6
T1 = 2 ;eco
T2 = 2 ;bam
T3 = 2 ;hind III

(11!/5!)(6!/2!2!2!)(1/12) = 2 494 800
```

The number of branches of the search tree is proportional to the number of leaves. Most branches are pruned early in the search so only a small fraction of the search tree is ever generated.

The running time of the serial search is limited by the number of partial descriptions generated; memory is not a primary consideration. Therefore being able to prune the tree as early as possible is the primary consideration for the serial approach. This implies that the pruning rules should be as effective as possible at weeding out losers early.

On CM, the tree is generated in parallel. Pruning rules are necessary because there is not enough storage to store the whole tree. The primary cost on CM is proportional to the communication costs of generating a new levels of the search tree which is proportional to the branching factor. Because of this limitation the storage space for representing a partial result should be as small as possible.

6.5 GA1 on the connection machine

The goal of the parallel implementation is to search the tree in parallel. Each partial description is stored at an individual CM cell. Generation, pruning, and placement can be done in parallel. The key factors in this application are the simple processors of CM and high communication costs. The SIMD processors require the template to be set up in such a way that a pruning rule be executed in parallel at each cell in the machine. A similar constraint applies to segment and site placement. The most important consideration is being able to generate new partial description efficiently because of the high communication costs. The amount of storage required for each partial description should be as small as possible to reduce the time needed to copy that data to new partial descriptions.

6.5.1 Template Structure

The partial description require a site-stack and a segment-stack to place new sites and segments. Unplaced-site-stack and unplaced-segment-stack are used to store unplaced sites and segments. When placing a site, a site is taken out of the unplaced-site-stack and placed on top of the site-stack.

6.5.2 Generator on CM

Given a partial description with N sites or segments to place will generate N new partial descriptions each with a different site or segment placed. This is accomplished by finding N free cells and enumerating them 0 to N-1 (call it I), copying the data to all the new cells, and then placing one of sites or segments as function of I. The limiting step is copying the data to the N new free cells.

Initially a segment is placed in a single partial description, the root of the tree. The generator is then applied to all partial descriptions alternately placing a site or a segment.

```
Generator:
1) first place segment in the root

LOOP UNTIL TEMPLATE IS FULL:
Generate new partial descriptions from all unpruned
partial descriptions in the last level. Branching factor
will be the number of unplaced sites or segments (depending
on which is being placed).  Call the branching factor N.
Each child is enumerated I from 1 to N.
2) alternately place site or segment
3) if site:
3.1) push the Ith unplaced site on the placed site stack
3) else if segment
3.2) push the Ith unplaced segment on the placed segment stack
```

A possible variation for site and segment placement: Instead of storing all unplaced sites and segments, this information can be broadcast to all partial descriptions. They would have to calculate which elements had not been placed. Selection of an unplaced element would still be a function of I. This method has the advantage of decreasing the amount of data that needs to be copied but increases the amount of processing and broadcasting that needs to be done. The feasibility of this approach depends on the communication speed and broadcast bandwidth of the machine.

6.5.3 Pruning rules

Pruning rules are executed in parallel on every partial structure. The rules are in the form of an instruction stream from the CC. Partial descriptions that are inconsistent with experimental data are marked for pruning (ie. they are forgotten and become free storage.)

After site placement: a site of type X has been placed a) all 1 and 2-digest segment sizes are sent out. For each digest XY, for each enzyme Y, the summation of segm...s since the last Y site was placed must be in the data for that digest unless the summation to the last Y site is greater than the distance to the last site of type X. Otherwise this branch may be pruned.

After segment placement: a) CC sends out total size. If summation of segments so far is greater than that size this branch may be pruned.

6.6 Generating New Levels of the Search Tree

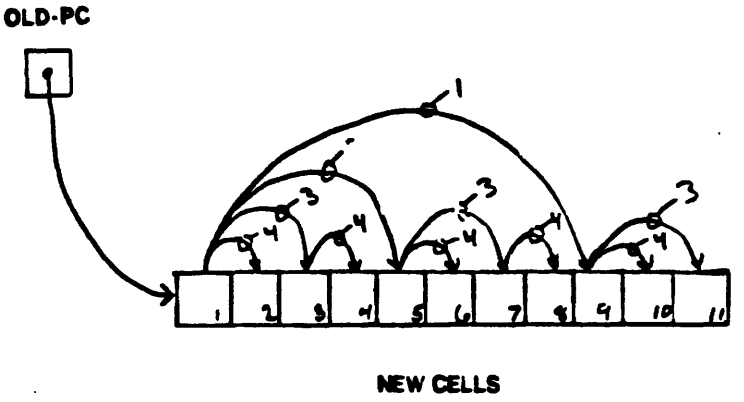
The speed of the search at each level of the tree is limited by the speed at which N new cells can be found and data copied to them where N is the branching factor at that level of the tree. The amount of data used to represent a partial state should be kept as small as possible to limit the amount of data that must be copied. Running the pruning rules and placing sites or segments are relatively fast compared to generating levels of the tree.

Problem Statement: To generate a new level of the tree each partial description at the fringe must find N free cells and copy its state to them. The new free cells must be uniquely enumerated 1 to N.

This can be done by using the free list consing algorithm to cons N new cells for each partial description at the fringe of the tree. The cells are enumerated by projecting a tree onto the linear structure in O(LogN) time. Data is copied from the old partial description into the first (number 1) new cell. The data is copied using the same projected tree.

Figure <Expanding a PD> shows how a partial description (call it the old-PD) would expand into 11 new cells. 11 new cells in a linear array are consed. The address of the first cell is known by the old-PD. The arcs show the tree that is imposed on the linear array. Each arc spans a distance of a power of 2 in the linear array. On the first iteration cell-1 sends a message to cell-9. The address is calculated by adding 8

Fig. 47. Expanding a PD



to the address of cell-1. The message contains the enumeration of the sender and the total number of new cells in that linear array. On each successive iteration cell-9 will also send a message to enumerate other cells. This procedure is repeated until all cells are enumerated. Note that a cell does not send a message to a cell that beyond its linear array. Four iterations are required to enumerate all cells. Copying data from old-PD to each new cell uses the same arcs.

6.7 Conclusions

Parallel Exploration of a search space is a good application for the Connection Machine. The implementation of GA1 described in this chapter utilizes the Connection Machines ability to allocate cells in parallel and test partial description in parallel. The potential gain in speed is proportional to the number of partial descriptions being considered in parallel.

7. Application: Combinators

This chapter examines a parallel implementation of Combinator Graph reduction (outlined in [Turner79]) for the connection machine. The first section of this chapter reviews evaluation of combinatory logic. The second section describes how combinator expressions can be represented as a graph. An implementation of a parallel graph reducing interpreter for combinator expressions on the connection machine will be discussed in the third section. The goal of this chapter is to show how a graph representing a computation can be reduced in parallel on the Connection Machine. The system for graph reduction outlined in this chapter is similar to the algebraic reduction program described in that chapter "Concepts".

7.1 Introduction to SKI Combinators

This section defines the translation of LISP lambda expressions to combinator expressions that contains no bound variables. Evaluation of combinator expressions is the same as LISP: The first element in an expression is a function which is applied to the rest of the elements. The value of the expression is the result of the functions. All functions return a value. Combinator expressions have the following properties:

```
All functions in combinator expressions take only one argument.
The translation of functions of multiple arguments to a function that
only takes 1 argument will be described below.

Three new functions S, K, and I will be introduced that are used
in combinator expressions.
```

Higher Order Functions

A higher order function takes a function as an argument and returns another function. A function of several arguments can be reduced to a higher order function that takes one argument. Consider the expression:

```
(+ 2 3)
```

This expression would be translated into the expression:

```
((plus 2) 3)
```

The expression (plus 2) returns a function that adds two to its argument. When this function is applied to 3 the result is 5. All functions in combinator expressions will take only one argument. Consider another example:

```
(if false 6 7) -> (((if false) 6) 7)  which evaluates to: 7
```

The interpretation of this is that (if true) returns a function that takes one argument. That function is applied to 6. The result is a function that takes one argument. That function is applied to 7. The result is 7.

Translating Lisp expressions: Removing Free Variables

Combinator expression use 3 new functions S, K, and I (known as combinators) defined below:

```
(((S f) g) x) => ((f x)(g x))
((K x) y)     => x
(I x)         => x
```

The translation of lambda expressions to expressions without variables is defined below:

```
Goal: Remove the variable x from (lambda (x) <expression>)
Notation: [x]E means remove the variable x from expression E.

[x](E1 E2) => ((S [x]E1) [x]E2)
[x]x       => I
[x]y       => (K y)

Where y is a constant or a variable other than x.
```

More than one variable can be removed from a expression by applying removing variables one at a time from the expression.

```
Removing more than one variable:
Goal: Remove the variable x and y from (lambda (x y) <expression>)

[x]([y]E)
```

An example translation is given below:

```
Example 1:
(defun plus1 (x) (plus 1 x))

[x](plus 1 x)
((S ([x](plus 1))) [x]x)
((S ((S (K plus))(K 1))) I)


The atom fact is bound to this expression.
```

An example evaluation of ((lambda (x) (plus 1 x)) 3) is given below. Only the left most reduction is

performed on each line.

```
(((S ((S (K plus))(K 1))) I) 3)
((((S (K plus))(K 1)) 3)(I 3))
(((K plus)(I 3))((K 1)(I 3)))
((plus (I 3))((K 1)(I 3)))
((plus 3)((K 1)(I 3)))
((plus 3) 1)
4
```

7.2 Representing Expressions as Graphs

Combinator expressions can be represented as graphs. The application of a function to an argument is represented by as a Application Cell. The car of the application cell is the function; the cdr is the argument. Figure <ski reduction> shows the graphical interpretation of S, K, and I and their associated reductions. Figure <reduction example> shows an example reduction of ((lambda (x) (plus 1 x)) 3).

The representation of the graph on the connection machine is strait forward. Arcs in the graph are represented as connections. Applications cells have two parts:

1) The function

2) the operand

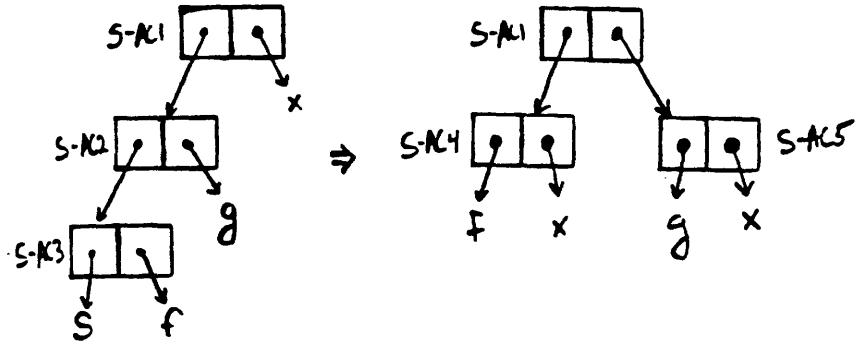
If more than one application cell points to something (either another application cell or an atom) a fan tree is used to hold the multiple connections. Figure <S reduction> in the next section shows an example of a fan tree holding multiple connections.

7.3 Parallel Reductions on the Connection Machine

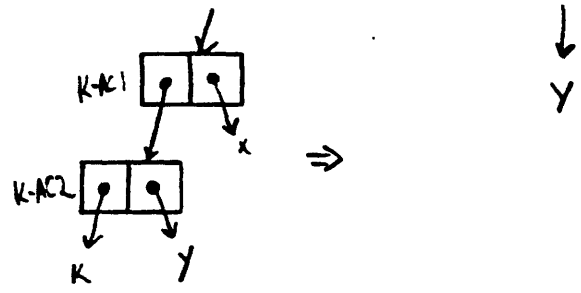
This section will describe how the functions S, K, and I are reduced in parallel. The method for reducing other functions such as *plus* and *if* will also be discussed.

At any time there may be several reductions that can be done. The order of reduction doesn't make any difference because the combinator expression has no side effects. In fact, all possible reductions at a given time could be done concurrently.

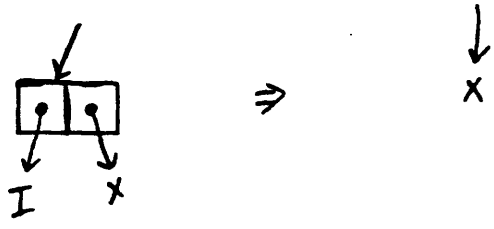
Fig. 48. SKI reductions



S Reduction: (((S f) g) x) => ((f x)(g x))



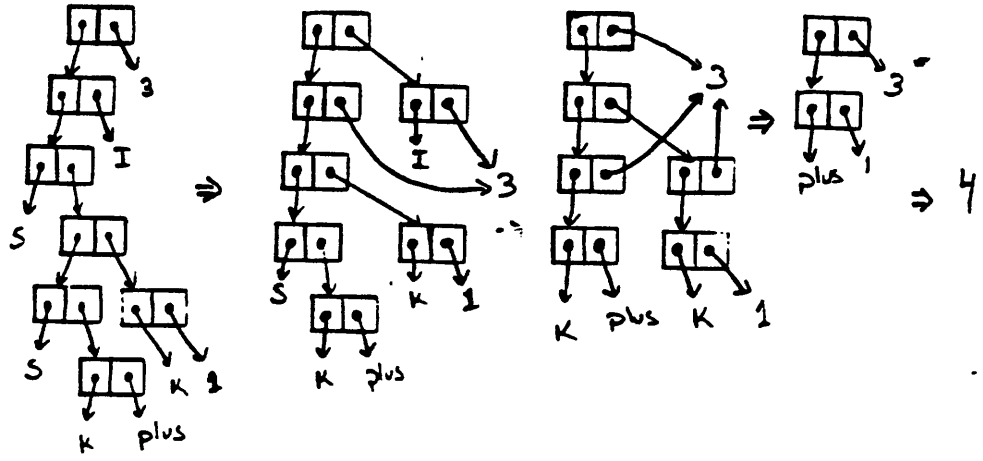
K Reduction: ((K y) x) => y



I Reduction: (I x) => x

Evaluation will be done by performing a series of reduction cycles. All possible reductions at the beginning of the reduction cycle will be done during the reduction cycle. After a reduction cycle new reductions will be possible. Reduction cycles are performed until there are no possible reductions left. At this point the evaluation is done.

Fig. 49. Reduction Example: ((lambda (x) (plus 1 x)) 3)



The first step in any reduction is to find all possible reductions. This can be easily done by local inspection of the graph in parallel. In the following discussion each application cell will know which part of a reduction it is part of. The term "Application Cell" will be abbreviated to AC for brevity.

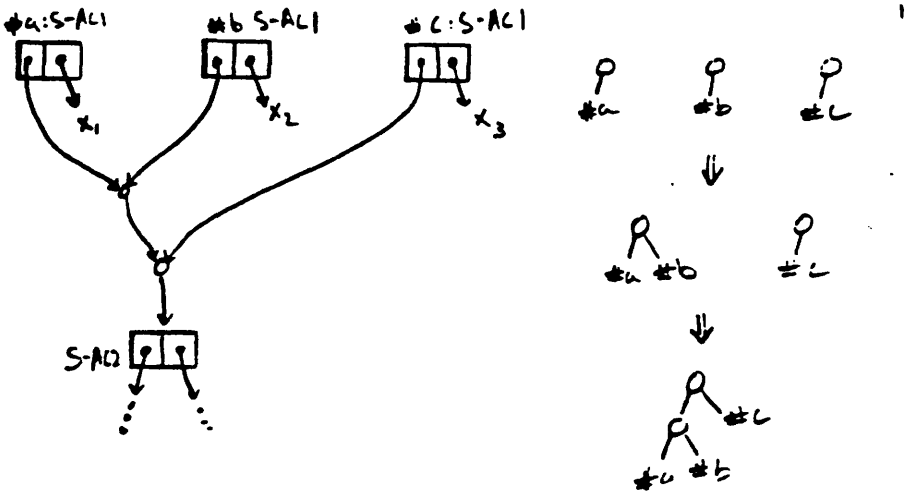
S Reduction

The S graph reduction is shown in figure <SKI reduction>. An S reduction is composed of 7 graph nodes: AC1, AC2, AC3, S, f, g, and x. For each S-AC1 cell the following steps are taken:

```
Step 1) Create two new S-AC cells: S-AC4 and S-AC5
Step 2) Add a connection from S-AC4 to f
Step 3) Add a connection from S-AC5 to g
Step 4) Add a connection from S-AC4 to x and from S-AC5 to x
```

An S reduction is the only reduction that produces new graph structure. Two application cell will be needed. Each S-AC1 cell create two new application cells which are called S-AC4 and S-AC5. There will be one AC4 and one AC5 for every S-AC1. The new ACs are created by consing which is described in chapter "N-cube algorithms".

Fig. 50. Adding Connections in Parallel



Adding the connections is more difficult. Notice that several S-AC1 cells can point to any single S-AC2 cell. There may also be several S-AC2 cells for each single S-AC3 cell. There may be several connections to add to a single cell. Consider step 3: A connection from each AC5 to g must be added. This operation can be done in parallel by collecting pointers to S-AC5 in the fan tree that connects S-AC1 cells to S-AC2 cells. This tree of connections can then be collected in the fan tree from S-AC2 to g. Figure <Collecting connections in parallel> shows this process. The final tree of pointers can then be added to g. Adding pointers from S-AC4 to f and S-AC4 and S-AC5 to x are handled in the same way. See figure <collecting pointers>. Algorithms for collection and adding pointers to trees are given in chapter <tree algorithms>. Adding each connections in Step 2 and Step 4 is handled in the same way as Step 3.

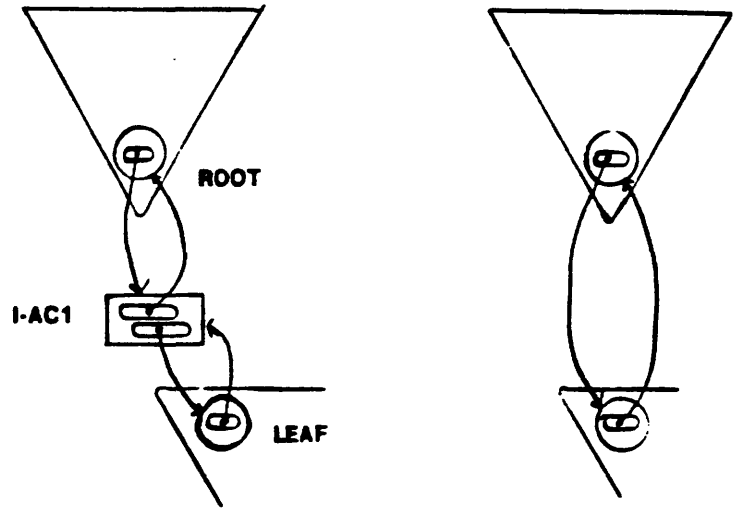
I Reduction

Reducing a I expression can be done easily by replacing the entire expression by one of its parts (x in this case). Assume that connections between cells and atoms always go through a fan-in tree. This

simplifies a problem when the I-AC1 of one reduction is the x of another reduction.[1] All AC cells that point to the I expression are held in a fan-tree that points to the I expression (the I-AC1 cell). Call this tree the fan-in-tree-I-AC1. The I-AC1 cell is connected to a fan-tree that holds the connections to x. Call this tree the fan-in-tree-x. The I reduction is done by connecting the root of the fan-in-tree-I-AC1 to a leaf of the fan-in-tree-x.

```
gorithm:

Step 1: Each I-AC1 cell sends the address of the fan-in-tree-x to the
        root of the fan-in-tree-I-AC1.  The root of the fan-in-tree-I-AC1
        stores the address of the fan-in-tree-x in the connection
        that used to point to I-AC1.  This is half of the new connection.

Step 2: Each I-AC1 cell sends the address of the fan-in-tree-I-AC1 to
        the leaf of the fan-in-tree-x.  The leaf of the fan-in-tree-x
        stores the address of the fan-in-tree-I-AC1 in the connection
        that used to point to I-AC1.  This completes the connection
        between the root of the fan-in-tree-I-AC1 and the leaf of
        the fan-in-tree-x.

Step 3: Each I-AC1 cell deletes its connection to its function and
        marks itself as garbage to be reclaimed.
```

Fig. 51. I reduction trees



1. This problem is analogous to the synchronization problem of the algebraic reduction program described in chapter "Concepts".

K Reduction

K reduction similar to I reduction. The reducing K expression is replaced by one of its parts. y in the case of a K expression.

Other Reduction

All other reductions replace the reducing expression with a function of their arguments. For example: ((plus 2 would replace itself with 5. Most expressions require that all arguments be reduced before that expression can be reduced. It would be difficult to reduce ((plus 2) <expression>) since plus is only defined for numerical arguments. Reducing these expressions is very similar to I or K reduction except that the arguments must be reduced.

One interesting exception is IF. IF only requires the predicate to be reduced before reducing itself.

```
(((IF predicate) then-expression) else-expression):

(((IF true) then-expression) else-expression) -> then-expression
(((IF false) then-expression) else-expression) -> else-expression
```

Once the predicate has been evaluated the expression can be reduced to either the then-expression or the else-expression depending on the value of the predicate. The other expression is thrown away.

7.4 Garbage Collection

During the course of evaluation many cells and connections will be created and thrown away. It is possible to throw away entire expressions that will continue to evaluate because they don't know they have been thrown away. Some of these expression could be infinitely recursive and will never terminate. Consider the example of factorial.

```
(defun fact (x) (if (= x 0) 1 (* x (fact (minus n 1)))))
```

When factorial is called on 0 the both branches of the IF expression are evaluated in parallel. The clause that will eventually be thrown away will be:

```
((* 0)(fact -1))
```

which is infinitely recursive. Garbage collection is needed to recover parts of the graph that have been thrown away.

There are several different methods that could be used for garbage collection.

```
1) Wait until the machine is full.  At this point mark all cells
that can be reached from the root of the expression.  Delete
all other connections.  Deletion of multiple pointers from
fan trees is described in chapter <tree algorithms>.  All good cells
are saved and all other cells are marked as free cells.

2) The connections in the graph are a built in reference counting
mechanism.  After each reduction cycle find all cells that are
not being pointed to.  Mark them as garbage and delete all
their connections.  Continue this process until all structure
that is garbage is collected.  The problem with scheme is that
a GC is required after every reduction cycle.

3) It is possible to collect garbage incrementally by modifying
method 2.  Instead of finding all garbage by tracing deleted pointers
on every reduction cycle
only trace garbage a fixed distance.  This scheme does not guarantee that
all garbage will be collected because the graph can grow exponentially
in depth, although if this is the case the machine will be filled
rapidly anyway so it is probably not a practical problem.
```

It is not clear which garbage collection scheme is the best in general. This will probably be determined empirically.

7.5 Conclusions

The point of this chapter was to show that the Connection Machine can be used as an interpreter that concurently evaluates expressions represented as a software graph of cells. It is not clear that the evaluation of SKI combinators or any conventional language (ex. LISP) represented as a graph is a good application for the Connection Machine; although, the idea of parallel graph reduction may be useful in some other context.

8. Application: Relational Data Base

This chapter discusses the implementation of a relational data base on the connection machine. The primary goal of this chapter is to show how global operations using the topology of the communication network can be used to implement a moderately complex system. An implementation of a relational data base is discussed using sort, cartesian product, and enumeration. These operations are described in chapter N-cube algorithms. A brief introduction to relational data bases is given, followed by a representation scheme on the connection machine. Algorithms for the operations Union, Intersection, Difference, Cartesian Product, Select, Projection, and Join are given for the representation scheme on the connection machine.

The definition of relational data base and the definition of the operations are taken from [Codd7?].

8.1 Definition of a Relational Data Base

Given a collection of sets $D_1, D_2, ..., D_n$ (not necessarily distinct), R is a *relation* on these *n* sets if it is a set of ordered n-tuples $\langle d_1, d_2, ..., d_n \rangle$ such that d_1 belongs to D_1, d_2 belongs to D_2, ..., d_n belongs to D_n. Sets $D_1, D_2, ..., D_n$ are the *domains* of R. The value n is the *degree* of R.

The table below illustrates a relation called PART, of degree 5, defined on domains P# (part number), PNAME (PART NAME), COLOR (part color), weight (part weight), and CITY (location where the part is stored). The domain COLOR, for example, is the set of all valid colors; note that there may be colors included in this domain that do not actually appear in the PART relation at this particular time.

Relation: PART

Fields: P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Yellow	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

8.2 Representing a Relational Data Base on CM

This section describes a representation of a relational data base on the connection machine. This is not necessarily the best way to implement a relational data base on the connection machine, although it does have some interesting properties. The purpose of this representation is to illustrate how global operations using the topology of the communication network can be used to do interesting things.

Each element of a set (E_{ii}) is assigned a unique ID within that set. IDs are contiguous numbers. For example: if there are 302 elements in a set then those elements are assigned IDs from 1 to 302. The number of elements may be larger or smaller than the address space of the machine.

Tuples are represented as cells. For each field a cell representing a tuple stores an ID relative to that field. Tuples themselves do not need to know which bits IDs are stored in. That information is known globally. Tuples are dumb; they are manipulated by the instruction stream. Tuples also store a tag which defines which relation it is a member of. A tuple can only be a member of one relation.

Relations have 2 parts. The first part is is a set of tuples defined by the fact that the tuples know which relation they are in. The second part is global information that defines how to access data in a tuples. Each relation has a unique ID so that tuples can be appropriately tagged.

The example below shows how the relation PART cold be represented on the connection machine.

```
Relation: PART    ID: 259
P#    PNAME    COLOR    CITY
P1:1  Nut  :1  Red  :1  London:1
P2:2  Bolt :2  Blue :2  Paris :2
P3:3  Screw:3  Green:3  Rome  :3
P4:4  Cam  :4           Athens:4
P5:5  Cog  :5
P6:6

The part would look like this (tuples are horizontal):
A single cell would contain 1 tuple.

P#  PNAME  COLOR  WEIGHT  CITY RELATION
1   1      1      12      1    259
2   2      3      17      2    259
3   3      2      17      3    259
4   3      1      14      1    259
5   4      2      12      2    259
6   5      1      19      1    259
```

8.3 Operations on Relational Data Bases

This section defines several high-level operations on relations. A user would manipulate the relational data base on the connection machine using these operations. The next section shall discuss how these operation can be implemented.

For the operators union, intersection, and difference, the two relations must be of the same degree, and the jth field of each relation must be from the same domain.

Union

The union of two relations A and B is the set of all tuples t belonging to either A or B (or both).

Relation: A Field: NAME	Relation: B Field: NAME	Relation: A Union B Field: NAME
a	b	a
b	d	b
c	e	c
d		d
		e

Intersection

The intersection of two relations A and B is the set of all tuples t belonging to both A and B.

Relation: A Field: NAME	Relation: B Field: NAME	Relation: A Intersect B Field: NAME
a	b	b
b	d	d
c	e	
d		

Difference

The difference between two relations A and B (in that order) is the set of all tuples t belonging to A and not to B.

Relation: A Field: NAME	Relation: B Field: NAME	Relation: A Difference B Field: NAME
a	b	a
b	d	c
c	e	
d		

Cartesian Product

The cartesian product of two relations A and B is the set of all tuples *t* such that is the concatenation of a tuple *a* belonging to A and a tuple *b* belonging to B.

Relation: A Field: NAME	Relation: B Field: NAME	Relation: A Cartesian Product B Field: NAME1 NAME2
a	b	a b
b	d	a d
c	e	a e
d		b b
		b d
		b e
		c b
		c d
		c e
		d b
		d d
		d e

Selection

SELECT is an operator for constructing a "horizontal" subset of a relation-i.e., that subset of tuples within a relation for which a specified predicate is satisfied. The predicate is expressed as a boolean combination of terms, each term being a simple comparison that can be established as true or false for a given tuple by inspecting that tuple in isolation.

Relation: A Field: Part	Weight	Color
P10	33	Red
P11	21	Blue
P12	17	Red
P13	27	Red
P14	25	Yellow
P15	16	Blue

Relation: A Select(Weight>20 Color=Red or Blue) Field: Part	Weight	Color
P10	33	Red
P11	21	Blue
P13	27	Red

Projection

PROJECT is an operator for constructing a "vertical" subset of a relation-i.e., a subset obtained by selecting specified fields and eliminating others. (and also eliminating duplicate tuples within the attributes selected). The set of fields that are to be eliminated is called the projection-domain. See figure <projection example>.

```
Relation: A
Field: Part  Weight  Color
-------------------------
       P10   33      Red
       P11   21      Blue
       P12   17      Red
       P13   27      Red
       P14   25      Yellow
       P15   16      Blue
```

```
Relation: A Project(Color)
Field: Color
--------------------------
       Red
       Blue
       Yellow
```

Join

JOIN is an operator that combines two relations over a common set of fields. The common set of fields is called the join-domain. The result of joining relation A on field X with relation B on field Y is the set of all tuples *t* such that *t* is a concatenation of a tuple *a* belonging to A and tuple *b* belonging to B, where x=y. This is called Equi-Join because equality is used in the comparison of the join-domain. Other kinds of joins can be defined using other comparisons (ex. greater-than, less-than etc.). See figure <Equi-Join>.

```
Relation: A
Field: Part  Weight  Color
-------------------------
       P10   33      Red
       P11   21      Blue
       P12   17      Red
       P13   27      Red
       P14   25      Yellow
       P15   16      Blue
```

```
Relation: B
Field: Color   Concept
----------------------
       Red     Ferrari
       Blue    Sky
       Yellow  Submarine
```

```
Relation: A Join B (over the Color Field)
Field: Part  Weight  Color    Concept
-----------------------------------
       P10   33      Red      Ferrari
       P11   21      Blue     Sky
       P12   17      Red      Ferrari
       P13   27      Red      Ferrari
       P14   25      Yellow   Submarine
       P15   16      Blue     Sky
```

8.4 Implementation of Operations on the Connection Machine

This section describes an implementation of the relational data base operations described above on the connection machine. When describing each operation there are two cases of interest: 1) the domain of interest (this could be several fields) is larger than the address space of the machine; and 2) the domain of interest is smaller than the address space of the machine. The address space of the machine is the number of processors that can receive a message. The size of a domain is $2^{(\text{number of bits that define the domain})}$. If the size of the domain is smaller than the address space of the machine then each element of a relation can send a message to the cell with the address that is equal to the bits that define the domain. This is a very useful operation as we shall see.

8.4.1 Domain size is smaller than address space

This section assumes that the domain size is smaller than the address space of the machine. Tuples can send mail to the address that is specified by the domain of interest.

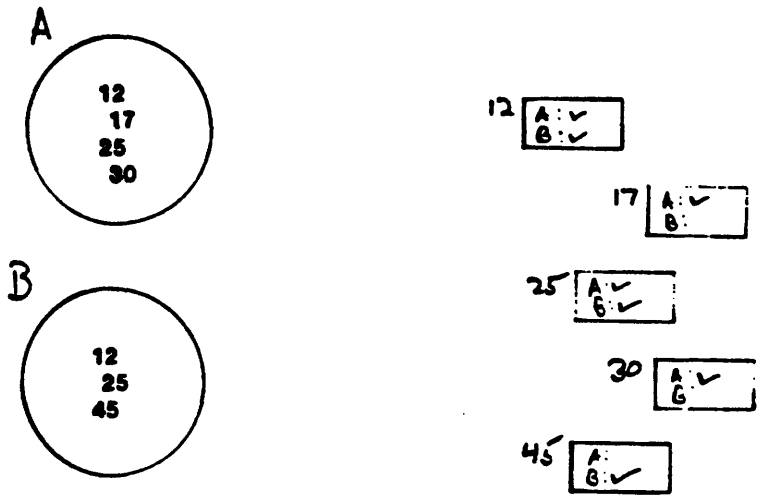
Union

```
A UNION B:
Step 1:
Every tuple in A sends a message (no content) to the processor
specified by the bits of the domain.

Step 2:
Every tuple in B sends a message (no content) to the processor
specified by the bits of the domain.

Step 3:
Any cell that receives a message during Step 1 or Step 2 create
a new tuple.  The value of the domain is the address of the cell.
```

Intersection

```
A INTERSECT B:

Step 1 and 2: Same as for UNION

Step 3:
Any cell that receives a message during Step 1 and Step 2 create
a new tuple.  The value of the domain is the address of the cell.

(optional example)
```

Fig. 52. Union Example (Address Arithmetic)



The union of set A and set B is done by using the entry as an address and sending a message to that address. If a cell in the machine recieves two messages then the entry that is that address is a member of the resulting set.

Difference

```
A DIFFERENCE B:

Step 1 and 2: Same as for UNION

Step 3:
Any cell that received a message during Step 1 and not during Step 2
creates a new tuple.  The value of the domain is the address of the cell.
```

8.4.2 Domain size is larger than the address space

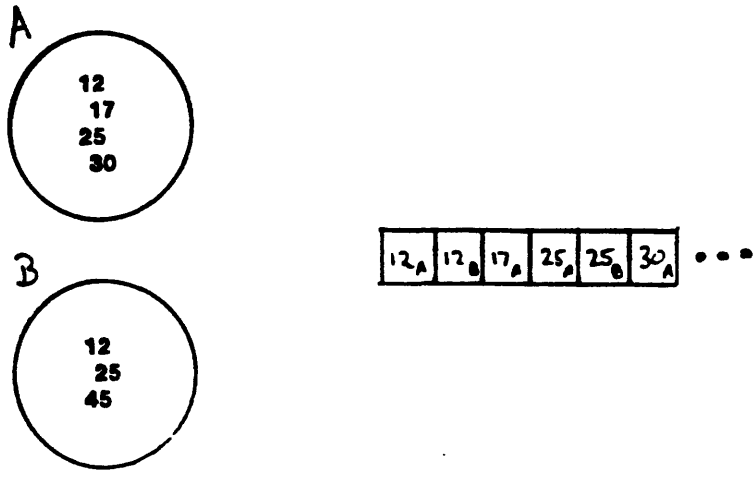
Now assume that the domain is too large to be used as the address of a cell in the machine. An alternative approach that uses sorting instead of hashing is presented below. These algorithms will work on relations with duplicate tuples. The resulting relations will not contain duplicates.

Union

```
A UNION B

Step 1:
Each tuple in A and B creates a datum that contains the domain as a
value.  The low order bit of this datum is 0 if the tuple is from set A
```

```
and 1 if the tuple is from set B.  The effect of this is that equal
an equal data from A will be next to an equal datum from B if one exists.

Step 2:
Sort these data into a linear ordered set of processors.

Step 3:
Each processor that has a datum looks at the datum stored in
the next processor.  If the datum stored at the next processor is equal
to the datum stored at this processor then mark this datum as a duplicate.
All processors that contain a datum from either A or B create a new
tuple whose domain is the value of the datum without the lowest order bit.
```

Fig. 53. UNION example (Sort)



In this case entries are too large to use as addresses. Sort the entries in set A and set B into another set. If there are two contiguous identical entries then that entry is a member of the resulting set.

Intersection

```
Step 1 and 2: Same as for UNION.

Step 3:
Each processor that has a datum from A looks at the datum stored in
the next processor.  If the datum stored at the next processor is
equal to the datum stored at this processor and is from B then
then create a new tuple whose domain is the value of the datum without
the lowest order bit.
```

Difference

```
Step 1 and 2: Same as for UNION.

Step 3:
Each processor that has a datum from A looks at the datum stored in
the next processor.  If the datum stored at the next processor is
not equal to the datum stored at this processor then a new tuple
is created whose domain is the value of the datum without the
lowest order bit.
```

8.4.3 Cartesian Product, Projection, Join

The size of the domain is not important for these operations because they do not use address hashing.

Cartesian Product

This algorithm is described in chapter N-cube algorithms.

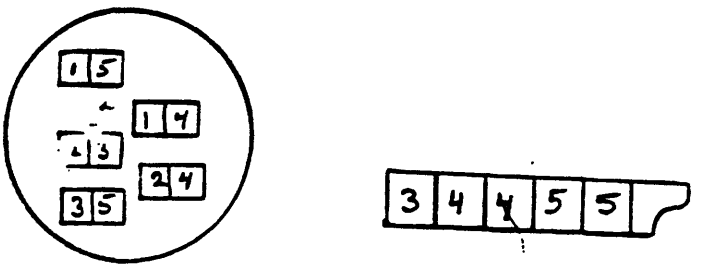
Projection

```
PROJECT A (over some set of fields called the projection-domain)

Step 1:
For each tuple in A create a datum that is the data of the
projection-domain.  Sort these data into a linear ordered set
of processors; one datum to one processor.
Sorting is described in chapter N-cube algorithms.

Step 2:
Once the data have been sorted each processor that contains a datum
sends a copy of the datum to the next processor in the linear
ordering.  If the datum received is equal to the datum stored then
mark the datum stored at this processor as a copy.

Step 3:
All processors that are storing data not marked as copies create a
new tuple whose domain is the value of the datum.  The result of the
projection is the set of all these tuples.
```

JOIN

A Join B (over the join-domain) can be done quite easily by forming the cartesian product of the two relations and doing the join comparison in parallel at every tuple of resulting relation. If the comparison is true then that tuple is a member of the resulting relation. Unfortunately this requires $|A| \cdot |B|$ processors.

Fig. 54. Projection Example

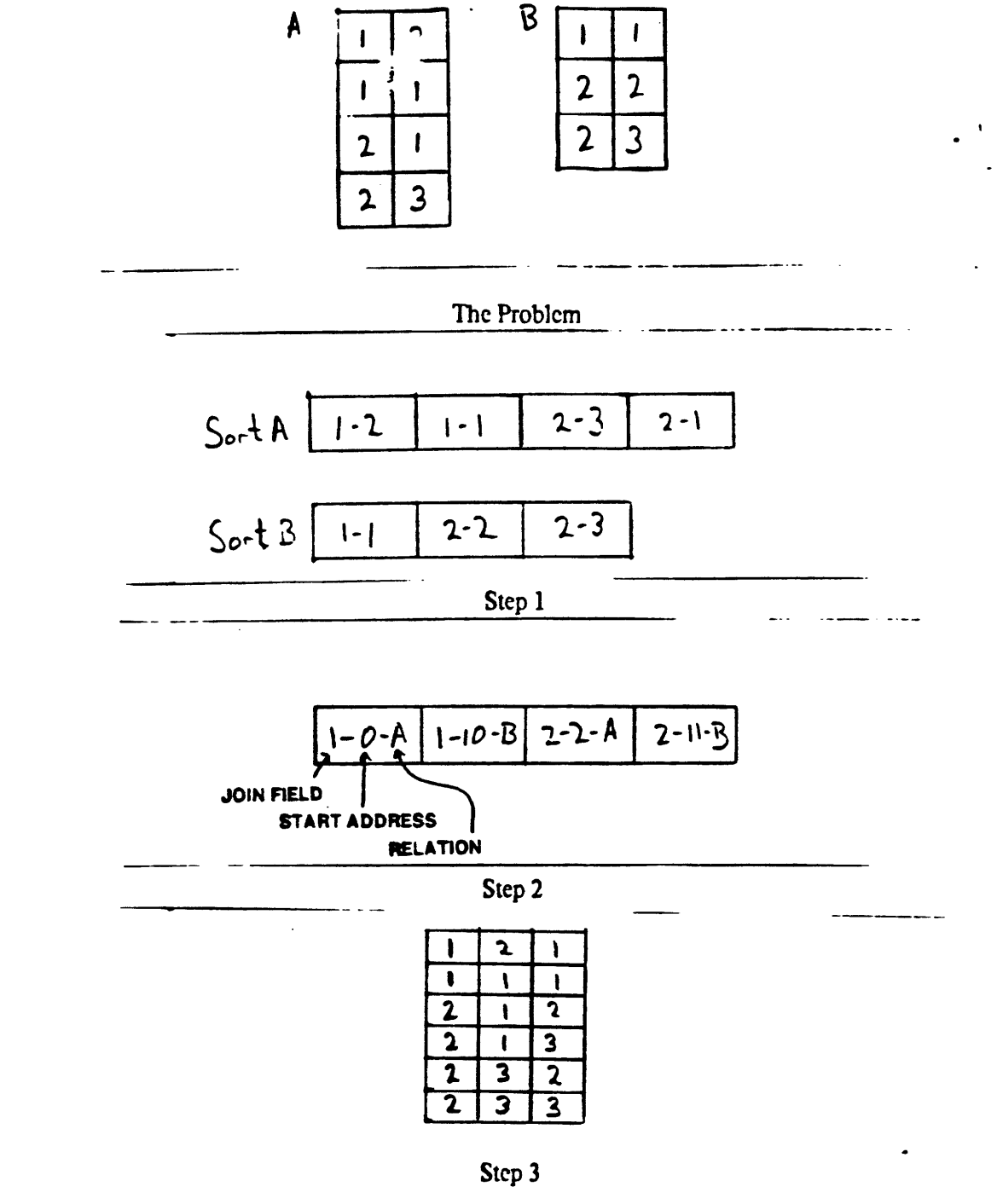


Equi-Join (comparison is equality) can be done more easily than by forming the cartesian product of both relations. The general idea is that tuples with equal join-domains are grouped together by sorting. Tuples with equal domains from relation A and B find each other and from a local cartesian product over the domain. The union of these cartesian products will be the result.

```
Step 1:
Each tuple in A and B forms a datum that contains its state.
Data from A is sorted into a set of linear contiguous processors.
Data from B is sorted into another set of linear contiguous processors.
Comparisons for sorting are just over the join-field.  The result is
that data with equal join fields are grouped together.  Call a set of
data with equal join-fields an "equal-set".

Step 2:
The object of this step is twofold: 1) To exchange the start address of
corresponding equal-sets from A and B and
2) find out how many cells are in each equal-set.  Each processor
with the first element of an equal set creates a datum that contains
its address, the join-field, and a bit that indicates if it is from set
A or B.  These data are sorted into a set of linear contiguous processors
using the join-field for comparisons.
The start address for the equal-set from A will be next to the start
address for the corresponding equal-set from B if it exists.  Also,
the number of elements in an equal set from one relation can be
determined by finding the next processor that contains a datum from
the same relation.  The return address allow this information to be
sent back to the first element of each equal set.

Step 3:
The cartesian product of corresponding equal-sets can now be done.
Corresponding equal-sets cons a block of linear contiguous processors.
This block will contain |equal-set from A|*|equal-set from B| processors.
```

All cartesian products can be done in parallel. The UNION of the resulting cartesian products will be the result of the EQUI-JOIN.

Fig. 55. Equi-Join



8.5 Conclusions

This chapter discused using the CM as a relational database processor. The use of relations is an alternate to semantic networks as a method of knowledge representation on the CM. Many of operations (sorting, enumeration, etc.) require that the entire machine be working because they depend on the topology of the routing network. A future goal of the CM project will be to make these algorithms fault tolerant.

applications use a combination of synchronization techiniques.

Several applications have been proposed for the Connection Machine using the graphical programming methodology. These applications include: Semantic Networks, Relational Data Bases, Constraint Networks, Graph Reduction Evaluation, and Data Flow Evaluation. The common property of all these applications is that each requires a large number of fairly simple computations and irregular communication patterns. The simple processors of the Connection Machine execute simple computations in parallel; the flexibility of the communication network allows irregular and dynamic communication patterns.

9. Conclusion

This thesis presents a programming methodology that exploits the highly parallel architecture of the Connection Machine. Using this methodology a computation is represented as a graph with a processor at each vertex. Two types of parallelism are exploited on the Connection Machine:

1) Each processor operates on its local memory in parallel.

1) Independantly addressed messages are delivered by the communication network in parallel.

The communication network allows parallel communication between connected vertices of an arbitrary graph represented on the Connection Machine as a data structure. The communication network is the feature that gives the Connection Machine its flexibility.

Three levels of abstraction for programming in the Connection Machine were introduced:

1) N-cube Level: Several low level operations quickly executed by taking advantage of the connection topology of the communication network.

2) Tree Level: Vertices are limited to 3 connections: a parent and two children.

3) Graph Level: Graph can have an arbitrary number of connections to other vertices in the graph.

Operations implemented at the N-cube and Tree level of abstraction are supplied as primitive operations for programming at the Graph level of abstraction.

Synchronization is the basic difficulty in parallel programming. Several methods of handling synchronization are used in the algorithms presented in this thesis. At the lowest level the single instruction stream of the Connection Machine allows direct control of synchronization. Enumeration by subcube induction is an example of an operation where it is important that all processors be synchronized tightly. Programming at this level is efficient but is very tedious. At a higher level of abstraction synchronization can be achieved by communication protocols between connected nodes. The Serialization algorithm uses this form of synchronization; when a datum is accepted an confirmation message is sent to the sender. It is not important that every processor be running in lock step. Most

10. Appendix 1: Algebraic reduction example in MP

This appendix describes an MP program for the algebraic reduction computation described in Concept Primer. This program gives the code for multiplication; addition would be similar.

```
VAR NODE-TYPE: {ROOT OPERATOR LEAF INACTIVE}
VAR OPERATOR-TYPE: {* +}
VAR LEAF-TYPE: {1 0 X}
VAR WAIT-FOR-RIGHT-CHILD: BOOLEAN
VAR WAIT-FOR-LEFT-CHILD: BOOLEAN
VAR REPLACE-LEFT-CHILD: BOOLEAN
VAR REPLACE-RIGHT-CHILD: BOOLEAN
VAR MESSAGE-TYPE: {TYPE, CHILD-REDUCING, REPLACE, UPDATE-PARENT}

;;;leaves send type to parent
(if (= node-type 'leaf)
    (send ('TYPE leaf-type) parent))

;;;operator nodes decide what to do
(if (= node-type 'operator)
    (progn
      ;;;if either branch is a zero
      (if (or (and (= left-child-mail true)
                   (= (get-msg left-child-mbx 2) 0))
              (and (= right-child-mail true)
                   (= (get-msg left-child-mbx 2) 0)))
          (progn
            (set node-type 'leaf)
            (set leaf-type 0)
            ;;;delete left and right child
            (set-up-send ('DELETE-POINTER) left-child)
            (set-up-send ('DELETE-POINTER) right-child)))
      ;;;if left and right are 1
      (if (and (and (= left-child-mail true)
                    (= (get-msg left-child-mbx 2) 1))
               (and (= right-child-mail true)
                    (= (get-msg right-child-mbx 2) 1)))
          (progn
            (set node-type 'leaf)
            (set leaf-type 1))
          ;;;else if only the left is a 1
          (if (and (= left-child-mail true)
                   (= (get-msg left-child-mbx 2) 1))
              (progn
               (set-up-send ('DELETE-POINTER) left-child)
               (set-up-send ('CHILD-REDUCING) parent)
               (set replace-with-left-child true))
              ;;;else if only the right branch is 1
              (if (and (= right-child-mail true)
                       (= (get-msg right-child-mail 2) 1))
                  (progn
                   (set-up-send ('DELETE-POINTER) right-child)
                   (set-up-send ('CHILD-REDUCING) parent)
                   (set replace-with-right-child true)))))
      ;;;reset mail
      (set left-child-mail false)
      (set right-child-mail false)
      ;;;if this branch is to be replaced notify parent
      (send-buffered-messages)))
;;;process the DELETE-POINTER message
```

```
(if (and (= parent-mail true)
         (= (get-msg parent-mbx 1) 'DELTE-POINTER))
    (PROGN
     (set node-type 'INACTIVE)
     (set parent-mail false)))
;;;Process the CHILD-REDUCING message
(if (and (= node-type 'operator)
         (= left-child-mail 1)
         (= (get-msg left-child-mbx 1) 'CHILD-REDUCING)
         (= REPLACE-WITH-LEFT-CHILD true))
    (progn
     (set WAIT-FOR-LEFT-CHILD true)
     (set left-child-mail false)))
(if (and (= node-type 'operator)
         (= right-child-mail 1)
         (= (get-msg right-child-mbx 1) 'CHILD-REDUCING,
         (= REPLACE-WITH-right-CHILD true))
    (progn
     (set WAIT-FOR-RIGHT-CHILD true)
     (set right-child-mail false)))
;;;STEP 2: intial step of reducing the tree
(if (and (= replace-with-right-child true)
         (= wait-for-right-child false))
     (set-up-send ('REPLACE Y) Z))
(if (and (= replace-with-left-child true)
         (= wait-for-left-child false))
     (set-up-send ('REPLACE X) Z))
(send-buffered-messages)
;;;loop
(while (and (= left-child-mail false)
            (= right-child-mail false)
            (= z-mail false))
       (dispatch-on-type
        left-child-mbx
        ('REPLACE
         (if (= wait-for-left-child true)
             (set-up-send ('UPDATE-PARENT SELF) left-child)
             (progn
              (set left-child (get-msg left-child-mbx 2))
              (set-up-send ('UPDATE-PARENT SELF) left-child)))))
       (dispatch-on-type
        right-child-mbx
        ('REPLACE
         (if (= wait-for-right-child true)
             (set-up-send ('UPDATE-PARENT SELF) right-child)
             (progn
              (set right-child (get-msg right-child-mbx 2))
              (set-up-send ('UPDATE-PARENT SELF) right-child)))))
       (dispatch-on-type
        parent-mbx
        ('UPDATE-PARENT
         (set parent (get-msg parent-mbx 2))))
       (send-buffered-messages))
```

11. Appendix 2: GA1 Pruning Rules

These rules are taken directly from [Stefik].

Canonical Form Rules: These rules prune reflected and rotated partial structures.

Rule F3. If circular structures are being generated, only the smallest segment in the list of initial segments should be used for the first segment.

Rule F4. If circular structures are being generated and the second segment is about to be placed and there are several segments to be placed and the segment is the largest of the remaining segments, then this branch of the generation can be pruned.

Rule F5. If circular structures are being generated and a segment equal to the first segment is about to be placed and the total mass is less than the molecular weight (so that at least one more segment will be placed) and all remaining segments are less than the second segment of the structure, then this branch of the generation may be pruned.

Rule F6. If circular structures are being generated and a segment equal to the first segment is about to be placed and the previous segment is less than the second segment, then this branch of the generation may be pruned.

Pruning rules: These rules prune partial structures that are not consistent with the experimental data.

Definition P3. Allowable sites for segments. Recognition sites are allowable for terminating a segment only if the segment appears in the 2-enzyme complete digests for the corresponding enzymes. (If there is only one enzyme in the experiment, then only its sites are allowable.)

Rule P4. If a segment is about to be placed and the previous site is not one of the allowable sites for this segment, then this branch of the generation may be pruned.

Rule P5. If a site is about to be placed and it is not an allowable site for the previous segment, then this branch of the generation may be pruned.

Definition P6. Required termination sites for segments. If only one enzyme was used in the experiment, then the site for that enzyme is required for every segment. If two enzymes were used, then for each segment which does not appear in a 1-enzyme digest, both enzyme sites are required. If three or more enzymes were used, then for each segment which appears in exactly one 2-enzyme complete digest, the sites for the enzymes involved in that digest are both required.

Rule P7. If a segment having required sites is about to be placed and the previous site is not one of them, then this branch of the generation may be pruned.

Rule P8. If a site is about to be placed and the previous segment has required sites and this site is not one of them, then this branch of the generation may be pruned.

Rule P9. If a site is about to be placed and the previous segment has two required sites and the previous site is one of the two required sites but this site is not the other one, then this branch may be pruned.

Rule P10. If a segment is about to be placed which would increase the mass of the current structure to be greater than the expected molecular weight and there are more sites to be placed, then this branch of the generation may be pruned.

Rule P11. If circular structures are being generated and the first segment is unique and appears in the 1-enzyme complete digest for enzyme E1, then a recognition site for E1 can be placed in front of the first segment.

Definition P13. Allowable inter-site segments. For recognition sites E1 and E2, a segment is said to be allowable between E1 and E2 when it appears in the appropriate digests. Specifically, if E1 is distinct from E2, the segment must appear in the 2-enzyme complete digest involving E1 and E2. Otherwise it must appear in the 1-enzyme complete digest for E1.

Rule P14. If a site E1 is about to be placed and there is another site E2 preceding it in the structure (and there is no site equal to E1 or E2 between them) and the sum of the intermediate segments in not an allowable segment for E1 and E2, then this branch of the generation may be

pruned.

12. Bibliograph

get references from thesis.bib or thesis.proposal which are offline now.