The Homogeneous Machine

Thesis by

Bart N. Locanthi

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1980

(Submitted January 21, 1980)

Acknowledgements

In the course of acknowledging those who helped me in formulating this thesis, I would be remiss in leaving out my mother and father who so profoundly influenced my early development and exercised my mind.

I also owe a great debt to Rod Serling, Joseph Stefano, Gene Roddenberry, and John Wayne, whose presence through the media kept me company as a child, and Isaac Asimov, Robert A. Heinlein, and Larry Niven, who captivated my imagination as an adult.

And lastly, I wish to extend my thanks to some of the people that made my stay at Caltech so rewarding:

to Sally Browning and Jim Rowson, with whom I shared many interesting projects, discussions, and experiences,

to Ivan E. Sutherland, who taught me how to sift through ideas,

to Carver A. Mead, who made me think and dared me to act,

and, most importantly, to my thesis advisor Chuck Seitz, whose endless patience with my slow progress and whose boundless enthusiasm for new ideas and adventures encouraged and guided me.

The research decribed in this thesis was sponsored by the Defense Advanced Research Projects Agency under contracts #N00123-78-C-0806 and #N00014-79-C-0597.

Abstract

The advance of semiconductor technology is bringing about rapid changes in the scale and performance of integrated systems, thus also in their economics and potential applications. The highly visible and readily quantified changes in measures such as the number of transistors are accompanied by more subtle but increasingly significant shifts in fundamental relationships affecting system design. Specifically, as transistors become smaller, faster, and lower power, the wires used to interconnect them are becoming slower. These shifts, along with the challenge of managing the complexity of designs with millions of switching elements, are forcing a new look at alternative computer architectures which use ensembles of computing elements under restricted and regular interconnection.

This thesis addresses the problem of orchestrating many computing elements in the performance of general-purpose computations. There are three major obstacles in the way of this goal. First, it must be possible to express programs in a notation that allows concurrency to be discovered and exploited. Second, it must be possible to map computations onto a physical structure for execution by multiple computing elements. Third, such computing elements must be provided rapid access to storage while at the same time avoiding contention.

This thesis presents a scheme which automatically detects and exploits concurrencies in computations expressed in an applicative subset of the LISP programming language. The mapping of numerical and symbolic computations onto array and tree structures is also investigated.

This thesis approaches the design of multiprocessor systems as a problem in bandwidth reduction. To this end, the concept of a multi-level cache is introduced. The discussion culminates with a description of a multi-level LISP system implemented on a tree of processors. This implementation provides each processor with a superset of the address space of its immediate ancestor. Memory allocation and garbage collection for this machine are described, and a simple example of its operation is given.

Table of Contents

List of Figures and Tables

Introduction

The changes brought about by the rapid advance of semiconductor technology are both quantitative and qualitative.

The quantitative changes are the highly visible and spectacular improvements in the economics of semiconductor devices, by now an expected, seemingly daily tradition. The result of these changes is of course that the same function will be made smaller, faster, cheaper, and lower power tomorrow than it is today.

As more complex functions are put on the "hit list" of LSI technology, people are wondering to what use the technology can ultimately be put. A favorite quip asks what IBM will sell for a million dollars once it becomes possible to put a System/370 on a $10 chip.

The increasing anxiety over just what to do with all those transistors is part of the qualitative changes brought about by advances in LSI technology. As people realize that building a million bit memory chip is really just a good way of not using 999,999 transistors at a time [Sutherland77], they have to sit back, fold their clammy hands, and wonder what else to do with the millions more transistors soon to find their way onto individual chips.

Unfortunately, increasing density and speed are not the only parts of the LSI story. Amidst all the wonderful news about density, the sad fact is that wires are getting slower. As the fundamental limits of semiconductor physics are approached, those tiny, fast devices have less and less drive, wires get thinner and flatter, and propagation times are more and more dominated by diffusion delays [Seitz79]. ("You mean signals won't go at the speed of ligh anymore?") This development puts an increasing importance to the concept of locality.

What, then, to do with all those transistors?

The concept of multiprocessing dates back to the early days of computers. With the realization that it may soon be practical to incorporate literally thousands of computing elements into one system, the subject of multiprocessing has been pursued with renewed vigor. Predictably, special-purpose multiprocessors have been proposed and occasionally built, mostly for signal processing or some simple forms of numerical analysis where brute parallelism can be put to good use.

This thesis takes a different emphasis, that of general purpose computation. The focus of this thesis is the orchestration of many computing elements in the execution of "useful" programs. Implicit in this goal is the hope that many computing elements can perform with many times the "power" of one computing element. Also implicit is the hope that "useful" programs are among those that many computing elements can perform with such power.

There are three major obstacles in the way of achieving this goal. First, it must be possible to express programs in a notation that allows concurrency to be discovered and exploited. Second, given a program that can be executed concurrently by several computing elements, these computing elements must be arranged so that the computation can be distributed effectively. Finally, given a programming notation and an arrangement of computing elements that distributes control of a computation effectively, storage resources and interprocessor communication must be managed so that computations by the individual computing elements can proceed relatively unimpared by the effects of their physical separation.

These three problems are interrelated. With the realization that computers are in fact physical systems subject to the constraints of physics, it is important to understand the physical implications of the use of a given programming notation. Similarly, given that a computation may be performed on physically separate computing elements, the concept of "global data" may be worthless, and this conclusion should be reflected in the programming notation.

In this thesis I attempt to treat the problems of notation, interconnect, and resource management individually. However, each discussion should be taken in the context of the other two, as the issues involved affect more than one area at a time. The reader is thus being asked to swallow a complete story, aware that each level of the house of cards is necessary for the support of the rest.

Chapter One

Concurrent evaluation of LISP expressions

This section describes a programming notation and a methodology for using it such that concurrency may be detected and exploited easily. In doing so I am aligning the discussion with the school of thought that advocates functional programming. My view is that people can think in terms of functional programming. As I will attempt to show, functional notation provides a convenient middle ground for the expression of computations as well as the easy discovery (by machine!) of concurrency in computations so expressed. The work of Davis [Davis78] and others suggest that concurrency can be detected in computations expressed in more conventional (procedural) notations, a much harder task. Since this thesis is primarily concerned with the machine, the mechanisms described and the examples chosen will be at the level of functional programming.

1.1 Functional programming

Algorithms are traditionally expressed as a sequence of steps, or procedure, by which a particular computation may be performed. Such descriptions are known as procedural descriptions. In order to get something done, one follows a procedure which describes how that something gets done. Cooking recipes are examples of procedural descriptions.

In fact, the bias toward procedural description pervades a great deal of human culture. Why this bias has carried into computer programming is less significant than the fact that it has. The result is that people generally program as if the computer used a pencil and paper and stored calculations in little named cubbyholes. Our procedual bias even shows itself in our terminology. The term scratchpad storage is a prime example.

An alternative to listing the steps by which a computation may be performed is to describe the answer. A much-used example ("much-used" is much said, by the way) is the factorial function

```
factorial[n] = 1 * 2 * 3 * ... * (n-1) * n
```

which can be computed in a sequence of steps as in

```
f = 1;
for i = 1 to n do
    f = i * f;
factorial = f;
```

or defined recursively as

```
factorial[n] = if n == 0 then 1
               else n * factorial[n-1];
```

The latter description is a functional one because it describes what a factorial is, whereas the former is a procedural one because it shows how to compute a factorial. It is of course possible to compute a factorial from the functional definition, though the process of doing so may strike some people as "inefficient".

Nevertheless, the functional description of factorial manages to get the essence of the idea across without worrying about the details of assigning values and sequencing computations. By contrast, the procedural description leaves nothing to the imagination, and in doing so closes off avenues for the discovery of concurrency.

Sermonette: As I have pointed out before, work has been done to discover concurrency in procedural descriptions of computations. Perhaps the term should be "recover" rather than "discover", since the major part of the task is filtering out irrelevant details put in because of the demands of procedural description. Programming is hard enough, but when the programmer is forced by convention to specify detail that is later to be discarded in the interest of "discovering" concurrency, effort must be wasted. _End of sermonette._

The central theme of the functional programming methodology, that of _describing the answer_, is generic to the _applicative_ style of programming that LISP shares with the so-called _reduction languages_ [Backus78] and other studies of _recursive programming_ techniques [Burge75]. While applicative notations may differ in syntax and semantics, they all possess a property known as _referential transparency_, which among other things insures that execution of algorithms expressed in such notations will not result in _side-effects_. This absence of side-effects is essential to the task of discovering concurrencies in computations, hence the renewed interest in functional notations.

The reader should be cautioned that functional programming is not simply expressing computations in the most general means possible. Rather, the practice of functional programming entails taking a different point of view from that taken in procedural programming. For example, one may know the procedural steps for inverting a matrix using Gaussian elimination, but how can he describe what the inverse of a matrix looks like? The obvious answer is Cramer's Rule, which unfortunately involves a tremendous amount of computation ($O(n!)$) as compared with the $O(n^3)$ of Gaussian elimination.

In order to express Gaussian elimination functionally, one first notes that the process involves successive transformations of the original matrix. Following this observation comes the description of what these transformations involve, and so on. The eventual result is a dual to the standard procedural description of Gaussian elimination, as shown in an example at the end of this chapter.

Until the change in viewpoint is mastered, generating functional descriptions can be difficult. Interestingly enough, since computer programming is an acquired skill, a student whose first exposure to programming is through functional notation will not have such difficulties. The assertion here is that understanding functional notation is not solely a property of computers. With this in mind, I will use the LISP language as a vehicle for illustrating functional programming and the goodness deriving from its use.

1.2 The LISP language

LISP has been around since the early days of programming languages as we know them today (or, as Perlis calls them, FBAPP: FORTRAN, BASIC, Algol, PL/I, and Pascal). Named after its supposedly original purpose, LISt Processing, LISP enjoys a loyal, bordering on fanatical, following of enthusiasts who revel in its power and flexibility. Indeed, the protean nature of the language has allowed it to survive nearly two decades of use, modification, extension, and other forms of abuse from the research community essentially unchanged from its original form. From its inception, and still today (1980), LISP is the standard language of the so-called Artificial Intelligentsia. LISP is a neat language.

By now, LISP has so many forms, dialects, and implementations that it can hardly be called "a" programming language. While this may result in some confusion when the single word "LISP" is not sufficient to specify what one is thinking about, it allows the author considerable freedom in saying just what he means by LISP. This gives me the opportunity to define a LISP-like functional programming notation and refer to it as "LISP", even though it is actually only that subset of most LISP systems which is functionally pure.

To the extent that a specification of my "LISP" will be required and its basic concepts need to be related to the central theme of this thesis, a short description is provided. However, the interested reader is referred to the superior discussions by Allen [Allen78], Berkeley [Berkeley64], and McCarthy [McCarthy65]. A recent special issue of a personal computing magazine [Byte79] provides an excellent introduction to LISP and several perspectives on its use. The following discussion is modeled after, if not plagiarized from, a discussion by Elson [Elson73].

LISP is set apart from the Tower of Babel of computer languages in three major ways:

1. the representation of all data as symbolic expressions (S-expressions)
2. the control structure of LISP
3. the ability to represent LISP programs as LISP data structures

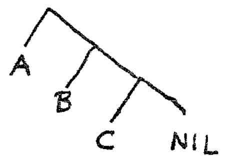
1.2.1 S-expressions

In LISP, all data is represented as S-expressions. An S-expression is either an <u>atomic symbol</u> or a pair of S-expressions. Atomic symbols correspond to what in other languages are called <u>tokens</u> or <u>identifiers</u>. For example, the word "FOO" is an atomic symbol. Numbers are also considered atomic symbols. The atomic symbols NIL and T are permanently defined and are generally used to denote falsity or truth.

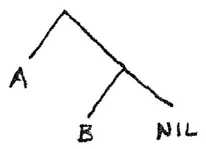
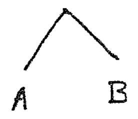
The definition for S-expressions is recursive. That is, data structures of indefinite size can be constructed from collections of S-expressions and atoms. Some possible S-expressions are shown below:

A	atom
(A . B)	pair
((A . B) . (C . D))	pair of pairs
(A . (B . (C . NIL)))	list
() = NIL	empty list

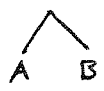
Unbalanced structures are used so often in LISP that a shorthand for list structure is used almost to exclusion of all else. The following S-expressions are identical:

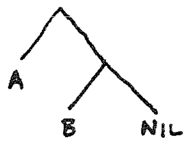
(A . (B . (C . NIL))) = (A B C) = 

Being so often used, list notation can sometimes be confused with dot notation. As long as the difference between (A B) and (A . B) is understood, there shouldn't be any problems.

 = (A B) ≠ (A . B) = 

The formal definition of LISP includes only five elementary operations for operating on S-expressions. The most basic operation, and the one that reflects the philosophy of LISP most clearly, is the construction operator <u>cons</u>. <u>Cons</u> takes two S-expressions as arguments and creates a new pair containing them, returning a pointer to the newly created object.

cons[A,B] = (A . B) = 

cons[A,cons[B,NIL]] = (A B) = 

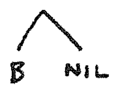
<u>Cons</u> sole purpose is to create new structures from existing ones. It is the <u>only</u> means in LISP of doing so. <u>Cons</u> does <u>not</u> destroy or modify existing data; such operations are not possible in LISP. (This marks my first departure from conventional LISP usage. If you know about <u>rplaca</u>, <u>rplacd</u>, and <u>setq</u>, you can forget about them here. If not, don't worry - funny names like that can't be useful anyway.)

The philosophy behind <u>cons</u> is the maintenence of an illusion, the illusion being that of an infinite quantity of storage for S-expressions. The user is neither informed, nor does he care, where the storage for a new pair is obtained. All that matters is that storage can be obtained whenever it is desired. In practice, this illusion is maintained with a finite quantity of storage by detecting when old S-expressions are not being used and recycling them. All this happens automatically, so the user need not worry until he *really* runs out of storage.

Accompanying the *cons* operator are the two *selection* operators *car* and *cdr*. Each operator takes a pair as an argument and returns one of the elements of the pair. *Car* and *cdr* of atomic symbols are undefined.

```
car[(A . B)] = A
cdr[(A . B)] = B
car[(A B)] = A

cdr[(A B)] = (B) =
```



```
car[cdr[(A B)]] = car[(B)] = B
car[A] = undefined
```

Note that the *cdr* of a list is still a list. The names *car* and *cdr*, artifacts of the IBM 709 computer on which LISP was first implemented, provide a convenient shorthand for commonly used combinations of the selection operators. The trick is to collapse a series of *car*'s and *cdr*'s into a c...r with the appropriate a's and d's.

```
cadr[(A B C)] = car[cdr[(A B C)]] = car[(B C)] = B
caddr[(A B C)] = car[cdr[cdr[(A B C)]]] = C
caar[((A))] = car[car[((A))]] = car[(A)] = A
```

Two *predicate* operators are provided for testing properties of S-expressions. *Eq* compares two atomic symbols for equality. *Eq* is undefined for non-atomic arguments.

```
eq[A,A] = T
eq[A,B] = NIL
eq[A,(A . B)] = undefined
```

Atom tells whether an S-expression is an atomic symbol or not.

```
atom[A] = T
atom[(A . B)] = NIL
```

From these five operators, all computable functions can be derived. For convenience, many more are usually provided for doing arithmetic and so on.

1.2.2 Control in LISP

Aside from functional notation, the only control structure of LISP is the conditional expression. The form of conditional expression is exceptionally simple:

```
[p1 -> e1; p2 -> e2; ... ; pn -> en]
```

works just like the Algol conditional expression

```
if p1 then e1 else if p2 then e2 ...
... else if pn then en
```

The predicates p are evaluated in sequence until one is found to be non-NIL. The corresponding expression is then evaluated and returned as the result of the conditional expression. The value of a conditional expression is undefined if all predicates evaluate to NIL. In practice, this is seldom found to be a concern, as the last predicate of a conditional expression is usually T, which is definitely non-NIL.

The notation for conditional expressions is illustrated in the following example of a function equal which compares two S-expressions for sameness.

```
     equal[x,y] = [atom[x] -> [atom[y] -> eq[x,y];
                                T -> NIL];
                   atom[y] -> NIL;
                   equal[car[x],car[y]]
                       -> equal[cdr[x],cdr[y]];
                   T -> NIL]
```

Note that functions are invoked in exactly the same fashion as primitive operators. Not only is this property absent from the FBAPP languages, it is also missing from the syntactically sugared functional notations of Backus et al [Backus78]. Whereas the manuals for other languages have syntax descriptions, type definitions, and parameter passing modes, the manual for a particular LISP system is the list of functions already defined. Since LISP has no syntax, what else can there be?

When a LISP function f[e1,e2,...,en] is invoked, the elements in the argument list are evaluated and bound to the formal parameters of the function f. The function body is then evaluated in the context of these new bindings. Nowhere in the definition of LISP is the order of evaluation of the arguments specified, nor does it matter since (1) they are all evaluated prior to the evaluation of the function body, and (2) the evaluation of one of the arguments cannot result in side-effects that could change the result of the evaluation of any of the other arguments. This is where strict adherence to functional programming pays off.

Profundity: The arguments to a function in LISP may be evaluated concurrently. The sole sequencing requirement of function evaluation in LISP is therefore that the arguments to a function must be evaluated before the function body can be evaluated. <u>End of profundity.</u>

As more details of LISP operation are revealed, it will become clear that even this sequencing requirement can be relaxed. For now, observing the restriction as is will not be troublesome.

The LISP analogy to sequential statements in the FBAPP languages is <u>functional composition</u>. The sequence

```
     a;
     b;
     c;
```

can be replaced by the composition

```
     c[b[a[]]]
```

where <u>c</u>, <u>b</u>, and <u>a</u> are defined as functions rather than statements. The bunched up parentheses at the right of the composition illustrates a familiar phenomenon which has earned LISP the alternate acronym of <u>L</u>ots of <u>I</u>rritating <u>S</u>ingle <u>P</u>arentheses.

1.2.3 Program is data

LISP fanatics will tell you that the ability to represent LISP programs as LISP data structure allows a program to create another program and have it executed. This is of course true, but the impact of "program is data" reaches farther than the abstractions of AI research. It just turns out to be very convenient to be able to store and manipulate programs easily. Almost all LISP systems have specialized editor and "prettyprint" programs that take advantage of the structure of LISP programs and make life easy for the LISP programmer.

More importantly, being able to represent program as data gives a new meaning to the concept of passing a function or an unevaluated expression as a parameter. If a function receiving such a parameter decided to print it, the actual source code for the expression would appear! Since most LISP systems operate from a uniform linear array of storage cells, the act of passing an unevaluated expression merely involves the creation of a new pointer to an old data structure and passing the pointer.

<u>Important observation:</u> If two functions do not operate out of the same memory, passing an unevaluated expression from one to the other must involve some copying of the program, that is, the data representing the program. Since program and data are represented the same way, a scheme which handles the distribution of data in a multiprocessor LISP system will also handle the distribution of program. This becomes especially significant when the sizes of programs exceed the storage capacity of individual processing elements which must execute them. <u>End of important observation.</u>

The ability to represent program as data also makes it convenient to illustrate the inner workings of a LISP system simply by writing it in LISP. This I will do in order to show how concurrent evaluation schemes differ from ordinary evaluators.

The mapping from functional notation into the standard S-expression form of LISP source code is very simple. A function invocation is simply a list with the function name as the first element and the argument list following.

```
atom[a] --> (ATOM A)
equal[x,y] --> (EQUAL X Y)
```

Capitalization inside S-expressions given here can be interpreted as a reminder that S-expressions are meant to be read by machine and are somehow less "refined" than the equivalent functional notation.

Conditional expressions take on an appropriate form:

```
[p1 -> e1; p2 -> e2; ... ; pn ->en] -->

(COND (P1 E1) (P2 E2) ... (PN EN))
```

where COND signals the beginning of a conditional expression. Note that conditional expressions take on the form of a function invocation, while the rules for evaluating conditional expressions are different from the rules (or lack thereof) for evaluating functions. COND is a so-called <u>special form</u>, or a function whose arguments are not evaluated. Most LISP systems allow the user to define special forms, mostly for the purpose of defining new control structures.

The implementation of special forms is illustrated in the function <u>eval</u> which evaluates an expression <u>e</u> in the environment <u>a</u>:

```
eval[e,a] = [atom[e] -> cdr[assoc[e,a]];
             atom[car[e]] ->
                [eq[car[e],QUOTE] -> cadr[e];
                 eq[car[e],COND] -> evcon[cdr[e],a];
```

```
                     T -> apply[car[e],evlis[cdr[e],a],a]];
                 T -> apply[car[e],evlis[cdr[e],a]a]]

    where

    assoc[x,a] = [equal[x,caar[a]] -> car[a];
                  T -> assoc[x,cdr[a]]]
```

Variable bindings (the environment) are stored on an <u>association list</u> as a list of pairs. The function <u>assoc</u> is used to search such lists. QUOTE and COND are the special forms defined in this version of eval. An S-expression of the form (QUOTE FOO) evaluates to FOO, while the evaluation of conditional expressions is passed to <u>evcon</u>. If the first element of an S-expression is not immediately recognized, it is assumed to be a function which is then applied to the argument list evaluated by <u>evlis</u>.

```
    evcon[c,a] = [eval[caar[c],a] -> eval[cadar[c],a];
                  T -> evcon[cdr[c],a]]

    evlis[l,a] = [null[l] -> NIL;
                  T -> cons[eval[car[l],a],evlis[cdr[l],a]]]

    where

    null[x] = eq[x,NIL]
```

<u>Evcon</u> evaluates until it finds a non-NIL predicate, while <u>evlis</u> evaluates until nothing is left.

None of the functions defined so far does anything to alter the environment <u>a</u> in which expressions are evaluated. Binding formal parameters to actual parameters is a crucial element of function evaluation. The function <u>pairlis</u> takes a list of variable names <u>x</u> and a list of evaluated expressions <u>y</u> and adds new bindings to an association list <u>a</u>.

```
    pairlis[x,y,a] = [null[x] -> a;
                      T -> cons[cons[car[x],car[y]],
                                        pairlis[cdr[x],cdr[y],a]]]
```

Formal parameters are specified through Church's lambda notation [Church41]. For example, the function

```
     norm2[x,y] = sqrt[plus[times[x,x],times[y,y]]]
```

can be represented as the binding of the atom NORM2 to the lambda-expression

```
     (LAMBDA (X Y) (SQRT (PLUS (TIMES X X) (TIMES Y Y))))
```

Lambda-expressions are simply lists in which the first element is the atom LAMBDA, the second is the list of names to be bound to the elements of the parameter list passed to the function, and the third is the expression to be evaluated in the context of these bindings (the body of the function). Such lambda-expressions are recognized by the function apply which also recognizes the primitive operators cons, car, cdr, eq, and atom.

```
     apply[fn,x,a] =
         [atom[fn] ->
             [eq[fn,CONS] -> cons[car[x],cadr[x]];
              eq[fn,CAR] -> caar[x];
              eq[fn,CDR] -> cdar[x];
              eq[fn,EQ] -> eq[car[x],cadr[x]];
              eq[fn,ATOM] -> atom[car[x]];
              T -> apply[eval[fn,a],x,a]];
          eq[car[fn],LAMBDA] -> eval[caddr[fn],
                                     pairlis[cadr[fn],x,a]]
          T -> apply[eval[fn,a],x,a]]
```

With the definition of apply, a complete interpreter for LISP has been given. More impressive than the conciseness of the interpreter is the shock one receives when he first understands how it works, because there is virtually nothing going on besides creating new variable bindings and replacing expressions by other expressions. (This is as it should be, since by the time the interpreter starts working on an expression, the programmer has already described the answer; the interpreter merely fills it out.)

In the course of evaluation, an expression gets handed back and forth between _eval_ and _apply_, unpeeling a little at each step. This mutual recursion is the "clock" by which the interpreter runs, as shown by the evaluation of norm2[3,4] in figure 1.1.

With the simplicity of the LISP interpreter comes another benefit from "program is data". With so few basic elements, the individual parts of evaluation are easily separable, and major changes in behavior can be wrought from changing key interpreter functions - and they're _all_ key interpreter functions. Indeed, much of the basis for this thesis comes from changing the way _evlis_ works.

1.3 Mechanisms for concurrent evaluation

1.3.1 Making evlis eager

The sole function of _evlis_ in the LISP interpreter is the evaluation of argument lists. So far as the rest of the interpreter is concerned, this function can be changed so that _evlis_ merely _arranges_ for the evaluation of argument lists. A simple way of doing this is for _evlis_ to take responsibility for evaluating the first element of an argument list and hand the rest to an _evlis_ in another processor. When both evaluations are completed, they are _cons_ed together and returned in the normal fashion. This operation necessarily involves some sequencing and is illustrated in an Algol-like notation

```
(NORM2 3 4)

-> eval[e,a]
e:  (NORM2 3 4)
a:  ( ... (NORM2 . (LAMBDA (X Y) (SQRT ... )))) ... )

-> apply[fn,x,a]
fn: NORM2
x:  (3 4)
a:  ( ... (NORM2 . (LAMBDA (X Y) (SQRT ... )))) ... )

-> eval[e,a]
e:  NORM2
a:  ( ... (NORM2 . (LAMBDA (X Y) (SQRT ... )))) ... )

-> apply[fn,x,a]
fn: (LAMBDA (X Y) (SQRT ... ))
x:  (3 4)
a:  ( ... (NORM2 . (LAMBDA (X Y) (SQRT ... )))) ... )

-> eval[e,a]
e:  (SQRT (PLUS (TIMES X X) (TIMES Y Y)))
a:  ((X . 3) (Y . 4)
     ... (NORM2 . (LAMBDA (X Y) (SQRT ... )))) ... )

...

-> 5
```

Evaluation trace for (NORM2 3 4)

figure 1.1

```
evlis[l,a] =
    [null[l] -> NIL;
     eager[] ->
        [pair p;
         p.cdr := MAIL;
         spawn[p,cdr[l],a];
         p.car := eval[car[l],a];
         while p.cdr == MAIL do [];
         return p];
     T -> cons[eval[car[l],a],evlis[cdr[l],a]]]
```

where spawn is defined in another processor to mean

```
spawn[p,l,a] = [p.cdr := evlis[l,a]]
```

If eager[] is non-NIL, the first thing that the new evlis does is spawn an evaluation in another processor. Having done that, it evaluates the first element of the argument list and waits for the spawned evaluation to return an answer. If eager[] is NIL, evlis functions as before.

Remark: The value of eager[] is assumed to depend on whether an additional processor is available or not. The point to remember is that the new evlis, an eager evlis, returns the same result as the ordinary evlis regardless of the value of eager[], which might just as easily be the coin flip function. End of remark.

The power of eager evlis is illustrated in the following contrived example of a function mirror which mirrors an S-expression s:

```
mirror[s] = [atom[s] -> s;
             T -> cons[mirror[cdr[s]],mirror[car[s]]]]
```

Eager evlis causes the two arguments to cons to be evaluated concurrently. By the nature of the example, the evaluations of these arguments requires roughly equivalent amounts of non-trivial work. Concurrent evalutation thus reduces the time to mirror a tree of _n_ leaves from O(n) time to O(logn) time. Wow!

The new _evlis_ clearly satisfies the condition of an earlier profundity. That is, all arguments to a function are evaluated before control is passed to the function. The concurrent evaluations are synchronized by a while-loop. As noted earlier, this synchronization is not strictly necessary.

Suppose the while-loop is removed from the new _evlis_. It then becomes possible for an incomplete argument list to be returned from _evlis_. In all cases, the first argument may be accessed, but any premature attempts to access the rest of the argument list results in following a pointer to MAIL. Once followed, such a pointer cannot be retraced, so some kind of checking is in order.

The requisite form of checker comes in the form of a _suspicious cdr_

```
     cdr[x] = [while x.cdr == MAIL do [];
               return x.cdr]
```

Synchronization is delayed until the first attempt to follow a _cdr_ pointer that points to MAIL. As will be shown in later examples, many functions can initiate substantial activity with an incomplete argument list.

Unfortunately, as the interpreter is presently defined, the first attempt to access the whole argument list occurs in _pairlis_ as new variable bindings are being added to the environment. This problem is an artifact of 1) the decision to define an interpreter rather than a compiler, and 2) the particular way in which the interpreter has been presented. The first point has been important to the development of the subject matter, and will continue to be so. The second is a result of wanting to present the interpreter clearly. The reader should be able to convince himself that a different interpreter can be defined in which the functions of _evlis_ and _pairlis_ are combined. (The author has in fact written and tested such an interpreter. Unfortunately it isn't pretty enough to be included with the text. See appendix.)

1.3.2 A lenient cons

Another source of concurrency comes from considering the basic philosophy behind _eager evlis_. That is, to return a result whether or not the parts of the result are completely evaluated. In LISP the basic means of producing results is _cons_. If _cons_ can be made to produce results without bothering to check that the parts are evaluated, synchronization can be delayed until the parts of the result are needed. How are results taken apart? With _car_ and _cdr_, of course.

Friedman and Wise [Friedman76] define such a _cons_ and call it a _lenient cons_. It is easily incorporated into the interpreter by considering _cons_ to be a special form rather than a primitive operator. Invocations of _cons_ are thus recognized in _eval_ rather than _apply_:

```
eval[e,a] = [atom[e] -> ... ;
             atom[car[e]] ->
                [eq[car[e],QUOTE] -> cadr[e];
                 eq[car[e],COND] -> evcon[e,a];
```

```
                         eq[car[e],CONS] ->
                            lcons[cadr[e],caddr[e],a];
                         T -> apply[ ... ];
                     T -> apply[ ... ]]
```

where lcons (in this case an asymmetrical lenient cons) is defined in a manner similar to the eager evlis:

```
     lcons[x,y,a] = [pair p;
                     p.cdr := MAIL;
                     cspawn[p,y,a];
                     p.car := eval[x,a];
                     return p]
```

and cspawn is defined in another processor as similar to spawn, invoking eval instead of evlis:

```
     cspawn[p,e,a] = [p.cdr := eval[e,a]]
```

As with eager evlis, synchronization is provided by suspicious cdr. Keller uses a symmetric form of lenient cons which requires car to be made suspicious as well [Keller78].

Interestingly enough, the old contrived example of the mirror function serves as an illustration of the way lenient cons works. As with eager evlis, the evaluation of

```
     cons[mirror[cdr[s]],mirror[car[s]]]
```

spawns separate subprocesses to evaluate the two arguments. However, a symmetric lenient cons will return with an answer almost immediately. Only when the result is examined will there be any synchronization. An analogy due to Hewitt is that of trading in commodities futures. Strawberries can be bought, traded, and sold, but the only time they have to exist is when they are eaten [Hewitt77]!

By returning a result, _any_ result, as fast as possible, _lenient cons_ is argued not only to be an effective means of discovering concurrency, but also a mechanism for _pipelining_ processes in separate processors. Although lacking in generality (nothing special would happen under _lenient cons_ if _cons_ was replaced with _plus_ in _mirror_), _lenient cons_ does represent a valuable concept in discovering and distributing concurrency.

1.3.3 Concurrent evaluation of conditionals

I have focused so far on schemes which partition work which is known to be useful. However, there are cases where anticipation of work to be performed can speed up the evaluation process.

In the course of evaluating conditional expressions, each branch is preceded by the evaluation of some predicate. Certain types of programs such as compilers, interpreters, and goal-directed AI programs often have conditional expressions with several predicates, the first of which is seldom true. Little advantage can be gained from evaluating simple predicates in parallel, since each takes so little time. On the other hand, complicated predicates involving several levels of recursion are prime candidates for evaluation in parallel.

An appropriate (eager) version of _evcon_ is easily defined.

```
     evcon[c,a] =
         [eager[] ->
             [pointer x;
              x := MAIL;
              fork[x,cdr[c],a];
```

```
                  if eval[caar[c],a] then [
                     kill[];
                     eval[cadar[c],a];
                     ]
                  else [
                     while x.cdr == MAIL do [];
                     return x;
                     ]
                  ];
              T ->
                 [eval[caar[c],a] -> eval[cadar[c],a];
                  T -> evcon[cdr[c],a]]
```

where <u>fork</u> is defined in another processor to be

```
     fork[x,c,a] = [x := evcon[c,a]]
```

and <u>kill</u> recursively terminates all child processes. <u>Kill</u> is included to stop activity which is known to be pointless.

Evaluating conditional expressions in this way represents a kind of gamble. The benefit from evaluating something before it is necessary must be weighed against the probability that it won't be needed. Care must be taken to avoid investing large fractions of the resources available in work that is not needed. In addition, due to the way conditionals are generally written, parallel evaluation of conditionals may result in illegal operations such as dividing by zero or taking the <u>car</u> of an atom. These are generally more catastrophic failures than those brought on by eager <u>evlis</u> operations. Care would have to be taken to insure these operations do not prove fatal.

Consider the following function <u>actsmart</u> defined in terms of a conditional expression:

```
     actsmart[in] =
         [match[in,pattern1] -> reply[in,response1];
          match[in,pattern2] -> reply[in,response2];
          .
```

```
        .
        match[in,patternN] -> reply[in,responseN];
        T -> nullreply[]]
```

If an eager evcon is unleashed on this function, and if invocations of match are costly, the proper response can be initiated in little more than the time required for a single match.

Unfortunately, very few LISP programs follow this pattern. Even the most ambitious conditionals appear to be similar to the form for interpreters:

```
      [null[x] -> ...
       atom[x] -> ...
       atom[car[x]] ->
           [eq[car[x],FOO] -> ...
            eq[car[x],MUMBLE] ...
            ...
```

Such conditionals hardly warrant concurrent evaluation. Worse yet, premature evaluation of some of the predicates invites disaster. It would seem that taking advantage of an eager evcon would involve considerable effort. Or perhaps the few programs that could use it can be recoded as functions operating on lists, as in

```
      itheval[firstone[pattern1,pattern2, ... , patternN],
              quote[response1,response2, ... , responseN]]
```

1.4 When is it worth it?

In the previous section I presented several mechanisms for exploiting concurrency. Each one takes a task following a general pattern and transforms it to a set of subtasks that can be evaluated concurrently. While it is certainly true that such schemes in general detect and exploit concurrency, the benefits from doing so are not equally pronounced.

1.4.1 Tail recursion vs tree recursion

A good example of the trade-offs involved is _tail recursion_, which is the LISP analogy to iteration in other languages. For example, the list reversal function (as opposed to tree reversal) can be defined as

```
rev[l] = rev1[l,NIL]

rev1[a,b] = [null[a] -> b;
             T -> rev1[cdr[a],cons[car[b],b]]]
```

There is no benefit to be derived from evaluating the arguments to _rev1_ concurrently, since they are so simple. Further, each invocation of _rev1_ initiates only one more invocation directly, so there is no useful partitioning of labor to be had.

As an intermediate example, consider the function _mapcar_ which maps a function _f_ onto a list of elements _l_ defined as

```
mapcar[f,l] = [null[l] -> NIL;
               T -> cons[f[car[l]],mapcar[f,cdr[l]]]]
```

Clearly, the tasks of evaluating f[car[l]] and mapcar[f;cdr[l]] do not require equal work. However, the degree to which this is important depends on the complexity of evaluating f relative to the work involved in setting up the next evaluation of f. We will call this latter quantity the _tail recursion work_ t. If the evaluations of f can be performed by separate processors, the time to evaluate mapcar on a list of length n is therefore w(f) + n*t

where w(f) is the work (time?) required to evaluate f. This is clearly better than the n*(w(p) + t) time required to evaluate mapcar in a one-processor system. However, it is possible that we can do better by avoiding the use of tail recursion. The degree to which the effects of tail recursion are noticed is determined by the ratio n*t/w(f).

An alternative to the use of tail recursion is mapping the function onto a tree structure instead of a list. An appropriately defined version of the mapping function is

```
map[f,t] = [null[t] -> NIL;
            atom[t] -> f[t];
            T -> cons[map[f,car[t]],map[f,cdr[t]]]]
```

Each evaluation of map invokes two evaluations of equal work, as opposed to the unbalanced partitioning of mapcar. If each new invocation can be given to a separate processor, the time to evaluate map on a tree containing n elements is

w(f) + log(n)*t

which is better, especially for large n, but is hardly worth the effort if w(f) is large enough to warrant parallel evaluation in the first place.

1.4.2 Automatic vs programmer specified

The matter of generality was also brought up during the discussion of lenient cons. Eager evlis was seen as somehow more general than lenient cons, yet an expression like

```
     cons[car[x],car[y]]
```

hardly yearns even for the capabilities of _lenient cons_. After all, spawning a process has to take a good deal more effort than taking two _cars_.

A glimmer of an answer is provided in the implementation of _lcons_, which is invoked from a new _eval_ which recognizes CONS as a special form. The new special form could just as easily be LCONS, leaving the task of recognizing CONS back in _apply_. Doing this would allow (or force, depending on your point of view) the _user_ to specify when he wants to use a _lenient cons_ and when he just wants to stick two things together with an ordinary _cons_. It should be fairly obvious to the user where computations are likely to be time-consuming, and LISP systems are interactive enough to let one try out several possibilities quickly.

In addition to _lcons_, the user could be allowed to define his own _extra-special forms_ ... or not. If he just wants to define a simple function without having to invoke the whole mechanism of _eager evlis_, so be it. Unfortunately, this would involve some duplication of function, as it were. Starting with _cons_ and _lcons_, we could have _plus_ and _eplus_, and so on. _Cons_ and _lcons_ are two different functions, but all the other function pairs would differ only in the kind of _evlis_ they use, which is in turn dictated by their usage in programs.

Perhaps the proper course is to have the user specify when eager evaluation is desired. A new special form could be used to differentiate usage.

```
     sumtree[t] = [atom[t] -> t;
                        T -> eager[plus[sumtree[car[t]],
                                        sumtree[cdr[t]]]]]
```

```
      sumlist[l] = [null[l] -> 0;
                    T -> plus[car[l],sumlist[cdr[l]]]]
```

A new _eval_ can thus be defined:

```
      eval[e,a] =
           [ ...
            atom[car[e]] ->
                 [ ...
                  eq[car[e],LCONS] -> lcons[cadr[e],caddr[e],a];
                  eq[car[e],EAGER] -> eeval[cdr[e],a];
                  ...
```

where _eeval_ applies a function to an argument list evaluated by _eevlis_

```
      eeval[e,a] = apply[car[e],eevlis[cdr[e],a],a]
```

and _eevlis_ is the previously defined _eager evlis_, given a different name in order not to conflict with the original evlis which is still around.

1.5 Three deluxe examples

1.5.1 Quiksort

Quiksort is typical of the divide and conquer class of algorithms ideally suited to parallel evaluation. It works by first partitioning a set of elements to be sorted into two sets, call them A and B, in which each element of B is greater than every element of A. The process is then repeated on the two sets A and B.

A classical implementation of _quiksort_ in Pascal is as follows:

```
type sortlist = array[1:n] of integer;

procedure quik(min,max:integer; var a:sortlist);
var middle:integer;
begin
    middle := partition(min,max,a);
    if middle > min+1 then quik(min,middle-1,a);
    if middle < max-1 then quik(middle+1,max,a);
end;

function partition(min,max: integer;
    var a: sortlist): integer;
var i,j,m,temp: integer;
begin
    m := a[max];
    i := min; j:= max;
    repeat
        while (i <> j) and (a[i] <= m) do i := i+1;
        while (i <> j) and (a[j] >= m) do j := j-1;
        temp := a[i]; a[i] := a[j]; a[j] := temp;
    until i = j;
    a[max] := a[i]; a[i] := m;
    partition := i;
end;
```

The best (and expected) time for quiksort applied to an array of n elements is O(nlogn). To achieve this it is best for the switching element m to be the median element of the array. In the algorithm shown m was merely taken from one end of the array. This is fine if the array is random, but if the array already has some order to it, partition will do a very lopsided job and the expected time will be equal to the worst case time of $O(n^2)$. In the interest of simplicity we will merely note this fact and continue the convenience of choosing m in an arbitrary manner.

Another item of interest is that quiksort as written in Pascal makes heavy use of array indexing for compare and exchange operations. If the "cost" of performing a compare or an interchange was proportional to the distance between the two elements involved, quiksort would be an $O(n^2)$ algorithm.

We can easily detect concurrency in this algorithm; after all it _is_ a divide and conquer algorithm. Clearly, the two recursive calls to _quik_ can be performed in parallel. They both unfortunately operate on the same array, but by the structure of the algorithm they operate on mutually exclusive parts of the array. If the recursive calls to _quik_ are partitioned out to separate processors, and _if_ the problems of contention for shared memory can be overcome, _quiksort_ would become an O(n) algorithm.

At this point the reader may be saying, "Who _cares_ if we can get a speedup from O(nlogn) to O(n)? They're both roughly the same time, and it takes that long to read in the information from disk. Besides, I never do any sorting anyway!" These points are well taken. The purpose of this exercise is to demonstrate the facility of discovering concurrency in algorithms expressed functionally. Of necessity we start with simple textbook problems. (Unfortunately, _I_ don't know of any interesting non-numerical $O(n^4)$ algorithms, so we will progress from simple textbook problems to more difficult textbook problems.)

With this said, we present a LISP version of _quiksort_:

```
     quik[l] = [atom[l] -> l;
               T -> partition[car[l],nil,nil,cdr[l]]]

     partition[m,a,b,l] =
         [null[l] -> conc[quik[a],list[m],quik[b]];
         greater[s,car[l]] ->
             partition[m,cons[car[l],a],b,cdr[l]];
         T -> partition[m,a,cons[car[l],b],cdr[l]]]
```

Although it looks very different, it really is much the same algorithm. The major difference is that _partition_ actually generates the two sets A and B separately from the input set. Thus, there are no side effects and parallel operations can proceed unimpaired.

In the Pascal version of _quiksort_, we knew that the recursive calls to _quik_ could proceed in parallel because we knew how _quiksort_ behaved. That is, the two sets to be sorted are always in different parts of the same array. A quick glance at the LISP version points this out immediately. All other operations are easily seen to be sequential in nature.

Again, I point out that the functional description of _quiksort_ is not so much a procedure for sorting a list as a description of _what the sorted list looks like_. The definition is concise and to the point. And, as in the LISP interpreter, there is _almost nothing going on_.

1.5.2 An NP-complete program - _The Traveling Salesman_

The NP-complete class of textbook problems are exceedingly expensive to solve exactly. NP stands for non-deterministic polynomial; NP-complete problems can be solved in polynomial time only by non-deterministic means. That is, whenever there is a choice of paths to take in the course of solution, the _correct_ path is taken. Deterministic solution of course requires the evaluation of each of these paths at each step of the solution. This exponential growth property forces the use of approximation methods on those who actually want to use the results.

However, being textbook problems, nobody is really interested in solving them. Rather, they are used as vehicles for studying computational complexity.

In a sense, then, the following example is a fanciful one. However, it is useful in illustrating the potential of applicative notation for mobilizing and coordinating computational resources. As we shall see, attempting to solve an NP-complete problem can mobilize many more computational resources than are feasible to build.

This observation becomes more meaningful if taken in a slightly different way. Recall that the mechanism for spawning subtasks does so only if extra resources are available. In the case of these NP-complete problems, we can look upon this characteristic as the ability to make use of whatever is available. For example, if you have an "intractable" problem that requires 10^{12} operations, isn't it reassuring to know that your 10^4 processors can be applied effectively to solve it in 10^8 time?

Browning and Mead [Browning79a] outline the general idea for distributing NP-complete problems over a <u>large</u> number of processors. Since exact solutions of NP-complete problems are necessarily combinatorial, why not simply generate all possible candidates for solution and pick the best one? This technique is illustrated in the following example:

```
"traveling salesman - find the shortest path"
travel[l] = descend[0,nil,l,l]

descend[s,x,y,z] =
    [null[y] -> cons[s,x];
     null[z] -> nil;
     T -> min[descend[plus[s,cost[car[x],car[y]],
                       cons[car[y],x],cdr[y],cdr[y]],
                   descend[s,x,rotate[y],cdr[z]]]

min[x,y] = [null[y] -> x;
            greater[car[y],car[x]] -> x;
            T -> y]

rotate[l] = rot1[car[l],cdr[l]]

rot1[x;l] = [null[l] -> x;
             T -> cons[car[l],rot1[x,cdr[l]]]]

"an example distance function"
cost[x,y] = abs[minus[x,y]]
```

In the _traveling salesman_ problem, we are faced with the task of finding the shortest path that passes through each city exactly once. The notion of _distance_ between cities can be generalized to that of the _cost_ of arcs in a graph. The more practical problem is amenable to approximation through geometric analysis, and some approximations so obtained are in fact quite good [Lewis78].

The algorithm shown solves the problem by generating all possible permutations of n elements (cities) and computing their cost functions along the way. After they have all been generated, they return through a filtering process which selects that permutation with the minimum total cost. There are n levels to the permutation process, and n^2 operations are required at each level to set up the computations for the next level down. A quick glance at the algorithm shows that the evaluations of _descend_, paired up as arguments to _min_, are really the only operations for which partitioning of effort is worthwhile. If a sufficient number (exponential) of processing elements exist, this algorithm takes $O(n^3)$ time, which is great if you have money but are short on time.

1.5.3 Solution of a linear system of equations

Numerical Analysis is a peculiarly fascinating blend of applied mathematics and computer science. While there is a familiar satisfaction accompanying the design of a "beautiful" algorithm, there is also an overriding concern for the "bottom line" - how fast the algorithm executes, how stable it is numerically, etc. While there is a concern for algorithmic complexity, asymptotic measures are not used to determine the efficacy of an algorithm. Rather than say Gaussian elimination takes $O(n^3)$ time, it is said that the time complexity is $3n^3/3 + n^2 - n/3$ [Isaacson66]. In fact, these expressions are usually broken down further into floating multiplications, divisions, and additions.

Certainly one reason for this is the fact that people crunch numbers in order to see the results. More important perhaps is the scale of problems attempted and the frequency with which they are attacked. Whatever problem is being solved today, there are always bigger or more complex problems on the shelf waiting for the next generation of faster computers. For example, straightforward application of linear algebra to the solution of Laplace's equation in three dimensions yields a problem whose complexity is of order n^7, where n is the size of the system along one of the dimensions. Fluid mechanics people will be fairly happy when this kind of problem can be comfortably solved for $n=100$. That's 10^{14} floating point operations, folks, and if the problem is not Laplace's equation but some non-linear subset of the Navier-Stokes equations, this operation will be merely an iteration toward the final solution [NASA78]!

One strange aspect of numerical analysis is the non-hierarchic makeup of most problems. While one might speak of the time required to solve a 100x100 system with the same concern a computer scientist refers to the time required for a procedure call, the end problem usually isn't very many levels above simple matrix inversion. The numerical analyst may be interested in solving a 1000x1000 system or applying 100x100 solutions as part of an iterative process. Whereas a computer scientist might build a many-level hierarchy of procedures or data structure out of simple components, the numerical analyst may be content with nested do-loops and large multi-dimensional arrays. Procedures are seen as a means of factoring out common subtasks, largely in the interest of saving typing.

All of the above is by way of introduction to the ubiquity of the solution of linear systems of equations, more popularly called matrix inversion (and they _aren't_ the same thing!). Coming from an engineering background, one of my early measures of the value of a programming language was the facility with which I could use it to solve linear systems of equations. Since my first introduction to LISP I have remembered the admonishments of the instructor to program by _describing the answer_, and wondered just how to describe what the inverse of a matrix looks like. Before going into that in detail, let me establish a reference point by presenting the algorithm in Pascal:

```
type matrix = array[1:n,1:n] of real,
     vector = array[1:n] of real;

procedure solve(n:integer; var a:matrix; b:vector);
var i,j,k,piv:integer;
    m,t:real;
begin
```

```
        "forward elimination with pivoting"
        for i:=1 to n-1 do
        begin
            "find pivot element"
            piv:=i;
            for j:=i+1 to n do
                if abs(a[j,i]) > abs(a[piv,i]) then piv:=j;
            "perform exchange"
            for j:=1 to n do
            begin
                t:=a[i,j];
                a[i,j]:=a[piv,j];
                a[piv,j]:=t;
            end;
            t:=b[i]; b[i]:=b[piv]; b[piv]:=t;
            "elimination step"
            for j:=i+1 to n do
            begin
                m:=a[j,i]/a[i,i];
                for k:=i+1 to n do
                    a[j,k]:=a[j,k]-m*a[i,k];
                b[j]:=b[j]-m*b[i];
            end;
        end;
        "back substitution"
        b[n]:=b[n]/a[n,n];
        for i:=n-1 downto 1 do
        begin
            for j:=i+1 to n do
                b[i]:=b[i]-a[i,j]*b[j];
            b[i]:=b[i]/a[i,i];
        end;
    end;
```

Solve is a good example of an algorithm well expressed in do-loops. Perhaps it's my cultural bias - this algorithm clearly and exactly describes to me the process of Gaussian elimination. What expressive power is lost due to the verbosity of over-specification (who *cares* about keeping track of i, j, and k or is interested in how an exchange is performed?) is compensated for by the familiarity of array indexing conventions and the pencil and paper nature of assignment statements.

Unfortunately, this exactness of specification gets in the way of extracting concurrency from the problem. It *seems* obvious where we can find concurrency in Gaussian elimination, but that is only because of our familiarity with it. Imagine being presented with this algorithm without being told what it does, and the point becomes clear. The indices i, j, and k assume only one set of values at a time, so parallel execution has to be directed through multiple sets of indices. But wait a minute. How can we be sure that smashing a[j,k] in one iteration doesn't interfere with a[i,k] in some other iteration? We can't, except by detailed analysis of indexing patterns or by advance knowledge of what the algorithm is supposed to do in the first place.

Some forms of concurrency go a lot deeper than instantiation of do-loops. For example, isn't it clear that the pivot element for the ith elimination step should be known upon completion of the i-1st elimination step? This fact is not at all apparent from the Pascal description of the algorithm, and putting it in is a non-trivial re-write.

During back substitution, it should be clear that the solution of b[i] can be applied immediately to partial computations of b[j] for j<i. If results are applied as soon as they are known, the back substitution process can be reduced to linear time instead of the quadratic time normally required. Again, deducing this fact from a Pascal program involves more analysis than I think is reasonable to expect of a compiler.

With these points in mind I present a functional definition of what the solution of a linear solution of a linear system of equations looks like:

```
"solution of system Ax=b"
x[a,b] = back[unzip[forward[pivot[zip[a,b]]]]]

"back substitution sweep"
back[x] = back1[car[x],cdr[x]]

back1[a,b] =
    [null[b] -> nil;
     T -> subst[car[a],car[b],back1[cdr[a],cdr[b]]]]

"solve for b[i] in terms of b[j], i<j<=n"
subst[ai,bi,x] =
    cons[quotient[minus[bi,
                             inner[cdr[ai],x]],
                      car[ai]],
            x]

inner[x,y] =
    [null[x] -> 0;
     T -> plus[times[car[x],car[y]],
                 inner[cdr[x],cdr[y]]]]

"forward elimination without pivoting"
forward[a] =
    [null[a] -> nil;
     T -> cons[car[a],
                 forward[pivot[elim[car[a],cdr[a]]]]]]

"eliminate using first row"
elim[p,m] =
    [null[m] -> nil;
     T -> cons[rowdiv[quotient[caar[m],car[p]],
                        cdr[p],
                        cdar[m]],
                 elim[p,cdr[m]]]]

"returns y-m*x"
rowdiv[m,x,y] =
    [null[x] -> nil;
     T -> cons[minus[car[y],times[m,car[x]]],
                 rowdiv[m,cdr[x],cdr[y]]]]

"pivot step"
pivot[a] = pivi[car[a],cdr[a],nil]

pivi[t,x,y] = [null[x] -> cons[t,y];
                 greater[car[t],caar[x]] ->
                     pivi[t,cdr[x],cons[car[x],y]];
                 T -> pivi[car[x],cdr[x],cons[t,y]]]
```

```
"utilities to join and separate augmented matrix"
"form augmented matrix from matrix a and vector b"
zip[a,b] = [null[a] -> nil;
            T -> cons[zip1[car[a],car[b]],
                      zip[cdr[a],cdr[b]]]]

zip1[a,b] = [null[a] -> b;
             T -> cons[car[a],zip1[cdr[a],b]]]

"form matrix and vector from augmented matrix"
unzip[l] = [null[l] -> list[nil];
            T -> combine[allbut[car[l]],
                         lastone[car[l]],
                         unzip[cdr[l]]]]

combine[x,y,z] = cons[cons[x,car[z]],cons[y,cdr[z]]]

"doing these separately ..."
allbut[l] = [null[cdr[l]] -> nil;
             T -> cons[car[l],allbut[cdr[l]]]]

"... is faster than trying to do them at once"
lastone[l] = [null[cdr[l]] -> car[l];
              T -> lastone[cdr[l]]]
```

At first it looks like a hopelessly jumbled set of definitions. There is a pattern, however, and close examination shows a fairly straightforward transliteration of the Pascal procedure. For example, instead of saying "forward elimination, then back substitution" the answer is defined as "back substitution of forward elimination of pivot of input".

Besides this simple transformation of sequential statements, for-loops have been replaced by the recursive LISP equivalents, and infix operators have been made prefix. I hope the reader will take the time to compare these two definitions of the same algorithm, as the similarity is striking.

Consider now the interaction of _forward_, _elim_, and _pivot_ under a combination of the _eager evlis_ and _lenient cons_ types of evaluation. _Elim_ _conses_ up rows as it performs the eliminations and returns them to _pivot_. _Pivot_ chases along its heels, comparing the first element of each new row as it is generated. Almost immediately after _elim_ has hit bottom, _pivot_ announces completion of its task to _forward_, which promptly begins the next elimination step. This automatic and possibly optimal form of pipelining comes for free once the effort is taken to describe the algorithm functionally.

An equally striking example can be found in the back substitution process. Almost as soon as the first pivoting operation takes place, _forward_ returns the first row of the eliminated matrix and _back_ lays the foundation for calculating x[1] in terms of x[i], i>1. The evaluation tree is laid out, terminating in MAIL nodes at the leaves. Soon the next row is supplied, and a corresponding tree is set up for x[2]. Note that x[1] depends on x[2], but they both depend equally on x[i], i>2. Eventually the whole back substitution tree is set up and the triggering value x[n] is supplied. Then, in a flurry of activity, MAIL nodes get replaced by results and these results ripple up the evaluation trees.

A similar but more massive mobilization of resources takes place during each elimination step. Each invocation of _elim_ starts a _rowdiv_ operation on the first row and simultaneously launches another _elim_ operation on the rest of the rows. Indeed, if pivoting were left out, _forward_ would be snatching the first results of _elim_ and inviting another elimination step while the original is still in progress. Again, the pipelining is automatic, yet more complete than could be specified by the human writing a Pascal program.

At this point something should be said about the reduction of theory to practice. While it is true that functional notation provides an extremely powerful avenue for directing the parallel evaluation of Gaussian elimination, the sad truth is that actual pieces of data have to be transferred from physical place to place in order for computations to take place. The flow of information during Gaussian elimination with pivoting is rather complicated and does not map particularly well onto a variety of structures. This problem of mapping of logical structures onto physical ones is the primary concern of this thesis, and these points will be discussed again later. For now, all that can be said is that functional notation does not get in the way of extracting and exploiting concurrency from problems.

Chapter Two

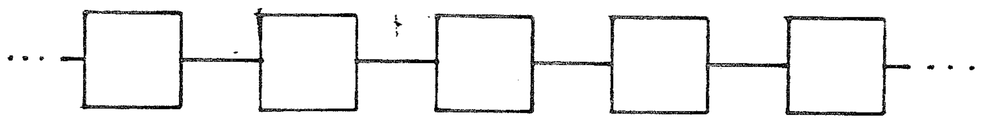
Structures for partitioning computations

I have so far concentrated on the logical problems involved in finding concurrencies and exploiting them, assuming the existence of some physical structure onto which the logical structure can be mapped. In this section I will turn the situation around and assume for each of several physical structures ~~a~~ corresponding logical structure capable of recognizing and distributing concurrencies.

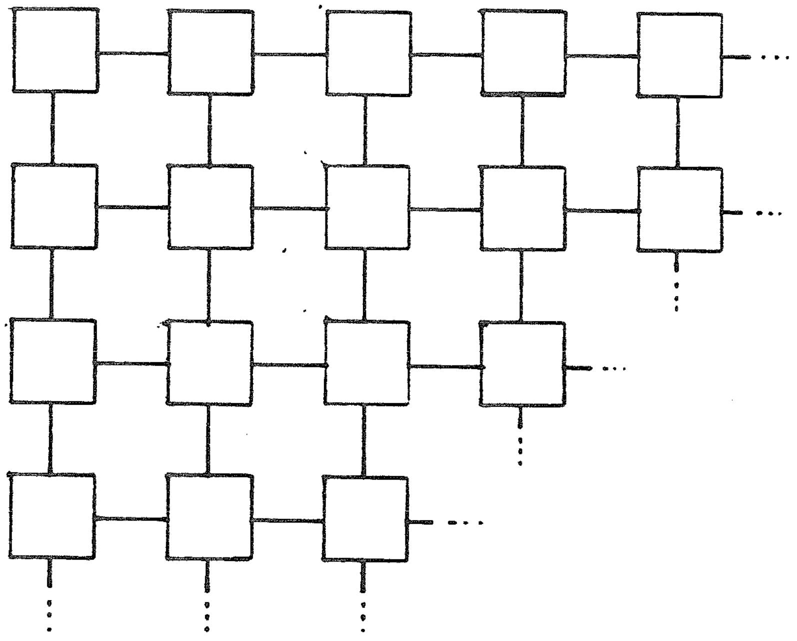
2.1 Array structures

The simplest and most often suggested structure for multiprocessors is the array of identical machines. Array interconnection in one and two dimensions is easily arranged (figure 2.1), and there is substantial economic incentive for the use of standardized parts.

One dimensional array machines have proven effective in a host of numerical analysis and signal processing applications. Some signal processing applications adapt so well to this structure that general purpose computers are seldom used for the individual processing elements in the array. Pipelined interconnection of specialized chunks of hardware lead to economic (ie commercially successful) construction of many stage digital filters, FFT boxes, and processors specialized for synthetic aperture radar [Cohen79].



1-D Array Machine



2-D Array Machine

figure 2.1

Arrays of general purpose processors are less common owing to their expense. Nevertheless, machines such as Illiac IV [Barnes68] are being used effectively in the solution of linear and non-linear systems of equations, and are particularly successfully applied to the solution of partial differential equations.

2.1.1 Numerical algorithms

Most numerical algorithms adapt well to use on one-dimensional array machines since they can be factored along at least one dimension. Matrices to be inverted can be divided into rows or columns. Uniform grids for partial differential equations can generally lose one dimension to parallelism.

Sometimes algorithms are not adapted without effort. For example, Gaussian elimination requires communication along both dimensions of the matrix. In a one-dimensional array machine this communication must in part result in the physical transmission of signals between processors. This is an added detail that the one-processor algorithm need not address, since all intermediate results occupy the same memory.

2.1.1.1 Array structure and algorithm structure

Some complications of adapting numerical algorithms reach beyond complexity of programming them and affect the numerical method used. An example is the solution of time-invariant ordinary differential equations with boundary conditions on a one-dimensional grid by the Gauss-Seidel relaxation method. Forgetting for the moment that this is a stupid way of solving the problem, we set up the sample problem

$$u''(x) = f(x), \quad a < x < b, \quad u(a) = ua, \quad u(b) = ub$$

by approximating the differential equation as a difference equation and solving for the value of u at each grid point. Assuming the interval is divided into n sub-intervals,

$$x[0] = a, \quad x[n] = b,$$

$$x[i+1] - x[i] = (ub - ua)/n = dx,$$

and that the differential equation can be approximated by

$$(u[i+1] - 2u[i] + u[i-1])/dx^2 = f[i],$$

we solve for u[i] and apply the result in Gauss-Seidel fashion and iterate the following loop until the maximum change in u[i] is small enough to indicate convergence.

```
for i=1 to n-1 do
    u[i] = F(u[i-1],u[i],u[i+1])
```

Note that relaxation implies the evaluation at grid points in sequence, using previously computed values wherever possible. If these evaluations occur in parallel, as shown by the change in notation

```
for each i in [1,n] do
    u[i] = F(u[i-1],u[i],u[i+1])
```

we suddenly discover that the grid evaluations occur synchronously, and the iterations are no longer Gauss-Seidel but Jacobi, which converge only half as fast.

This difficulty could in principle be overcome by overlapping successive Gauss-Seidel iterations (ie pipelining), except that Gauss-Seidel iterations are often applied in alternating directions to improve convergence. Clearly, alternating the direction of pipelined operations at each iteration "empties the pipeline" at each iteration, so the improvement in convergence has to be weighed against the loss in parallelism.

Granting that a gross penalty in speed may be a bad bargain for a slight improvement in convergence, we still have to choose between the straightforward and speedy (yet slowly convergent) Jacobi iteration and the more complicated and rapidly convergent Gauss-Seidel iteration. (Ironically, Gauss-Seidel is <u>simpler</u> than Jacobi when programmed for a sequential machine, so the choice is seldom difficult.) Ordinarily, alternating the direction of iteration of Gauss-Seidel iterations is a trivial matter. However, when dealing with a parallel machine, even a one-dimensional array machine, the rules of the game have changed.

2.1.1.2 Direct solution vs iteration

As I mentioned before, using simple iterative methods on one-dimensional ODEs with boundary conditions is usually a bad choice. (The exception to this is when time-dependent behavior is desired.) If the ODE is linear, direct solution merely involves the solution of a banded system of linear equations. (Non-linear ODEs can be solved by a few Newton iterations of such direct solutions, a process called <u>quasi-linearization.</u>) Array machines can be gainfully applied to the solution of full linear systems of equations. However, solving a banded system is a simple two-sweep (down and up) process, which does not lend itself especially well to parallelism.

While one would normally choose a direct solution method when programming for a sequential machine, we have seen a case where the iterative solution of a problem speeds up dramatically while the direct solution does not. Not only are we forced to deal with the details of a solution method, but we must re-evaluate the <u>nature</u> of the method desirable for computing the solution on a parallel machine.

2.1.1.3 Time solution and PDEs

Solution of time-invariant partial differential equations is similar to solving the corresponding time-dependent equations by simulating the behavior from rest or initial conditions. Jacobi iteration differs from time simulation in that some grid points are advanced more quickly in time than others. In time simulation, all grid points are advanced with the same time step, generally the maximum step allowed for the most sensitive grid point. This time step is chosen to be the maximum for which stability of the solution can be guaranteed.

Generally, time simulation is not performed unless time-dependent behavior is desired. This is because direct solution methods are much faster on sequential machines. However, as the dimensionality of the target machine increases, not all methods adapt equally well. Time simulation and Jacobi iteration will always take full advantage of array machines, but the situation for direct solution methods is less clear.

Direct solution methods operate on mathematical abstractions of physical situations, whereas simulation methods map the geometry of the physical situation directly onto the computer. Parcels of fluid transfer heat and momentum only to the elements surrounding them, so a simulation method will require only nearest neighbor communication within an array computer. The process of inverting a matrix is an operation on a mathematical abstraction which does not preserve the locality of the physical situation.

Matrix operations such as pivoting or eliminating a column have no analogy in the physical world. This fact can be ignored in a sequential machine where all memory accesses cost the same, but an array machine (indeed, <u>any</u> parallel machine) must by nature impose some characteristics of the physical world on the algorithm designer. It may be that the existence of multi-dimensional array computers will bring a resurgence in the use of iterative methods simply because they follow more closely the physics of the problem being solved. Unfortunately, a 2-D array computer performing an iterative solution of some problems may be slower than a 1-D array computer performing a direct solution.

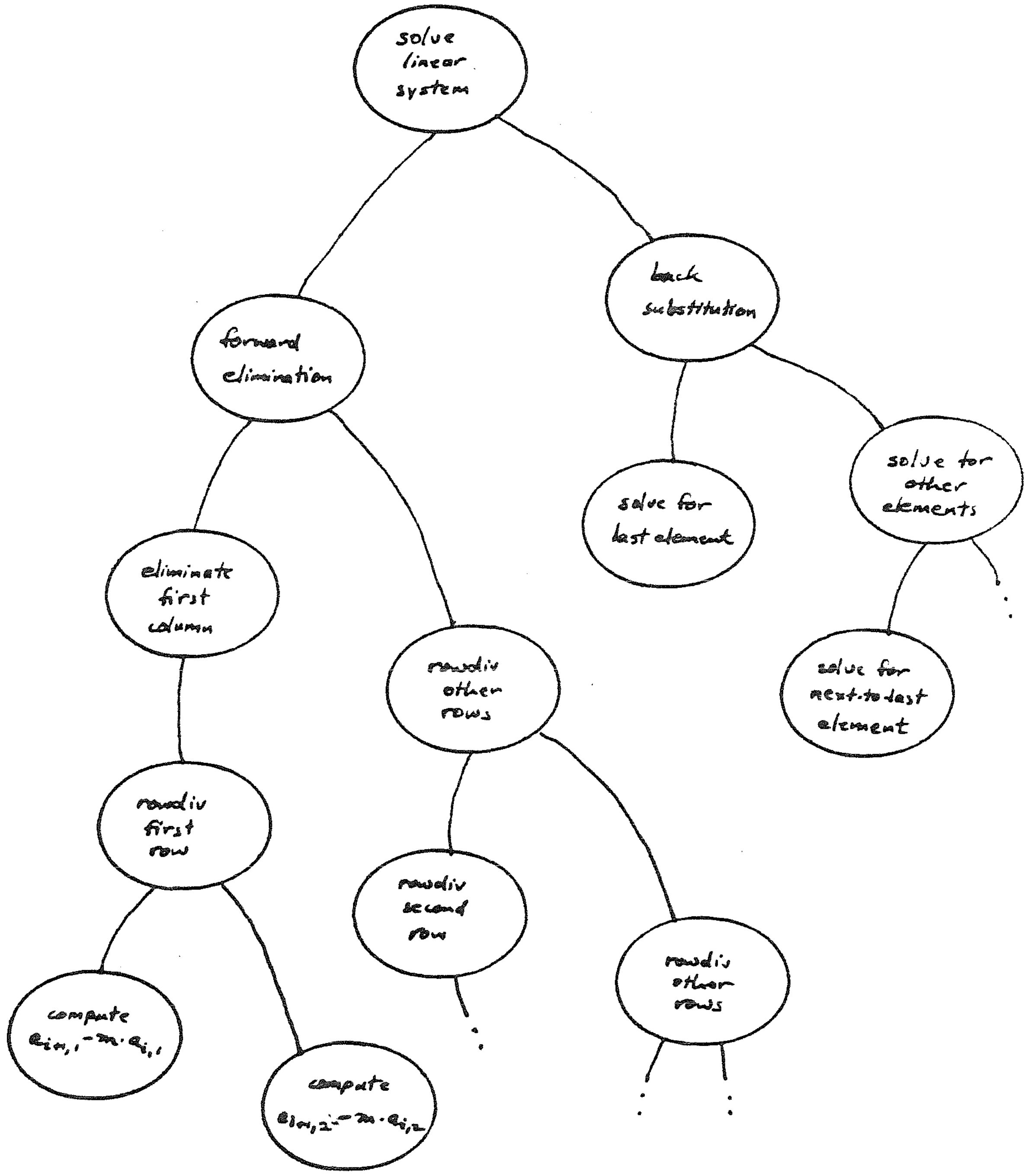
2.1.2 Array machines and non-numerical problems

In applying array machines to numerical problems it is usually possible to make assumptions about how to map problems onto the array. One is generally not afforded this luxury in the course of trying to map non-numerical problems onto array structures. The generality of storage management for non-numerical programming languages such as LISP, Simula, SNOBOL, and to a lesser extent Pascal reflects the needs for a world based on tree structures and records and not arrays. It is not at all clear how tree structures might best be represented in, say, a two-dimensional array machine.

One tree structure I would like to represent is the hierarchy of procedure activations generated by a concurrent evaluation scheme (figure 2.2). When each activation generates two or more new ones, the primary task can request the use of an exponentially increasing number of processors at each step. Machines being limited to the reality of two or three dimensional space, the resulting mayhem is quite interesting to watch.

If resource allocation is handled on a local basis, an individual processing element will attempt to partition out subtasks to its immediate neighbors, some of which may already be busy with work handed out by other processing elements. Communication paths will not have any preferred direction; a node coordinating the activities of its neighbors may finish its work and subsequently become a slave to one of those same neighbors. Activity will spread out and contract as an amorphous blob, with subtasks competing for resources.

An unfortunate property of arrays is that a spreading wave front of activity in a 2-D array will at best encounter a linearly increasing number of elements as it progresses, whereas tree-structured computations can grow exponentially with the distance from the root. Not only can such processes swamp an array very quickly, but many subtasks will be blocked from potential resources. This phenomenon, analagous to pinch-off in semiconductor physics, may result in local shortages in situations of global surplus so as to



Tree of procedure activations

figure 2.2

prevent even large problems from using the full resources of the array. Figure 2.3 illustrates this phenomenon as time progresses for a problem of dimension 5x5x5x5x5 running on a 10x10 array machine.

If the partitioning part of the parallel evaluation scheme is made sensitive to "crowding" by sending new activations some distance away, this problem can be alleviated somewhat, at the cost of using some processors solely for communication. An ideal scenario for such a scheme would be for processing nodes to recognize conflict situations and make big communication jumps on the basis of local information. A demand-driven policy for allocating idle nodes would either have to be based on unselfish attitudes towards limited resources or some centrally managed scheme based on global information.

Both schemes have problems. The former, even assuming honest and civic-minded processing nodes, has the possibility of causing artificial shortages of resources simply by using the same allocation strategy in each node. Contention for resources could even result in deadlock problems. The latter class of schemes, on the other hand, could avoid local shortages in situations of global surplus, but would be handicapped by the need for global communication.

While there might be some strategy to deal with the allocation problem, it seems unnatural to force all programs to map onto the procrustean bed of array machines which seem best adapted for a special class of numerical problems.

```
                                       |
3 - 3 - 3   3   3   3 - 3 - 3   3   3  |  3 - 3 - 3   3   3   3 - 3 - 3 - 5   *
        |   |   |   |           |   |  |          |   |   |   |
3 - 3   2   2 - 2 - 2   3 - 3 - 3   3  |  3 - 3   2   2 - 2 - 2   3 - 3   *   *
    |   |   |           |           |  |      |   |   |           |
3   3 - 2   2   2 - 2 - 2 - 2 - 3 - 3  |  3   3 - 2   2   2 - 2 - 2 - 2 - 3   *
|       |   |   |                      |  |       |   |   |
3 - 3 - 2   1 - 1 - 1 - 2 - 4 - 4 - 4  |  3 - 3 - 2   1 - 1 - 1 - 2 - 3 - 4 - 5
        |   |           |           |  |          |   |       |   |       |   |
4 - 4 - 2 - 1 - 1 - 4   2 - 3 - 3   4  |  4 - 4 - 2 - 1 - 1   4   2   4 - 4   5
                |   |   |       |      |                    |   |   |       |   |
3 - 2 - 2 - 2 - 2   4   2 - 3   3 - 3  |  *   2 - 2 - 2 - 2   4   2 - 3   5   5
|       |   |   |   |   |   |          |          |       |   |   |   |   |
3   3 - 3   3   3   4   2   3 - 3 - 3  |  *   *   3   *   3   4   2   3   5   *
|   |       |   |   |   |              |                    |       |
3   3   3 - 3   3   4   3 - 3 - 3 - 3  |  *   *   *   *   3 - 5   3 - 3 - 3   *
|   |   |       |   |               |  |
3   3   3   3 - 3   5 - 5 - 5 - 5   4  |  *   *   *   *   *   *   *   *   *   *
|   |   |   |   |                      |
4   4   4   4   4 - 4 - 4 - 4 - 5   *  |  *   *   *   *   *   *   *   *   *   *
                                       |
                  a.                   |                  b.
-------------------------------------------------------------------------------
                                       |
*   3 - 3   3   3   3 - 3 - 3   *   *  |  *   *   4 - 3   3   3   *   *   *   5
        |   |   |   |                  |               |   |   |               |
*   3   2   2 - 2 - 2   3   *   *   *  |  *   *   2   2 - 2 - 2   3   *   *   5
    |   |   |           |              |          |   |           |           |
*   3 - 2   2   2 - 2 - 3 - 5 - 5   *  |  *   *   2   2   2 - 2 - 3 - 5 - 5   5
        |   |   |                      |          |   |   |                   |
3 - 3 - 2   1 - 1 - 1 - 2 - 3 - 4   *  |  *   *   2   1 - 1 - 1 - 2 - 3 - 4 - 5
        |   |       |                  |          |   |               |   |
3 - 3 - 2 - 1 - 1   5   *   *   *   *  |  3 - 3 - 2 - 1 - 1 - 5   4 - 3   4 - 5
|           |                          |  |       |   |   |       |       |   |
4   5 - 5 - 5   *   *   *   *   *   *  |  4   4 - 4   5   4   5   4   4 - 4   5
|                                      |  |               |   |   |   |   |   |
4 - 5 - 5   *   *   *   *   *   *   *  |  4 - 5 - 5   5 - 5   4 - 4   5   5   5
|       |                              |  |           |           |   |   |   |
4 - 5   5   *   *   *   *   *   *   *  |  4 - 5   *   5   5 - 5 - 5   5   5   5
|   |                                  |  |   |                   |   |
4   5   *   *   *   *   *   *   *   *  |  4   5   *   *   *   *   *   5   5   *
|                                      |  |                               |
5   *   *   *   *   *   *   *   *   *  |  5 - 5   *   *   *   *   *   *   5   *
                                       |
                  c.                   |                  d.
```

"Pinch-off" phenomenon

figure 2.3

2.2 Tree structures

Almost everything in the physical world, while not resembling trees of the redwood and other kinds, is structured as a tree. People have limbs, which have bones and muscles, which are made of cells. Buildings have floors, rooms, furniture, and so on. Corporations have presidents, vice-presidents, managers, foremen, and factory workers. Except for corporations, which are composed of a collection of structurally very similar people, these hierarchies are heterogeneous. Instead of describing the skeletal structure of humans, one can talk about the hierarchies of various internal organs. For example, kidneys (yecch!) are made of nephrons (ulp!), which are made of cells, which are made of protoplasm (gak!) ... well, you get the idea.

Computers are no exception. The obvious physical hierarchy is that of computer, cabinet, rack, board, integrated circuit, and transistor. Memory systems are beautiful examples of this kind of hierarchy. However, the hierarchy of the memory system is used as a filter for requests _from_ the root, so most of the system is idle. We seek a different hierarchy, similar to that of the corporation, which somehow manages to harness the working energies of many like individuals. The hope is that a processing element can be designed _once_ and replicated indefinitely to achieve a combined computational power on the order of the sum of the powers of the individual processing elements.

The array approach to fulfilling this hope corresponds to setting up a corporation made from a large pool of executives, all of equal standing. Human beings are adept, perhaps uniquely so, at alternating between the roles of giver of orders and receiver of orders [Niven74]. The role of receiver of orders is in some measure made palatable by the identification of a single identity called boss. Deference to the boss is tolerated because there are underlings conditioned to the role of receiver of orders. (If this concept seems tenuous, ask a professor how he would feel about being ordered around by a graduate student.) In the case of the array machine, this approach leads to an uncertain identity of processing elements. No element had a fixed relationship to its neighbors - it might be a slave one instant, a master the next.

A tree machine as it will be discussed here will be a collection of processing elements whose interrelationships correspond to the physical arrangement of the tree. Each node in the tree has exactly one parent, which is also its master, and some number of descendants, which are its slaves. The fixed master-slave relationship between a node and its descendants is a simplification over the amorphous expanding blob described for the array machine. Since each node has only one parent, there can be no competition for control of any given node.

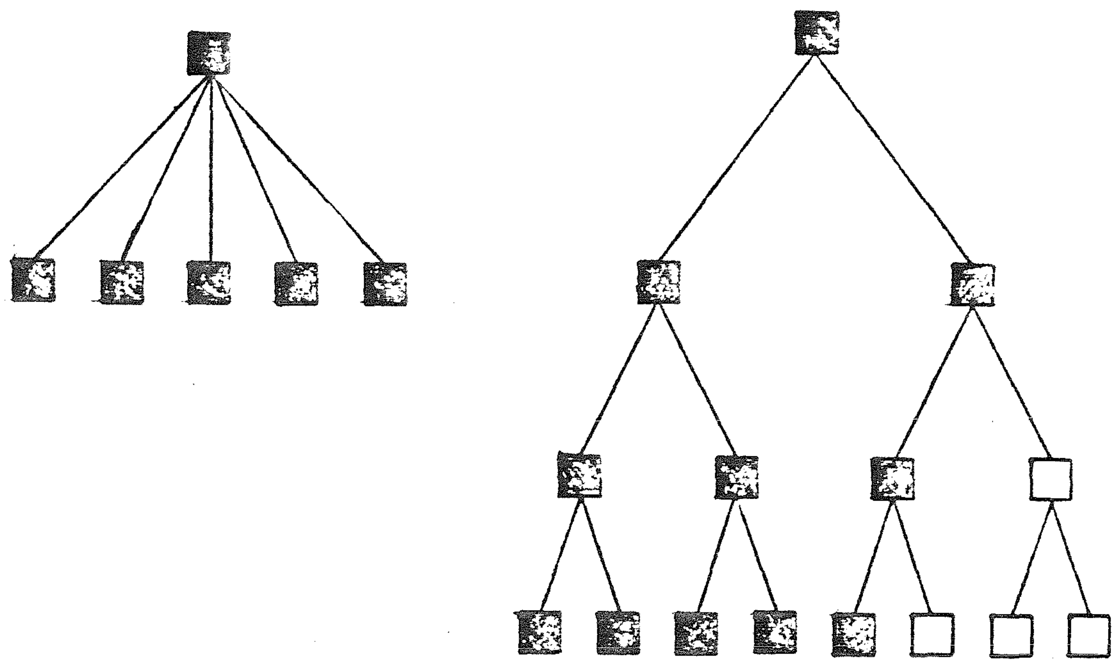
2.2.1 Tree machines and non-numerical problems

Trees fit naturally into the procedure/data structure hierarchy mold of computation with which we are most comfortable. Indeed, the eager evlis mode of evaluation builds up a hierarchy of processes each coordinating the activities of its component subtasks. This arrangement corresponds quite nicely to a physical structure in which each node has exactly one parent and some number of children except at the leaves.

The trouble with this arrangement is that the structure of the machine, although a tree, may not match the tree structure of procedure activations generated by a particular problem. Any resources lost to part of a computation cannot be picked up by another part of that computation because each node has only one possible master - its parent. Whereas one can hope a problem will expand to fill an array machine, a computation which is imbalanced will result in an imbalanced use of nodes in a tree machine.

What choices are there for dealing with this problem in resource allocation? The easiest (and in another sense the hardest) way is to put the burden of allocation on the application programmer. This technique has been explored in programming several algorithms for a machine configured as a binary tree [Browning80]. In these algorithms the binary tree is made to appear as an n-ary tree by using several levels of the tree for communication as in figure 2.4.

A side "benefit" from being forced to allocate resources manually is that the programmer has control over what goes on in the tree. At best this is a mixed blessing; I hear very few people complain about having a garbage collector in a programming system, and fewer still who are used to having a garbage collector around who are willing to return to a system without one. So it will be with manual allocation of processors <u>unless</u> there is a marked penalty in performance from automatic allocation.



Simulation of 5-tree on a binary tree

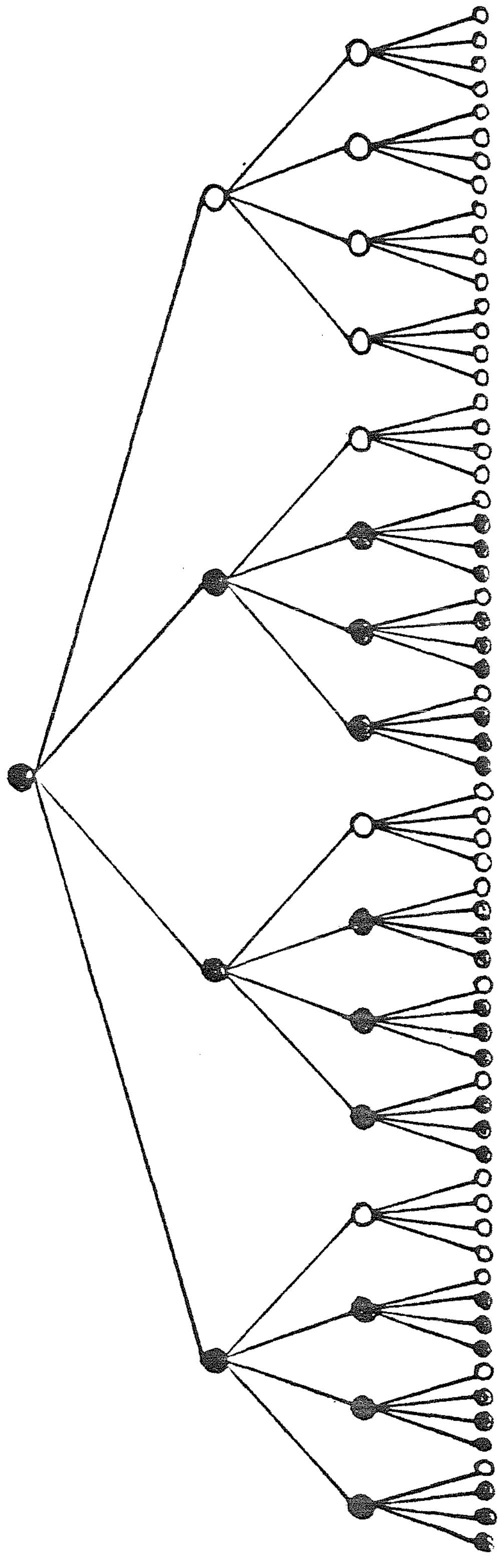
figure 2.4

The concurrent evaluation scheme discussed in chapter one does not take the effort to determine the branching ratio of a problem in advance. A process splits if it can be divided. Beyond that, nothing is known about the potential of the sub-processes for acquiring and exploiting computational resources. Whatever potential there is must wait until it is given resources to exploit. In practice, this cavalier approach to resource allocation clearly would have to be modified in the interest of providing an equitable division of resources.

2.2.2 Exponential waste

As pointed out earlier, tree machines have the marvelous property that an exponentially increasing number of nodes can be made available to a computation for every level of partitioning. Unfortunately, it is also very easy to waste an exponentially increasing number of nodes.

In fact, the situation is worse than one might at first expect. Consider the simulation of a 3-tree on a 4-tree, shown in figure 2.5. Obviously, one fourth of the tree is going to be wasted, right? Look again. One fourth of the _first_ level of the tree is going to be wasted, which takes out one fourth of the _whole_ tree right away. In each successive level, less and less of the machine remains available. By the time the fifth level is reached, there are $3^5 = 243$ nodes in use ... out of $4^5 = 1024$ nodes! In fact, the ratio of wasted nodes to total nodes rapidly approaches unity. (For _n_ large, $(3/4)^n$ is a very small number indeed.)



level	used	unused
1	1/1	0/1
2	3/4	1/4
3	9/16	7/16
4	27/64	37/64

Exponential waste

figure 2.5

This problem of exponential waste can rear its ugly head in any tree machine of branching ratio greater than two. Eradicating the problem completely requires the use of a binary tree and a miserly allocation methodology. That is, a computation whose branching ratio is not a power of two should be given fewer computational resources than it can use.

Suppose for example that an exponential computation of branching ratio three is shoehorned into a binary tree. At the first level of partitioning only two out of a possible three subtasks are initiated and the third must wait for the completion of the other two. This effectively doubles the computation time for the subtasks at that level. Since each level suffers the same degradation, the total slowdown is the product of the individual contributions, or 2^n.

A result of this policy is that exponential computations performed on a tree machine will again take exponential time. However, solving the problem on a tree machine takes exponentially less time than solving it on a single processor machine ($3^n/2^n = (3/2)^n$). The point of view one takes depends on whether the goal is performing computations in the minimum possible time or keeping busy as many processors as can be afforded.

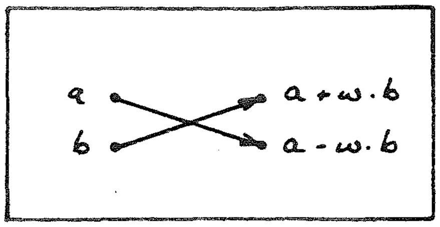
2.2.3 Tree machine and numerical problems

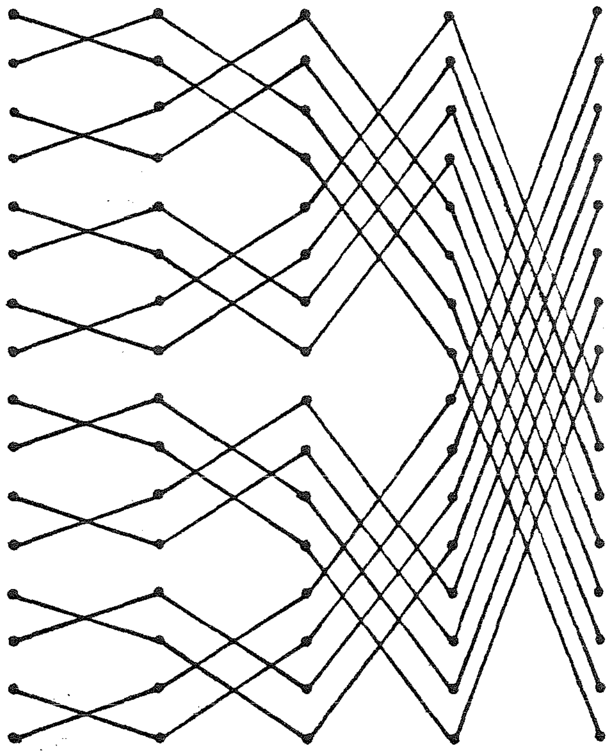
In discussing the merits of array machines I treated the problems of mapping numerical and non-numerical computations separately. In the interest of equal time, I will continue the practice here. However, from the standpoint of a tree machine, an array-structured computation is simply a badly balanced, or list-structured, computation, so there isn't much to say.

One point about numerical computations that I will make concerns their generally physical derivation. In physical systems every element affects every other either directly, as in E & M or potential flow problems, or indirectly, as demonstrated by iterative solutions of field equations. The possibility of each element requiring communication with every other element is just the situation that a tree machine is _designed_ to prevent.

Partitioning computations out in a tree relies on the existence of _independent_ sub-computations that can be evaluated by separate processors. Functional descriptions of numerical computations do manage to maintain this separation. However, the price for this is passing at the time of partitioning _everything_ needed by the sub-computations. This isn't so bad for matrix inversion, where the matrix can at least be divided into a hierarchy of rows, but computations involving matrix multiplication, such as computing a Discrete Fourier Transform, insist that _each_ element of a vector have access to _every_ other element.

Ah, you say, but the Discrete Fourier Transform can be reformulated as the Fast Fourier Transform, a perfect example of a divide-and-conquer algorithm. While computing an FFT takes fewer arithmetic operations than computing a DFT, the effect is still that each element of the input vector has to be combined with every other element through the well-known _butterfly_ operation (figure 2.6). A straightforward divide-and-conquer implementation of the





FFT butterfly computation

figure 2.6

FFT on a tree machine would transmit half of each sub-butterfly over each of $\log_2 n$ levels, yielding a total computation time of O(n). This result is the same for the matrix-vector multiply obtained in the course of solving a linear system of equations [Browning80].

To the extent that numerical problems can be expressed hierarchically, preferably not as hierarchies of shuffles as in the FFT, tree machines can be useful in solving them. The multi-grid method [Brandt77] is a hierarchical interpretation of iterative methods for solving partial differential equations. The interested reader is directed to the beginning of what should be an expanding body of literature.

2.3 Wirability analysis of interconnect structures

This analysis is presented as an attempt to replace some intuitive feelings I've had with some hard analysis. For some time we have at least been aware that there will be penalties in attempting to implement various structures, but we have seen precious little analysis of just what the penalties are. With this analysis I hope to put at least some of the speculation behind us.

Throughout we will be concerned primarily with two things. First, we will seek measures of wiring cost in terms of total length of wire. Secondly, and as a result of the first, we wish to know the effects of wiring on the packing density of structures we build. This, after all, is the bottom line of the cost of wiring - how sparsely must we populate circuit boards and integrated circuits in order to accommodate the interconnect wiring?

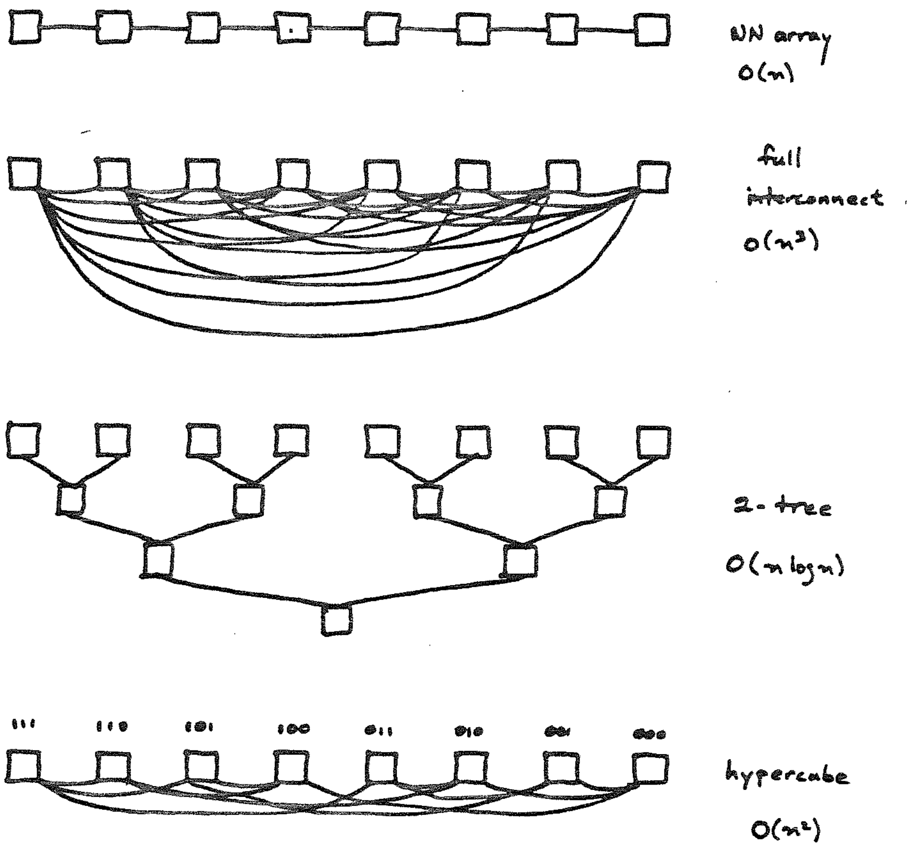
I have selected four examples of particular interest. Nearest neighbor interconnect and full interconnect represent lower and upper bounds of things we might want to consider, and tree interconnect and hypercube interconnect are examples of structures actually proposed for the construction of multiprocessor systems.

2.3.1 Nearest Neighbor Array

The nearest neighbor interconnect structure is presented first because it is both the simplest to build and analyze and the last resort if all others prove infeasible in some way. Analysis is of course trivial, with the wiring cost of order n for n nodes in both 1-space and 2-space. NN array layouts are included in figures 2.7 and 2.8 for completeness and also comparison with other structures.

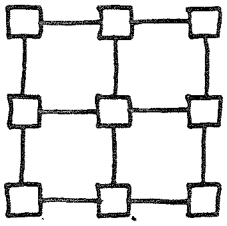
2.3.2 Full Interconnect

Full interconnect represents an upper bound in the sense that it is in principle desirable but in practice unimplementable. There are two facets of difficulty, the first of which is the tremendous _number_ of wires required to achieve full interconnect ($O(n^2)$). There is also the problem of _average wire length_, which must be proportional to the characteristic dimension of the array. To zeroth order the wiring costs are $O(n^3)$ and $O(n^{5/2})$ for 1-space and 2-space, respectively. However, the randomness plus the amount of wiring rapidly influences the packing density and thus increases the wiring estimates dramatically. In short, it is _bad_ (see figures 2.7 and 2.8).

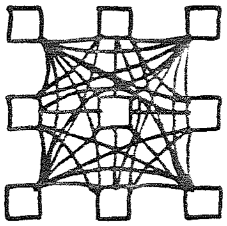


1-dimension

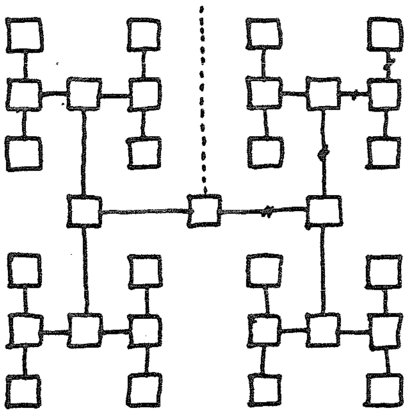
figure 2.7



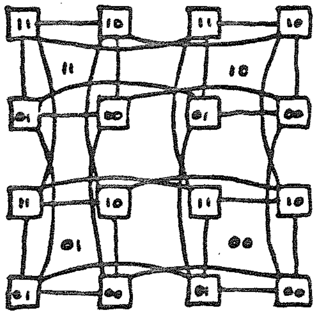
NN array
O(n)



full
interconnect
$O(n^2\sqrt{n})$ for small n



2-tree
O(n)



hypercube
$O(n\sqrt{n})$
[$O(n^2)$ for very large n]

2-dimensions (figure 28)

2.3.3 Trees

For now I will confine the discussion to 2-trees, for which the analysis is particularly simple. In 1-D, a quick glance at the figures should convince the reader that the total wiring cost is

$$\sum_{i}^{\#\text{ levels}} (\#\text{ nodes at level } i) * (\text{av wire lengths at level } i)$$

$$\sum_{i}^{\#\text{ levels}} (n/2^i)\, 2^i = n\log_2 n$$

Another way of looking at this result is that the cost of wiring a tree in 1-D grows faster than the cost of the nodes used in the tree. Hence, the width of the wiring channels will eventually affect the component spacing significantly.

I would calculate exactly when this happens, except this problem goes away in 2-D. As you can see from the figures, the wire lengths need not double for every level. In fact, the figure shows a layout where the wire lengths double every _two_ levels, or

$$\sum_{i}^{\log_2 n} (n/2 + n/4)\ 2^0 + (n/8 + n/16)\ 2^1 + \ldots$$

$$= 3n/2 \sum_{i}^{\log n/2} 1/2^i \;=\; 3n/2\ (1 - 1/n^{1/2})$$

$$= O(n)$$

In other words, 2-tree connectivity has _no influence_ on packing density.

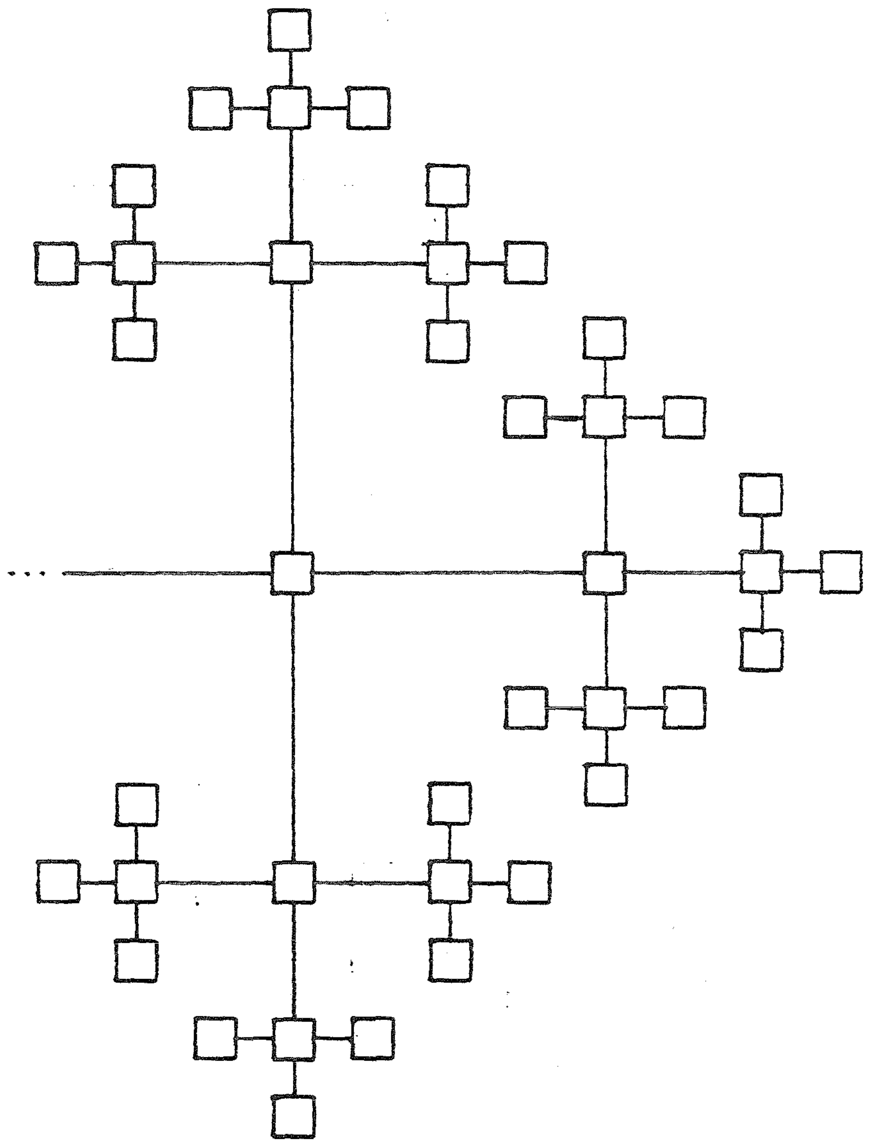
Clearly, 3-space can do the same thing for 4-trees that 2-space does for 2-trees. (Take a look at some TV antennae and you will see what I mean.) However, it is not clear that added dimensionality is mandatory for higher degree trees. For example, figure 2.9 illustrates an arrangement for a 3-tree for which the cost is

$$\sum_{i}^{\log_2 n} (n/3^i)\ 2^i = 3n(1 - (2/3)^{\log_2 n}$$
$$= O(n)$$

In general, the important parameter is the ratio of wire length increase to the branching ratio of the tree. When it is not less than 1, the wiring cost again becomes appreciable.

We are now in a position to investigate the placement of n-trees in 2-space. Picking up from where we left off earlier, we see that the wiring cost for a tree structure is

$$\sum_{i}^{\log_a n} (n/a^i)\ p^i$$



3-tree in 2-space

figure 2.9

where a is the branching ratio of the tree, and p is the ratio of wire lengths as we go from level to level. The form of the result clearly depends on the ratio p/a, for which we have

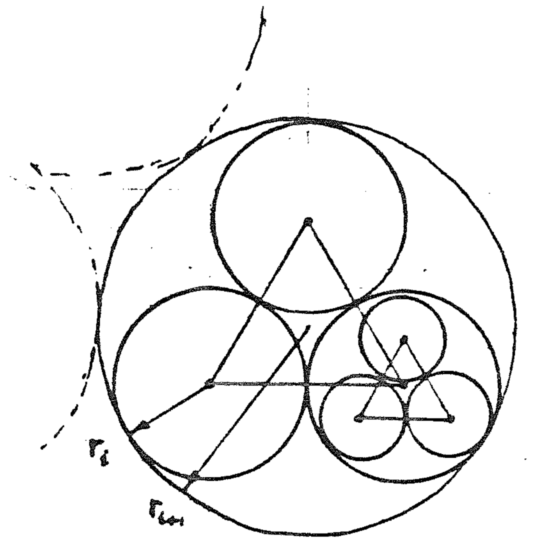
$$\sum_{i}^{\log_a n} (n/a^i)\, r^i = \begin{cases} n \dfrac{1 - (p/a)^{1+\log_a n}}{1 - p/a} \\ \qquad \longrightarrow \; O(n) \text{ for } p/a < 1 \\ n\log_a n \text{ for } p/a = 1 \\ n \dfrac{(p/a)^{1+\log_a n} - 1}{p/a - 1} \\ \qquad \longrightarrow \; O(n^{1+\log(p/a)}) \text{ for } p/a > 1 \end{cases}$$

Clearly we like to have $p/a < 1$. We have seen existence proofs of 2- and 3-trees where this is so, but they were generated in an ad hoc manner. Figure 2.10 shows an attempt at regular constructions for 3- and 4-trees which I will christen <u>recursive stars</u> for want of a better name.

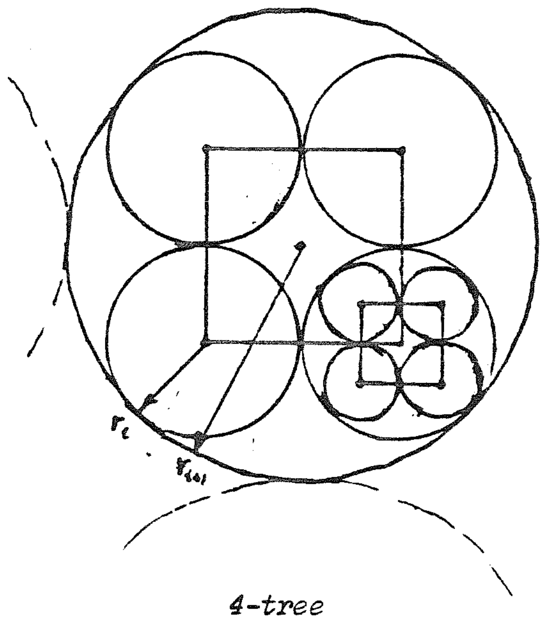
Each level of the star has a characteristic radius r_i which is somewhat larger than the wire length l_i. In fact, we see that $r_{i+1} = l_{i+1} + r_i$. To find $p = r_{i+1}/r_i$ we introduce the recurrence relation

$$r_{i+1} = r_i + r_i d_a$$

where d_a is defined in the construction of figure 2.11. So we have



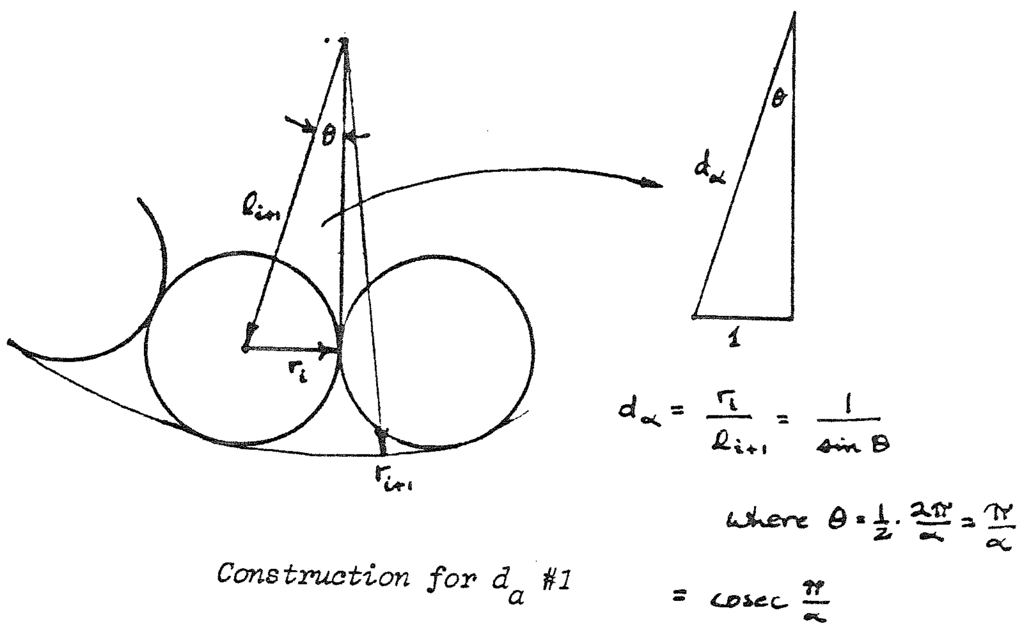
3-tree



4-tree

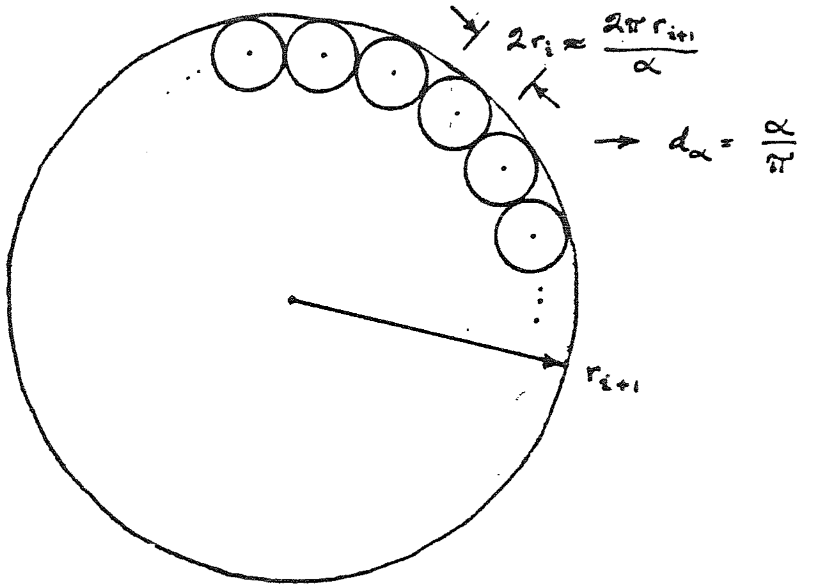
Recursive stars

figure 2.10


θ
ℓ_{i+1}
r_i
r_{i+1}
d_α
θ
1
$d_\alpha = \frac{r_i}{\ell_{i+1}} = \frac{1}{\sin\theta}$
where $\theta = \frac{1}{2}\cdot\frac{2\pi}{\alpha} = \frac{\pi}{\alpha}$
$= \operatorname{cosec}\frac{\pi}{\alpha}$

Construction for d_a #1

figure 2.11


$2r_i \approx \frac{2\pi r_{i+1}}{\alpha}$
$\rightarrow d_\alpha = \frac{\alpha}{\pi}$
r_{i+1}

Construction for d_a #2

figure 2.12

$$r_{i+1} = r_i + r_i \operatorname{cosec}(pi/a)$$

$$p = r_{i+1}/r_i = 1 + 1/\sin(pi/a)$$

and the quantity of consuming interest is

$$p/a = 1/a\ (1 + 1/\sin(pi/a))$$

We can tabulate some values to get an idea of just when this brings trouble. Let's see now,

$$a = 2 \Rightarrow d_a = 1 \Rightarrow p/a = 1/2\ (1 + 1) = 1$$

$$a = 3 \Rightarrow d_a = 2/3^{1/2} \Rightarrow p/a = .72$$

$$a = 4 \Rightarrow d_a = 2^{1/2} \Rightarrow p/a = .6$$

$$a = 6 \Rightarrow d_a = 2 \Rightarrow p/a = 1/3$$

Wait a minute, this was supposed to blow up, right? Well, in the limit of increasing a, sin(pi/a) --> pi/a and we are left with

$$\lim_{a \to \infty} p/a = \lim_{a \to} 1/a\ (1 + 1/(\sin pi/a)) = 1/pi$$

and the cost goes to

$$\lim_{a \to \infty} n \frac{1 - (1/pi)^{\log_a n+1}}{1 - 1/pi} = n\ pi/(pi - 1)$$

Clearly, then, not only is added dimensionality unnecessary for higher degree trees, life gets <u>easier</u> as the branching ratio increases. Looking back at the construction for d_a, the result for large a is rather obvious. In fact, it's the ratio of diameter to circumference! (See also figure 2.12.)

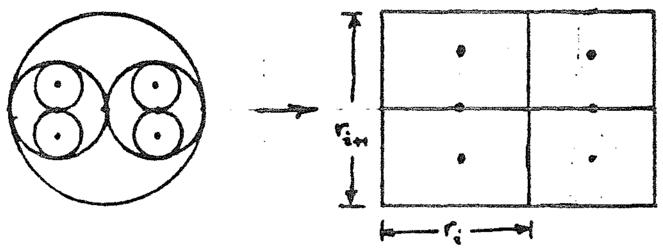
The reader may have noticed that p/a = 1 for a = 2. Fear not - the recursive star approximation is intended to simplify the calculations for non-trivial a. Adapting this stuff to 2-trees is literally the same as squaring off corners in a circle (figure 2.13) so that $p/a = 1/2^{1/2}$ which is similar to the previously obtained result for 2-trees.

2.3.4 Hypercube

Mention hyper-dimensional interconnect and your audience may consider you unfit for life outside your own private dream world. Hypercubes are obviously impossible or expensive to build, so thinking about them is a futile exercise. Or is it? Proponents [Sullivan77] of hypercube systems have brought out several advantages accruing from the generality of hyper-dimensional interconnect, so it seems a worthwhile exercise to at least measure the cost.

The idea behind the hypercube is that each node has an address and is connected to its $\log_2 n$ <u>Hamming neighbors</u>, that is, nodes whose addresses differ by one bit. In this way, any node can communicate with any other node in at most $\log_2 n$ jumps. Also, since there are $O(n\log_2 n)$ communication paths, communication can be arranged so that at any given instant <u>every</u> node is connected to some other node <u>without interference</u>.

To comprehend the connectivity of hypercube interconnect, consider a block of <u>p</u> words of storage interleaved across <u>p</u> different nodes in the hypercube network. Since each node is connected to $\log_2 n$ other nodes, there are $p\log_2 n$ paths into this one block of storage! Managed in this way, hyper-dimensional interconnect gives a new meaning to the concept of shared memory.



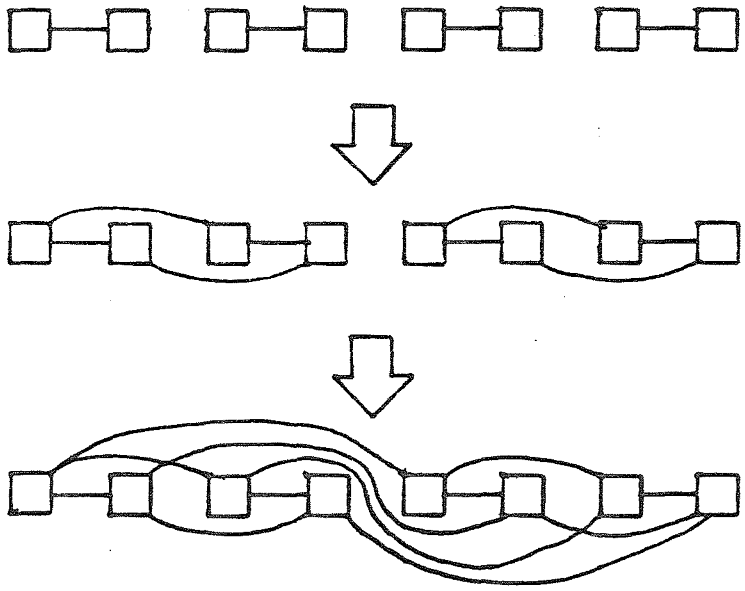
$$\frac{r_{i+1}}{r_i} = \frac{r_i}{r_{i-1}} = \frac{r_i}{r_{i+1}/2}$$

$$\Rightarrow \frac{r_{i+1}}{r_i} = \sqrt{2}$$

$$\Rightarrow \frac{\rho}{\alpha} = \frac{1}{\sqrt{2}}$$

Squaring a circle

figure 2.13



Hierarchical construction of a 1-D hypercube

figure 2.14

Hypercubes map quite nicely into hyperspace. Unfortunately, there is of course no convenient mapping onto real space, so one may expect average wire lengths to be of the order of the characteristic dimension of the layout. Naive analysis of the cost for 1-D would thus bring

$$(\# \text{ of nodes}) * (\# \text{ wires/node}) * (\text{av wire length})$$
$$= n^2 \log_2 n$$

which is nearly as bad as full interconnect at n^3. Remember, this is still not considering the effects of devoting space to wiring channels. For completeness, the naive result for 2-D is $n^{3/2}\log_2 n$, since the characteristic dimension is $n^{1/2}$.

Rather than remain naive, we may choose to calculate actual wire lengths. An obvious mapping of addresses onto 1-D space (figure 2.7) reveals that each of the $\log_2 n$ wires leaving a node has a different length 2^i where i is the bit which is different in the two nodes connected by the wire. The total cost is thus

$$n \sum_{i}^{\log_2 n} 2^i = O(n^2)$$

In 2-space the wire lengths double for every <u>two</u> bits (figure 2.8), so the cost is

$$n \sum_{i}^{\log_2 n} (2^i + 2^i) = O(n^{3/2})$$

which is hardly the horrible toll one would at first intuit. The figures in fact bear this out by their relatively un-busy appearance.

Given that the relative cost of wiring a hypercube increases with its size, we should calculate the impact on the layout due to widening wiring channels. The first item of business is to calculate just when it becomes significant. Clearly, when we consider the effects of wiring channel widths, the characteristic dimension (and thus the wire lengths) is multiplied by $1 + ad$ where a is the "thickness" of a wire relative to a node ($\ll 1$), and d is the number of wires each wiring channel must accomodate in width. For packages on a printed circuit board we may reasonably expect a to be 1/10 to 1/20. For integrated circuit layouts a could easily be as small as 1/100.

We are left with the task of determining d. The reader will first note that the 2-D layout for the hypercube uses only vertical or horizontal wires. Remember, all wires connect nodes whose addresses differ in one bit. Each row and column can thus be considered as a separate 1-D problem, i.e. there is no glut of wires in the center! Consider now the hierarchical construction of a 1-D bypercube shown in figure 2.14. Note the uniform doubling of wiring channel widths along the length of the array. Clearly, d is one half of the characteristic dimension, which in 2-space means

$$d = (n/4)^{1/2} \Rightarrow 1 + ad = (1 + n^{1/2})/32$$

for a convenient value of $a = 1/16$. Thus we need not consider the effects of wiring channel width for PC board layout until $ad = 1/32\ n^{1/2}$ is roughly 1, or $n = 1032$! If we assume $a = 1/64$ for IC layout, we needn't worry until $n = 16,384$!

Recovering from hysterics, we see that the total wiring cost for a hypercube in 2-space is thus

$$n^{3/2} (1 + an^{1/2}/2)$$

which is $O(n^2)$ if n is large enough.

In summary, hypercubes are not nearly as difficult to build as one might expect, although payment must be paid eventually for the generality thus obtained. Whereas both tree and nearest neighbor interconnect incur linear cost increases, the hypercube does get less dense with size. My conclusion is that with tree and array processors one could afford nearly any form of interconnect, including bundles of wires (flat cables), while with hypercube systems one immediately insists on serial communication between nodes.

Chapter Three

Memory bandwidth and contention in multiprocessors

The basic concern of this thesis is to discover how to utilize the many millions of transistors soon to be offered by LSI technology. In chapter one I outlined mechanisms for discovering and distributing concurrency in programs. In the previous chapter I presented some interconnect structures which might prove suitable for such concurrent evaluation. So far no attention has been paid to some fundamental issues of logistics, otherwise known as system design. This chapter addresses just those issues.

Partitioning as a means of extracting concurrencies from a computation produces processes which are conceptually copies of the original computation. The processes are literally snapshots of the original computation as it would appear in the memory of a single processor machine at a specific moment in the execution of the computation. These processes are of course independent of each other in execution, but they are derived from a common origin and ultimately must share access to some global structure.

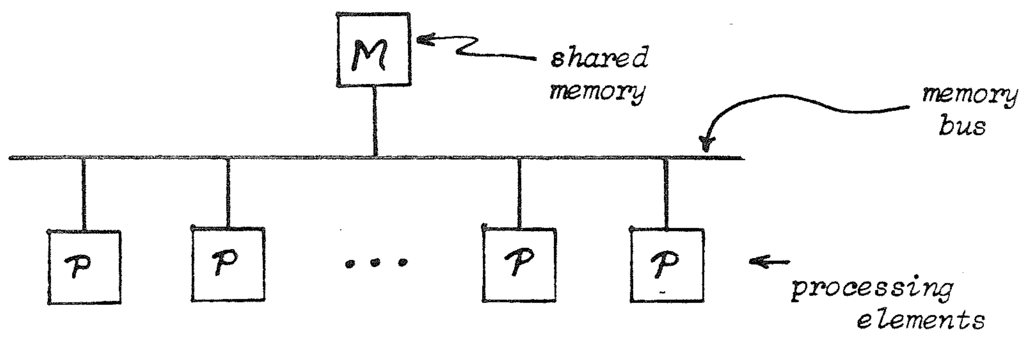
The idea of multiple processors sharing access to common resources such as primary memory brings about a question of balance. Single processor machines are constructed so that the bandwidth of primary memory is roughly the same as the maximum potential bandwidth that the processor can request. What happens to the balance of such a system when several processors are suddenly put in contention for access to a storage unit designed to handle the bandwidth of only one processor? Such a multiprocessor system (figure 3.1) will only perform with the power of a single processor system, possibly even worse because of overhead due to resolving bus conflicts.

Fundamental assertion: Given that any multiprocessor must provide shared access to certain resources, the bandwidth to those resources required per processor must be reduced by roughly the number of processors in the system. _End of fundamental assertion._

This chapter deals with a number of schemes to deal with the bandwidth problem. The first three sections form a progression leading to a tree structured scheme which borrows from existing bandwidth reduction technology. The last section is essentially a diversion - an attempt to extend some ideas learned from the technology and lore of fixed head disks.

3.1 Multi-level memory

The construction of multi-level storage systems is largely motivated by cost. The advent of _cache memory_ has made it possible to build what appears to be a large fast memory for the cost of a large slow memory plus the cost of a small fast memory - a bargain indeed. The concept of _virtual memory_ [Denning70] is even more astounding, considering the tremendous difference in speed between semiconductor memory and the electromechanical activator of a moving head disk drive.



Shared memory multiprocessor
figure 3.1

technology	bandwidth	ratio
cache memory	5 M words/sec	1:1
main memory	1 M words/sec	1:5
disc file (peak)	0.5 M words/sec	1:10
disc file (block)	12.5 - 5 K words/sec	1:400 - 1:1000
disc file (character)	50 - 20 words/sec	1:100,000 - 1:250,000

Bandwidth relationship of devices
figure 3.2

Whatever the economic motivation, a principal design goal of multi-level memory is bandwidth attenuation. While such attenutation is ordinarily applied to allow the use of a slow storage medium, an equally valid application is the sharing of a fast storage medium by several processors. This point should be kept in mind during the following discussion.

3.1.1 Characteristics of storage media

Figure 3.2 compares the relative bandwidths of several kinds of storage. The numbers in the table are taken as a representative sample from the DEC PDP-11 line of computers and are rounded for convenience. Three sets of numbers are given for the disk file to illustrate the difference in bandwidth of various accessing modes.

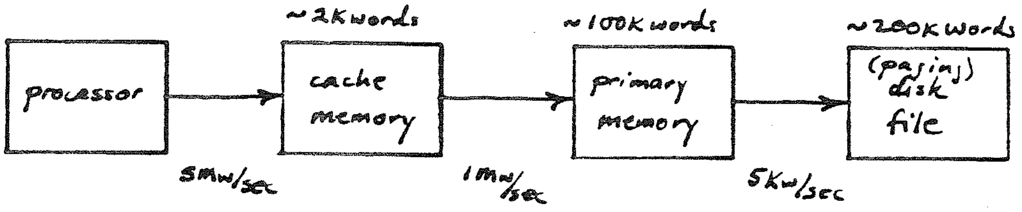
The peak bandwidth of the disk file is no less than the rotational speed of the surfaces times the bit density. At half a million words per second, the disk is half as fast as primary memory. However, in order to realize such a transfer rate it is necessary to transfer more than 20,000 words at a time. Such disk operations are limited to mass transfer operations such as the swapping of core images.

When data is retrieved in blocks (256 words on the PDP-11) the effective bandwidth goes down to between 5 and 12.5 thousand words per second, assuming latencies of 50 and 20 milliseconds. The first case is based on the average access time assuming no correlation between successive disk accesses, while the latter is an observed average transfer time on the UNIX operating system [Ritchie78]. The difference between observed behavior and average expected behavior is usually attributed to a phenomenon called locality of reference. (I call it a phenomenon because it is more easily measured than created.)

Disk blocks retrieved in the operation of a paging system [Kilburn62] are not guaranteed to contain 256 equally useful words of data, useful being defined as likely to be referenced in the very near future. In the worst case, where only one word of a retrieved disk block is ever referenced, the disk becomes a device for accessing individual words and its bandwidth is between 20 and 50 words per second! Again, locality of reference comes to the rescue by assuring that worst case behavior is not approached in practice.

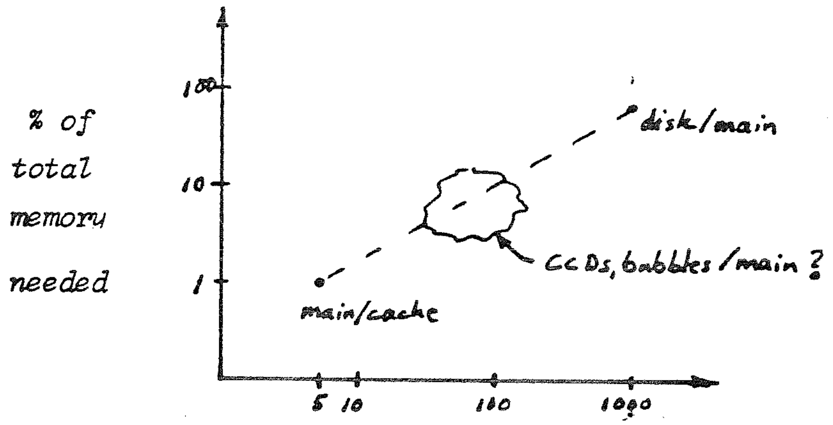
Figure 3.3 illustrates a typical storage hierarchy built from the various storage media of figure 3.2. A fast processor accesses primary memory through a cache buffer, allowing primary memory to be slower than the processor. Primary memory is in turn backed up by a moving arm disk, which is accessed very infrequently due to the extreme bandwidth difference between primary memory and disk files.

The size ratios given in the figure are typical of real systems. Cache sizes of a few thousand words are considered adequate to make a few hundred thousand words of primary memory appear to have nearly the bandwidth of the five times faster cache. Miss ratios for successful operation are typically 5%-15%. On the other hand, the tremendous bandwidth differential of primary memory and disk storage requires that primary memory be large enough to hold nearly all of a running computation. In terms of



Typical multi-level memory system

figure 3.3



bandwidth slowdown

A continuum of cache performance

figure 3.4

miss ratio, only 0.0004% to 0.001% of all memory references can be permitted to result in a page fault. Because of this, moving arm disk files are essentially useless as paging devices [Brinch-Hansen73].

The numbers given here may seem a bit nebulous and imprecise. They are. As can be expected, caching and paging statistics depend heavily on the nature of programs tested. Indeed, variations in the languages used (even their implementations), page replacement algorithms, cache write-through algorithms, etc., can affect the statistics gathered more than the individual computations used for the test. Graphs illustrating caching statistics seldom even have labelled axes! Even those that do [Strecker78] are only applicable to a very restricted class of machines. What conclusions could possibly be made from comparing cache statistics for a Burroughs B6700 performing symbolic integration in LISP to a PDP-11 computing Fast Fourier Transforms in FORTRAN?

<u>Blanket statement:</u> Most caching statistics are phony. <u>End of blanket statement.</u>

For the purposes of this thesis, it will be useful to establish a continuum of required cache size as a function of relative bandwidth. Unfortunately, in addition to the imprecision and variability of existing statistics, there are no statistics gathered for storage devices which are, say, 20-100 times as slow as the processor they serve, since there have been no examples as yet of devices with such characteristics from which to build paging systems.

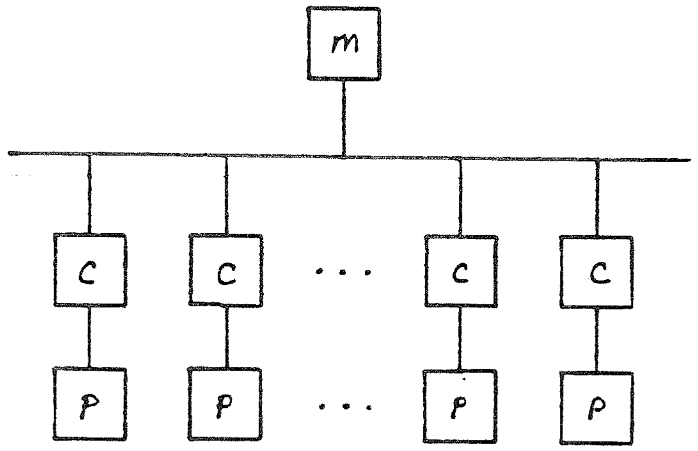
The absence of data for intermediate bandwidth differentials makes it difficult to assume that the difference in performance of cache memories and paging systems is a matter of degree, and that the underlying principles are the same. Nevertheless, I will define a continuum, illustrated in figure 3.4. The shape of the curve, an exponential tail, is suggested by Coffman and Varian [Coffman68] in a discussion of paging systems.

3.2 Shared memory multiprocessor with cache

A cache memory with a miss ratio of 5%-10% reduces the required bandwidth to main memory by an order of magnitude. If instead of buffering access to a slower memory, the cache is used to buffer access to a memory of the same bandwidth as the cache, one would expect to be able to place several processors with cache on the same memory bus (figure 3.5).

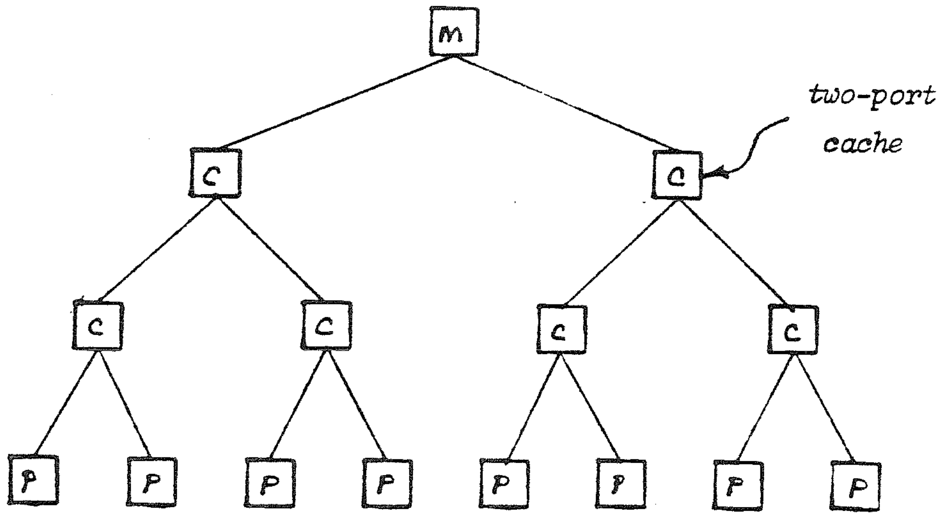
Clearly, using a cache that attenuated bandwidth by a factor of 10 should allow 10 processors to share the same memory comfortably. The existence of paging systems prove that it is feasible to build a cache memory with a miss ratio of 0.01%. Does this mean that 10,000 processors can be made to share the same memory bus simply by building large caches for them?

Ignoring the basically intractable electrical problems of such an arrangement, there are serious problems in accommodating interprocessor communication. That is, if 10,000 processors are forced to share the same bus for



Shared memory multiprocessor with cache

figure 3.5



Multi-level cache

figure 3.6

communication, the bus will be completely saturated. This after all was the point of the previous chapter. However, assume for the moment that the electrical problems are solved and a separate communication network can be provided. What further obstructions are in the way of this scheme?

3.2.1 Write-through

The course of computation can be viewed as a sequence of state vectors, each derived from the previous. The transformation of state is the intended result of having a processor execute a program. Since all state is stored in memory, it follows that some accesses to memory must be to write data.

This is in fact the case, though the statistics governing write behavior are as subject to misrepresentation and error as caching statistics. Consider for example a typical microprogrammed computer, and suppose it writes data on 20% of its memory references under certain conditions. Now suppose the microprogram store is mapped into main memory. Since the microprogram never changes, the percentage of writes will decrease. If internal registers are mapped out, the percentage of writes will increase. If instead of compiling into machine code, a language system relies on another level of interpretation, the percentage of writes will again drop. As a reference point, forcing all writes in a PDP-11 cache to be written through to memory results in a slight change in performance [Bell78], as if a write occurred for about 5% of all memory references. (The reader is left to speculate on the relative power of the PDP-11 instruction set.)

The impact of write statistics on the performance of a shared memory multiprocessor with cache is subtle. Even if writes occur 20% of the time, the performance of a virtual memory system will not degrade significantly. All virtual memory systems mark pages that have been altered with a "dirty bit", which forces the page to be written only when it has been selected for replacement. In this way, writes to memory are buffered in a similar fashion to the way memory reads are buffered.

Unfortunately, a multiprocessor system cannot afford this luxury. Activity initiated in another processor must have access to the same environment from which it was spawned. Most of this environment is shared and cached by both processors, but some of it, namely the parameters computed for the new activity, is different and should be written through to memory for the other processors to read. In other words, each new process will require essentially a flushing of the cache belonging to the processor spawning the process.

The need for periodic cache flushing defeats the write buffering necessary for an extremely low miss ratio cache. Hence, there must be a limit to how many processors can share a memory bus. My own feeling, which is probably about as accurate as most caching statistics, is that the limit lies between 10 and 20 processors - certainly a far cry from the thousands of machines soon to be affordable.

3.2.2 Stale data

This is a problem that keeps system designers from putting even two cached processors on the same memory bus. Consider a location in memory that two processors have accessed recently and hence both have in their caches. Suppose one processor writes into that location. Eventually the new contents of that location will migrate out to memory, but the other processor will never be able to access it. After all, the contents of that location had previously been fetched and stored in cache. Such data, changed elsewhere by another processor, has become _stale_.

Gloat: For systems that allow only functional prgramming, the _stale data problem_ is a _non-problem_. Since stale data can only be created through side effects, a functional programming system will not suffer the stale data problem. _End of gloat._

Implementations of functional programming systems occasionally make use of side effects. The most obvious example is the _cons_ operator, an essential component in the maintenance of the illusion of infinite memory. The implementation of _cons_ generally involves alteration of a _free list_, which is not something that two processors should be doing simultaneously. The obvious answer in this case is to maintain a free list for each processor. Other means of implementing _cons_ in multiprocessor systems will be discussed as part of memory management consideration.

3.3 Multi-level cache and tree machine

Consider a tree-structured application of the concept of multi-level memory as shown in figure 3.6. This arrangement is different from the traditional multi-level storage system in two ways. First, each level of storage is implemented in the same medium, enabling shared access to common storage. Second, each level of storage branches out with the levels of the tree.

In this way, each cache serves both as a filter for requests made to a larger store and as a shared resource for caches lower in the hierarchy. The intent of the tree structure, aside from avoiding the electrical problems of massive bus sharing, is to sufficiently filter memory requests so that the bandwidth available from the root node is enough to permit unimpeded access. How much filtering is sufficient?

Bandwidth condition: If the branching ratio of a level in the tree is $\underline{a}$, then the miss ratio of a cache at that level must be less than or equal to $1/a$. If this condition is always satisfied, then the bandwidth at the root required by the rest of the tree is less than or equal to the maximum bandwidth required by a single processor. End of bandwidth condition.

Attempting to satisfy this condition immediately raises several questions. First, although the required miss ratios are given in relative terms, the cumulative miss ratios required near the root of the tree are quite small. For a cache near the root to filter the required percentage of accesses, it has to have information that caches at lower levels do not have. For example, a cache at the second level of a ten-level binary tree has to be able to function as a cache with a miss ratio of $(1/2)^9$, or 0.2%, to half of the nodes at the leaves of the tree. Does this mean that nodes near the root of the tree have to have a substantial fraction of the storage capacity of the root?

The second question relates to write-through. If the caches are all mapped to memory in the root, is not the bandwidth problem of write-through the same as for the shared memory multiprocessor with cache, that is, intractable?

The last question (that I can think of) concerns the access time through the tree. If it turns out to require large quantities of storage just to satisfy the bandwidth condition, will it require even more to make references to memory appear acceptably fast? Recall that the requirements for miss ratios are relaxed relative to typical cache miss ratio requirements.

As presented this scheme appears to share some of the same difficulties as the scheme of the previous section. As presented so far, this tree structured cache organization says nothing about a possible communications network. The time has come to combine two ideas from separate chapters and describe their interactions. I will answer some of the preceding questions in the course of describing the combination of ideas.

3.3.1 A tree machine with cache

Until now I have treated multi-level caches as passive elements in a tree structure, assuming that all processing happens at the leaves and that requests for data filter up through the tree of multiple port caches. There is no reason to suspect the validity of using the normal caching models for this structure, since each multiple port cache can be thought of as a conglomeration of single port caches. If the bandwidth problems can be resolved as specified, there can be no problems with such a conglomeration sharing communication paths.

Suppose now that this cache hierarchy is overlaid with a hierarchy of processing elements. When evaluating a function, a single processing element in this tree views the world above it as a multi-level cache, and thereby some distribution of data referenced by the element is set up. Now, if the evaluation splits and forms new processes running the node's children, each new process starts out with an empty local memory. Most of the data (and program) needed by the new processes can be found one level up, and the first few moments of execution are spent largely in filling local memory. This cache flushing behavior is similar to that found in cached single processor systems supporting multiprogramming.

In practice, this kind of cache flushing is not thought to be a serious problem, since caches tend to recover quite rapidly. However, if the machine switches context very rapidly, the cache may never reach a state where the ratio of misses to hits remains constant. For a PDP-11, even if a new process executes as few as 300 instruction fetches, the miss ratio for the cache is only degraded to 30%. This gives some measure of how large a process must be before it is worthwhile to partition it to some processing element lower in the hierarchy.

3.3.2 Write-through, locality, and address spaces

In the shared memory multiprocessor, parameters to new processes are transmitted through shared memory. This requires changed data to be written out to shared memory independent of the cache's ability to buffer write access. In a tree machine, computations are parcelled out to descendant processors in the tree. Hence, write-through is not necessary for transmission of parameter information in tree machines.

In fact, except for returning results from functions (a one-level write-through), the concept of write-through is superfluous in the context of a tree machine. The functional programming methodology forbids side-effects, so there is no need to maintain mappings to upper levels for newly generated data. Selective mapping of data, while not strictly in the tradition of cache memory, eliminates the write-through problem altogether.

Selective mapping also provides a means of preserving locality of reference. Consider a function which in recursing upon itself is handed off to a descendan processor. The code for the function body is at least partially stored locally, so a cache miss in the descendant processor will travel only one level up the hierarchy. Similarly, the parameters for the new function are also generated locally, not even mapped to higher levels, so parameter access also requires data transmission across only a single level.

In other words, locality of reference allows some aspects of caching behavior to be considered in relative terms. This means that the problem of maintaining reasonable miss ratios near the root of the tree may not be intractable after all. Cache memories with miss ratios of 50% are certainly very easy to build, especially when the alternative is a 0.2% miss ratio cache.

Consider also the effect on locality of descendant processors performing similar computations. For example, if they are evaluating the same function (with different arguments), the ancestor processor has to fetch the code for the function body only once. This meets the bandwidth criterion exactly.

3.3.3 Multi-level cache performance

One of the questions raised earlier was whether or not the bandwidth requirement of tree structured cache hierarchies is stringent enough to provide the expected performance benefits of cache memory. That is, will access to memory through the cache be roughly as fast as accesses to local memory?

Since all levels of the multi-level cache will use semiconductor storage, and there is no compulsion to use cheaper storage for the larger caches, accessing upper levels of the cache will not incur the extreme access time penalty of single processor cache hierarchies. The first jump, from the internal memory of a node to the memory of the parent node, will mean roughly a factor of 10 degradation in access time. However, having to go <u>two</u> levels up instead of one will not bring another factor of 10, but will only be <u>twice</u> as slow. In a 4-tree, for which we require a miss ratio of no more than 25%, a factor of two increase in access time will not be noticed. In fact, since the penalty for missing at a level gets milder for each step up, the delays associated with misses <u>will not be noticed</u>.

Assuming a 4-tree with a miss ratio of 25% and a delay progression that goes something like (1, 10, 20, 30, 40 ...), the average access time for a given node in the tree will be

$$3/4 + (1/4 * 10) + (1/16 * 20) + (1/64 * 30) + \ldots$$

or roughly a factor of 5 slower than local memory accesses. Note that the first jump contributes the most to this factor. In other words, if the bandwidth requirement is satisfied, the speed of access to memory is bounded, if not entirely satisfactory.

<u>3.4 Broadcast methodologies</u>

The previous sections assumed that bandwidth reduction through demand driven scheduling is desirable, if not unavoidable. However, the handling of the various details of cache management - mapping algorithms, replacement policies, lookahead policies (if any) - can be complicated and time consuming. Perhaps it would be easier to provide raw bandwidth and repeatedly broadcast everything to all processors. Would the results be as good? This section is then an exercise in point of view, and is perhaps best introduced by the example of elevator scheduling.

3.4.1 Elevators

Suppose we have a single elevator serving several floors and lunchtime has arrived. Nobody brought a sack lunch to work, so requests for elevator service come from all floors. Furthermore, service requests are directional and varied because there are several floors of parking structure.

The most obvious (and worst!) strategy is to handle requests in the order they come in, transporting one person or group at a time from floor to floor. To service all requests the elevator must on the average traverse the entire height of the building for each request. The service is intolerably bad, mitigated slightly by the shift of would-be elevator users to the stairs. Unfortunately, most of the stair users had already pushed the call button before turning to the stairs in disgust.

A much better policy is to consolidate several service requests into each trip. This approach does not improve the service to a single elevator user, but service degrades slowly in the face of demand from several users. Quality of service now depends somewhat on the capacity of the elevator and the time spent at each floor picking up and depositing passengers. Considerable thought can be put into this kind of scheduling; it is even possible to find an optimal one given the properties of the elevator and some measure of cost [Knuth73].

An intermediate policy is to send the elevator up and down constantly and have it stop at floors for which service requests point in the direction the elevator is currently traveling. This policy will not provide optimal service in the sense of delivering the most people to the most floors in the least time, but it will provide a lower bound for service as well as a reasonably well controlled expected time. Also, it is by far the simplest policy of the three to implement.

3.4.2 Fixed-head disks

This elevator example also corresponds well to the characteristics of rotating media. Although elevators do have inertia, they have no preferred direction of motion once they are stopped. Moving head disk arms are the same way, which is why the second kind of policy is most often used for moving head disk strategy. But the medium itself rotates, and with considerably more momentum than that of the twirling patches of magnetization we call bits. With several surfaces rotating at 3600 revolutions per minute, a lot of information is whizzing by in each 17 millisecond revolution. In fact, if there is a read/write head for every track of recorded data on the disk, everything you could <u>possibly</u> want to touch passes by in those 17 milliseconds [McMahon79].

This observation forms the basis of a nearly defunct lore of fixed-head disks and their use on paged timesharing systems. Rather than take the point of view that each I/O operation will take place in so much time, the promise can almost be made that all I/O operations presently queued will be <u>completed</u> in a fixed amount of time. The validity of such a promise depends of course on the ability of the system to perform I/O operations on several tracks concurrently, since there will at times be multiple operations scheduled for different tracks in the same sector of the disk.

This prospect is enormously attractive in a timesharing environment where several processes are generating I/O requests and page faults asynchronously. The TENEX [Bobrow72] and BBN LISP [Bobrow67] systems are examples of systems that take good advantage of head-per-track disks in this way. Unfortunately, the days of the fixed head disk are apparently past, and modern timesharing systems can be purchased only with moving head disks.

3.4.3 (Multiple head)-per-track disk

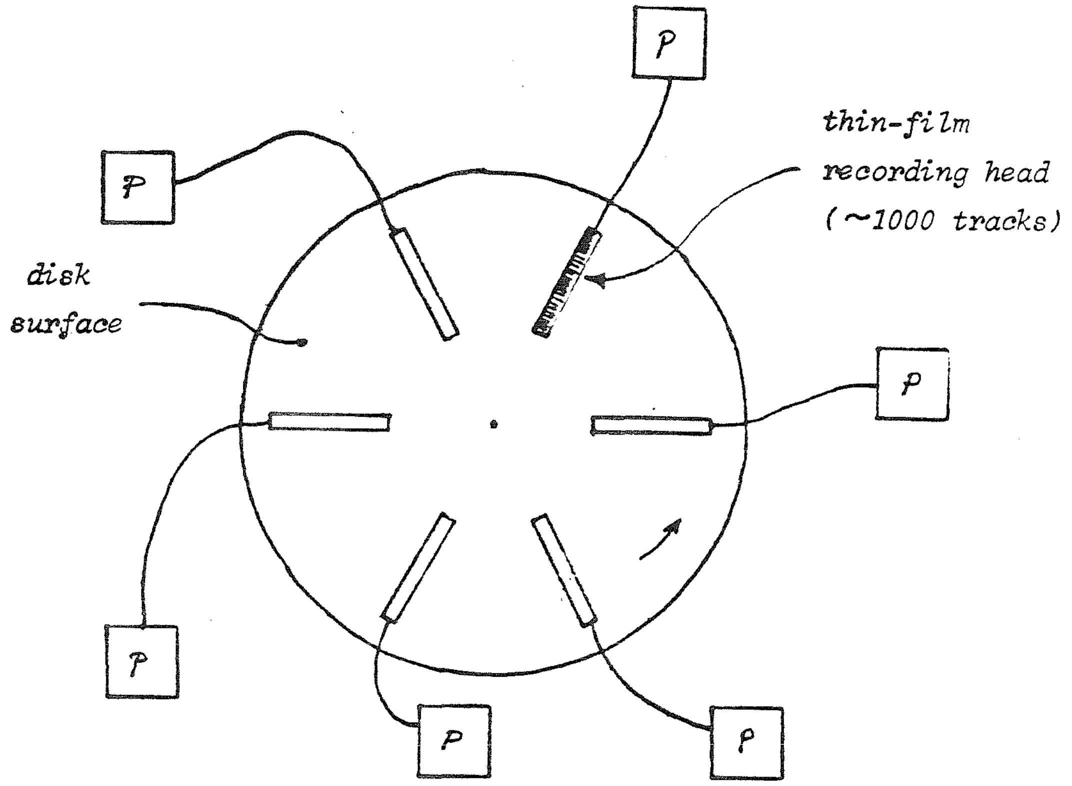
Consider the enormous potential bandwidth of individual read/write heads each hovering over a separate track. This bandwidth is usually thrown away by enabling only one at a time. However, the fact remains that even under a heavy load the maximum access time is only double the average access time. More importantly, every bit on the disk passes under a read/write head for every revolution of the disk.

This marvelous property first came about from the difficulties in building disk arms that moved without machining material from disk surfaces. With the advent of moving arm technology and its ever-increasing track density, the extravagance of installing (and maintaining!) a head for each track became less and less justified and fixed head disks have all but died away.

Today's (1980) densities of thin film technology recording heads are in the hundreds of tracks per inch. While one can hope for a comeback in fixed head disks, another possibility presents itself. Back in the days when disk drives were (relatively) cheap and computers were not (when men were men and giants walked the earth), it made sense to spend a little more to make the disk faster. This situation has very clearly reversed - very few personal computers have hard disk drives. Given the inevitable, that multiple computers will share the same recording medium, why not give each computer its own set of heads? Further, what is to stop us from placing these sets of heads over the same tracks as in figure 3.6?

The advantages are twofold. First, all machines have access to the same storage, and this storage passes by each machine every revolution. If the address space of each machine is mapped to disk then we have achieved the goal of multiple machines sharing memory. In fact, all storage is shared save the local storage in each machine which acts more as a cache than anything else.

The second advantage is that since the heads are physically placed at different locations over the disk surface, the transfer rate to and from each machine is not reduced. Looking at it another way, the bandwidth of the disk is multiplied by the number of machines connected to it. This total shared memory configuration so valuable for multiprocessors comes without the penalty of reduced bandwidth.



(Multiple head)-per-track disk

figure 3.7

Communication between processors is limited by the rotation speed of the disk, so in a very real sense the disk becomes the clock by which partitioning of parallel subtasks takes place. More than anything else, this affects the size of subtasks below which there is no advantage to partitioning. Given the overhead one might expect to incur for partitioning, this may not be a major problem. There also has to be a multiple free list scheme to prevent contention for write access to individual areas on the disk.

I leave it to the reader to construct his own analogies to this kind of system in CCD and Bubble memory technologies.

Chapter Four

Structure of a multi-level LISP system

In previous chapters I have presented schemes for extracting concurrencies from computations, interconnect structures for multiprocessors, and ways of providing adequate memory bandwidth to individual processors in a multiprocessor system. In this section I attempt to combine the results obtained and apply them in a real-world system.

This chapter describes an implementation of a LISP programming environment on a tree machine whose memory operates as a multi-level cache. The purpose of this exercise is to lend a sense of reality to previous discussions. In this regard a LISP system can be brought arbitrarily close to reality; many LISP systems are more operating system than progamming language.

The implementation of a LISP system also brings with it the necessity of solving some very real problems almost universally ignored yet vital to the success of a truly functioning system. For example, the matter of how program code gets to the processors that need to execute it has never to my knowledge been dealt with satisfactorily.

Also, in the course of managing the limited resources of a node in a tree machine, the matter of managing program which might not completely fit is often ignored by treating program code as special. Such problems reflect symptoms of a memory management scheme insufficiently general to handle loading and execution of programs while using only those primitives provided by the programming language. In LISP, program _is_ data, forcing the implementer to face up to the realities of managing program space.

Treating programs as data offers that additional advantage that a scheme that handles data structure well will also handle program storage. This provides a convenient test of a multi-level caching system intended to make access to data appear as fast as the fastest memory in the system. As pointed out earlier, LISP programs can be represented as simply a particular form of list structured data, and the same arguments that apply to data referencing behavior should apply equally well to program reference behavior. If not, somebody is lying and the reader is admonished to start checking for the _floating point syndrome_ which often appears amidst phony caching statistics.

Another aspect of life in the real world which is often overlooked is automatic memory management. Several languages, including Simula [Birtwistle73], APL, SNOBOL, SAIL, and LISP, provide this very useful service to programmers, often at the expense of having their colleagues accusing them of laziness or living in an illusory world. The existence of a garbage collector frees the programmer from the strictures of stack programming, just as stack allocation in Algol and its descendents freed programmers from the strictures of FORTRAN programming.

Pedantic statement: That there still exists a respected body of people who either fail to acknowledge the automatic memory management functions of almost all operating systems or don't realize that their favorite programming language uses dynamic storage for strings, or worse yet feel that confining oneself to the strictures of stack programming is somehow "good" or "virtuous" is a constant source of amazement to me. <u>End of pedantic statement.</u>

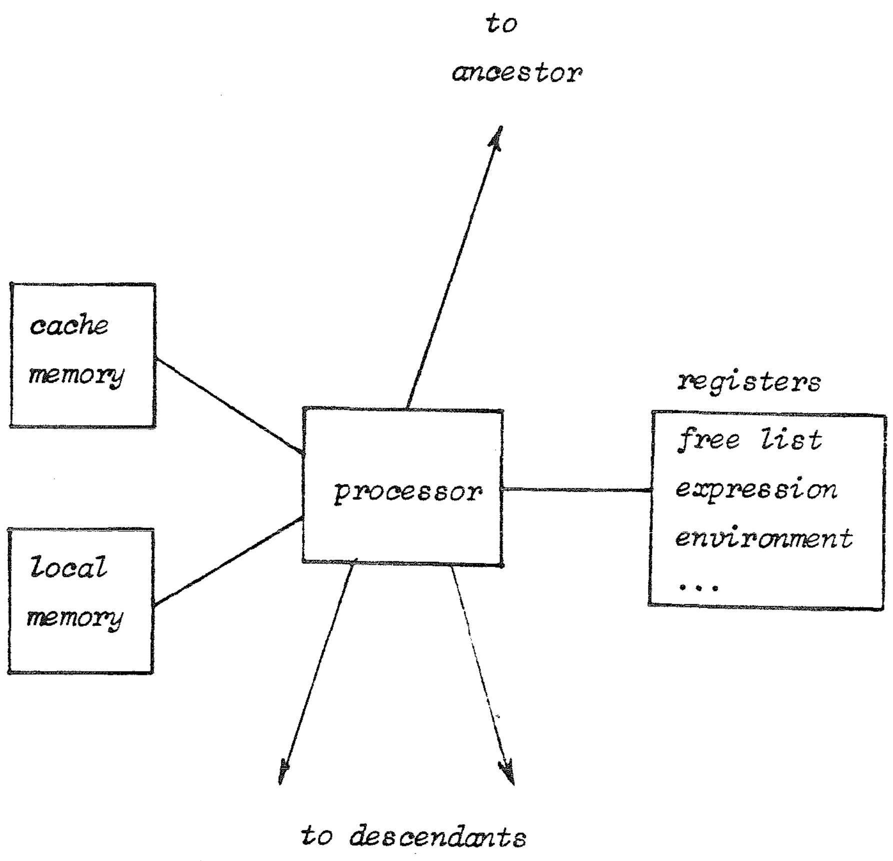
Again, the implementation of a LISP system provides direct exposure to a real world problem whose solution is testimony to the effectiveness (or failure!) of one's ideas. However, because of the uniform structure of list structure, a solution for LISP will not necessarily work for languages (eg Simula) which provide automatic management of variable sized objects [Arnborg72]. As I will point out later, <u>that</u> is a hard problem.

I will thus content myself for now with mapping a LISP environment onto a multi-level system. This task combines some of the realities involved in providing a <u>useable</u> system with the potential of a programming language sufficiently powerful as to be able to write an operating system in it.

<u>4.1 A hierarchy of address spaces</u>

Imagine a multi-level machine in which the address space of each node is a superset of the address space of its immediate ancestor. Each node can thus address everything accessible by its parent, plus more which is accessible only to the node and its descendants.

A suitable configuration for a node in such a machine is given in figure 4.1. Each node is comprised of a processing element, a cache memory through which access to ancestral data is provided, and a local memory which can be written



A node in a multi-level LISP tree machine

figure 4.1

only by the processing element in the node. The processor is connected to its immediate ancestor and descendants in a tree structure.

The sole purpose of the cache is to buffer read accesses to ancestral data. The processor is prohibited from writing in the cache. Hence, the concept of write-through is meaningless in this machine. As discussed in the previous chapter, eliminating write-through removes a potential bandwidth problem and prevents the so-called stale data problem.

Activity in a node is initiated by the parent spawning the evaluation of an S-expression. The node is handed pointers to the expression and the environment for its evaluation, from which the node may access any relevant piece of program or data. Items so referenced are buffered in the cache and are subject to the standard replacement algorithms of cache memory.

In the course of evaluating the expression, the processor in the mode may create new data or new combinations of old data. Space for such data is allocated in the local memory of the node. Note that this data can be made available to any of the node's descendants should pointers to them be passed down in the spawning of an evaluation.

The interesting part of the evaluation process comes when it is time to return the results of the evaluation to the parent node. Since the result is most likely to be created in local memory, which is not addressable by the parent, any local data returned must be mapped and copied into the parent's address space. All pointers to such data also have to be re-created in the parent's address space. This process will be described more fully later.

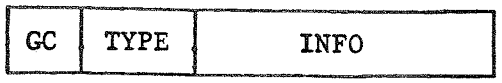
4.2 Data representations

A possible representation for LISP data is illustrated in figure 4.2. All data is typed, allowing type-dependent decisions to be made on inspection. LISP systems have historically dealt with data typing on a very informal basis. For a multi-level system it is important to separate the notions of type and address space.

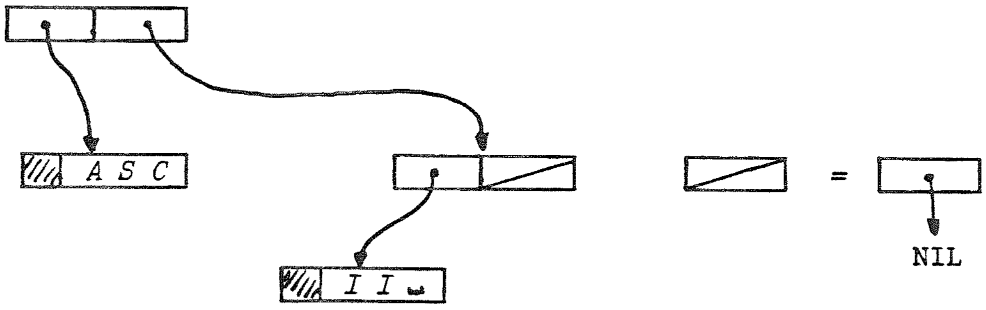
All data objects, or words, are identical in size, consisting of three fields: a _type_ field, an _information_ field, and a small (1 or 2 bit) field for garbage collection purposes. For the sake of convention I will assign 24 bits for the information field, 6 bits for the type field, and 2 bits for the garbage collection (GC) field, a total of 32 bits for each data object.

The primitive data types chosen are _integer_, _character_, _atom_, _pair_, and _free_. In the _free_ data type, the information field is used to hold an address to the next element of a _free list_. All data objects are allocated from the free list and returned after use by the garbage collector.

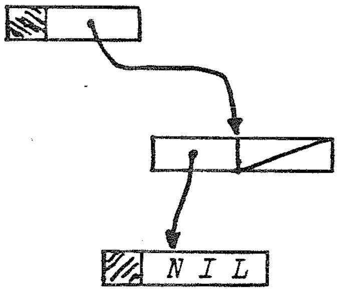
Since LISP pairs have two fields _car_ and _cdr_, a pair is considered to be two consecutive words in storage. The information field of the first word is the _car_ field, while the _cdr_ resides in the information field of the second word



LISP data object
figure 4.2



Representation of "ASCII"
figure 4.3



Representation of NIL
figure 4.4

of the pair. Two free lists are kept in anticipation of having to allocate two words at a time for pairs and one word at a time for everything else.

As a matter of interest, the MIT LISP machine [Greenblatt77] makes use of clever encodings of the type field in order to represent some LISP pairs in a single word. This scheme relies on the existence of a _copying garbage collector_, an assumption I am not prepared to make for this multi-level system.

An _integer_ object is a word containing the _integer_ type field and a signed two's complement integer in the information field.

The _character_ data type uses the information field to store a small number of characters. A 24 bit information field is room enough for 3 ASCII characters. Unused character positions are represented by null bytes.

Strings of indefinite length can be represented as lists of character objects. For example, the string "ASCII" is represented in figure 4.3, occupying six words of storage.

The information field of an _atom_ object is simply an address of a string object. The string object is the _print name_ of the atom and is created when the atom is read in from the keyboard. Note that atoms are created only in the address space of the root node. The atom NIL is shown in figure 4.4.

The representations given were chosen for clarity rather than compactness. Many variations abound in the annals of

LISP culture, each with its own particular advantages. One of the earliest recorded non-standard representations is Deutsch's compact encoding for atoms in PDP-1 LISP [Deutsch64]. The important properties of the representations chosen here are the separate type field and the (nearly) constant size of primitive objects.

Since this is a multi-level system, addresses are broken down into two fields: a short field denoting the _level_ of the tree to which the address refers, and the physical address within the local memory of the ancestor node at that level. Thus, the combination of a 4 bit _level_ field and a 20 bit physical address field is sufficient to identify 2^{20} elements in each of 16 levels of the tree

Note that addresses are given only in terms of the current node and ancestors of the current node. Addresses may be duplicated across various nodes at the same level of the tree without ambiguity, since the address spaces can be differentiated by the physical location of the nodes.

4.3 Memory management and garbage collection

Each node in the tree is responsible for maintaining the validity of pointer data in its own address space, as well as managing the allocation and reclamation of free storage. Because each node is part of a larger whole, it makes sense to adopt certain conventions for behavior in managing memory. These conventions will of course restrict the freedom of action of individual nodes, but they will also allow many kinds of operations to be easily accommodated.

The fundamental property of this multi-level system is the hierarchy of address spaces. In terms of access rights, each node can access its own storage as well as the storage in ancestor nodes. Conversely, local storage in any node is accessible only to that node and its descendants.

4.3.1 Returning results

This situation has its costs. For example, consider a data structure being returned by an evaluation in a descendant node. This data structure contains references to data accessible by that descendant node. Unfortunately, the node receiving the result does not have access to data in the local memory of the descendant node.

Some data created in the descendant can be copied without translation. The representations of integer and character objects are independent of the level of their creation. In copying such objects back, <u>free</u> objects are merely taken from the free list and the data stored in them. Addresses, however, are a different matter, including the addresses of objects whose representations are independent of level.

An object containing an address to another object in the local memory of the descendant cannot be copied literally. A <u>free</u> object can be taken from the free list and given a new type immediately, but the address field cannot be filled in until the data pointed to is copied <u>and its new address known</u>. If this data also points for objects in the descendant's local memory, the process has to be repeated.

Returning results from a spawned function is thus a recursive process. Each attempt to copy an object pointing into the local memory of a descendant initiates another recursion. Attempting to copy a <u>pair</u> can result in two branchings of the recursion. Recursion along such a branch terminates when an <u>integer</u> or <u>character</u> object is copied, or when an address encountered points to an object in the current address space. Such objects cannot have been created or altered by the descendant processor because the descendant processor does not have write access to any memory but its own local memory. Hence, following an address pointing to "old" data cannot lead to any data which is not "old".

Note the significance of the restriction that prevents a node from writing into its cache. If it were possible to splice new data into an existing list, there would be no way of determining at what point to stop a recursive copying operation.

4.3.2 Garbage collection

The process of computation produces results by creating new data structures from previously defined data structures. In LISP, new data structure is stored in objects taken from the free list. By the physical nature of storage devices, the free list must have a finite length. However, one of the primary goals of a LISP system is the maintenance of the illusion of infinite memory. In order to maintain this illusion, unused objects must eventually be recycled to the free list. This is customarily accomplished through the garbage collection process.

Garbage collection is normally initiated when the free list runs out of objects. Without telling the user that he just ran out of memory, the garbage collector reaches out and marks every object touchable by the user's program and data structures. The garbage collector then sweeps through physical memory returning unmarked (and therefore unreachable) objects to the free list and unmarking those objects that got marked in the first phase. Simple.

4.3.2.1 GC scare #1

Garbage collection becomes a more complicated memory management operation if the system on which it runs relies on virtual memory. This is because objects in memory do not become unused in the same order they are used. What starts out as a nicely ordered free list becomes random after several garbage collections, which plays havoc with the paging behavior of virtual memory systems.

Schemes designed to alleviate this problem range from a <u>clever cons</u> [Bobrow67] to full <u>linearization</u> of memory through a copying garbage collection [Baker78]. <u>Clever cons</u> works by separating the free list into a free list for every page in the hope that new data <u>cons</u>ed together will have better than random referencing behavior. Copying garbage collection works simply by copying all referenceable data from one part of memory to another, linearizing references along the way. Both schemes work by attempting to give the paging system what it wants: non-random reference patterns.

The question arises: what does the multi-level cache scheme want? The answer of course is "nothing", since all levels of storage in the system are made from semiconductor memory, objects can be retrieved one at a time nearly as rapidly as in large blocks.

This is very fortunate from the standpoint of garbage collection algorithms, since moving data around in memory is dangerous in a multi-level system. Why? Because objects in the local memory of node may be pointed to from descendant nodes. Rearranging storage in a node would thus cause a lot of caches to suddenly go stale, a situation worth avoiding.

4.3.2.2 GC scare #2

Consider now the marking phase of garbage collection, where all reachable objects are accessed and marked. The implementation of a hierarchy of address spaces certainly gives each node access to a lot of memory. Marking all that memory for each garbage collection in each node of the tree would take a great deal of time.

Also, consider those objects in local memory pointed to from descendant nodes. Can any of these objects become unreachable from local memory and still be pointed to from outside? If so, marking must proceed from the leaves of the tree towards the root. Such forced synchrony of operation would be devastating to the performance of the system as a whole.

Fortunately, all objects reachable from a spawned computation can be traced from the information passed to it at the time of spawning. That is, the expression to evaluate and the environment in which to evaluate it provide all the information necessary to determine what can be referenced from a descendant node. By retaining this information, marking can therefore proceed in a node without concern for what descendant nodes actually do reference.

Looking from the point of view of such a descendant node, it doesn't matter what portions of ancestral storage it can reference. All such references are of course made through the cache. The obvious conclusion is that addresses to non-local memory need not be following during garbage collection!

<u>Astounding observation:</u> Garbage collection in a node of a multi-level LISP system can proceed completely independently of other nodes in the tree. It is therefore impossible to generate a cache miss in the course of performing a garbage collection. <u>End of astounding observation.</u>

This ability to garbage collect without generating the equivalent of page faults is of course not shared by ordinary virtual memory systems. However, to be fair, the multi-level scheme presented has no recourse in the event that a computation exhausts local memory. In providing individual nodes with access to large quantities of storage through the tree, and so allowing individual nodes to have less memory than would otherwise be needed, preventing writes from migrating up the tree hierarchy is an essential part of maintaining independence of operation and reasonable bandwidth to memory. Providing an overflow scheme is an additional complication that may have to be faced in the construction of a multi-level system.

4.4 A simple example

This section illustrates the operation of a multi-level LISP system in the course of evaluating a simple example. That example is the old contrived example of the *mirror* function

```
     mirror[t] = [atom[t] -> t;
                  T -> cons[mirror[cdr[t]],mirror[car[t]]]]
```

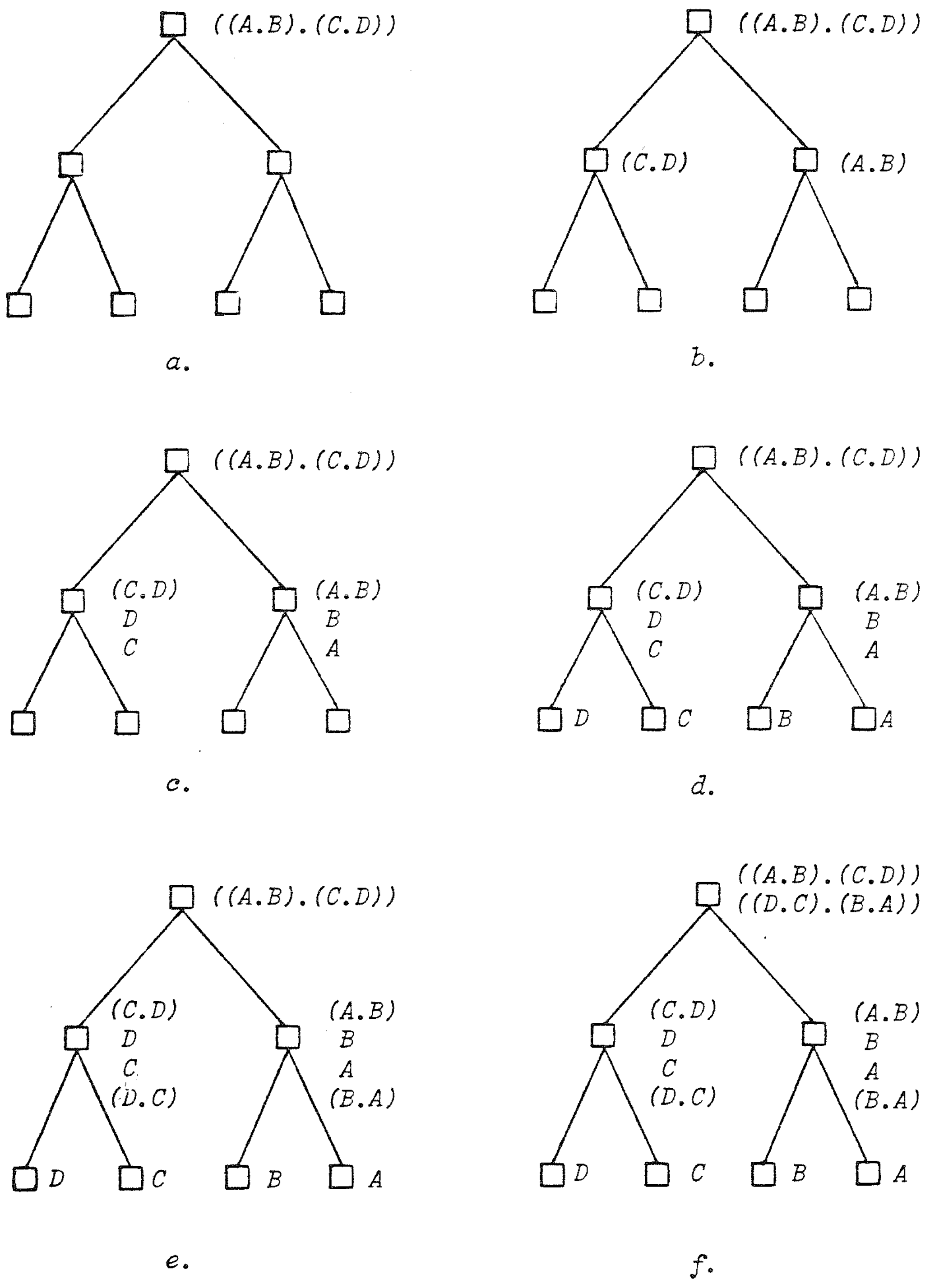
applied to the S-expression

```
     ((A . B) . (C . D))
```

The computation begins with the definition of *mirror* and its argument contained in the memory of the root node (figure 4.5a). *Mirror* determines that its argument is non-atomic and proceeds to *cons* together the results of the mirrored sub-trees. Under *eager evlis* evaluation, the root hands these sub-computations to descendant nodes and waits for the result before actually performing the *cons*.

At this point the descendants have pointers to the function to be evaluated (*mirror*) and the S-expression to which it is to be applied. The left descendant is charged with the task of evaluating mirror[(C . D)], while the right descendant is given mirror[(A . B)] to evaluate (figure 4.5b).

In the course of evaluating the *mirror* function, each of these descendants must bring a copy of the relevant *mirror* code into its cache. Since *mirror* is not an iterative function, and each invocation spawns computations in descendant nodes, all references to parts of *mirror* result



Time history of multi-level cache

figure 4.5

in cache misses and must be retrieved from the parent, in this case the root.

Consider now the left descendant of the root, which is evaluating mirror[(C . D)]. Upon inspection of the pointer to (C . D) it determines that the argument to mirror is non-atomic, and proceeds to set up another pair of evaluations. These evaluations involve the cdr and car of (C . D), so the pointer to (C . D) is followed and the actual pair is cached (figure 4.5c). At this point the node can give pointers to D and C to its descendants along with pointers to mirror (figure 4.5d). A similar sequence of events happens n the right descendant of the root evaluating mirror[(A . B)].

There are now four evaluations of mirror[t] initiated in which the arguments t are atomic. The tests are made, caching the code for mirror from one level up in the tree, until the leaf nodes discover that their arguments are indeed atomic, and they must do something their parents had never done. Until this point, the part of

```
atom[t] -> t;
```

that returns t has never been executed. Assuming no special attention has been paid to managing program segments as such, a series of cache misses ensues, with the requisite code being brought down from the root.

Once the leaves find out that in fact their arguments (D, C, B, and A) are their results, pointers to the appropriate atoms are returned and are consed together by the nodes in the second level of the tree (figure 4.5e). Since the newly created cons pairs (D . C) and (B . A) are in fact the results of the mirror evaluations at this level, they must be returned to the root. Because local data is not mapped to a parent's address space, the new pairs are re-created in the address space of the root. The root in turn conses these results together to return the answer

((D . C) . (B . A))

as shown in figure 4.5f.

An obvious question in light of this example is that of the observed cache miss ratio. It seems as if virtually all acesses to data in each node below the root result in cache misses. Obviously, this does not auger well for the performance of a multi-level cache system.

The retort to this question is of course, "How could it be otherwise?" The computation necessarily begins with exactly one copy of the program and the initial data, most of which is eventually distributed throughout the tree. The operation of the multi-level cache in this example is analogous to the behind-the-scene activity necessary for a system embodying the philosophy of letting the compiler handle everything. In the worst case, the multi-level cache operates as a recursive loader, at all times satisfying the bandwidth condition since two requests for the same item generate only one cache miss.

For computations which are less of a "one-shot" nature, such as matrix inversion, one would expect certain functions such as inner product to migrate to the leaves of the tree for repeated invocation during the computation. Needless to say, cache memory works best in situations like these where "the past predicts the future".

4.5 Summary and critique

In this thesis I have attempted to treat the problems of notation, interconnect, and resouce management individually. In the first chapter I introduced several mechanisms whereby concurrencies in LISP programs can be detected and exploited easily. In the second chapter I showed how various kinds of computations might be expected to map onto physical structure, and presented a short analysis of the wirability of several interconnect structures. In the third chapter I addressed system design issues from the standpoint of maintaining acceptable bandwidth for each computing element, resulting in the development of the concept of the multi-level cache.

In this the final chapter I have brought the results of these discussions together in the description of a multi-level LISP system. This description is complete in the sense that very little is left to the imagination regarding its construction. In fact, given an appropriate collection of microprocessors, support circuits, and memory, the system described could be built today.

The multi-level LISP system presented here has the potential of being able to involve many computing elements usefully in the performance of general-purpose computations. Whether or not this universally sought after goal is achieved depends among other things on whether the underlying principles of cache memory really apply favorably across many levels.

To be sure, the design attempts to take advantage of locality in computations. (In fact, I don't know how it could be more so.) This design is in keeping with the trends in semiconductor technology mentioned in the introduction, which promise to reward locality handsomely as the technology progresses.

The irony of this presentation is that the vehicle chosen for searching out the benefits of locality is the LISP programming language, whose reputation for generality is matched only by its reputation for non-locality. LISP, the bane of many a paged timesharing system, used in a demonstration of locality in design!

4.5.1 What could be

A natural question at this point is whether or not the ideas presented in the implementation of a multi-level LISP system apply to other programming notations. Some of the schemes presented, such as lenient cons, are indeed very specific to LISP, and their applications to computation in general are perhaps not immediately clear. Could this thesis then be construed as an ultimatum from the LISP world?

Certainly not. Aside from the material in chapter one, most of the discussion leading up to this chapter has been divorced from the specifics of any programming notation. If a concurrent evaluation scheme were developed for, say, Pascal, then all of the discussion of interconnect structures and multi-level caches would apply.

In particular, the successful operation of the multi-level cache would still depend on the absence of side-effects. If partitioning in Pascal is keyed on procedure calls as with the eager evlis mechanism for LISP, the standard Pascal scope rules would have to be modified and the var parameter passing mode eliminated. In addition, assignment to local variables accessible from parameters passed to descendant nodes would somehow have to be prohibited.

Also, since results returned from Pascal procedures can only be of simple types or pointers to records, structures returned must be mapped and copied in a manner similar to that described for list structures in LISP. Such considerations beg the question of storage management in Pascal, which is very much of the do-it-yourself variety.

In short, it would not be easy to adapt Pascal for concurrent execution on a multi-level machine. However, other functional notations [Burge75] [Backus78] [Turner79] have much in common with LISP, and one might expect implementation on a multi-level machine to be similar in complexity to that of LISP.

LISP of course has many properties which, if not essential to the success of the ideas presented, are at leeast useful in illustrating mechanisms for handling generality. The existence of an internal form for LISP programs ("program is data") exposes and solves logistical questions usually avoided in discussions of concurrent machines.

The constant size of primitive objects in LISP allows for a general storage management scheme whose operation is not affected by the inability to move objects in storage. The use of variable-sized objects coupled with a need for garbage collection, as in a Simula system, forces the use of copying algorithms in storage management. As I pointed out earlier, such algorithms are not suitable for use in multi-level systems.

4.5.2 What isn't

LISP is used in this thesis as an example of a self-sufficient environment which is known to be useful, and I am comfortable with the implementation described. Yet the view taken by LISP, indeed by all of the FBAPP languages and most functional notations, is that of a centrally defined demon operating on passive objects. The multi-level system described spreads the demon over a hierarchy of active elements, yet the data remains passive.

The view taken by the Simula programming language is more consistent with the real world. Each instance of a Simula <u>class</u>, while possibly owned by another object, is nonetheless the master of its own internal state. Objects in the real world have handles, buttons, and furry paws through which the external world is interfaced. No centrally located demon controls the pop-up mechanism of a toaster or the digestion processes of a small kitten. Those are matters for the internals of toasters and small kittens.

Objects in the real world are long-lived. When a man cuts himself with a razor, no new instance of the man plus a cut is created and the old one thrown away. You can meet someone you've never seen or heard of before. All of these observations are alien to the LISP world of functional expression, yet are consistent with the viewpoint of Simula.

I don't know what a multiprocessor Simula system would look like, but I am sure it would not resemble a multi-level LISP system. For example, a basic precept of cache memory is that program precedes data. That is, a data object can be fetched only if there is a program to request it. When the program goes away, the data loses all of its meaning.

Suppose an object is left in some descendant node of the tree. It can't be touched, because all pointers point <u>up</u>. If it is left dormant, later to be accompanied by some program which gives it meaning, there is no way of finding the object.

In a Simula system, pointers would point <u>down</u>.

It is easy enough to imagine a system comprised of individual function blocks, each a <u>class</u> object capable of influencing others. How can behavior be shared, as for multiple instances of a class? Perhaps the idea of shared behavior is superfluous, since each instance of a small kitten has its own copy of everything a small kitten should have.

In short, there is much work to do.

Appendix

An eager evaluator

The interpreter presented here combines several concepts from the first chapter. Incorporated are an explicit lenient cons and an explicit eager evaluation scheme in which the functions of pairlis and eager evlis are combined. As before, eval and apply form the main body of the interpreter.

```
    eval[e,a] = [atom[e] -> cdr[assoc[e,a]];
                 atom[car[e]] ->
                    [eq[car[e],QUOTE] -> cadr[e];
                     eq[car[e],COND] -> evcon[cdr[e],a];
                     eq[car[e],LCONS] -> lcons[cadr[e],caddr[e],a]
                     eq[car[e],EAGER] -> lapply[cadr[e],cddr[e],a]
                     T -> apply[car[e],evlis[cdr[e],a],a]];
                 T -> apply[car[e],evlis[cdr[e],a]a]]

    apply[fn,x,a] =
        [atom[fn] ->
            [eq[fn,CONS] -> cons[car[x],cadr[x]];
             eq[fn,CAR] -> caar[x];
             eq[fn,CDR] -> cdar[x];
             eq[fn,EQ] -> eq[car[x],cadr[x]];
             eq[fn,ATOM] -> atom[car[x]];
             T -> apply[eval[fn,a],x,a]];
         eq[car[fn],LAMBDA] -> eval[caddr[fn],
                                    pairlis[cadr[fn],x,a]]
         T -> apply[eval[fn,a],x,a]]
```

In this interpreter, lcons performs a symmetric lenient cons function:

```
    lcons[x,y,a] = [pair p;
                    p.car := MAIL;
                    p.cdr := MAIL;
                    carspawn[p,x,a];
                    cdrspawn[p,y,a];
                    return p]
```

where carspawn and cdrspawn are defined in separate processors as

```
     carspawn[p,x,a] := [p.car := eval[x,a]]

     cdrspawn[p,y,a] := [p.cdr := eval[y,a]]
```

This requires both car and cdr to be suspicious:

```
     car[x] = [while x.car == MAIL do [];
               return x.car]

     cdr[x] = [while x.cdr == MAIL do [];
               return x.cdr]
```

If the user specifies eager evaluation, eval passes control to lapply rather than apply. Lapply is different in that it does not expect the argument list x to be evaluated:

```
     lapply[fn,x,a] =
         [atom[fn] ->
             [eq[fn,CONS] -> cons[eval[car[x],a],
                                  eval[cadr[x],a]];
              eq[fn,CAR] -> eval[caar[x],a];
              eq[fn,CDR] -> eval[cdar[x],a];
              eq[fn,EQ] -> eq[eval[car[x],a],
                              eval[cadr[x],a]];
              eq[fn,ATOM] -> atom[eval[car[x],a]];
              T -> lapply[eval[fn,a],x,a]];
          eq[car[fn],LAMBDA] -> eval[caddr[fn],
                                     pairevlis[cadr[fn],x,a]]
          T -> lapply[eval[fn,a],x,a]]
```

where pairevlis is defined to perform the functions of pairlis and eager evlis:

```
     pairevlis[x,y,a] =
         [null[x] -> NIL;
          eager[] ->
             [pair p;
              p.cdr := MAIL;
              spawn[p,cdr[x],cdr[y],a];
              p.car := cons[car[x],eval[car[y],a]];
              return p];
          T -> cons[cons[car[x],eval[car[y],a]],
                    pairevlis[cdr[x],cdr[y],a]]]

      spawn[p,x,y,a] = [p.cdr := pairevlis[x,y,a]]
```

As I said before, it isn't pretty. The reader is invited to formulate his own, prettier, evaluator.

References

"This bibliography is presented both to acknowledge the great debt of the author to the work of others, and to share with the reader a compendium of provocative reading." [Kay67]

[Acton70]
Forman S. Acton
Numerical Methods That (Mostly) Work
Harper & Row, 1970

[Allen78]
John Allen
Anatomy of LISP
McGraw-Hill, 1978

[Arnborg72]
Stefan Arnborg
"Storage Administration in a Virtual Memory Simula System"
BIT 12:9, 1972, p. 125-141

[Backus78]
John Backus
"Can Programming be Liberated from the vonNeumann Style? A Functional Style and its Algebra of Programs"
Comm ACM 21:8, August 1978, p. 613-641

[Baker78]
Henry G. Baker, Jr.
"List Processing in Real Time on a Serial Computer"
Comm ACM 21:4, April 1978, p. 280-294

[Barnes68]
George H. Barnes et al
"The ILLIAC IV computer"
IEEE Trans Comp 17:8, August 1968, p. 746-757

[Bell78]
C. G. Bell, J. C. Mudge, and J. McNamara
Computer Engineering
Digital Press, 1978

[Berkeley64]
Edmund C. Berkeley
"LISP - A Simple Introduction"
in <u>The Programming Language LISP: Its Operation and Applications</u>
MIT Press, 1964, p. 1-49

[Birtwistle73]
Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard
<u>Simula BEGIN</u>
Petrocelli/Charter, 1973

[Bobrow67]
Daniel G. Bobrow and Daniel L. Murphy
"Structure of a LISP System Using Two-Level Storage"
<u>Comm ACM</u> 10:3, March 1967, p. 155-157
155-159.

[Bobrow68]
Daniel G. Bobrow
"Storage Management in LISP"
in <u>Symbol Manipulation Languages</u>
North Holland, 1968, p. 291-301

[Bobrow72]
Daniel G. Bobrow
"TENEX, a Paged Time Sharing System for the PDP-10"
<u>Comm ACM</u> 15:3, March 1972, p. 135-143

[Brandt77]
Achi Brandt
"Multi-Level Adaptive Solutions to Boundary-Value Problems"
<u>Mathematics of Computation</u> 31:4, April 1977, p. 333-390

[Brinch-Hansen73]
Per Brinch-Hansen
<u>Operating System Principles</u>
Prentice Hall, 1973

[Browning79a]
Sally A. Browning
"Computations on a Tree of Processors"
<u>Caltech VLSI Conference Proceedings</u>
California Institute of Technology, 1979, p. 453-478

[Browning80]
Sally A. Browning
The Tree Machine: A Highly Concurrent Computing Environment
PhD Dissertation, Computer Science Dept.
California Institute of Technology, 1980

[Burge75]
W. H. Burge
Recursive Programming Techniques
Addison-Wesley, 1975

[Byte79]
Special issue on LISP
BYTE Magazine 4:8, August 1979

[Church41]
Alonzo Church
The Calculi of Lambda-Conversion
Princeton University Press, 1941

[Clark77]
Douglas W. Clark and C. Cordell Green
"An Emprical Study of List Structure in Lisp"
Comm ACM 20:2, February 1977,p. 78-87

[Coffman68]
E. G. Coffman and L. C. Varian
"Further experimental data on the behavior of programs in a paging environment"
Comm ACM 6:7, July 1968, p. 396-408

[Cohen79]
Danny Cohen and Vance Tyree
"VLSI System for SAR Processing"
Caltech VLSI Conference Proceedings
California Institute of Technology, 1979, p. 151-172

[Davis78]
A. L. Davis
"Data Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems"
Utah Technical Report UUCS-78-108
University of Utah, October 1978

[Denning70]
Peter J. Denning
"Virtual Memory"
Computing Surveys 2:3, September 1970 p. 153-189

[Deutsch64]
L. Peter Deutsch
"The LISP Implementation for the PDP-1 Computer"
in _The Programming Language LISP: Its Operation and Applications_
MIT Press, 1964, p. 326-375

[Elson73]
Mark Elson
Concepts of Programming Languages
Science Research Associates, 1973, p. 213-227

[Fenichel69]
Robert R. Fenichel and Jerome C. Yochelson
"A LISP Garbage Collector for Virtual-Memory Computer Systems"
Comm ACM 12:11, November 1969, p. 611-678

[Friedman76]
D. Friedman and D. Wise
"CONS Should Not Evaluate its Arguments"
Proc 3rd Int Colloq on Automata, Languages, and Programming
Edinburgh University Press, July 1976, p. 257-284

[Gibson67]
D. Gibson
"Considerations in block-oriented systems design"
AFIPS Conference Proceedings 30, Spring 1967, p. 75-80

[Greenblatt77]
Richard Greenblatt
LISP Machine Progress Report memo 444
MIT AI Lab, August 1974

[Hansen69]
Wilfred J. Hansen
"Compact List Representation: Definition, Garbage Collection, and System Implementation"
Comm ACM 12:9, September 1969 p. 499-507

[Heinlein66]
Robert A. Heinlein
The Moon is a Harsh Mistress
Berkely Bantam Books, 1966

[Hewitt77]
Carl Hewitt
"Viewing Control Structures and Patterns of Passing Messages"
Artificial Intelligence 8, 1977, p. 323-364

[Isaacson66]
Eugene Isaacson, Herbert Bishop Keller
Analysis of Numerical Methods
John Wiley & Sons, Inc., New York, 1966

[Kay67]
Alan C. Kay
The Reactive Engine
PhD Dissertation, Computer Science Dept.
University of Utah, 1967

[Keller78]
Robert M. Keller, Gary Lindstrom, Suhas Patil
"An Architecture for a Loosely-Coupled Parallel Processor"
Utah Technical Report UUCS-78-105
University of Utah, October 1978

[Kilburn62]
T. Kilburn
"One-level storage system"
IRE Trans Elec Comp 2:4, April 1962, p. 223-235

[Knuth68]
Donald E. Knuth
Fundamental Algorithms
Addison Wesley, 1968, p. 420-422

[Knuth73]
Donald E. Knuth
Sorting and Searching
Addison Wesley, 1973, p. 357-360

[Kung79]
H.T. Kung and Charles E. Leiserson
"Algorithms for VLSI Processor Arrays"
in [Mead79], p. 271-292

[Lewis78]
H. R. Lewis and C. H. Papadimitriou
"The Efficieny of Algorithms"
Scientific American 238:1, January 1978, p. 96-109

[McCarthy65]
John McCarthy
LISP 1.5 Programmer's Manual
MIT Press, 1965

[McMahon79]
E. McMahon and J. Carson
"It's not a very big book..."
The Tonight Show
National Broadcasting Company, 1979

[Mead79]
Carver A. Mead and Lynn Conway
Introduction to VLSI Systems
Addison-Wesley, 1979

[Niven74]
Larry Niven and Jerry Pournelle
The Mote in God's Eye
Pocket Books, 1974

[NASA78]
"Future Computer Requirements for
Computational Aerodynamics"
NASA Conference Publication 2032, 1978

[Pease77]
M. Pease
"The Indirect Binary N-Cube Microprocessor Array"
IEEE Trans Comp 26:5, May 1977, p. 458-473

[Ritchie78]
Dennis M. Ritchie and Ken Thompson
"The UNIX Time-Sharing System"
The Bell System Technical Journal 57:6,
Part 2 (Special issue on UNIX Time-Sharing System)
July-August 1978, p. 1905-1930

[Seitz79]
"Self-timed VLSI Systems"
Caltech VLSI Conference Proceedings
California Institute of Technology, 1979, p. 345-356

[Steele75]
Guy Lewis Steele, Jr.
"Multiprocessing Compactifying Garbage Collection"
Comm ACM 18:9, September 1975, p. 495-507

[Strecker78]
William D. Strecker
"Cache Memories for PDP-11 Family Computers"
in [Bell78] p. 263-268

[Sullivan77]
Sullivan
"... Boolean N-Cube ..."
Columbia University, 1977

[Sutherland77]
Ivan E. Sutherland and Carver A. Mead
"Microelectronics and Computer Science"
Scientific American 237:3, September 1977, p. 210-228

[Teitelman74]
Warren Teitelman
Interlisp Reference Manual
Bolt, Beranek & Newman, Xerox Corporation, 1974

[Turner79]
D. A. Turner
"A New Implementation Technique for
Applicative Languages"
Software Practice and Experience 9:1,
January 1979, p. 31-49