FFP machine support for language extensions

David Middleton

Bruce T. Smith

FFP machine support for language extensions

David Middleton

Bruce Smith

University of North Carolina at Chapel Hill

Abstract

An important characteristic of the FFP machine is its ability to create an autonomous virtual computer for each of the reducible applications in the machine. This work investigates creating these independent computers for purposes other than evaluating primitive FFP function applications; these purposes include supporting features available in other functional languages and improving the parallelism available to tasks.

Introduction

The Functional Programming languages of Backus [Ba78] provide obvious parallelism in their model of program execution. Function applications can be nested and are evaluated from the inside outwards. Innermost applications may be evaluated in any order, possibly at the same time, and hence are called ***reducible applications*** or RAs. The FFP machine [MM84] is a parallel, language-directed computer based on the Formal Functional Programming language, which is a low level language for implementing FP languages. The machine creates, for each of the RA's it contains, an isolated network of processors dedicated to evaluating that single RA. During execution, the machine continually creates new networks to match the changing innermost applications.

We are interested in exploiting the creation of independent virtual computers to obtain parallelism in more ways than just simultaneously evaluating RAs. The simplest use is evaluating a single RA using more than one virtual computer. There is a tradeoff

because while the computers do not interfere with each other's operation, they also cannot communicate. Next, we describe a stream-like mechanism, in which a composition of functions to be applied to a sequence of values is re-organized so that later functions can begin operation before earlier functions have finished generating a complete result. The third mechanism proposed for the FFP machine is a non-deterministic choice operation which allows innermost applications to be preëmpted by external events. The innermost reduction rule in FP yields eager rather than lazy evaluation, and an important capability of lazy evaluation is the opportunity to search for an answer with a number of procedures, some of which may not terminate.

Extensions to FP languages may affect both the semantics of FP and its algebra of programs. None of these proposals are concerned with such FP language issues. Rather, these are facilities which the FFP machine can provide with no fundamental change to the hardware design.

1. Partitioning the FFP machine

The FFP machine consists of a linear array of processors called *L cells* into which an FFP program is written as a string of symbols. This *L array* is connected by a tree of communication nodes called *T cells*. One aspect of the parallelism in the FFP machine is that it creates for each **RA** a separate *virtual computer*, that is, an isolated subset of the physical machine resources. The FFP machine operates in a three phase machine cycle. First the machine is decomposed into separate virtual computers. Next, the virtual computers generate decriptions of the FFP structures they contain, receive microprogram segments appropriate to their FFP functions and operate independently to evaluate the **RA**s. Then the program text is shifted through the L-array to provide empty space where it is needed.

At any stage of execution, FFP programs can contain a number of subcomputations ready to be performed. The virtual computers allocated from the physical machine are conceptually linear groups of processors connected by a communication network. The virtual and physical networks are designed around different goals: the virtual networks should provide high throughput and low contention, and the physical network should provide physical closeness and low engineering complexity. These goals are not independent; the physical machine must be able to re-allocate its resources rapidly to the virtual computers as the subcomputations they are to support change during execution. As a consequence of this constraint, tree networks have been chosen for both the virtual computers and the physical machine. In general, the two networks need not be identical, and the two types of trees can be seen to be independent in Figure 1. The virtual trees are not aligned with the physical tree hierarchy, and more than one virtual tree may involve a particular T cell.

The evaluation rule for FP allows for simple recognition of the RAs. An *area* is a string of text not containing any parentheses other than its enclosing ones. Because non-innermost applications are broken into many areas, separated by the RAs, some areas may not have a parenthesis at each end. An innermost application is an area which does have a matching pair of parentheses. The physical machine partitions itself so that each area is identified with its own virtual computer; those virtual computers associated with areas that are not innermost applications do not perform any computation. Those T cells that occur between the L cells holding an area (in an inorder traversal of the physical tree), contribute a node to the virtual tree supporting that area. Connections between these virtual nodes may exist in the physical tree outside this inorder relationship.

A T cell is concerned with at most three areas, as shown in Figure 2. A T cell views the machine as three pieces reached through its left child, its right child, and its parent, and it is only concerned with those areas lying in more than one of these pieces. Areas entirely contained within one child's subtree or the outside machinery do not involve this

cell. Figure 3 shows the four possible patterns for areas in the L array than can involve a particular T cell, along with the corresponding configurations of the T cell for these cases. The T cell configurations are accomplished by the setting of two channel switches connecting the cell to its parent. A more detailed explanation is given in [MM84], but the correlation can be seen between a subtree containing a parenthesis and the setting of the channel switch on that side. Partitioning is accomplished in the FFP machine, by each subtree, starting with the individual L cells, sending a one bit message meaning "My leaf cells hold a parenthesis". Each T cell sets its two channel switches on the basis of the two bits it receives, and sends the logical sum to its parent. A further two bits are sent from each subtree describing the outside parentheses in that subtree, in order to determine which areas contain RAs. In the current FFP machine, an L cell generates its one bit message by examining the FFP symbols that it contains. In order to create virtual computers for other purposes, an L cell needs to generate the messages using different information.

There are two kinds of costs associated with continually creating virtual computers to match the changing programs during execution: the cost of accomplishing the partitioning, and the costs of the resulting system. The hardware required for partitioning is negligible, and, in isolation, the time required to perform it is logarithmic in the size of the machine, However, since each T cell is configured upon receiving its partitioning messages, it can immediately participate in its virtual computers. This pipelining results in an apparently constant time cost of three gate delays for the machine to accomplish partitioning. The fundamental benefit of partitioning is that the virtual computers are isolated. Thus, in contrast with the majority of tree-structured parallel computers, there is no bottleneck at the root of the physical machine, and very little contention of any kind between separate computations. Within an area this contention at the root of a virtual computer remains. The choice of a tree for a virtual computer network sacrifices low contention in situations

where that would be useful, to better satisfy other constraints. As a result, communication between processors evaluating FP primitive applications takes time which is linear in the number of messages transferred.

Partitioning takes effectively constant time to create isolated virtual computers which can then perform independent computations simultaneously. The simplicity of this operation suggests that isolated (and therefore non-communicating) computers may be exploited for other purposes.

2. Using many virtual computers to reduce one application

The FFP machine creates a disjoint machine for each available subtask. These virtual computers do not interfere, but this is at the cost of their not being able to communicate. This does not hinder the evaluation of FFP programs, since RAs are independent. This lack of interference suggests the possibility of evaluating a single RA using more than one virtual computer, so long as they can be given tasks that are be isolated. As a simple example, we consider grid operations of the sort used in the method of finite differences by Pargas [Pa82]. Each point in a rectangular grid has an associated value that is to be updated on the basis of the values in its four neighbors, shown in Figure 4. In the current FFP machine, this communication is performed in a single tree-connected virtual computer, as shown in Figure 5a. In an $n \times n$ grid, it costs $O(n^2)$ time to move the messages through the root of the virtual computer. Figure 5 (parts b, c and d) shows an alternative scheme in which three sets of virtual computers, used consecutively, accomplish the same communication in only $O(n)$ time. In the first stage, each row is placed in an isolated machine that handles the communication between horizontal neighbors. In the second stage, each even row and the single row below is placed in an isolated machine. Points in even rows can now communicate with their southern neighbors, and points in odd rows can communicate with their neighbor to the north. In the third stage, each even row is grouped with the row above to finish the data movement. During these three stages, the

RA is supported by O(n) virtual computers, each handling O(n) messages, instead of one virtual computer handling O(n^2) messages.

This creation of extra virtual computers is accomplished by inserting *fake parentheses* during the evaluation of the RA, and delaying the operation of the microprogram until the next physical machine cycle. That is, the L cells send up a partitioning message, not on the basis of the FFP parenthesis symbols they contain, but rather on the basis of their position within the internal structure of the RA. These fake parentheses are shown in Figure 5 (parts b, c and d) as hollow parentheses. No other changes are required; the loading of microprogram segments, and generation of local structure descriptions are not repeated.

3. A Stream-like mechanism

Lazy evaluation means computing only part of a result and seeing whether that is sufficient for the situation. Some parallelism may be gained by allowing this partial result to be used, while more of the result is being computed. This is in contrast with eager evaluation in FFP where all results must be completely evaluated before any part of them may be used. The following mechanism provides this form of parallelism, and still allows the enclosing FFP programs to receive constant objects as operands. A framework is built up in which an assembly-line of user provided functions operate on individual elements of a sequence. The final sequence accumulates until the stream computation has completed, and then passes as a completely evaluated object to the surrounding FFP program. This conveyor-belt model matches well to the machine structure consisting of a linear array of processors. The user-provided functions are required to behave in an incremental fashion.

The basic units of the assembly-line are called *filters*, reflecting the sense of modifying values as they pass by. The user supplies for each filter, a transition function and an initial state. The filter applies the transition function to an initial state and input value, to yield

an output value and a new state. Since a filter might remove values from a stream, the values passed along will be sequences of zero or more elements of the stream. In Figure 6b, a sequence of virtual computers are applying the filters to the objects on their right. The result, shown in Figure 6c, is a sequence of object-filter pairs. (The primes show that both the objects and the internal states of the filters may have changed.) This apparent movement of objects past filters is accomplished locally.

At some level, an FFP program wants to begin a stream computation and receive a fully evaluated result. The stream shown in Figure 6 is controlled by a *terminator*. Once the stream computation is begun, the terminator repeatedly regroups values and re-applies the component filters. This continues until a predicate, also provided by the user, indicates that the computation should terminate. At this point the terminator deletes itself, the filters, and any further partial results, and returns a fully evaluated result.

This mechanism is powerful enough to accomplish operations like the Sieve of Eratosthenes implemented with filters and generators in KRC/SASL by Turner [Tu82]. Each filter would remove values from the stream that were divisible by a number it kept internally. The terminator, on receiving a value, would save it in an accumulating sequence of primes and also start up a new filter with this number as its internal gauge, as shown in Figure 6f. The predicate inside the terminator might count the number of primes set, and upon reaching some limit, prompt the terminator to finish.

4. A Non-deterministic choice mechanism

It is often desirable to obtain results from a set of function applications, others of which may not terminate. One example arises in searching for a value in a tree or graph. A left to right depth-first search may encounter an infinite branch of a tree, missing the desired value if it occurs to the right of that branch. The choice mechanism proposed here resembles the "alternative" command of Hoare's CSP [Hoar78]. Given a sequence

of function applications, it returns a subsequence of the values of those that successfully terminate. There is no guarantee that all successful function applications will be included. Two non-deterministic choice functional forms are natural for FP: one resembling the *construct* functional form, the other resembling the *apply-to-all* functional form.

Figure 7a shows a possible execution trace for a computation involving FP's construct functional form. The function **g** depends on the values returned by all four f_i functions on the argument **x**. In this case, $(f_1\ x)$ and $(f_4\ x)$ yield non-terminating computations, and so the values returned by $(f_2\ x)$ and $(f_3\ x)$, 5 and 6 respectively, can not be used. For the the function **h** shown in Figure 7b, any of the four values would suffice. Evaluating the expressions $(f_i\ x)$ one at a time, however, will not, in general, help. In this case, searching either from left to right or from right to left will encounter a non-terminating, inner-most application.

The **ND** functional form in Figure 7b shows one way of implementing a non-deterministic choice. Like construct in Figure 7a, **ND** creates four independent, inner-most applications. Each, however, is surrounded by an application of **ND´** , and all four RAs are surrounded by a pair of hollow braces. When $(f_2\ x)$ and $(f_3\ x)$ terminate, **ND´** places a pair of outward-pointing hollow braces around each value. During a special machine cycle (in the box), these braces become fake parentheses, like the ones described in section 2, and create virtual computers whose task is to erase the virtual computers evaluating The regular parentheses, which would normally prevent the evaluations of $(f_1\ x)$ and $(f_4\ x)$ from being preëmpted, do not participate in partitioning in this special "non-determinism cycle". During partitioning in normal machine cycles, on the other hand, the hollow braces are treated as ordinary FFP symbols, and ignored.

To prevent the whole computation from being erased during an earlier non-determinism cycle, the fake parentheses generated by the outer pair of hollow braces cannot match each other. The rule for matching, during this cycle, is that at least one parenthesis must be

a "prime". Also, if applications of **ND** are to be nested, then the fake parentheses must also be tagged with a level number. The simplest answer to when non-determinism cycles should occur is to have them occur with some fixed period, say every 100 machine cycles. For this reason, it is not necessarily true that the values returned will be from the first function evaluations to terminate.

Conclusions

This work has shown some of the possibilities for exploiting reconfigurability in the FFP machine. Its ability to rapidly create independent networks of processors allows it to support efficiently a number of mechanisms that have been proposed for various functional languages.

References

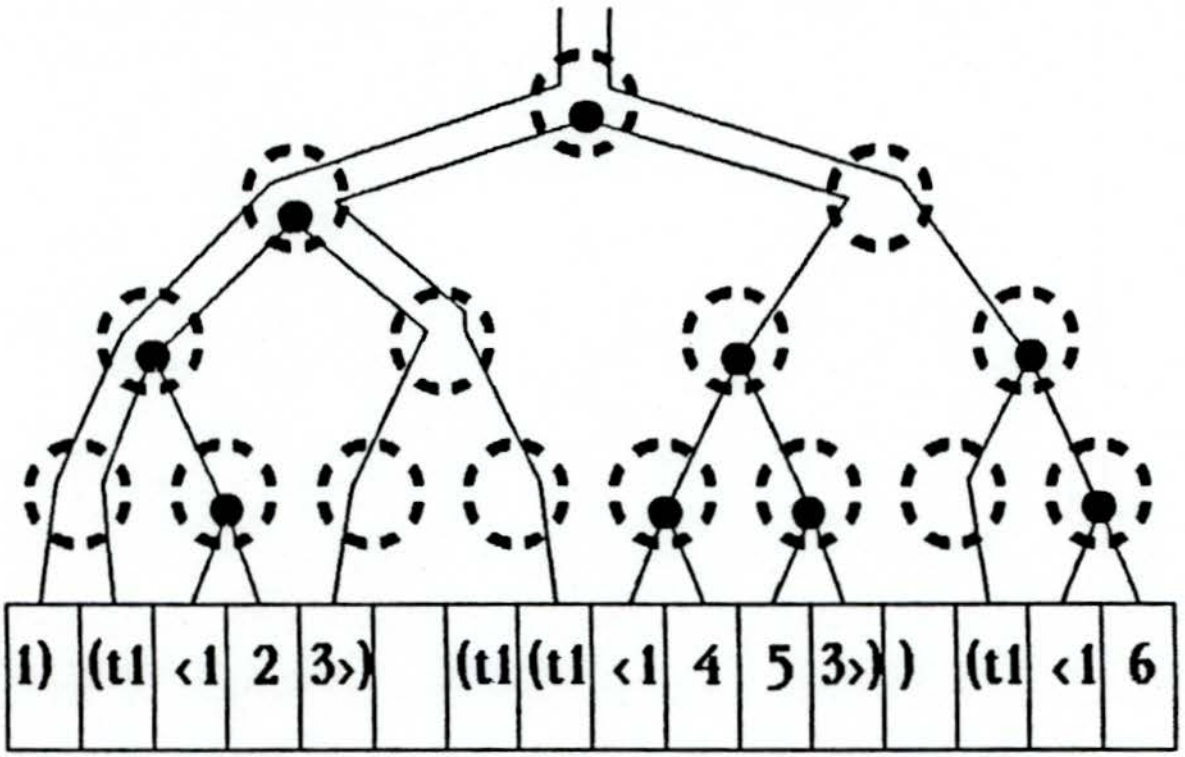
[Ba78] Backus, J. "Can Programming be liberated from the von Neumann style? A functional style and its algebra of programs" *Communications of the ACM 21,8*, August 1978, pp. 613-641.

[Hoar78] Hoare, C. A. R., "Communicating Sequential Processes." *Communications of the ACM 21*, 8 (August 1978), pp. 666-677.

[MM84] Magó, G. A., and Middleton, D. "The FFP Machine – A progress report" *Proceedings of the International Workshop on High-Level Computer Architecture*, Los Angeles, May 1984, pp. 5.13-5.25.

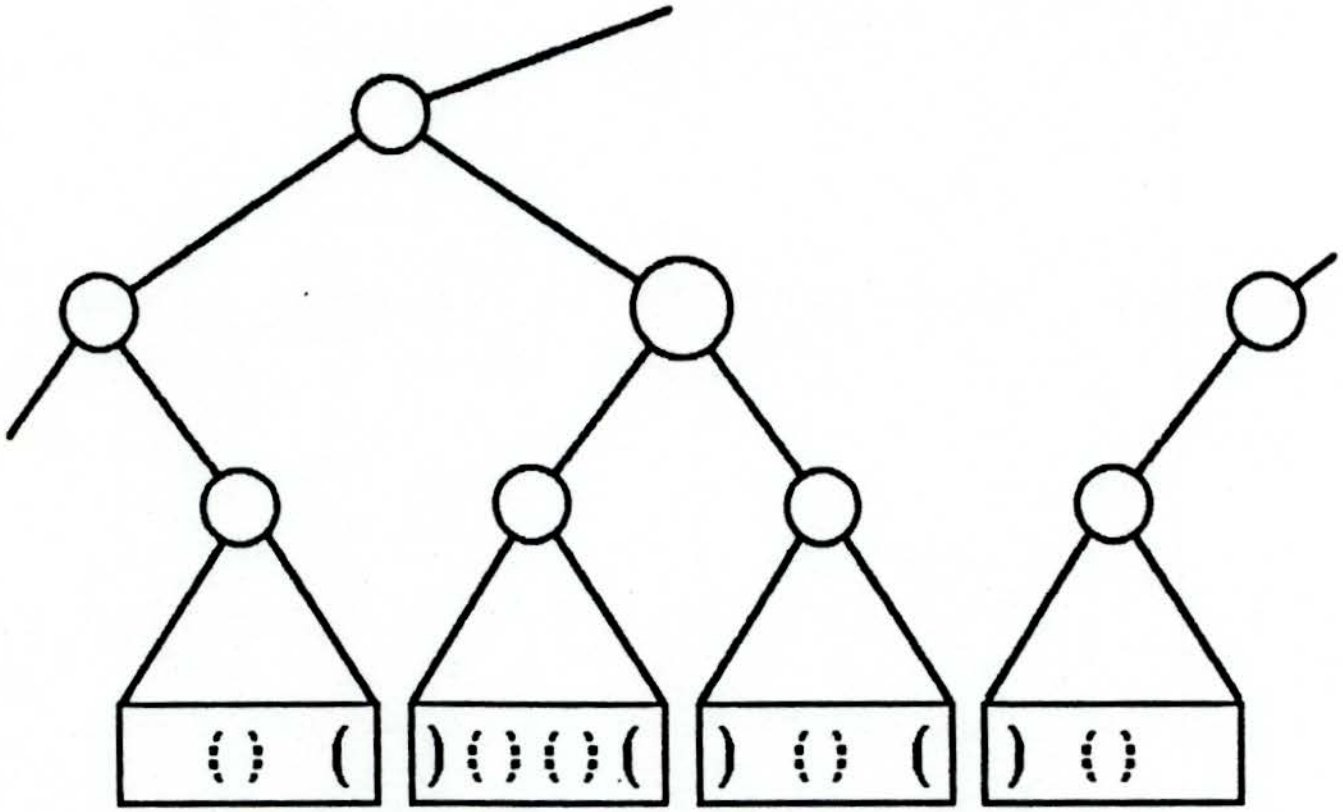
[Tu82] Turner, D. A. "Recursion Equations as a Programming Language" *Functional Programming and its Applications*, J. Darlington, P. Henderson and D. A. Turner, eds., Cambridge University Press, pp. 1-28. 1982

[Pa82] Pargas, R. P. "Parallel Solution of Elliptic Partial Differential Equations on a Tree Machine." Ph.D. Thesis, University of North Carolina Computer Science Department Report TR82-002, 1982.



L cells, at the bottom of the tree, hold FFP symbols, e.g. 'tl', the tail function.
T cells, shown as dashed circles, contain the internal nodes of the virtual machines, shown as solid circles.

Figure 1. Partitioning the physical machine into disjoint virtual machines.



For the enlarged T cell, the solid parentheses show all the areas that can need support. Areas shown by dashed parentheses are entirely supported either within one of the children, or by hardware exterior to this T cell.

Figure 2. A T cell must support at most three areas.

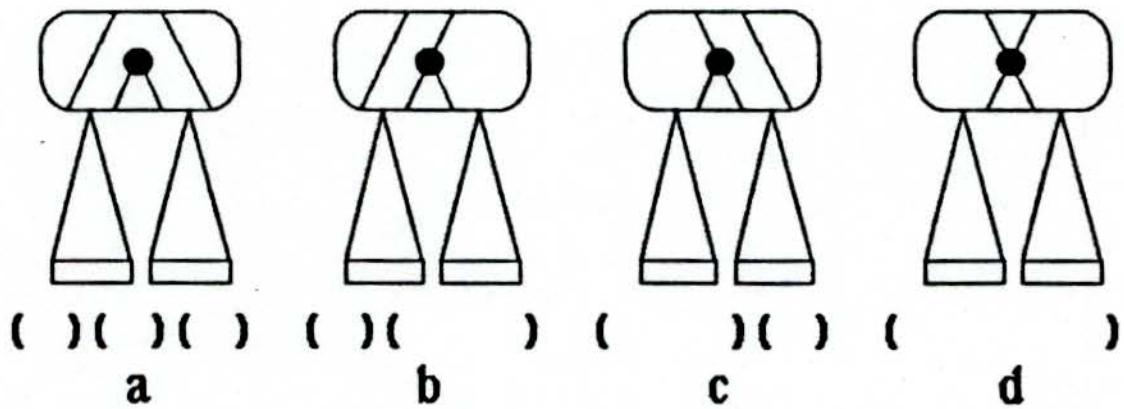
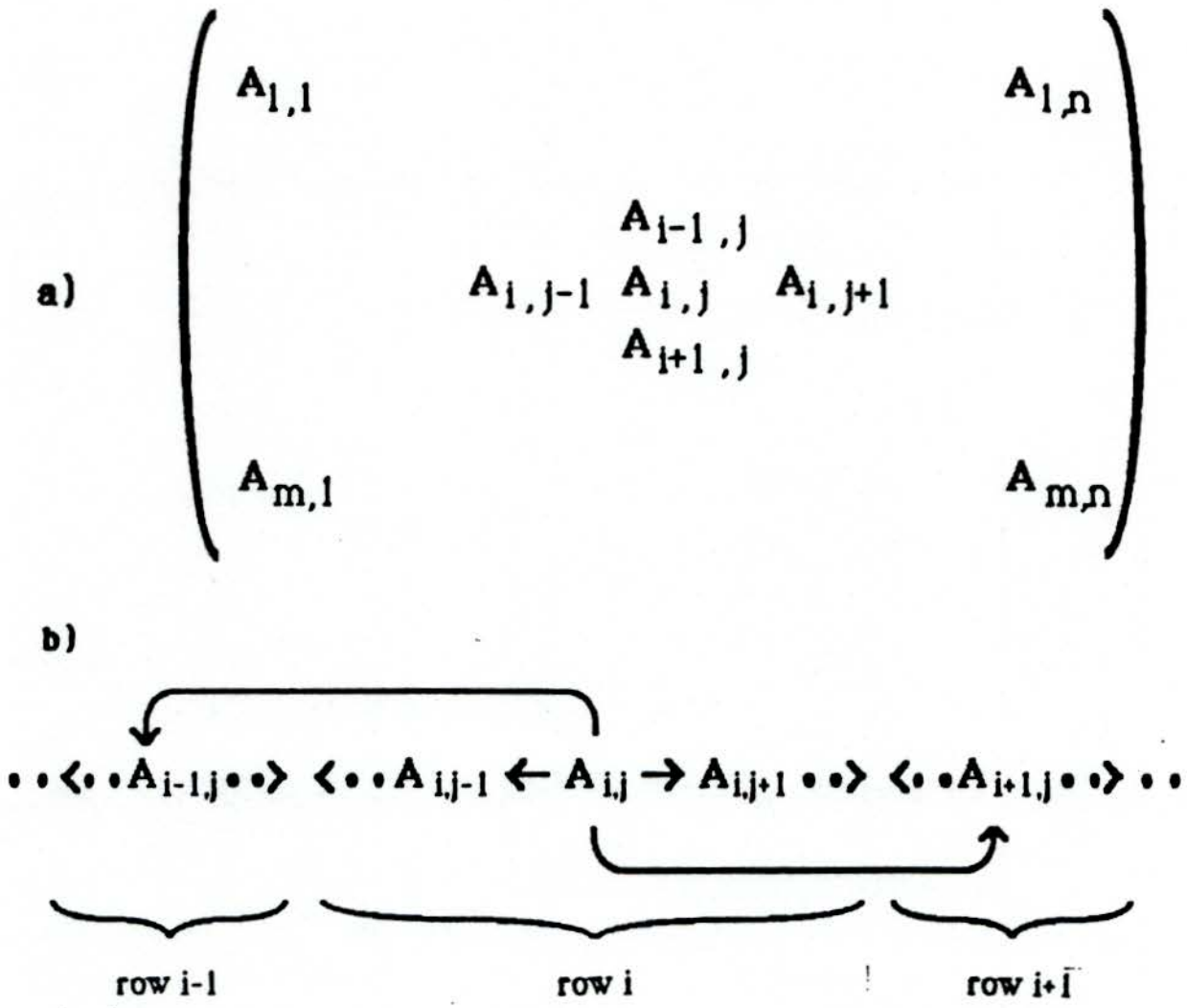
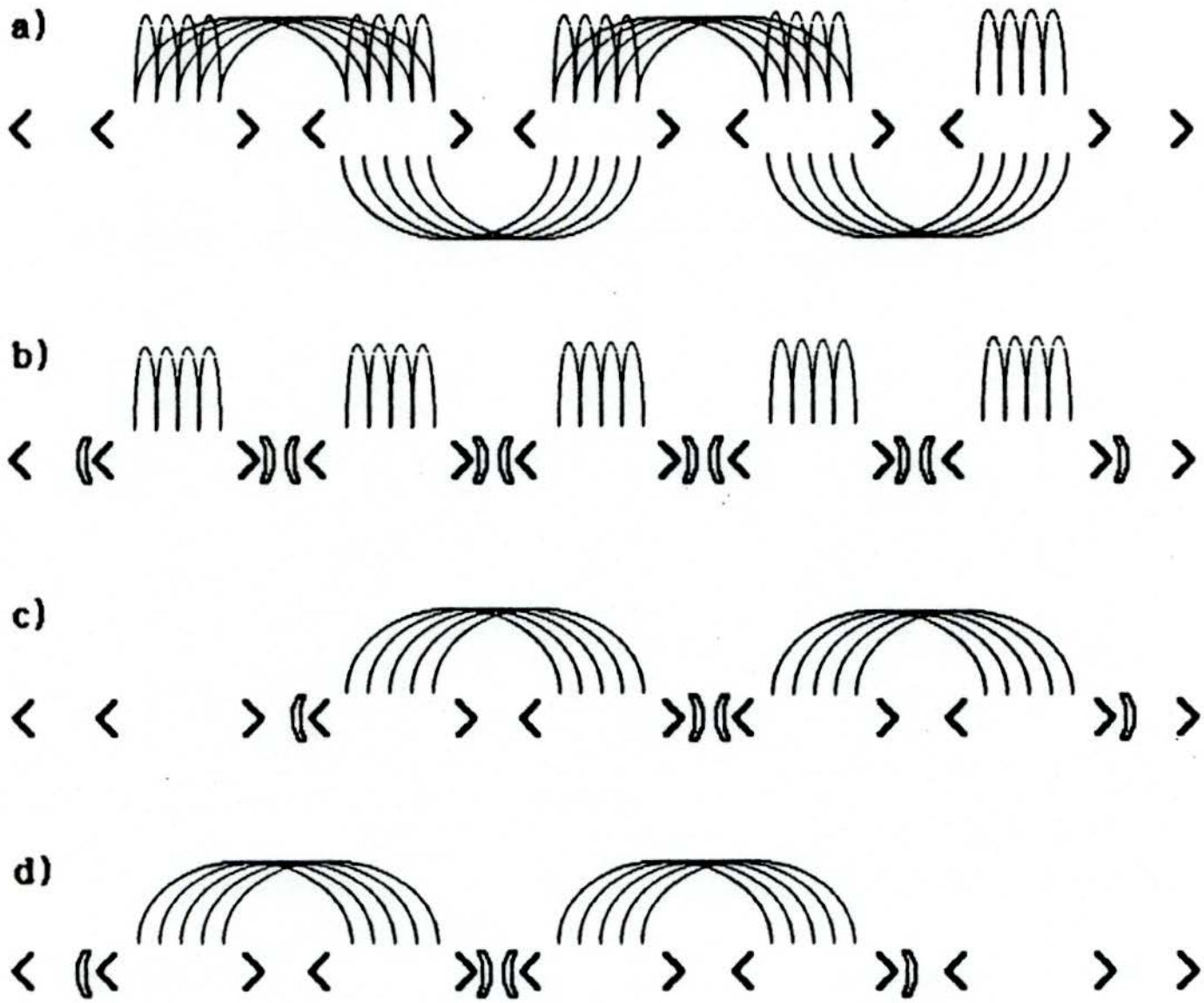


Figure 3. Internal configuration of T cells.



4a shows element A(i,j) and its four neighbors, whose values will affect its next value. 4b shows how a matrix is represented as a text string. Nearest neighbors in a two dimensional structure are no longer adjacent in the one dimensional representation.

Figure 4. Communications in a grid and in the FFP format



5a shows the interference between independent grid communications laid out in a linear representation. 5b, 5c and 5d show the separation of this communication into a series of less congested stages. 5b isolates communication to within rows. 5c and 5d, respectively, isolate communication to within adjacent even-odd and odd-even rows.

Figure 5. Separating communication into multiple stages reduces interference.

a $(\mathbb{T} \quad \langle r \langle f_0\ o_0 \rangle \langle f_1\ o_1 \rangle \cdots \langle f_n\ \emptyset \rangle \rangle)$

b $(\mathbb{T}' \quad \langle r \quad (f_0\ o_0) \quad (f_1\ o_1) \quad \cdots \quad (f_n\ \emptyset) \quad \rangle)$

c $(\mathbb{T}' \quad \langle r \langle o_0'\ f_0' \rangle \langle o_1'\ f_1' \rangle \cdots \langle o_n'\ f_n' \rangle \rangle)$

d $(\mathbb{T}'' \quad \langle r \quad o_0' \quad \langle f_0'\ o_1' \rangle \cdots \langle f_n'\ \emptyset \rangle \rangle)$

e $(\mathbb{T} \quad \langle r' \langle f_0'\ o_1' \rangle \langle f_1'\ o_2' \rangle \cdots \langle f_n'\ \emptyset \rangle \rangle)$

OR

f $(\mathbb{T} \quad \langle r' \langle f_{-1}'\ o_0' \rangle \langle f_0'\ o_1' \rangle \cdots \langle f_n'\ \emptyset \rangle \rangle)$

6a is the start of a cycle. T makes each pair into an application (6b) and replaces itself with T'. After their parallel evaluation (6c), T' re-associates each modified value with the next filter (6d). The leftmost value is isolated, and is available for T'' to incorporate into its accumulating result, r. T'' may either terminate, returning r', or else it may continue the stream computation, either with the same set of filters (6e), or with an added filter on the left.

Figure 6. Stream-like mechanism.

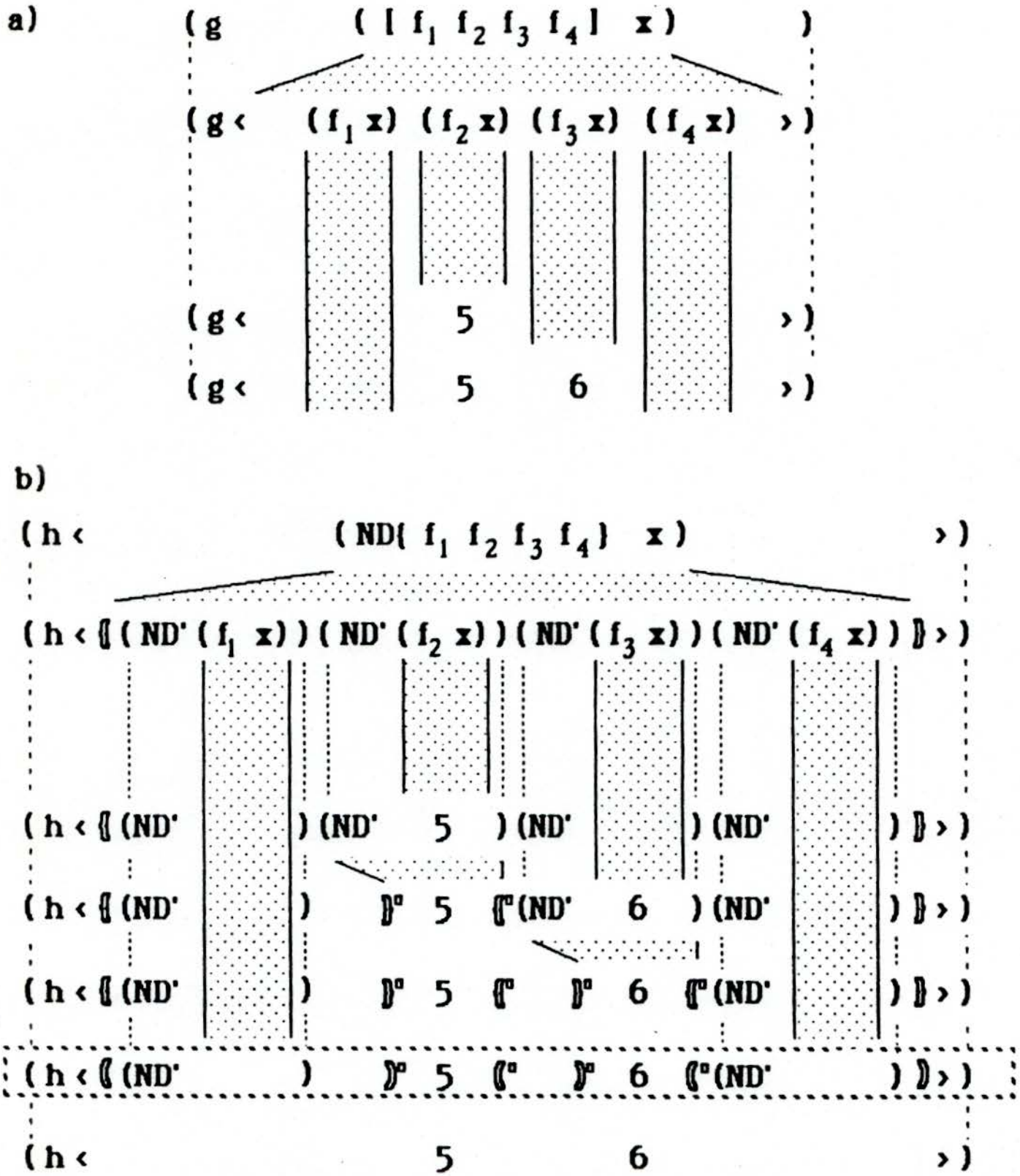


Figure 7. Non-deterministic choice.