MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structures Group Memo 62

A Look at "The Controlled Execution of Parallel Programs Operating on Structured Data" by Ian Campbell-Grant

Paul J. Fox

October 1971

The work discussed in this article was done at Project MAC, MIT, and was supported in part by the National Science Foundation under research grant GJ-432, and in part by the Advanced Research Projects Agency, Department of Defense, under Naval Research Contract N00014-70-A-0362-0001.

A Look at "The Controlled Execution of Parallel Programs Operating on Structured Data" by Ian Campbell-Grant

Paul J. Fox

A recent thesis in the Computation Structures Group of Project MAC at M.I.T. contains several new ideas in the area of computational schemata. In particular it is the first work known to deal with determinate parallel computations on arbitrary structures (i.e. data structures which include acylic sharing and directed cycles as well as pure trees). This memo has two main purposes. The first is to serve as a "readers guide" to the thesis. In doing so, I have attempted to use the terminology of the thesis to aid the reader in this task. No attempt has been made to accurately summarize the whole work. The second purpose is to begin the critical examination of the thesis — particularly as it relates to the rest of the work of the Computation Structures Group.

A computation may be viewed as transformations of data. The transformations involved with a program using structured data may be divided into three parts: changes in the structural relationships, reading data items from and writing data items into the structure, and changes to an individual data item. Because Ian Campbell-Grant desires to allow determinate parallel execution, computations in this thesis have a fourth part — changes in the potential access to the data structure.

Because our structured data is represented by a directed graph with nodes storing data items, changes in the structural relationships are simply the addition or removal of arcs and nodes. Access to the data items is controlled by means of pointers to nodes in the data structure. In addition to indicating a particular node in the data structure, a pointer also conveys a particular degree of access privilege (read, write, for example). Given a pointer, one is allowed to obtain a pointer to any node accessible (by following directed links) from the node referred to by the given pointer. Thus, pointers not only give access to the contents of a data cell, they also convey the potential for access to portions of the data graph which are accessible from the data cell referred to. The thesis preserves the determinacy of the parallel computations — ie. all executions of the same program yield the same results for the same set of input values — by insuring that at no time are there any two active pointers which could potentially cause an indeterminacy. Since this thesis is based on a fundamental assumption of ignorance about future actions, this requirement includes considering all those pointers which may be potentially derived from the two pointers in question.

With the first two parts of a computation — changes in the structural relationships and changes in potential access (pointers), we have not actually changed the value of any individual data item (although we may have destroyed some). This is crucial to the concept of computation on structured data. The actual operations on the data (or the "data changing" phase) takes place off the data structure. To carry out any "data change", we must first read the data item from the structure. If the data item has been changed it must be written back into the structure. To insure determinism neither the read nor write operation can take place without a pointer to the particular data cell having the appropriate access privileges. The remaining step is, of course, the actual operation upon the data item(s).

The thesis [1] concentrates only upon the first three parts of a computation, leaving responsibility for the actual operations upon the data cell contents to some other party. The fact that no data item can enter or leave the data structure except as the result of an explicit operation (discussed above) allows Campbell-Grant to define an effective interface between the portion of the total computation dealt with in his work and the remaining portions.

Description

With this overview, it is probably simplest to first explore the formal structure of Campbell-Grant's thesis. The first item the thesis defines is a Computation State (Fig. 1) consisting of two parts — the Data Structure and the Program Execution Structure (PES). The Data Structure stores all current data and the current structural relationships between data items. The Program Execution Structure (PES) contains the instructions, all current pointers into the Data Structure, and all current restrictions on the use of these pointers. The history of a computation can be shown by the progression of Computation States.

Data Structure: The formal definition reflects the thesis' objective of allowing arbitrary structures, including cycles. The structure is composed of Data Cells which consist of a unique identifier, called the pointer, and a contents which is some form of data item. Since the thesis is concerned with the restrictions on access to the data as opposed to the complete computation, the exact forms of data representation are not dealt with. Data Links are then defined as ordered pairs of Data Cell pointers. The lack of restrictions on what Data Links may be created (as long as they are created in a deterministic manner), allows one to form a completely arbitrary data structure. It is worth noting that there is no distinction in this thesis between interior nodes and leaf nodes. All nodes are the same, with each containing an item of data. Given a data structure, one is able to define the substructure of any cell in that structure. The substructure is all nodes that may be accessed from the given cell by the traversal of the (directed) Data Links. The nodes marked by an X in Fig. 2 are the sub-

structure of cell E. Note that one may as easily refer to the substructure of a pointer as of a cell. Selectors are functions which map a cell onto a unique cell within that cell's substructure (or onto no cell). Campbell-Grant does not require a 1:1 relationship between selectors and Data Links as does all the other work of the computation Structures Group. As an example, in Fig. 2 one could have a selector that mapped cell E onto cell C whereas in other works one would only have selectors from E to H and I. The significance of this difference is not clear to me. I suspect it makes no real difference.

Program Execution Structure (PES): Having described the data structure, we now consider the portion of the Computation State which deals with access privileges — the Program Execution Structure. Before going into the details of the PES, it is worthwhile to discuss some of the basic concepts employed. The first concept is that of the stream. A stream is analogous to a "process." In addition to parallelism among streams, which the thesis deals with, there may also be parallelism inside a stream but the responsibility for maintaining determinism inside the stream is that of the stream. Reflecting this parallelism is the fact that an "instruction" in this thesis is really a directed graph of many elementary instructions, some of which may be executed in parallel. The elementary instructions consist of three types — Primitive Instructions which transform the Computation State ie. the access privileges and structural relationships of the data, Sequencing Instructions, and Computation Instructions, which affect the data contents of the Data Cells ("data changing").

The streams are formally represented by the set "Potential Parallel Process Streams" (PPPS). Each element of the PPPS, called a stream*, is a five-tuple of the Invoking Stream Name, Absolute Stream Name, Relative Stream Name, Stream State, and Instruction. The Absolute Stream Name is a unique identifier of the particular stream. The Invoking Stream Name is the absolute stream name of some other stream which spawned this stream. The Relative Stream Name is provided for syntactic purposes and hence is not discussed in this thesis. The role of the Stream State which may be "blocked," "unblocked," or "applying," will be discussed later. The Instruction, as discussed above, is really a directed graph of elementary instructions.

---

*To remain in conformity with the thesis, the word "stream" is used to refer to both the "process-like" concept and the formal five-tuple representation. It should be clear from the context which is being referred to.

Pointers in the PES are items which give their owners certain access privileges to a specific cell in the Data Structure and to the substructure of that cell. There are three kinds of access privileges — read, write data, and write structure. Read (R) only gives the privilege of reading the contents of a Data Cell. Write Data (WD but sometimes in the thesis W) gives the privilege to not only read a Data Cell's contents but to change it. Write Structure (WS) is the ability to change the Data Structure (Data Links and the existence of Data Cells) in addition to the contents of a Data Cell. All pointers have the ability to "read structure," a concept to be discussed later. The pointers are formally found in the Pointer Instance Set (PIS) which is a set of three-tuples — Name, Pointer, Access. The Name is the absolute name of the stream in the PPPS which owns the pointer. The Pointer is the unique identifier of a Data Cell in the Data Structure, while Access is R, WD, or WS as discussed above.

Given several pointers into the Data Structure there exists a need to control the interactions between various streams, each of which wants to utilize the privileges its own pointers hold. The device which does this is the Constraint Set (CSET) which is a set of ordered pairs of PIS elements. The ordered pairs of PIS elements form a directed graph which defines precedence relationships between the various PIS elements. PIS element A <u>precedes</u> PIS element B if and only if there is a sequence of CSET elements from A to B. It is shown by theorem (not definition) that the CSET is always acylic.*

<u>Primitive Instructions</u>: At this point we will follow the pattern of the thesis and defer discussion of anything having to do with WS capability. This permits a simpler and more orderly presentation of the concepts in the thesis. We first look at the Primitive Instructions:

---

*Again following the pattern of the thesis, the word "CSET" is used for both the constraints themselves (as in this sentence) and the formal ordered pair representations of these constraints.

CI-S ~ no mnemonic
P-S ~ Pointer Select
F-P ~ Free Pointer
I-S ~ Initiate Stream
T-S ~ Terminate Stream
A-P ~ Apply Pointer

CI-S enables an issuing stream to hand an access ability it possesses (in the form of a PIS element) to another stream. Formally a new element of the PIS is created with a pointer to the same Data Cell and with the same access ability as the PIS element belonging to the issuing stream. If the new PIS element must be ordered with respect to the old (to be discussed below), then a parameter, SEQ, tells in which order.

P-S(I, S) creates a new PIS element that is in the same stream as PIS element I and has the same access abilities as I but points to the cell obtained by applying selector S to the cell pointed to by I.

F-P(I, Type) releases some or all of the access abilities PIS element I possesses. If Type is All, then I disappears. If Type is WD, then only the write data ability is given up (while the read ability is retained).

I-S and T-S initiate and terminate a stream. Streams are initiated without PIS elements and must have no PIS elements before they can be terminated. The instruction Free Pointer is used to accomplish this.

Apply Pointer ~ A-P is the means by which the thesis structure interacts with computations on the data. This primitive instruction precedes any attempt to read or write the contents of the cell pointed to by an element of the PIS owned by the issuing stream. The issuing stream is blocked (ie. the state of the PPPS is changed to "blocked") until there are no constraints (ordered pairs in the CSET) terminating on the PIS element. When this occurs the stream is changed to applying and, upon the completion of the read or write operation, the state reverts to unblocked. Until it reverts to unblocked, the issuing stream may not execute any other primitive instructions.

Forming the Constraint Set: With this set of primitive instructions, we have developed the capability for parallel computations, but we have not yet provided for determinate computation. Suppose we execute the Primitive Instruction CI-S giving stream 2 a PIS element J which is a duplicate (same access privilege and pointer to the same Data Cell) of PIS element I belonging to stream 1. If the access ability of I was R then no harm has occurred. But suppose I had WD access, we now have two streams with write access to the same Data Cell. If both were to attempt to use this access ability to write into the contents of the data cell, we would no longer have a determinate computation. In the language of computational schemata we would have a "conflict" — a situation in which two operators are unordered and the output of one operator affects the input of the other or the output of one can change the results of the other operator. In the terminology of this thesis a potential conflict occurs whenever there are two PIS elements to the same cell with one or both having WD access. The solution to this problem is to make the two PIS elements ordered. The mechanism for doing this is the CSET. Any two PIS elements which point to the same cell must be ordered in the CSET if either one has WD access, unless they both belong to the same stream. If they both belong to the same stream, any ordering restrictions are considered to be the stream's, not the system's, responsibility.

However, as discussed in the introduction, pointers convey potential access to the whole substructure, not just the data cell pointed to. In Fig. 3, PIS elements 1 and 2 point to different cells (A and B), so there is no immediate conflict. The stream owning PIS element 1 could read the contents of cell A (ie. execute Apply-Pointer) at the same time the stream owning 2 could write into cell B. The computation would remain fully determinate. However, suppose that instead of executing Apply-Pointer the streams owning PIS elements 1 and 2 both executed Pointer-Select operations resulting in PIS elements 1' and 2' each pointing to cell C(Fig. 3b). We now have an immediate conflict if PIS elements 1' and 2' (and 1 and 2) are owned by different streams. If they are owned by different streams, then 1 and 2 must be ordered in the CSET. Because of the thesis' fundamental assumption of

ignorance about future actions, we see that our concept of "conflict" must be expanded to include any case where the substructures of two PIS elements belonging to different streams overlap and at least one has WD access. The concept of substructures overlapping is used so often that a formal operator, Structure, is defined. The Structure operator has as its arguments either two Pointers or Data Cells. If the substructure of these two pointers or cells have no nodes in common the operator returns the value Φ. If, on the other hand, they overlap, some form of $\neq \Phi$ is returned. From the point of view of this thesis, Structure need only return one of two values — equal or not equal to the null set. Campbell-Grant suggests that in an actual implementation one might want to return the intersection of the two overlapping substructures. In Fig. 2 Structure (E, F) = Φ , whereas Structure (D, E) $\neq \Phi$. Note also that in Fig. 3 Structure (Pointer (1), Pointer (2)) $\neq \Phi$.

The use of the operator Structure allows us to formally define a conflict free computation state as one in which, for any two elements (I and J) of the PIS, property S holds, where property S is as follows:

S
- If Structure (Pointer(I), Pointer(J)) $\neq \Phi$
- Then Either Access(I) = Access(J) = R
  - Or [Stream]Name(I) = [Stream]Name(J)
  - Or Either (I precedes J) Or (J precedes I)

One way to prove that all computations maintain Property S is by induction. The fact that all computations start from an initial state of only one stream possessing only one PIS element provides the basis for the proof. Proving that the sections of the Primitive Instructions which define what if any additions or deletions from the CSET are necessary to maintain Property S are complete is sufficient to provide the inductive step. This proof is done in Theorem 3.5 of the thesis. Using the same technique Theorem 3.4 proves that the CSET is always acylic, while Theorem 3.6 shows that there are no unnecessary constraints in the CSET. Formally it is shown that for every element (C_1, C_2) of the CSET, Property T holds:

$$T \quad \begin{cases} \text{Structure}(\text{Pointer}(C_1), \text{Pointer}(C_2)) \neq \Phi \\ \underline{\text{Not}}\ (\text{Access}(C_1) = \text{Access}(C_2) = R) \end{cases}$$

Write Structure Access Privileges (WS): Up until now we have not had any way to change the structural relationships in the data structure (ie. use WS access privileges). If we now allow the use of WS privileges we must first introduce the new and modified Primitive Instructions.

Free Pointer ~ F-P is simply expanded to include a Type WS which reduces the access privileges from WS to WD.

Amend Structure ~ A-S has three forms:

Amend Structure (Attach Cell) ~ $A\text{-}S_{AC}$ creates a link to a cell not previously connected to the Data Structure. A WS PIS element pointing to the cell to which the new cell will be attached is required.

Amend Structure (Create Link) ~ $A\text{-}S_{CL}$ creates a link to a cell already in the data structure. If a stream wishes to create a link from cell i to cell j it must possess a WS PIS element pointing to i, a PIS element of any access privilege pointing to j, and a PIS element of WS privilege which has both i and j in its substructure.

Amend Structure (Delete Link) ~ $A\text{-}S_{DL}$ destroys a link between two cells. To execute this Primitive Instruction a stream needs a WS PIS element pointing to the first cell in the Data Link to be deleted.

There is no explicit instruction to delete a cell since a cell is "deleted" by simply disconnecting all data links between it and the rest of the data structure.

To illustrate the difficulty WS privileges cause, let us look at the example shown in Fig. 4. With the situation as shown in Fig. 4a let us suppose that by means of Primitive Instructions CI-S, F-P, and P-S, PIS element 0 is converted into two PIS elements — 1 and 3. The result (shown in Fig. 4b) is PIS elements 1, 2, and 3, each belonging to different streams. Since Structure(Pointer(1), Pointer(3)) = Φ, there seems to be no reason to order 1 and 3. However, suppose PIS element 2 is used to create PIS elements with WS privilege pointing to cells C and E. We now have the necessary conditions to allow execution of Amend Structure (Create Link) creating a link from cell C to cell E. The result is shown in Fig. 4c. If Free-Pointer is now executed to remove PIS elements 2, 2a, and 2b, the result is shown in Fig. 4d. Note that property S (conflict free) no longer holds since Structure(Pointer(1), Pointer(3)) $\neq \Phi$ and Access (1) = WD.

The problem arose when we concluded that PIS elements 1 and 3 did not need to be ordered with respect to each other. We made that conclusion on the basis of the structural relationships existing at the time of Fig. 4b. Because PIS element 2 had WS access privilege, it had the ability to change those structural relationships upon which the conclusion — to leave PIS elements 1 and 3 unordered — was made. In other words, we cannot allow any case in which a preceding PIS element with WS can, by changing the structural relationships, cause a conflict where none had existed.

This notion can be formally represented in the operator (WS) Structure which is formally defined for PIS elements K and J as:

(WS) Structure(K, J) $\neq \Phi \Leftrightarrow$

Either Structure(Pointer(K), Pointer(J)) $\neq \Phi$

Or $\exists$ a sequence $(A_1, \ldots, A_n)$

s.t. $A_i \in$ PIS, Access(A_i) = WS $(1 \leq i \leq n)$

Structure(Pointer(K), Pointer(A_1)) $\neq \Phi$

Structure(Pointer(J), Pointer(A_n)) $\neq \Phi$

If $n>1$, Then Structure(Pointer(A_i), Pointer(A_{i+1})) $\neq \Phi$, $(1 \leq i < n)$

$A_i = K$ Or $A_i = J$ Or ((A_i precedes K) And (A_i precedes J)).

Fig. 4 was an example of a case in which n = 1 with K = 1, A = 2, and J = 3. It turns out that if the restrictions imposed on the Primitive Instructions (to be discussed next) are obeyed, Theorems 3.4, 3.5, and 3.6 remain valid with the substitution of the operator (WS) Structure for Structure.

The restrictions necessary to insure determinism with WS access are found in the Association Sets. Association Sets consist of those primitive instructions whose immediate execution would result in a situation similar to that in Fig. 4 — the presence of a preceding PIS element with WS ability which may change the structural relationships, causing a potential conflict. In the case of Fig. 4, the creation of PIS elements 1 and 3 must be suspended until the elimination of PIS element 2. Formally, an Association Set is associated with each ordered pair of PIS elements (owned by different streams). The Set is nonempty whenever a

suspended Primitive Instruction would have used the second PIS element. In the example shown in Fig. 4, the suspended attempt to create PIS elements 1 and 3 would be associated with the directed pair (2,0). Whenever the first PIS element is removed, the Association Set for that pair is activated and the suspended instructions are executed. In the case of Fig. 4, upon the deletion of PIS element 2, the creation of PIS elements 1 and 3 is executed. At this time (Fig. 4d) it is clear that PIS elements 1 and 3 must be ordered.

In his concluding chapter (5), Campbell-Grant defines a new set of primitive instructions. These are defined in terms of the previous primitive instructions so that the properties (acylic, conflict free, no unnecessary constraints, etc.) of the first set remain valid. The purpose of this modified set is to create a set of instructions that are simpler to use in actual computations. Campbell-Grant believes there may be some restriction on the amount of parallelism allowed with this new set, but that the simplicity gained is worth this cost.

Analysis

It is clear from the thesis that Campbell-Grant equates conflict free (property S in Theorem 3.5) with determinate.* Although never stated explicitly this is presumably based on previous work in the Computation Structures Group such as the 6.232 Class Notes [2].

In comparing this thesis with the other work in the Computation Structures Group (C.S.G.), I find three major areas of difference. The first is obviously the ability to maintain determinism with arbitrary data structures. The second is the separation of "reading the structure" from reading the data. The third significant area is the execution time nature of the system.

What is meant by the concept "reading the structure"? Reading the structure" is the movement of PIS elements (pointers) around the data structure without actually accessing (reading or writing) a data item. During these

---

*There is the additional requirement that the constraints in the CSET are never inverted. (Section 3.3)

movements one is really "reading" the structural relationships of the data items. In this thesis it is possible to carry on "read structure" operations with PIS elements that cannot currently be used to read (or write) data items, ie. execute Apply-Pointer. One simply adds or deletes the necessary constraints in the CSET. At some point it may be necessary to suspend "read structure" operations with the Association Sets because of the presence of WS PIS elements with the ability to invalidate the structural relationships on which the CSET is being formed. This separation of "read structure" ability from read data is in sharp contrast to other work in the Computation Structures Group (C.S.G.). For an example let us look at some work by J. B. Dennis in C.S.G. Memo 32 [3]. In Memo 32 there is no equivalent to the CSET. Any active pointer is completely without constraints on its use — including its use in reading (or writing) data items. In other words there is no separation of "reading the structure" and reading a data item — a pointer can do both or neither.

It seems that working on arbitrary data structures as opposed to pure trees makes separation of read structure from read and write data desirable. As an illustration let us consider Fig. 5 which shows an n dimensional matrix shared between two data structures. Suppose Streams 1 and 2 wanted to write into adjacent cells in the nth dimension (j and j + 1 in Fig. 5). In this thesis with its separation of structure reading from data accessing the two PIS elements could move down the matrix with some ordering in the CSET until they separated at the nth dimension. With one pointing to the jth cell and the other to the j + 1th there would be no need for an ordering of PIS elements 1' and 2' so that both could immediately execute Apply-Pointer. In contrast, Memo 32 with structure reading and data accessing combined would force the complete access (including going down the structure) to be ordered even though no actual conflict ever developed.

The third area of contrast is the execution time character of this thesis. Unlike other work in the Computation Structures Group, this thesis' system allows an instruction to be initiated long before its execution can be determinate. The system checks each initiated (primitive) instruction at execution time and simply suspends those whose immediate execution would be indeterminate. This is done with the Association Sets and in the case of Apply-Pointer by

blocking the Stream. Similarly, those instructions which may be executed may still introduce additional restrictions on succeeding instructions. These restrictions are found in the CSET. It is important to remember that there is no predetermined order in which instructions are suspended or elements of the CSET added or deleted. The system simply deals with the instructions in the order in which they are initiated by the different streams — something impossible to know by the very definition of asynchronous parallel computation.

This "execution time" character contrasts sharply with the other work in the Computation Structures Group. Again looking at C.S.G. Memo 32, we find a system which is capable of being implemented before execution time such as at compile time. Further — and this goes to the heart of the difference — it is possible in advance to graph the order in which the different access abilities (pointers) will be given to different instructions. In particular, one knows exactly which instructions will be held up pending execution of which preceding instructions. While the absolute times are unknown, the relative order in which these events occur is known ahead of execution time.

There may be a simple explanation for why Memo 32 uses a compile time system and Campbell-Grant's thesis uses an execution time system. Memo 32's system was intended for parallelism inside a single procedure. Campbell-Grant's work, however, is to coordinate parallelism between separate processes (or streams as he calls them). It seems that an execution time system needs less knowledge about the other processes it is interacting with than does a compile time system. The same distinction between parallelism among processes versus inside a process may be relevant to the questions of separation of "read structure" from read data. The problem of Fig. 5, parallel access to different items in a matrix, may not require two separate pointers inside a single process. Inside a single process it may be possible to move one pointer down the matrix to the nth dimension and then split it up — something that may easily be too difficult to do with separate processes.

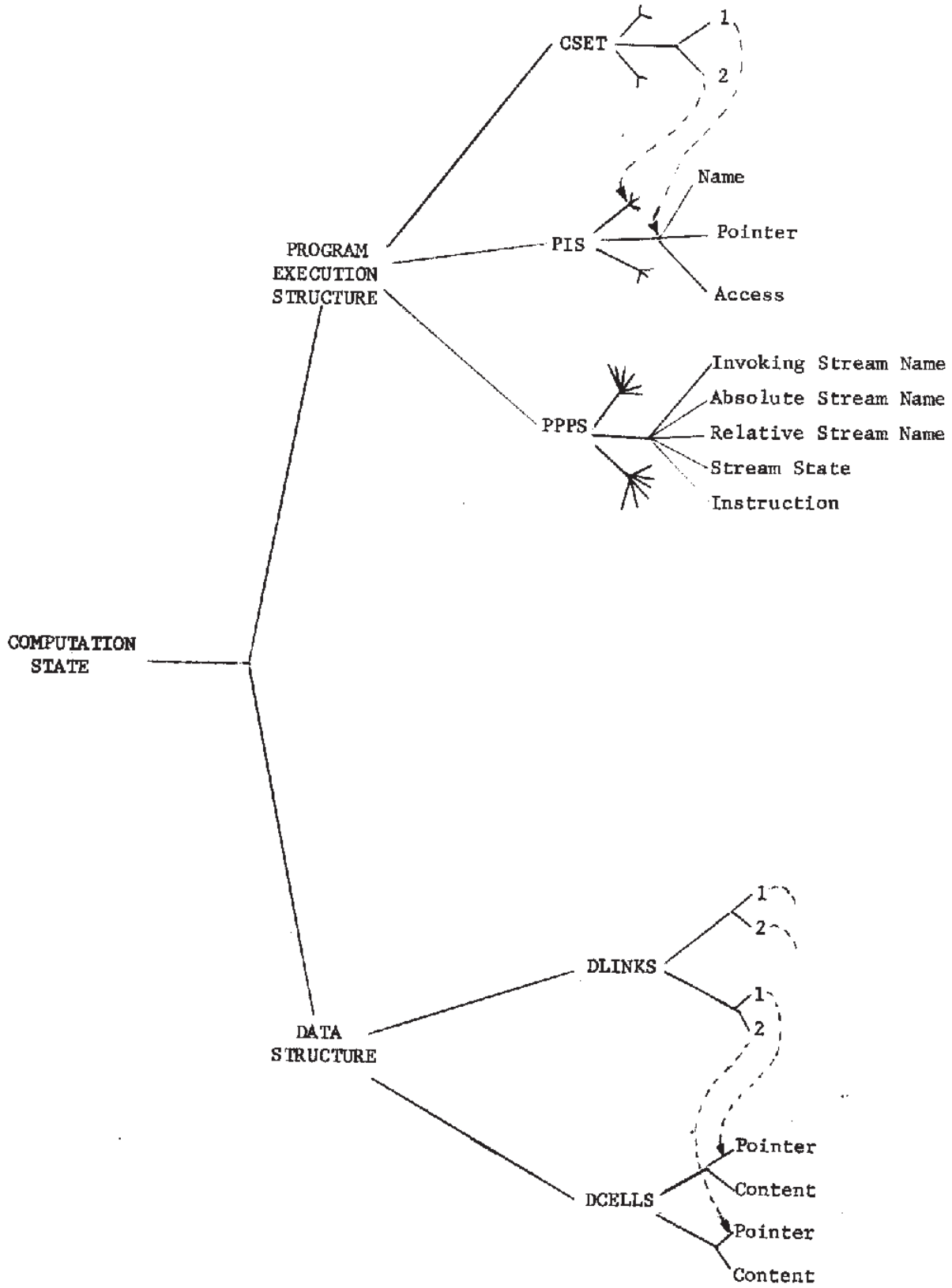
As the last point indicated, these three areas of difference are by no means disjoint. In fact, their interrelationships are clearly one of the major areas for future research. It may be that there is a need for separate schematas to allow parallelism between and inside processes. As an example, Campbell-Grant makes no provision for iteration or conditionals. Presumably an individual process (stream) could have such operations within it, but there still remains the need to insure that these operations are carried on determinately.

References

[1] Ian R. Campbell-Grant, The Controlled Execution of Parallel Programs Operating on Structured Data, Electrical Engineer Degree Thesis, Department of Electrical Engineering, M.I.T., 1971

[2] J. B. Dennis, Computation Structures. Notes for Subject 6.232, Department of Electrical Engineering, M.I.T., Cambridge, Massachusetts, 1970.

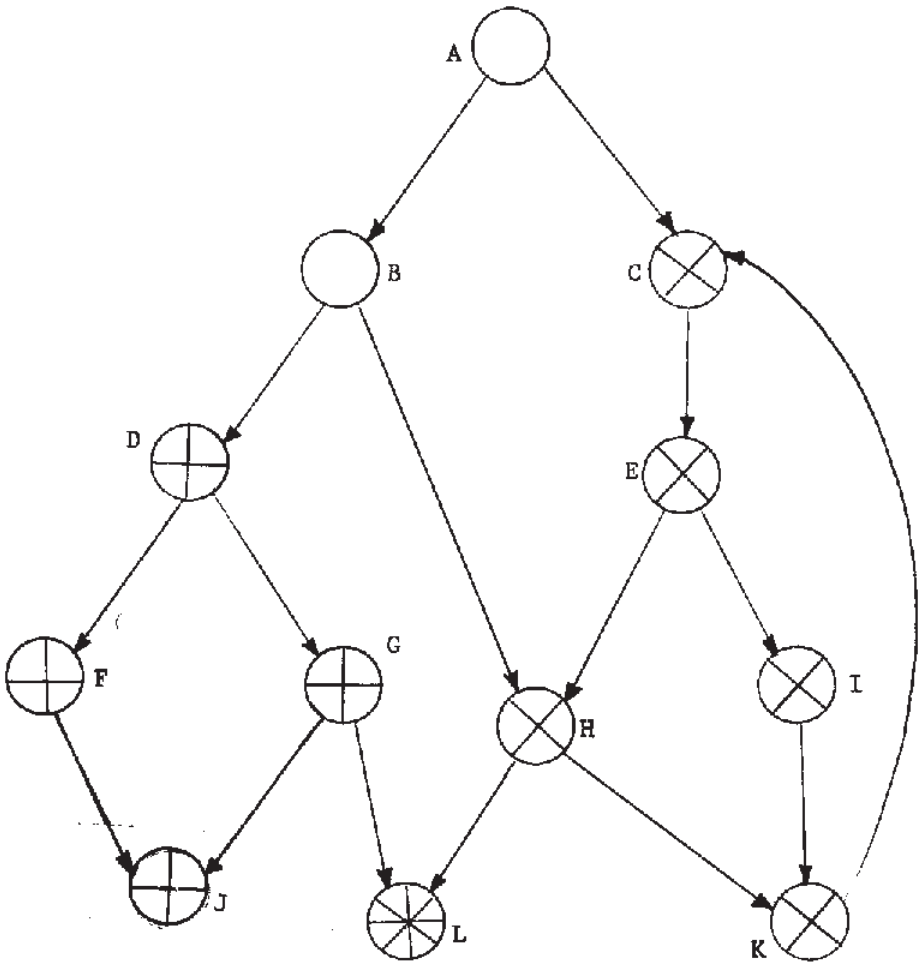
[3] J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture." Computation Structures Group Memo 32, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, August 1968. [Also Information Processing 68, North-Holland, Amsterdam 1969, pp 484-492.]



This is Fig. 2.1 on p. 22 in [1].

FIGURE 1

A DATA STRUCTURE



Substructure of E

Substructure of D

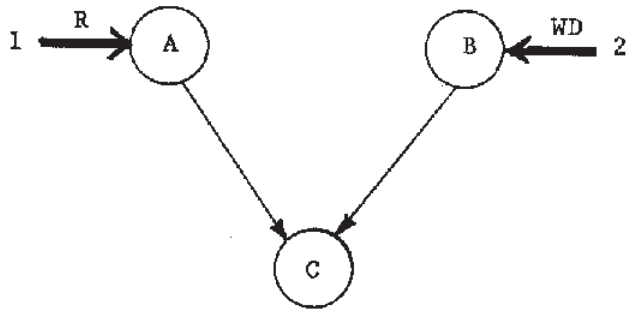
FIGURE 2

POTENTIAL CONFLICT

Data Cells {A, B, C}

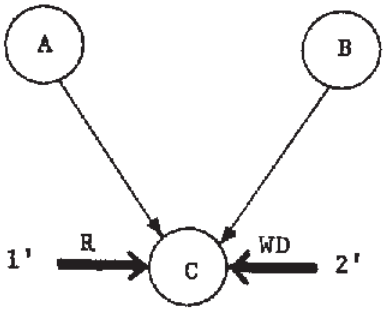
Data Links {(A, C), (B, C)}

(a)



PIS Elements {1, 2}

(b)



PIS Elements {1', 2'}

FIGURE 3

POTENTIAL CONFLICT WITH WS ACCESS

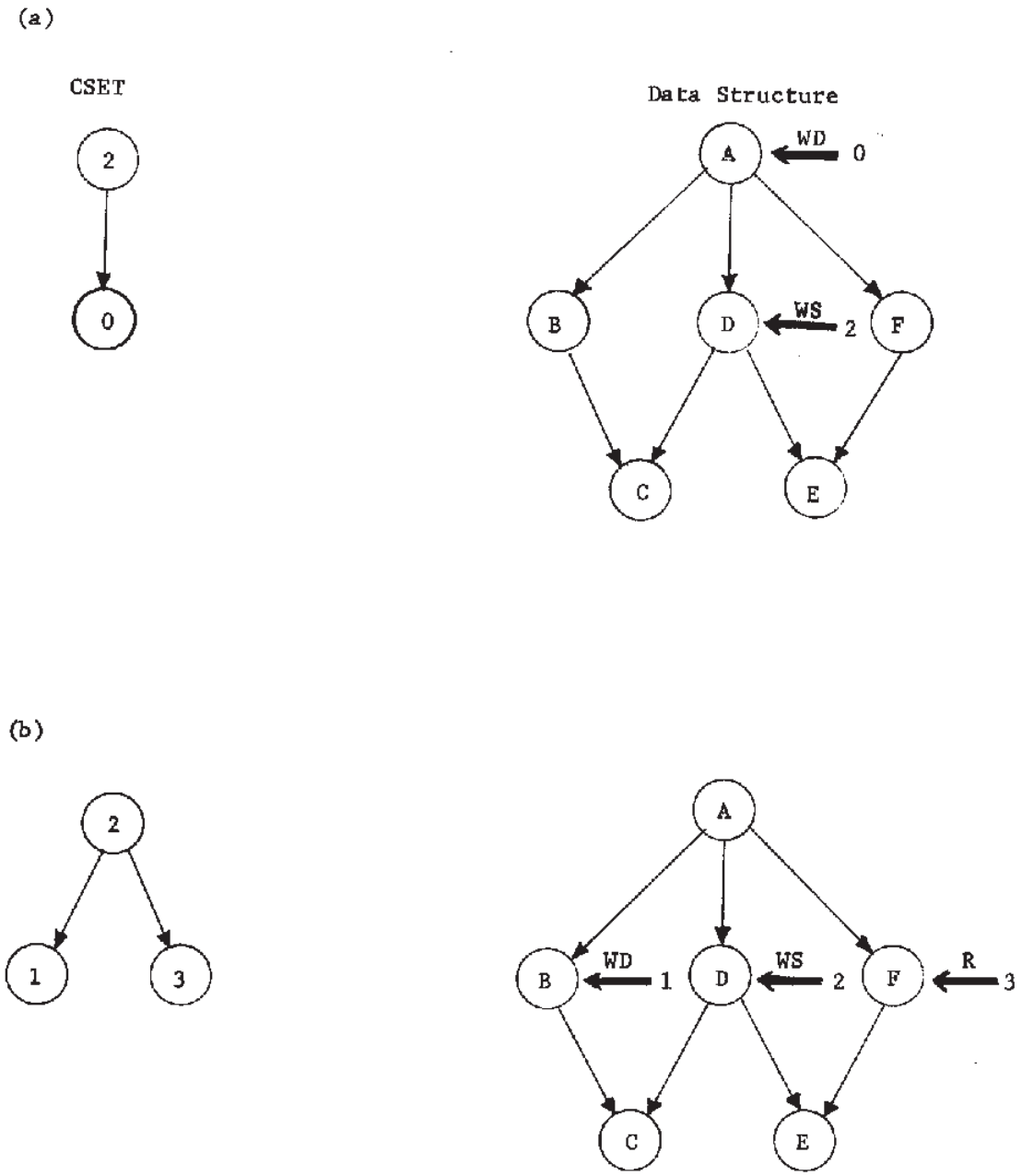
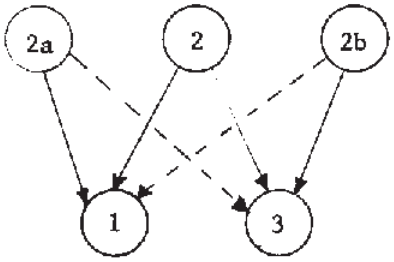


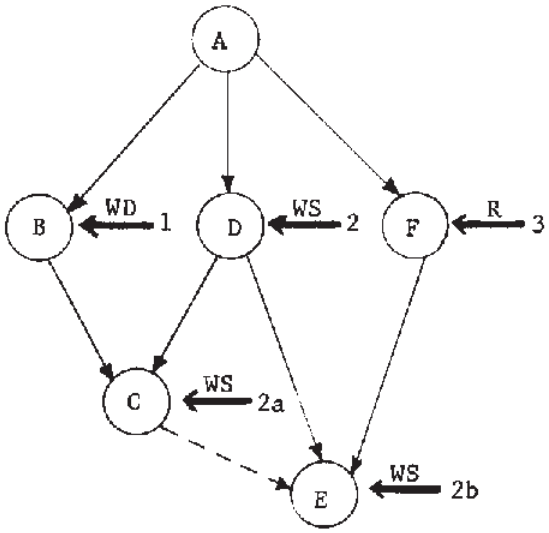
FIGURE 4

(c)

CSET



Data Structures



(d)

empty

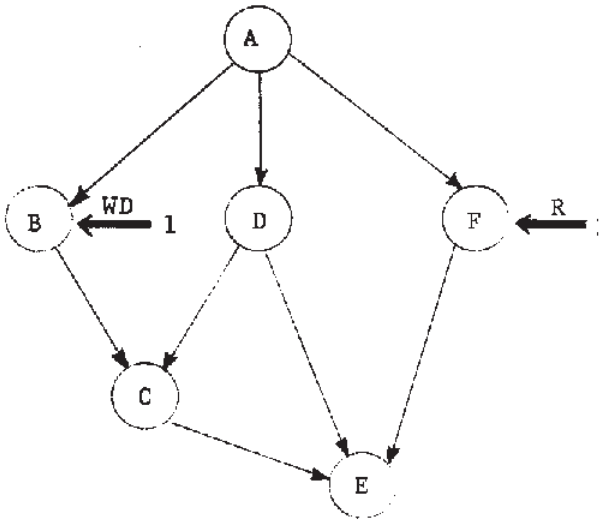


FIGURE 4

SHARING A MATRIX

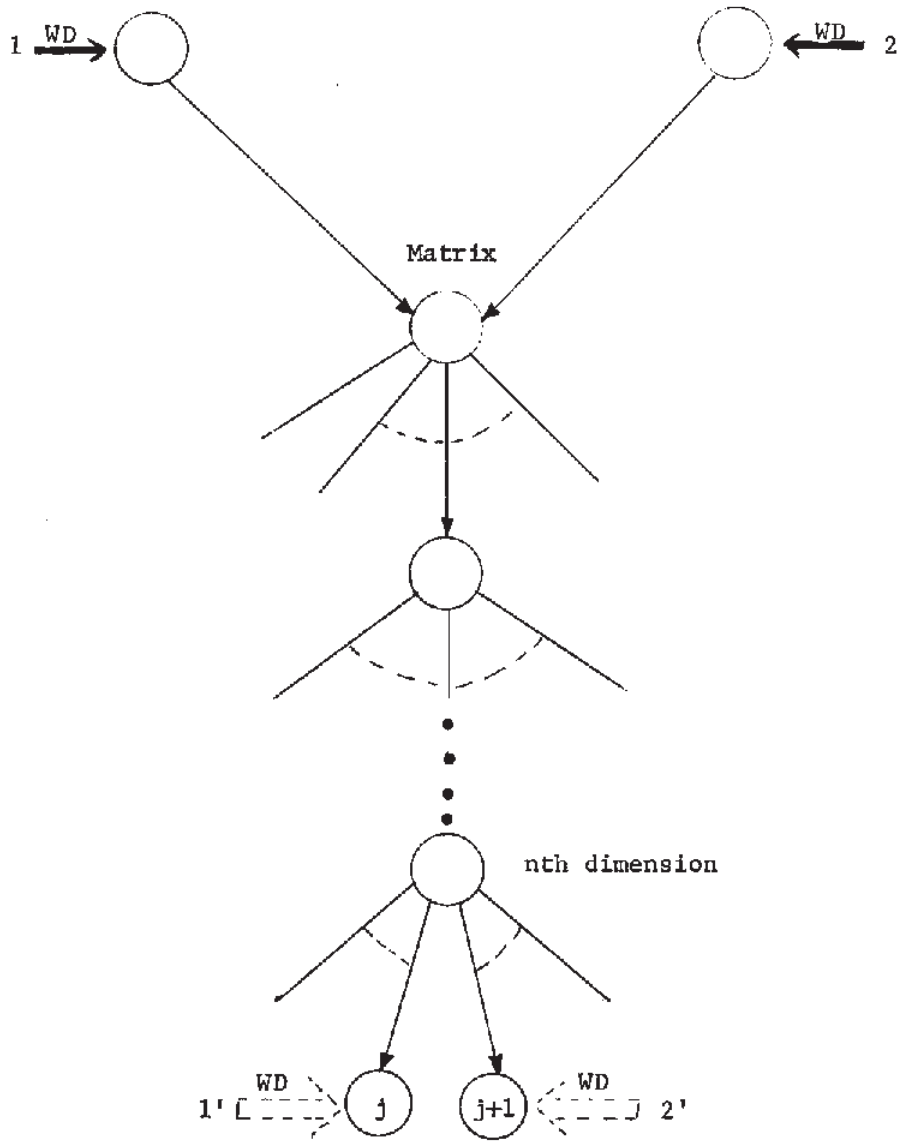


FIGURE 5