MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo No. 55

Sharing in J. B. Dennis' *Programming Generality, Parallelism and Computer Architecture*

Paul J. Fox

January 1971

ABSTRACT

This paper contains a detailed discussion of sharing within the framework of J. B. Dennis' CSG Memo 32, "Programming Generality, Parallelism and Computer Architecture" (a condensed version appears in Information Processing 68 (1968 IFIPS). This paper is not intended to stand alone and should be read only after having read Memo 32.

Sharing in C.S.G. Memo 32

Paul J. Fox

The problem of sharing data can be looked at on many different levels. From the systems viewpoint Dean Vanderbilt's recent work [1] offers valuable suggestions. In his approach the data segment itself is never shared but is always owned by a unique procedure. It is the procedure which may be shared and it is the procedure which solves any conflict in requests to it. It is important to note the distinction between system determinacy and user determinacy. A shared node can accept the first request to it but reject any other request until the first request is finished. The order in which two requests are acted upon then depends upon their relative speed down the heirarchy— an unknown quantity. The result is system indeterminacy, but the individual user's computation may still be fully determinant. Generalizing this point a little, many sharing problems can be greatly simplified if some indeterminacy is allowed (the fastest request gets served first). Vanderbilt's work utilizes such indeterminacy. But what about sharing and parallelism inside a single data segment by the procedure? Although I think there may well be some situations where indeterminacy can be tolerated, there clearly are going to be some situations where this cannot. It is with such situations where indeterminacy cannot be tolerated that this paper deals.

J. B. Dennis' C.S.G. Memo 32 [2] provides a data structure and operations designed to work on the individual procedure-data segment level (as opposed to the systems level of Vanderbilt's work). Memo 32 controls parallel operations through the use of pointers to the data structures. Each pointer

has either read or write (which includes read) capability. These pointers and their associated capabilities are propagated down the tree. At each node a write pointer usually implies the ability to write on any part of the tree subordinate to that node. Because of this structure the constraints insuring determinism are all local in nature. To me the local nature of the constraints is a very important property.

Unfortunately there is very little sharing. In fact the only sharing actually allowed is during the application of a procedure to a data structure. This sharing is provided by the operations of *bind* and *attach*. The *bind* operation establishes a link from a node with a write pointer to another node which also has a write pointer. I can find nothing wrong with the *bind* operation. The *attach* operation does, however, force constraints which are global in nature rather than local.

The *attach* operation upon receiving a write pointer, q, a read pointer, p, and a branch name, n, establishes a branch named n from the node pointed to by q to the node pointed to by p and a done is returned by q. Upon receiving a write pointer r the n branch is deleted and dones are returned on p and r.

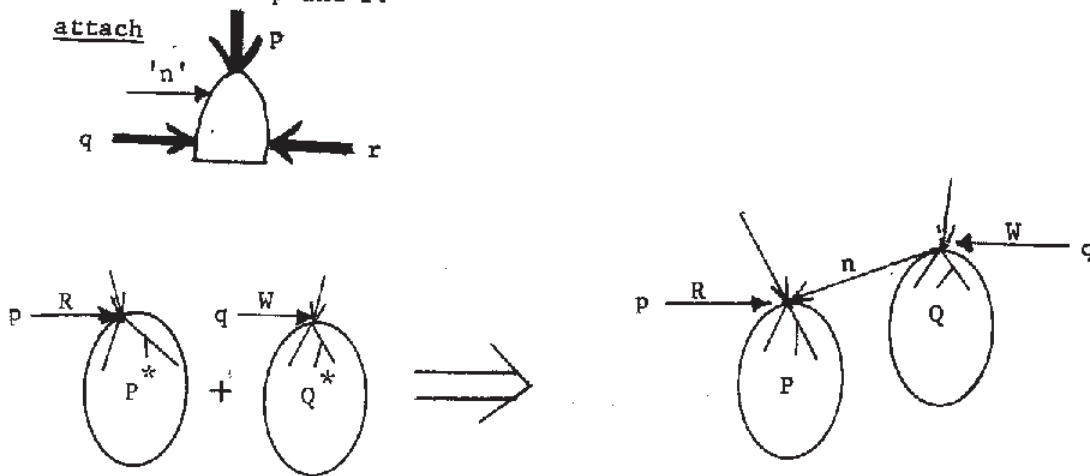

Fig. 1

*Arbitrary Data Structures

As the above diagram shows, the result is a write pointer, q, pointing to a node which has below it items (in P) to which there can only be read access. What's worse, this structure is now passed through <u>execute</u> to a different procedure where it may be passed through countless other procedure calls. Ensuring that the data in P is not changed could be difficult.

It seems to me that the simplest way to cure this problem is changing <u>execute</u> from a binary operator to a trinary operator (See Fig. 2). At present it requires a procedure structure with read access and a data structure with write access. I propose instead that there be two data structures — one having write access and the other only read access. The next problem is just what kind of operations do we want to allow between the <u>create</u> and the <u>execute</u> operations. We clearly do not want any changes to the data in the read only portions (P in Fig. 1), but can we allow attachments to read only structures? To answer that question we need to look at the following simplified problem.

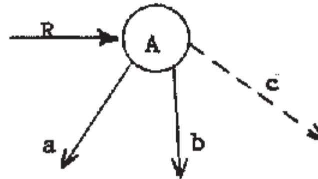

Fig. 3.

If we have acquired a read pointer to A, can we add branch c onto the structure only for the duration of a procedure call? It is indeed possible, but three conditions must be enforced.

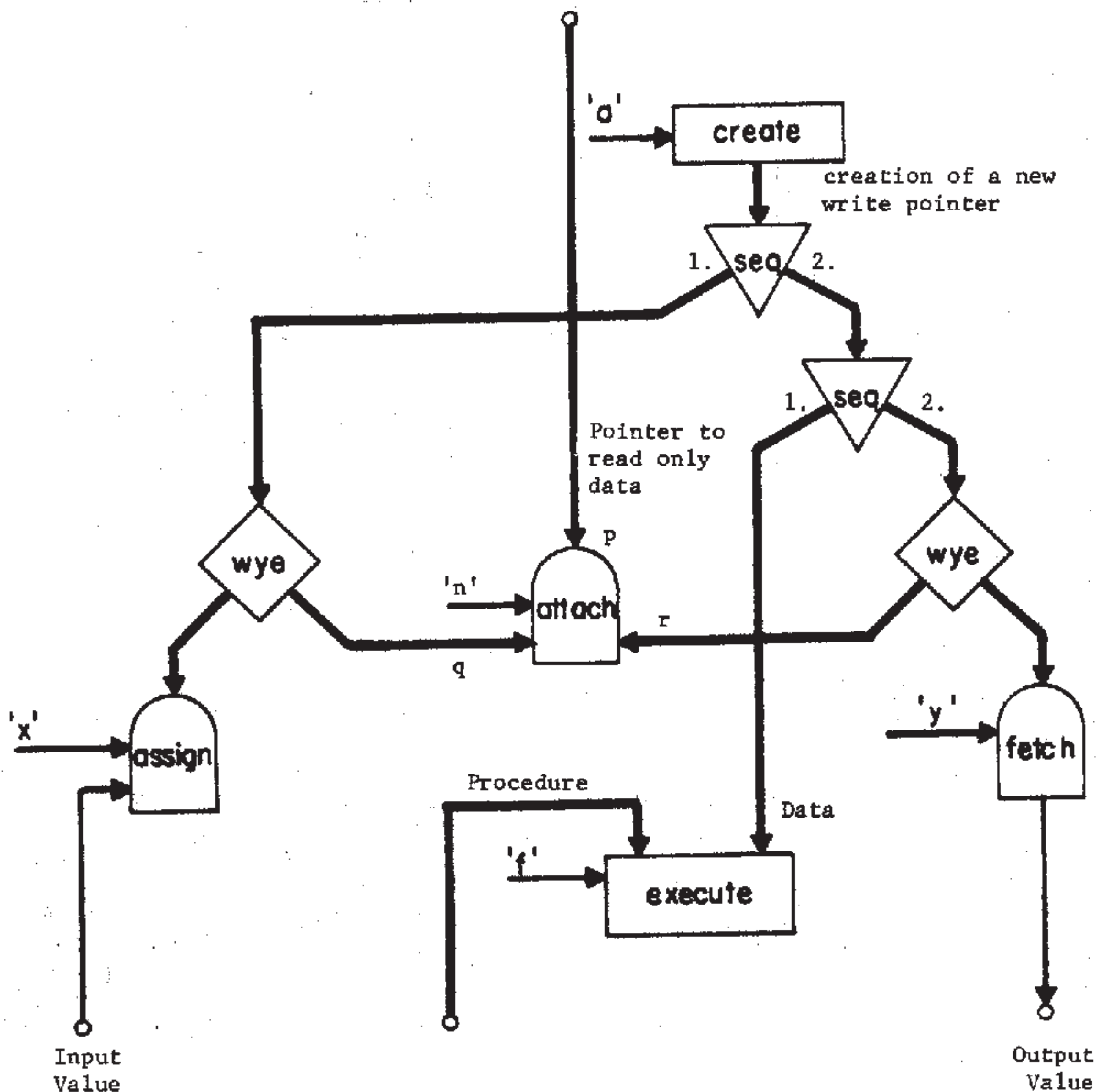

Fig. 2

This is Fig. 11 (p. 32) of CSG Memo 32 [2] with some of the notation changed to conform with the notation of this paper.

1. No other process can have "add on" capability — otherwise they can add on a branch c too.
2. No branch c already exists.
3. Any other read pointers to A (or below) do not use c.

   The problem is that ordinarily a mistaken use of c would result in an undefined state. However, when the branch was added, this would result in defined but erroneous behavior.

The first two conditions are not difficult to meet, but the third is clearly the sticker. The result is that read only data structures can be built up in any manner desired by the _append_ operator and sharing through the _bind_ operator. It cannot, however, build upon the data structures acquired through _attach_, even to just attach further structures to it. How can these restrictions be implemented? It turns out that the present _attach_ operator does the job fairly well — as long as we remember that we can never have a write pointer inside the "attached" structure (inside P in Fig. 1). The write pointer is then changed to a read pointer as it passes through the _execute_ operator.

What sequencing restrictions must be placed on this structure? There turn out to be three separate phases which must be ordered.

1. formation of the data structure
2. execution
3. dissolution of the data structure

But have we really changed anything? Yes! Whereas before we had a global restriction that had to be checked down through a whole procedure(s), we now have a local constraint that need only be checked between the create and execute nodes.

Storing Restrictions in the Data Structure

Another way to implement sharing is by means of storing the restrictions in the data. One reasonably straightforward method of implementation is as follows. Any node with more than one father is made a blocking node. This kind of node has the properties that a write pointer cannot gain access to this node or any node below it, but a read pointer can. Assign becomes straightforward. A block is simply inserted at the node to which p pointed.

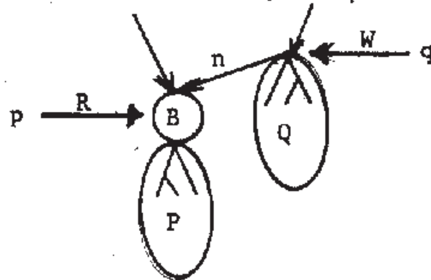

Fig. 4.

As a result, execute can revert to its original binary form. In addition, it is possible to add a new class of pointers called super write. A super write pointer would have the ability to pass through a block and change this or any lower node. In order to do this there can only be one super write pointer in a known disjoint structure. In addition there cannot be any read pointers in the system. To enforce this last condition it will probably be necessary to exclude all pointers, since a write pointer could become a read pointer anywhere in the structure. It would be very useful

if this disjoint requirement did not need to include the links formed by apply. Let us examine the situation (Fig. 5). The three problems are

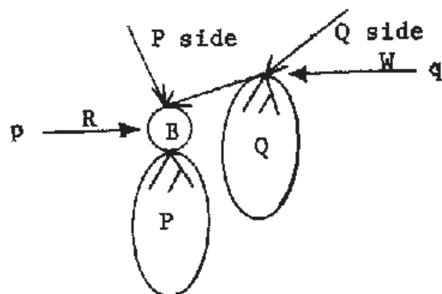

Fig. 5.

1) SW from P; R from Q side. 2) R from P side and SW from Q. 3) two SW's.

1st case: pointer p was needed to form the link and is not acknowledged until the link is dissolved. Therefore, an SW can never come down P while the link is present.

2nd case: Another read pointer can indeed pass through pointer p. Can a SW come down Q? Only if q (the write pointer which caused the branch's formation) is acknowledged back up the system before r arrives. If q is never acknowledged back up the system until r arrives, no SW can come down the Q side. This requirement is met in practice so its explicit requirement causes no real difficulty.

3rd case: combine cases 1 and 2.

As a result it is not necessary to consider links caused by attach/bind in the definition of disjoint, provided the condition of case 2 is met.

There is some question of whether the restrictions should be applied to the branch or the node. It seems to me that from a semantic point of

view one can say the restriction applies to the node if all links coming into the node must have the same restriction. If on the other hand, the restriction need not apply to all branches to a node; it should be called a branch restriction. Can these blocks be branch restrictions or must they be node restrictions. If we are only dealing with apply sharing they can be branch restrictions, since pointer p already protects the shared data from the P side while a block on branch n would protect it from the Q side. If, however, we wanted to allow permanent sharing, the restriction could not be a branch restriction.

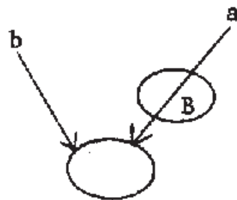

Fig. 6.

In Fig. 6 a write pointer from b could conflict with a read pointer from a.

What good is the above proposal for blocking nodes? I think it provides a clean, simple procedure for allowing shared data in situations where there is either very little shared data, or the ratio of read operations/write operations is large. While the sequencing restrictions on super write pointers are cumbersome, it does provide a deterministic way to update shared data without the extreme complexity of Campbell-Grant's work. [3] I believe this simplicity is an extremely important property in primitive operations. The cost of this system is, of course, that every operation involving the data structure must include a check for a block. How costly this would be depends on the implementation.

1. D. Vanderbilt, "Controlled Information Sharing in a Computer Utility," MAC Technical Report TR-67, 1969.

2. J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture," CSG Memo 32, Project MAC, M.I.T.

3. I. R. Campbell-Grant, S.M. Thesis Proposal. Department of Electrical Engineering, M.I.T., 1970.