MDC 2/1
5/15/75

MUMPS DEVELOPMENT COMMITTEE

SUBCOMMITTEE ON IMPLEMENTATION

MUMPS GLOBALS AND THEIR IMPLEMENTATION

A Type B Release of MDC Subcommittee #2, Implementation

Anthony Ira Wasserman

Anthony Ira Wasserman, Chairman
MDC Subcommittee #2

The reader is hereby notified that this document neither reflects MUMPS specifications nor any implied support by members of Subcommittee #2 of the MUMPS Development Committee or their sponsors, but that it is being offered for possible consideration by Subcommittee #2. It is being made available in order to establish better communication between MDC Subcommittee #2 and that segment of the public interested in MUMPS language development.

Anyone reproducing this release is requested to include this introduction.

MUMPS GLOBALS AND THEIR IMPLEMENTATION

Table of Contents

Acknowledgments

This paper has benefited from the ideas and work of many people. We have had many fruitful discussions with implementors in the MUMPS community, and would especially like to thank Jack Bowie, Bob Rees, and Paul Egerman. The MUMPS standardization effort has aided us greatly in sharpening our understanding of current implementation techniques. We are grateful to the many members of the MUMPS Development Committee for sharing their thoughts on globals with us.

Finally, and perhaps most importantly, we wish to express our deep appreciation to Tina Walters who, with assistance of Marina Mancia, did the typing of this manuscript.

MUMPS GLOBALS AND THEIR IMPLEMENTATION

Anthony I. Wasserman, David D. Sherertz,
Charles L. Rogerson

INTRODUCTION

The multiprogramming system MUMPS[1], first developed at the Laboratory of Computer Science at Massachusetts General Hospital in the late 1960's, supported a high-level interpretive programming language, also known as MUMPS. MUMPS was designed to facilitate the creation of conversational programs which can share a data base on a small time-shared computer. The data base of MUMPS is hierarchically organized and consists of tree-structured files called global arrays or, simply, globals.

Since its initial development, MUMPS has been used principally as a data base management system. As with any data base management system, the time required to service an interactive user is highly dependent upon the speed with which a given piece of information can be retrieved from its storage location. As a result, efficient utilization of mass storage devices has always been a primary consideration in MUMPS implementations.

This document is a report of the results of a study done on the MUMPS data base mechanism as part of a larger overall study of implementation techniques for MUMPS. This report is divided into five sections. Section one describes and defines some of the basic notions of the theory of graphs and trees which are essential to a complete understanding of a hierarchical data base organization. Section two describes and defines the ways in which globals are used in MUMPS and gives some illustration of this use. Section three gives a detailed description of the method used to implement global arrays in existing

1 Massachusetts General Hospital Utility Multi-Programming System

MUMPS systems. Section four gives consideration to optimization of the performance of global implementation methods, focusing upon the relationship between storage allocation and effect upon access time. Section five analyzes the strengths and weaknesses of MUMPS globals, and also gives some recommendations for providing a more powerful and efficient data base management capability within the existing framework of MUMPS.

The material presented in this report assumes some prior knowledge of MUMPS and MUMPS globals[2]. The aim of this report is to give an overview of the MUMPS data base structure and to provide sufficiently detailed information to implement globals.

2 The reader who is unfamiliar with MUMPS is referred to [Johnson, 1974].

PART I - THEORY AND USE OF MUMPS GLOBALS

1. Tree Structures: Terminology and Definitions

1.1 Graph theory and information structures

The mathematical area known as the theory of graphs is extremely important in developing a thorough understanding of information structures which can be used to store and interrelate data. Graph theory has proved to be a very effective analytic tool in computer sciences, since a graph may be drawn to represent an arbitrarily complex data structure or to represent the execution profile of a computer program. Because of their inherent simplicity, graphs are often very useful in describing classes of data structures in a formal way.

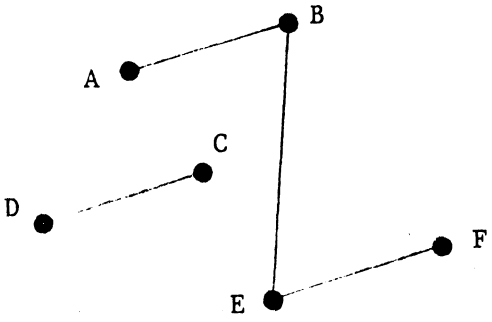
Unfortunately, there is as yet no standard terminology in this field, and a variety of terms have been used to mean the same thing. In this report, we shall conform to the terminology used by Knuth [Knuth, 1974] unless stated otherwise[3].

A graph is generally defined to be a set of points (called vertices or nodes) determined by a set of lines (called edges) adjoining certain pairs of distinct vertices.

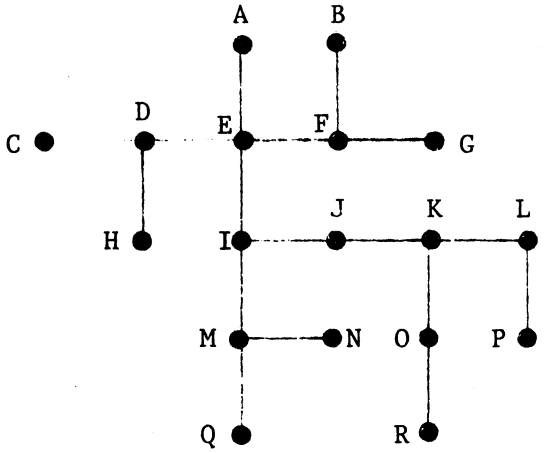
Figure 1 shows four graphs. Figure 1a is a graph with six nodes and four edges. Figure 1b is a connected graph made up of 18 vertices and 17 edges. A connected graph is one in which it is possible to construct a "path" between any two vertices. Figure 1c is a connected graph with a cycle. If we use Knuth's definition that there is at most one edge joining any pair of vertices, then a cycle is a path of length three or more from a vertex to itself. The path ABEDA and the path BCEB are cycles in Figure 1c.

One type of graph which is of particular interest is called a tree. Trees have been called "the most important nonlinear structure arising in computer

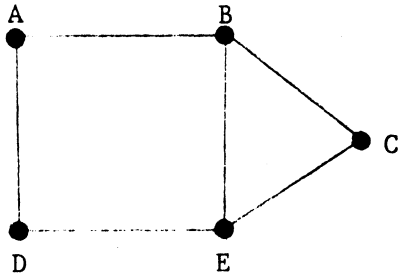
3 This use of terminology is occasionally different from terminology used elsewhere in the MUMPS literature.



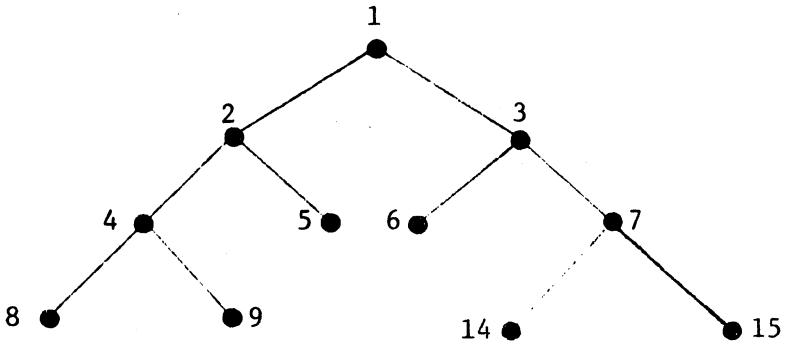
a) A graph with six nodes and four edges



b) A connected graph



c) A connected graph with a cycle



d) A binary tree

Figure 1 - Graphs

algorithms" [Knuth, 1974]. Tree structures have been studied and used for many years, long before the advent of computers, and have been applied in a wide variety of areas.

A tree is formally defined as a connected graph with no cycles. Figure 1b is a tree, as is Figure 1d.

A forest is defined to be a set of zero or more trees.

In our discussions, we will generally refer to oriented trees, in which a particular node of the tree is designated as the root. By convention, a tree is usually drawn with the root at the top and all of its branches beneath it (See Figure 2). When a tree is drawn in this manner, it becomes apparent that the tree may be separated into levels determined by the number of edges which must be traversed to reach the given node from the root. The height of the tree will then be defined as the maximum distance from the root to any node.

There is a considerable amount of terminology that has been developed for discussing trees, not all of which is consistent. Most commonly used are the terms from genealogical charts (family trees), in which subsequent generations of an individual are depicted.

For any given node in the tree, nodes on the path between that node and the root are called ancestors. The most immediate ancestor is generally called the parent.

Similarly, those nodes that can be reached from a given node by heading away from the root toward the leaves of the tree are called the descendants of the node. The most immediate descendant is usually called a child. A node which has no descendants is called a leaf of the tree, or a terminal node.

A group of nodes which have the same parent are termed siblings and are said to be members of a filial set.

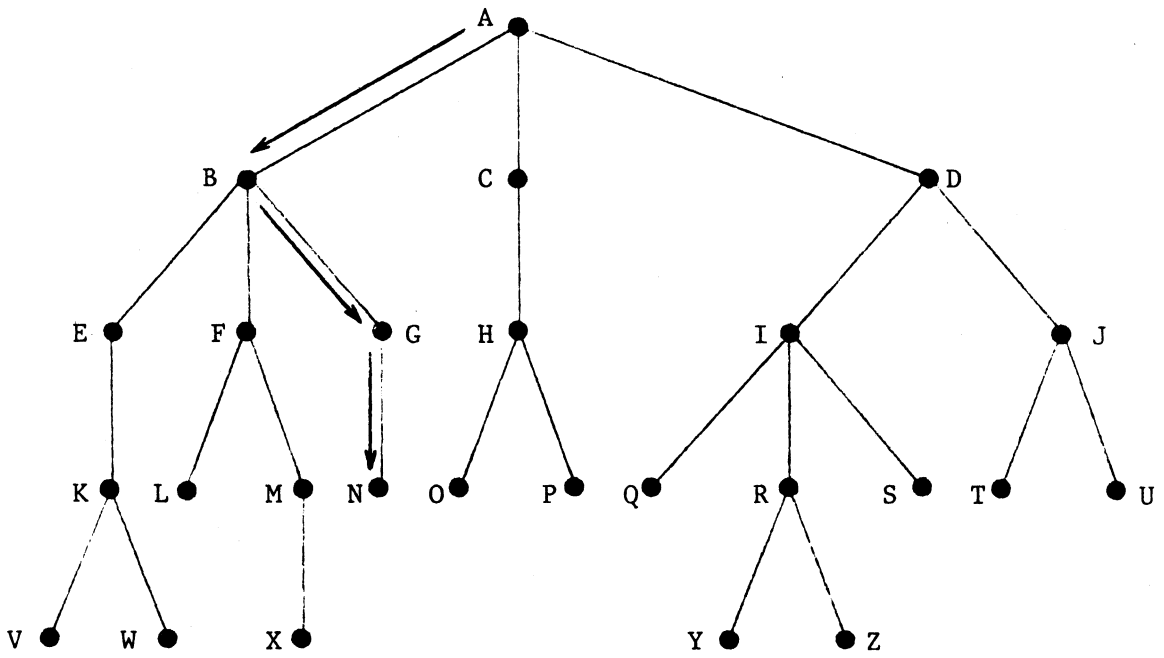


Figure 2 - A tree with root A

As is clear from Figure 2, there is a unique path from the root to any given node of the tree, and these nodes may be identified and referenced by a logical addressing notation, called a trace. A trace is a set of nodes representing the sequence of nodes which must be traversed to reach the given node. Thus, in Figure 2, the trace for node N is (A,B,G,N). To find the unique path for the route to the node at level N, we construct a sequence of length N+1, with the root as the first element, an element from level one (one of the children of the root) as the second element and so on, until the desired node becomes element N+1.

1.2 Binary trees

One tree structure which has been of particular interest to mathematicians and computer scientists is a binary tree. A binary tree is defined to be a tree in which no node has more than two children. A binary tree has the advantage that each node can be uniquely numbered, with the number for a node at level M in the range between 2^M and 2^{M+1} -1. The binary tree is also well

suited to various kinds of branching logic, in which true and false answers determine the path to be taken from a particular node. Figure 1d above shows a typical binary tree, which has been numbered according to the scheme suggested here.

It has been shown that any tree may be converted to a binary tree by a sequence of steps, as follows:

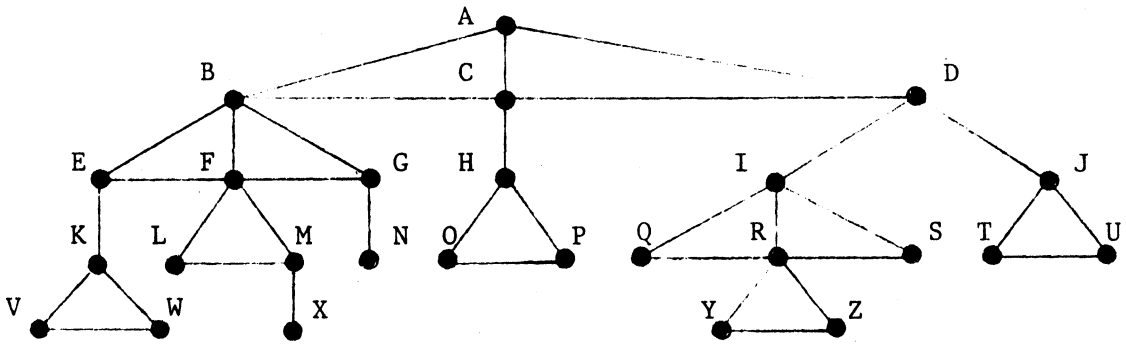
1) For each node, connect it with all of its siblings;

2) For each node, delete the path between the node and all of its children other than the first;

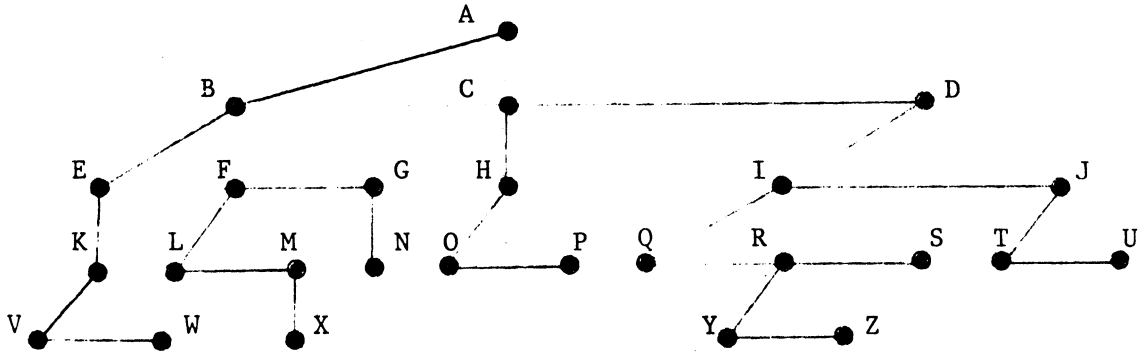
3) Tilt the diagram so that it resembles a tree.

Figures 3a, 3b, and 3c show the three steps of this transformation as applied to the tree in Figure 2. (This transformation process can be extended to a forest of trees by simply connecting the roots of the various trees as if they were siblings, and applying the same three steps.) Note that the relationships among the nodes are apparently different.

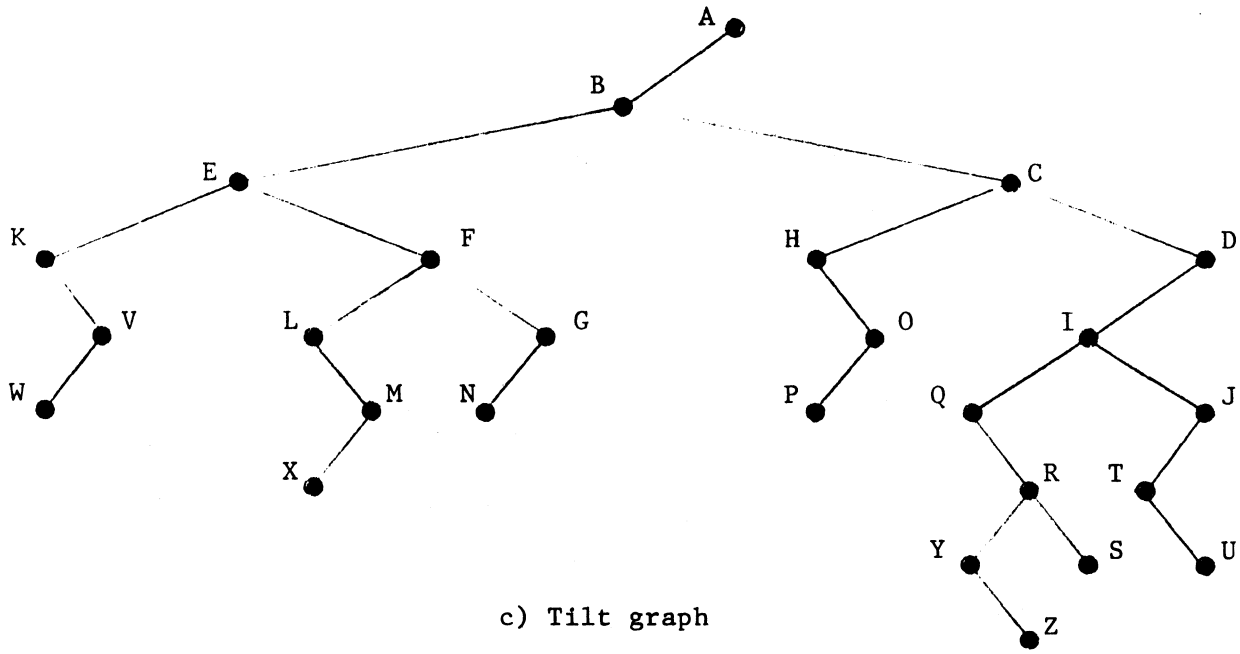
There is much more that could be shown in the area of graphs and tree structures. However, they are not essential to understanding of the material which follows. Readers desiring more information on trees and graph theory are referred to [Busacker and Saaty, 1965], [Deo, 1974] , [Knuth, 1974], or [Ore, 1963].



a) Connect siblings



b) Delete paths between node and all children other than the first



c) Tilt graph

Figure 3 - Conversion of tree to binary tree

2. Tree Structures and MUMPS

2.1 Hierarchical structures

The tree structure described above is extremely flexible, in addition to being intuitively attractive and powerful. While it can be shown that certain kinds of information structures cannot be handled with a tree representation, the vast majority of structures which are desirable to represent on a digital computer can be organized within the framework of trees. Programming languages such as COBOL and PL/1 incorporate a hierarchical data definition capability, equivalent to trees. Other programming languages such as PASCAL, SIMULA 67, and ALGOL 68, incorporate even more powerful data structuring capabilities.

The popularity of the hierarchical approach to data organization is due to the fact that many bodies of information can be described as trees and are frequently more clearly conceptualized when this is done. For example, the structure of a textbook, as reflected in a table of contents, is a tree structure. The title is the root, the chapters are the children, the various sections are the children of the chapters, and the text of the sections are the terminal nodes or leaves.

This hierarchical approach to data base organization has been incorporated into the programming language MUMPS. Within MUMPS, tree structured files are known as global arrays, or simply globals. With the possible exception of the root, each node of a global array may contain data. The MUMPS language makes a strong attempt to preserve the close relationship between the general notion of the tree and its utilization in a programming language. Although the globals represent a form of file system, the syntax of the MUMPS language is intended to make most of the details of the file system invisible to the programmer and user.

Because MUMPS was developed for an environment in which most of the applications would involve access to and modification of the data base, it is necessary to give considerable attention to the problem of implementing globals efficiently. The speed with which a user's request can be serviced is highly dependent upon the time it takes to locate a given node in a global and provide that information to the user. Unlike COBOL or PL/1, in which the hierarchical structures are static, MUMPS allows dynamic trees, in which the amount of information stored in each node can be changed, and in which the number of nodes can also be altered. As we shall see, this degree of flexibility can be achieved only at the expense of implementation overhead. Considerable effort on the part of previous MUMPS implementors has been devoted to the problem of efficient allocation of storage of MUMPS globals, combined with techniques to improve the efficiency of access.

2.2 The usage of globals in MUMPS

MUMPS supports a data base with an arbitrary number of globals. The MUMPS language allows the programmer to reference nodes of the global and to assign values, either numerics or strings, to individual nodes; assignment of a value to a node automatically creates that node. Nodes or entire globals may also be explicitly deleted. All linkages between nodes and levels are automatically updated with the creation or deletion of nodes.

Within MUMPS, global variables are treated much the same as local variables, so that global values may be used in expressions in a consistent manner. (Local arrays can be created analogously to the global tree structure.) Because MUMPS is a multi-user system, there are facilities for reserving all or part of a global (or local) array, in order to facilitate multi-user access to the shared data base and to prevent deadlock. In addition, two functions are provided in MUMPS for working with global (and local) arrays: (1) the $DATA

function, which allows tracing the entire node and data structure of a global and (2) the $NEXT function, which allows tracing the ordering of nodes at a given level of a global.

The language makes no restrictions about the value of the "subscripts" at the various nodes. Because MUMPS is a declaration-free language, the system makes no assumptions about the lower bound or the upper bound of such subscripts. Instead, a MUMPS global is treated as a sparse tree, and the only nodes which are created are those to which a value is explicitly assigned or those which must be created in order to provide a path from the root to a node which has been assigned a value. This generality allows the programmer to utilize nodes via a mechanism which is appropriate for the particular task being programmed. However, it should be noted that this degree of generality involves a trade-off of space against time when compared with a system in which a linear ordering of subscripts is required (see subsection 3.5).

In general, then, a global name, followed by an arbitrary number of subscripts, may refer uniquely to a node in a tree structure, to which a value may be assigned or from which a value may be retrieved. There is one major exception to this rule, however - a syntactic structure called the naked reference, which is introduced for compactness of notation and improved efficiency of execution. Many tree processing algorithms involve traversal methods or processing sequences in which the program will work with a group of siblings or with the descendants of a given node. Accordingly, the language includes a scheme whereby the invariant part of a global reference may be omitted, and only those subscripts differing from the previous global reference are included.

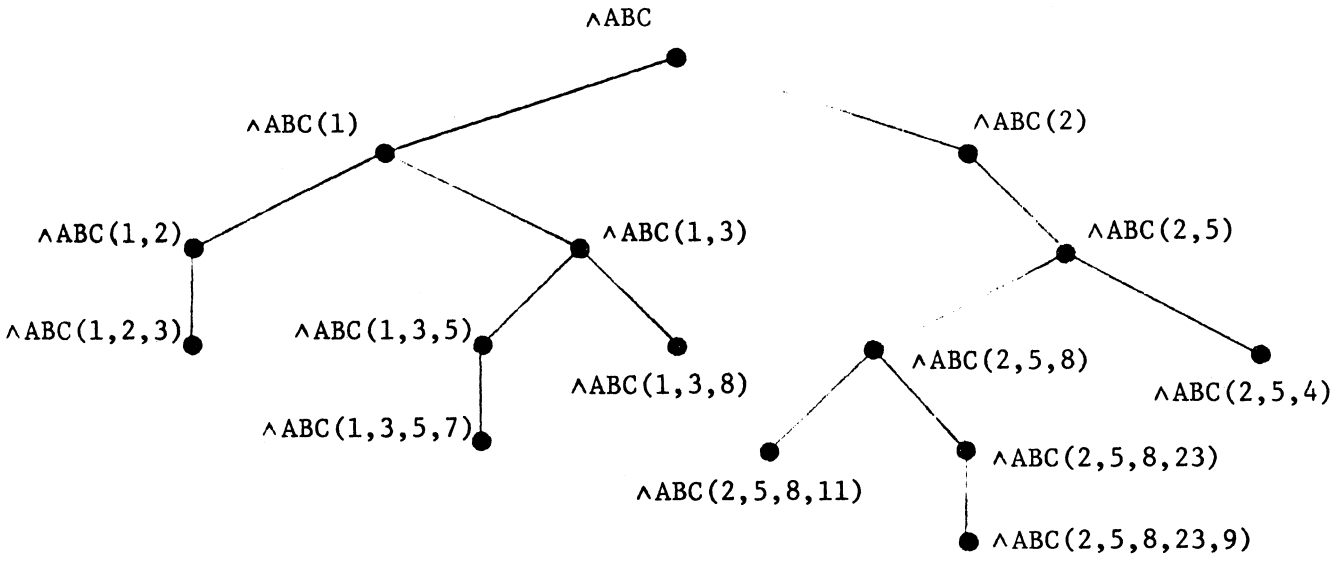


Figure 4 - A global in MUMPS (Addresses only; data unspecified)

Thus, in Figure 4, if one has most recently referred to ^ABC(2,5,8,11) and then wishes to refer to ^ABC(2,5,8,23), it is possible to abbreviate the latter reference to ^(23). This notion is extendable to lower levels as well; for example, a desire to reference ^ABC(2,5,8,23,9) next could be abbreviated to ^(23,9). Note, however, that the extension is in one direction only - away from the root node; to reference ^ABC(2,5,8,11) would now require a full reference. The abbreviation is unambiguous, but dependent upon the order of execution of statements during interpretation of the program. For example:

```
SET X=^ABC(1,2,3) IF Y SET X=^XYZ(23,16,6)
SET Z=^(4)
```

If Y=0, then Z obtains the value of ^ABC(1,2,4); otherwise, it obtains the value of ^XYZ(23,16,4). It also requires the programmer to know that evaluation of global references on the righthand side of an assignment statement occurs before the assignment to the variable itself. Thus, if one wished to assign the sum of ^ABC(1,2,3) and ^ABC(1,3,8) to the variable ^ABC(1,3,5), a proper notation would be:

```
SET ^(5)=^ABC(1,2,3)+^ABC(1,3,8)
```

As shall be shown later, use of this naked reference is reflected in the implementation of global accesses, so that it becomes unnecessary to re-trace the path from the root of the global to the level at which the reference is being made. This syntactic abbreviation improves the efficiency of execution, and is thus a valuable tool for the MUMPS programmer if used properly.

2.3 An illustrative example

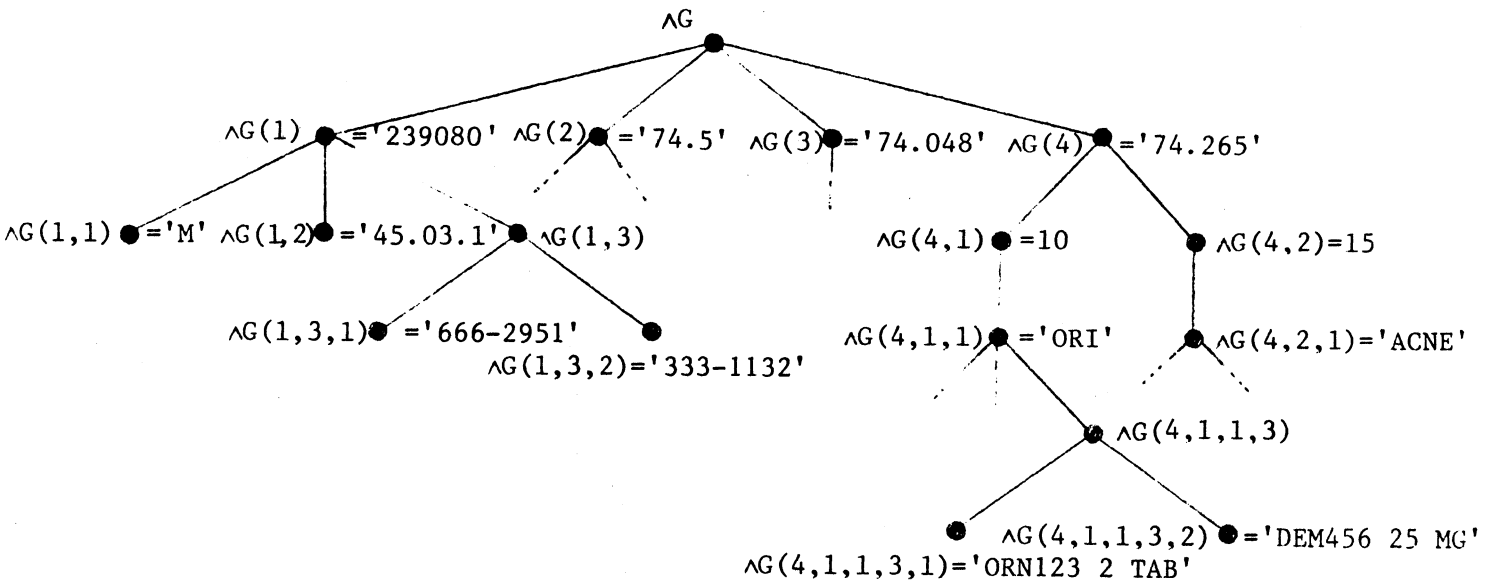


Figure 5 - A pictorial representation of part of a MUMPS global

Consider the global ^G depicted in Figure 5, which might be considered to be a part of a patient's outpatient medical record summary. The information stored in each node is of varying type. ^G(4,1) has an integer value; ^G(4,1,1,3,1) has a string value; ^G(4,1,1,3) has no associated value, but simply a pointer to its descendants. Given this partial representation of a global, the use of some MUMPS commands upon this structure can be examined. Table I illustrates the effect of these commands on the global shown in Figure 5.

Table I

Commands	Effect
SET ^G(1,3,1)='666-2951'	value of node ^G(1,3,1) becomes '666-2951'
SET VAL= ^G(4,2,1)	value of variable VAL becomes 'ACNE'
WRITE "HOME PHONE IS ", ^G(1,3,2)	"HOME PHONE IS 333-1132" is output
KILL ^G(2)	node ^G(2) and its descendants are deleted
KILL ^G	entire global is deleted
LOCK ^G(4,1)	the node ^G(4,1) and its descendants are reserved to user; no ancestor of ^G(4,1) may be reserved while LOCK is in effect
LOCK	all reserved globals/nodes are released

Functions	
$DATA	returns a code indicating the type of node
SET X=$DATA(^G(4,1,2))	node does not exist; X returns 0
SET X=$DATA(^G(1,1))	node is terminal; X returns 1
SET X=$DATA(^G(4,1,1))	node contains pointer and value; X returns 11
SET X=$DATA(^G(1,3))	node contains pointer only; X returns 10
$NEXT	returns value of next subscript
SET X=$NEXT(^G)	error
SET X=$NEXT(^G(1))	X returns 2
SET X=$NEXT(^G(4,1,1,3,-1))	X returns 1
SET X=$NEXT(^G(4))	X returns -1; no next node

It should again be noted that the assignment of a value to a node is not dependent upon the pre-existence of nodes at lower levels (those between the referenced node and the root). If a path does not already exist from the root to the referenced node, one will be created by the system, consisting of nodes containing only pointers, but with no data. Referring back to Figure 5, the creation of ^G(4,1,1,3,1) caused the creation of ^G(4,1,1,3) in order to form a path from the root.

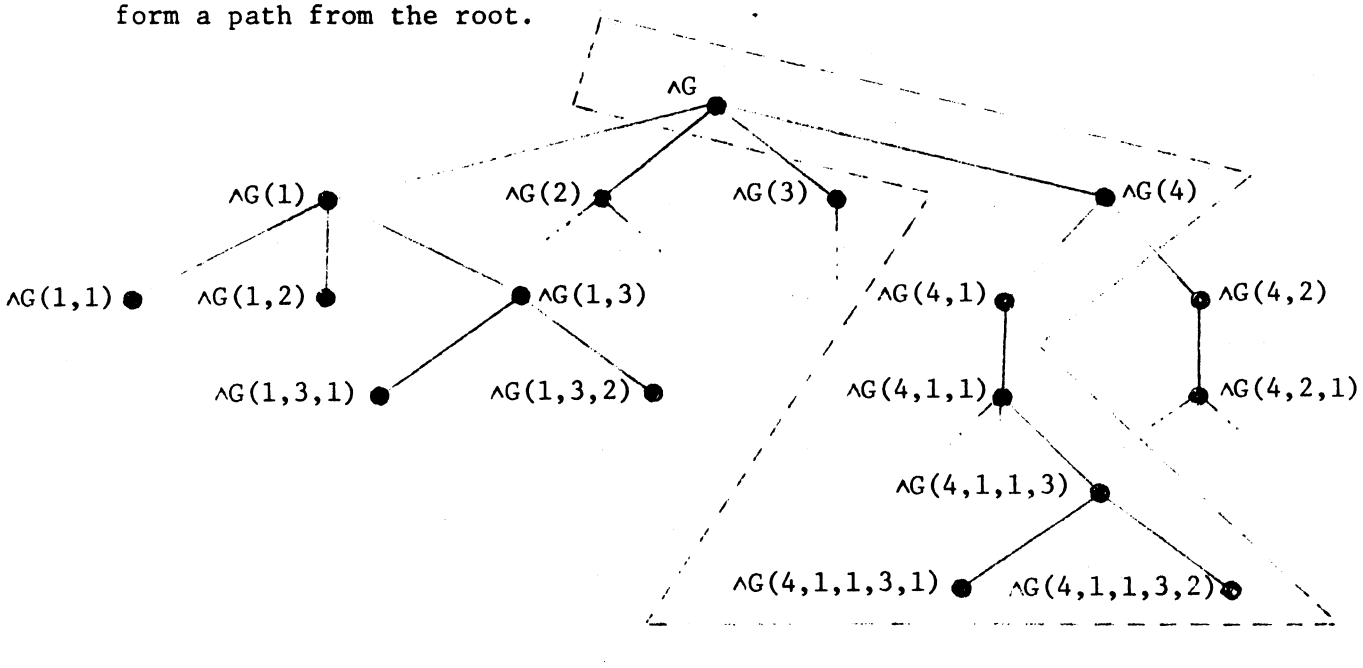


Figure 6 - Effect of LOCK ^G(4,1) on access to ^G

In Table I, the effect of LOCK ^G(4,1) on the global G shown in Figure 5 was explained briefly. Figure 6 illustrates the effect of this command in more detail. It may appear that other users are now restricted from using the entire global, but this is not the case. They are simply prevented from specifically performing a LOCK on ^G(4) or ^G, as well as on any of the descendants of ^G(4,1). (Note that it would be possible for another user to perform LOCK ^G(2) or LOCK ^G(4,2).) This restriction is made in order to prevent two users

from interfering with one another. If a user were permitted to lock ^G while another user locked ^G(4,1), then the two users could attempt to write the same global information simultaneously or the user who had locked ^G could delete the entire global while the other user was trying to reference some node in the global. The MUMPS language, since it is also a programming system, must make provisions for mutual exclusion among the various concurrent users of the system. The LOCK command is one mechanism which can be used to accomplish this.

It is not the intent of this document to discuss detailed usage of MUMPS globals or to give more than passing attention to some of the programming considerations involving the effective utilization of MUMPS globals. Readers wishing that information should consult [Johnson, 1974], especially Chapters 9 and 11, [Peck and Greenes, 1974], particularly Section A, or other introductory MUMPS literature. The goal rather is to treat some of the implementation considerations so that the reader can implement globals on an arbitrary computer on adequate secondary storage capacity. Accordingly, the remainder of this document deals with those implementation issues and with some of the problems involved in trying to optimize storage access and allocation.

PART II - IMPLEMENTATION OF MUMPS GLOBALS

3. Existing Implementation Techniques

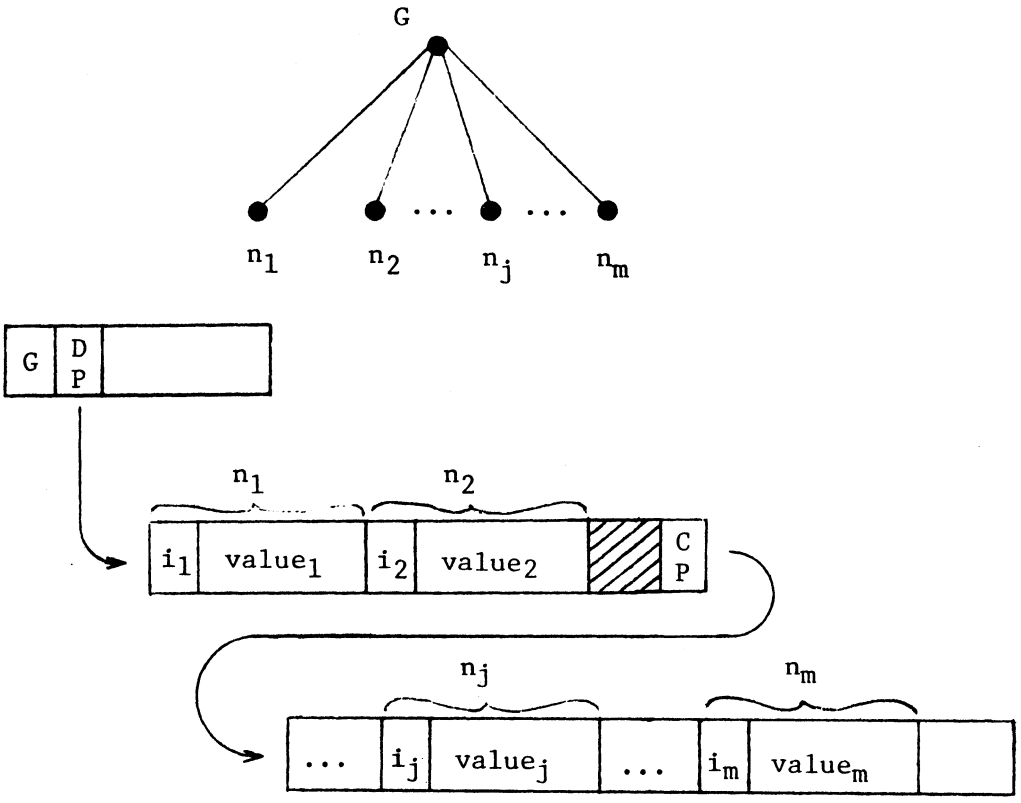
3.1 General strategy

The general implementation strategy for the tree structure of hierarchical data bases is a combination of disc addresses, and physical disc contiguity [Bernstein, 1974]. All of the existing MUMPS systems use random access discs for storage of globals, and almost all have been implemented in assembly language. As a result, they are highly dependent upon the structure of the computer system on which they run, and upon the random-access I/O methods of its implementation language. The implementation approach for globals is similar among various MUMPS systems, with differences arising mostly from varying word and disc sector lengths, and the use of alternative disc strategies [Bowie, 1973].

Globals have always been implemented as a set of chains of one or more fixed length disc blocks, the first of which is called the head block or header for that level. The disc block containing the root node is referred to as the header for the entire global, and is often kept with other root nodes in a global directory for the entire data base (see subsection 3.2.4). Nodes are stored in these blocks; the value of the subscript is stored with its current value. Because the MUMPS language is declaration-free, nodes are created only upon explicit assignment and removed only upon explicit deletion. Therefore, the storage structure treats MUMPS globals as sparse multi-dimensional arrays with unordered subscripts.

Present implementations of globals also require all information about any particular node (that is, its subscript value, the pointer to its children, and its value) to fit within a single disc block. Some approaches to extending this to allow a single node to occupy multiple disc blocks will be discussed below.

When it becomes impossible to store all the nodes for a set of siblings in a single block, continuation blocks are automatically created and linked to the previous block by means of a continuation pointer. There is also a need for a mechanism to handle multi-level globals, so that a parent node can point to its descendants. The technique used is a pointer called a down pointer, which points to the header for a node's children. The conceptual notion of continuation pointers is illustrated in Figure 7, and Figure 8 shows the gross structure of a multi-level global with down pointers (both these figures show only the general node structure, without the representational details for each node).



In the diagram

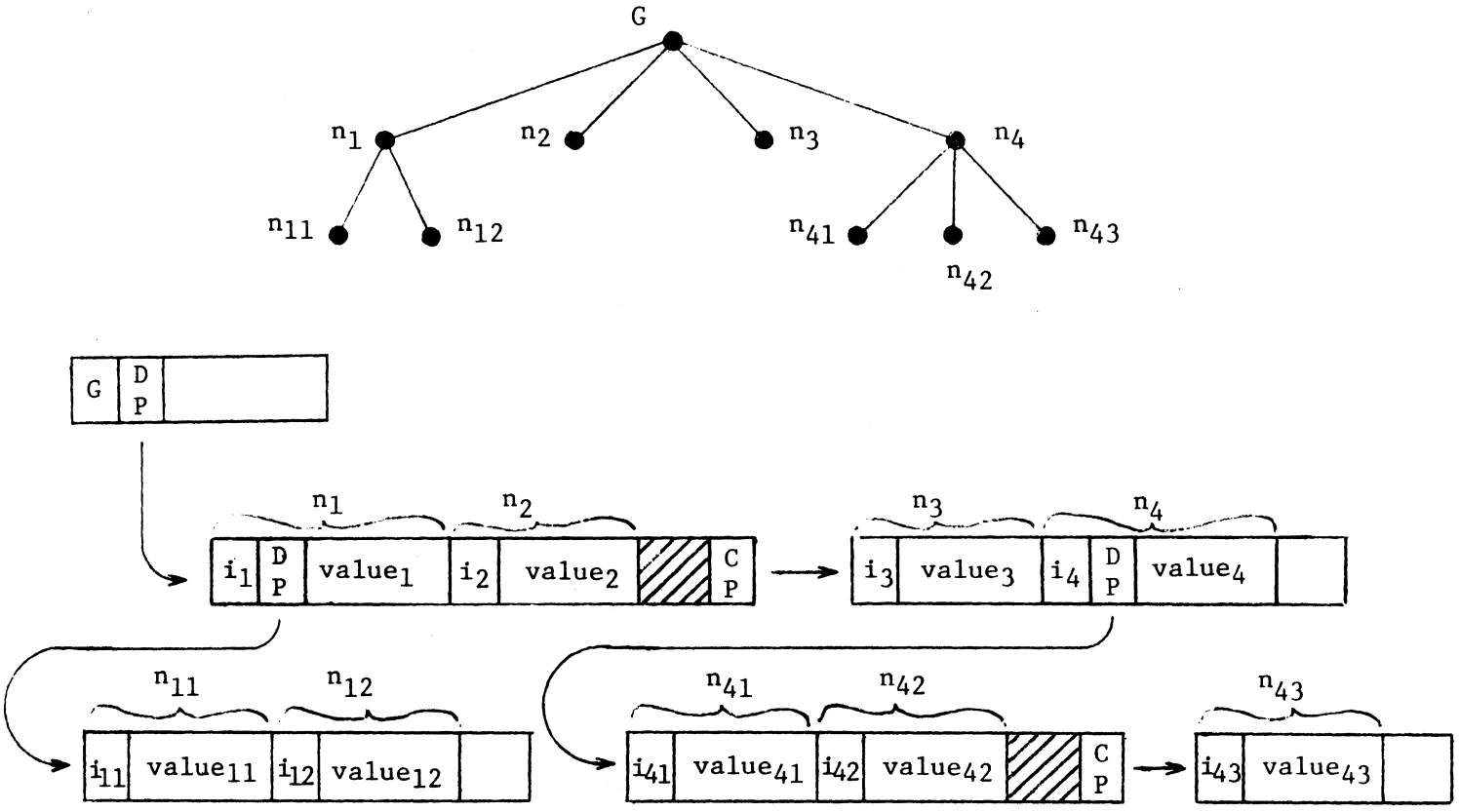
i_k represents the subscript for node n_k, $0 < k < m+1$

$value_k$ represents the value for node n_k

DP is a down pointer

CP is a continuation pointer

Figure 7 - Use of continuation pointers



In the diagram

i_j represents the subscript for node n_j

$value_j$ represents the value for node n_j

DP is a down pointer

CP is a continuation pointer

Figure 8 - Typical representation of a multiple-level global structure

In Section 1, a method was outlined for converting any general tree structure to a binary tree. This transformation closely resembles the actual representation used for MUMPS globals. All the implementations surveyed have only one pointer from a parent node to the set of its children. When an arbitrary global is transformed to a binary tree, it can be seen that traversing the left subtree represents tracing a down pointer, while traversing the right subtree represents continuing within a block or following a continuation pointer. Figure 9 is a redrawing of Figure 8 to permit comparison between the binary tree and the global structure.

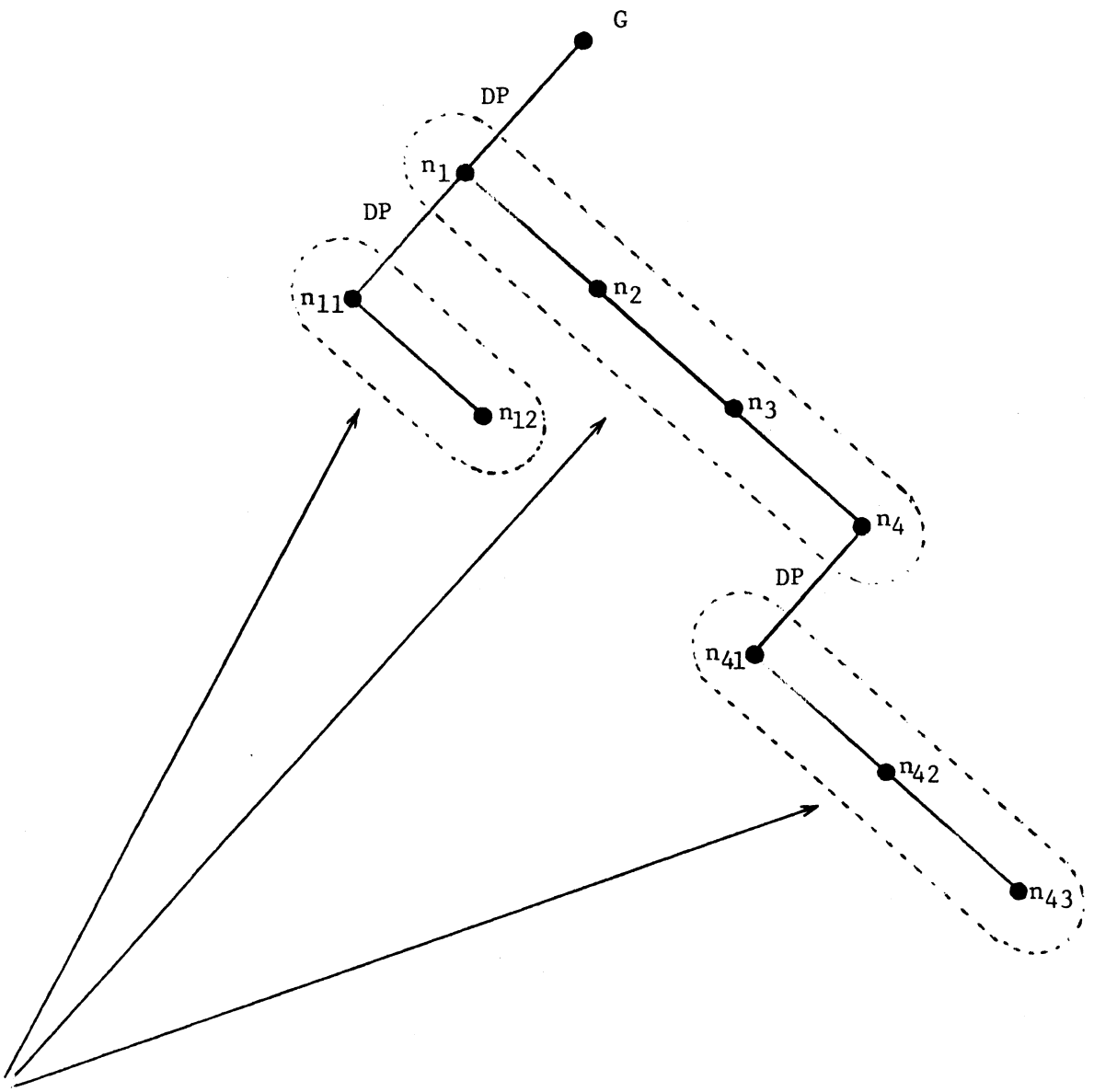


Figure 9 - Global of Figure 8 redrawn to show storage structure as a binary tree

3.2 Global data structures

3.2.1 A data representation technique

Each node is stored in a disc block as a subscript (its index), along with data consisting of a down pointer and/or a value. A value associated with a node may be either numeric or string. The structure of the node is dependent upon the existence of a down pointer and upon the type of data (if any) associated with the node. For the MUMPS language, the following types of nodes can be identified:

1) Integer value

2) Real numeric value (noninteger)

3) String value

4) Pointer only

5) Integer value plus pointer

6) Real numeric value plus pointer

7) String value plus pointer

Figure 10 shows ways in which each of these node types can be represented on disc storage in an unambiguous way. Various specific implementations may choose to modify this scheme to achieve a more compact storage scheme or to accommodate specific ranges of subscripts or values. Within the present discussion, the following assumptions have been made:

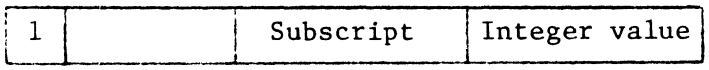
1) Subscript values require four bytes of storage[1]

2) Numeric values require four bytes of storage

3) A string node will fit in one sector.

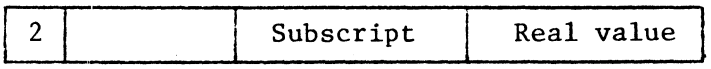
1 By limiting the subscript value range to 29 bits instead of 32, for example, a 3-bit code indicating the type of node could be packed into the same byte with some subscript information in order to effect compression of storage.

Number of bytes



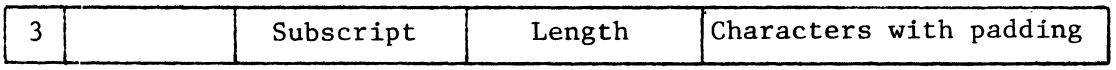
1 3 4 4

a) Integer value node 12



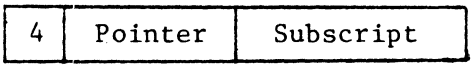
1 3 4 4

b) Real value node 12



1 3 4 4 N((Length+N-1)/N)

c) String value node (N = characters per word)
(String is padded with nulls to an even word boundary) 12+N((Length+N-1)/N)

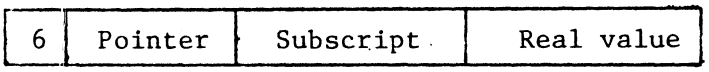


1 3 4

d) Pointer value node 8

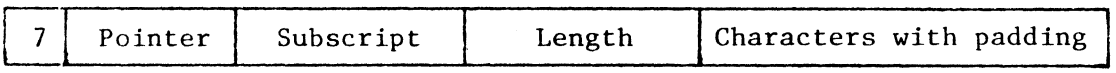
1 3 4 4

e) Pointer with integer value node 12



1 3 4 4

f) Pointer with real value node 12



1 3 4 4 N((Length+N-1)/N)

g) Pointer with string value node (N = characters per word)
(String is padded with nulls to an even word boundary) 12+N((Length+N-1)/N)

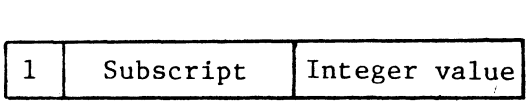
Figure 10 - Representation of different types of global nodes

3.2.2 Data storage compression

3.2.2.1 Pointer and numeric optimization

Some comments should be made about the representations shown in Figure 10. In several instances, more space has been used than might seem necessary. In particular, four bytes are used for storing the length of a string in characters (Figures 10c and 10g), and four bytes are used for storing the type of node when there is no pointer (Figures 10a, 10b, and 10c). The reason for illustrating things in this manner is the intent of treating these values consistently as integers, since many computer systems are best designed for dealing with integers. Also, a high-level language implementation of MUMPS would work most efficiently with that representation.

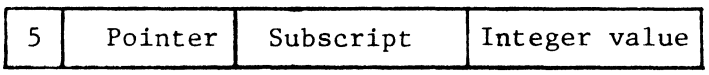
It is clear that assembly language implementations of globals on byte-oriented machines could use disc storage more effectively, by storing the string length in a single byte (for implementations where the maximum string length is 255 characters) and by distinguishing between integer values with and without pointers as shown in Figures 11a and 11b.



a) Integer value node

Number of bytes

9



12

b) Pointer with integer value node

Figure 11 - Byte-oriented representations of global nodes for integers

This approach saves three bytes for each node where there is an integer value with no down pointer. The same approach can be applied for real number and string valued nodes where there is no pointer. For a given global, then, the possible savings in bytes is three times the number of terminal nodes (leaves).

Three bytes have been assigned for the pointer in these representations. The pointer is used to point to a block elsewhere on the disc and thus can take on a range of integers equivalent to the number of blocks on the disc. For some smaller discs (such as cartridge discs), the number of blocks can be represented with as few as fourteen bits, or two bytes, permitting a further reduction in space utilization. However, since larger disc systems require more than sixteen bits for their addressing scheme, implementors are cautioned against restricting pointer values to two bytes, since eventual conversion problems will result with the changeover from a small to a large disc system.

Only four bytes have been allocated for the storage of real values. This is based on the assumption that real values will be handled through the use of floating point arithmetic, and not through decimal arithmetic or string arithmetic, which are possible alternatives. The 32 bits used for floating point values on most computers permit the representation of decimal numbers with an absolute value in the range 10^{-63} to 10^{63} with a precision of approximately 7 decimal places. For implementations in which the given range is inadequate or in which greater precision is required for computation, more bytes should be allocated for floating point values. The allocation of six or eight bytes for floating point values produces the effect of double precision arithmetic and can be used to allow a wider range of values or a greater degree of arithmetic precision, or both. Readers interested in the problems of floating point arithmetic are referred to [Sterbenz, 1974].

3.2.2.2 String storage considerations

The representation for strings contains the length of the string in characters as an explicit value. Previous implementations of MUMPS (on the PDP-11 for example) stored an offset which pointed to the first word of the next node. These methods are functionally equivalent; length as used here

more clearly illustrates the space requirement for the character string itself (see below). The number of words required is a function of the number of characters in the string (L) and the number of characters stored per word on a given computer system (N). The formula is given by:

words = (L+N-1)/N

where the division is an integer division. Thus, on a computer where four characters are stored in a word, a string of length 18 would require five words of storage; i.e., (18+4-1)/4 = 21/4 = 5.

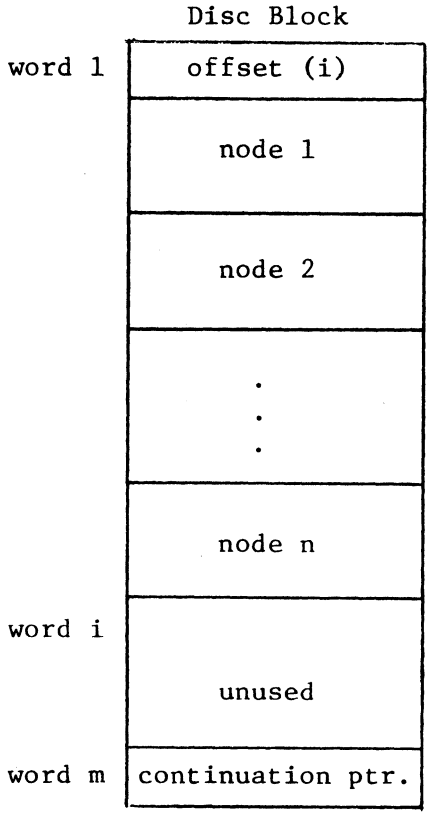
The MUMPS Level 1 Standard provides a maximum string length of 255 characters, which may be in excess of the physical disc block size for some computer systems. It is thus important to have a mechanism for storing a string when its length is greater than the size of a disc block. As in previous MUMPS implementations, a node is stored in a single block, unless the size of the node itself exceeds the block size. For example, if the remaining space in a block is 10 two byte words and an existing node within that block acquires a string value of 30 characters in length, the node will be removed from the old block, and a new block will be attached to the sibling chain. The node and its string value will then be stored in the new block, and the last word of the old block will hold a continuation pointer to the new one.

The simplest solution to the problem of storing a string larger than the physical block size is to continue to utilize the last word of the block for a continuation, as just described. Then, part of the string is placed in the first block, and the remainder in the next (and possibly succeeding) blocks. By using the length of the string along with a sequence of continuation pointers, it is possible to locate nodes following the "long string" node, or to retrieve the "long string" in pieces and assemble the entire string. This technique requires the string length to be explicitly stored with the node. It should be noted, however, that this approach adds complexity to the searching algorithm, and to the compaction method described below.

3.2.3 Data base structure for globals

It is now possible to examine the total representation of globals by combining the information about pointers with the information about structure of individual nodes. Each global uses one or more disc blocks of length m, which are laid out as in Figure 12. The first word of the block indicates how many words in the block are used, the last word of the block is a continuation pointer (or 0 if there is no continuation), and intermediate words hold information on the nodes.

Using this information, then, the part of the global G in Figure 5 from Section 1 can be depicted in Figure 13 as it would be implemented according to this scheme. (Figure 13 is slightly stylized for ease of comprehension and does not correspond precisely to the actual implementation).



Assume fixed-length disc blocks of m words

Figure 12 - Global disc format

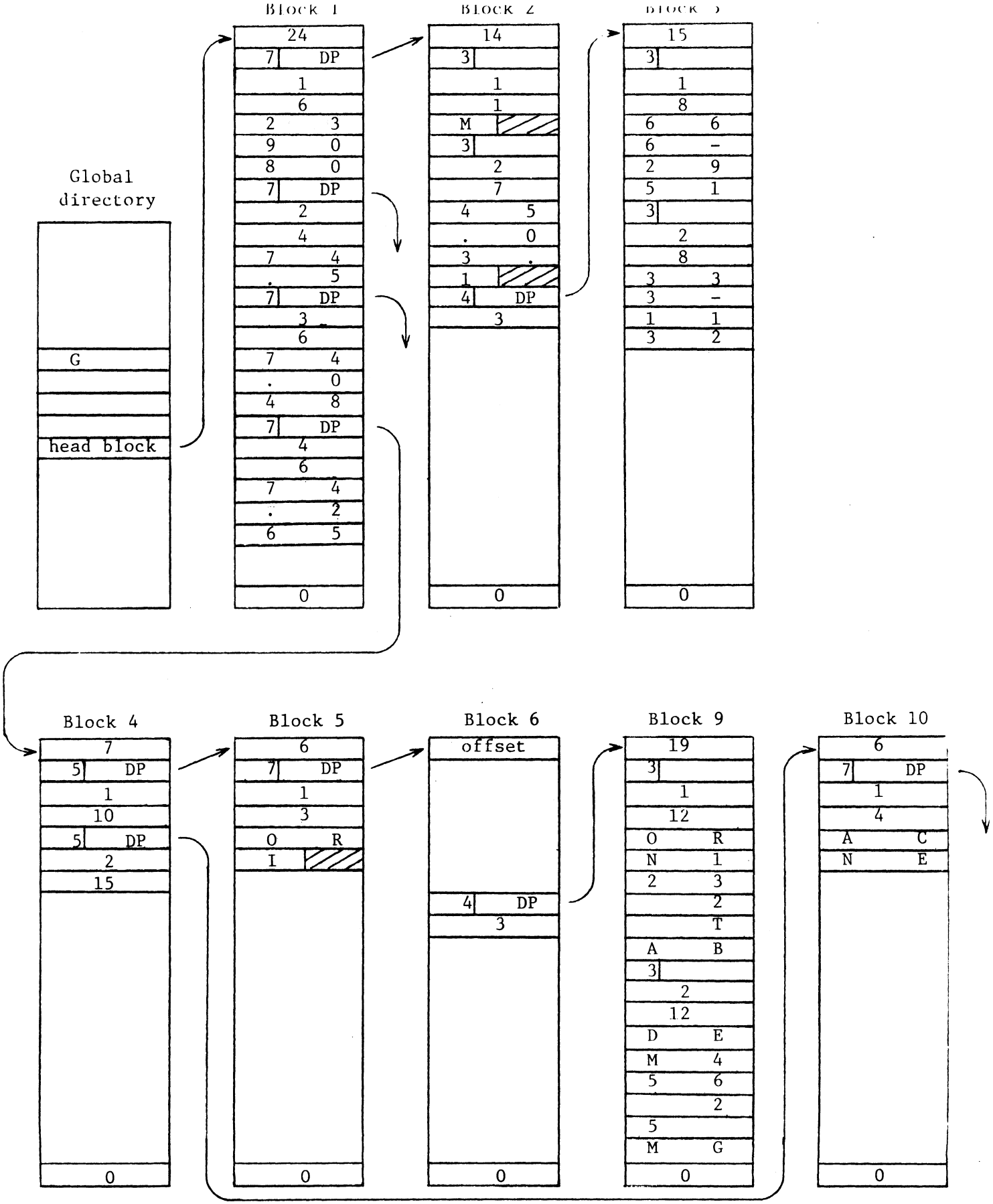


Figure 13 - Implementation of a portion of global in Figure 5

3.2.4 Global directories and global creation

Figure 13 introduces the notion of a global directory. The global directory is used as the initial entry point table for globals within the data base. It contains the root nodes for all global references. In handling global directories, existing implementations use one of the two general techniques discussed below, or variations of them.

The first method treats the root nodes of the globals differently from lower-level nodes. Typically, a fixed number of words are allocated for each root node entry in the global directory; this allows a faster search of the directory. In this scheme, the entire global data base is accessible to all users of the MUMPS system, so that there is only one global directory, which is normally held in primary memory for efficient searches. Each directory entry contains the symbolic name of the global, and a pointer to its first-level head block. The initial word of the directory may contain a count of the number of globals in the data base.

With this scheme, globals cannot be created dynamically with ease, so usually, the global creation process is an off-line task, using special utility functions. Also, the root nodes cannot be deleted dynamically. The advantage of this method is that it gives stability to the global data base, especially if the globals created off-line can pre-allocate disc blocks (see Section 4.3). This helps prevent globals from becoming overly fragmented on a large disc. The disadvantages are that the fixed data base is rather inflexible, and that the root nodes are not handled in a manner consistent with lower-level nodes. Also, the root node usually cannot have a value associated with it. Thus, although this method has been used, it is primarily of historical interest, as the MUMPS Standard allows the root node of globals to take on a value. Once the root node must accommodate a variable-length value, the fixed-length directory is inappropriate,

so the second method below is more reasonable. Implementors who wish to disallow values at the root node may still use this technique, however.

The second method treats the root nodes of the globals exactly like the lower levels. Thus, the global directory can be viewed as the set of first-level nodes for a higher-level structure. The root node of the hierarchy which points to the global directory is usually part of the user information table, so that each user has his own set of globals. When a user enters the system, the pointer to his global directory is loaded once into his information table. This pointer is usually to a head disc block for the directory (since more than one block may be necessary to hold the directory). The same disc block layout and searching algorithm used for lower-level global nodes is employed for the directory. Some systems even transform the name (root node) of a global to an integer value which "looks" like a subscript (hashing). This "subscript", and possibly a value, are stored in the directory block, along with a pointer to the first-level head block. Figure 14 illustrates this hierarchical directory technique.

Under this scheme, globals can be created and deleted dynamically, using the MUMPS language. An assignment to a global whose name (root node) is not in the global directory creates a new node in the directory as described above. This dynamic mechanism facilitates on-line global creation and deletion by user programs. Also, some measure of security is provided by giving each user his own set of globals.

Usually, there are also "library globals" on such systems, which are primarily read-only globals which any user may reference, but not modify. In this case, each user information table also has a pointer to the system library global directory. Some convention is usually established to distinguish library global names from user globals (for example, all library global names may begin with "%");

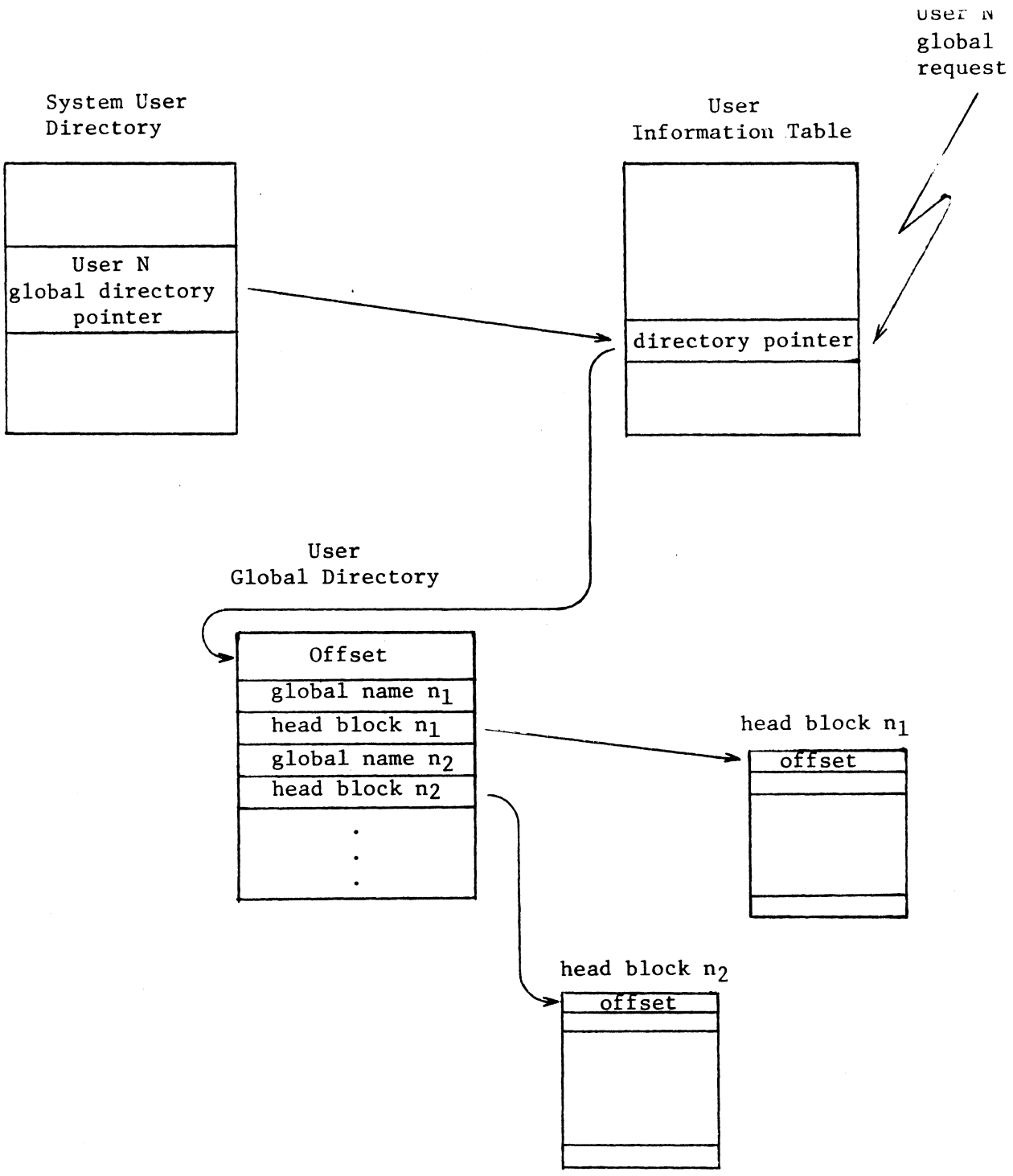


Figure 14 - MUMPS global directory structure

this prevents the system from always searching both directories, eliminating considerable overhead. Library globals are normally created and modified by special "privileged" functions, and are used as common data bases among a similar class of users.

The flexibility of this scheme introduces some added complexity to the disc management. More attention must be given to disc allocation techniques, as a multi-user system dynamically creating and deleting globals could badly degrade efficient accessing. Also, one user should be prevented from expanding his global data base until he monopolizes disc usage at other users' expense. In general, though, this method has been employed by more recent MUMPS implementations, as its flexibility and consistency of node treatment are far more advantageous.

3.3 Search structures

3.3.1 Node references and modifications

Within a MUMPS program, reference can be made to a global value or an assignment can be made to a global node. In both cases, it is necessary for the system to search for the existence of a node with a given set of subscripts. The general rule for searching is as follows:

1) Go to the disc address pointed to by the global directory for that global.
2) Compare the subscript reference in the program with the subscript value in the global node on the disc. If there is a match, proceed to step 3; otherwise, go to the next node in that disc block or to the first node in the continuation block if that block is exhausted. If there are no more nodes, the search fails.
3) If all of the subscripts in the program reference have been matched, the search is successful. Otherwise, follow the down pointer from the last successfully matched node on the disc to compare with the next subscript in the global reference. Return to step 2.

As an example of this, consider a program reference to ^G(4,1,1,3,2) in Figure 13. The global directory points to block 1. Block 1 is searched with unsuccessful comparisons for 1,2, and 3, until 4 is matched. The successful match on the first subscript means that search should proceed, looking for a match on the second subscript. The down pointer from the match points to block 4, where a search for a 1 occurs. That search is successful and the search for the third subscript (1) occurs in block 5. That is also successful, so the search for the fourth subscript occurs in block 6, according to the down pointer from the successful match in block 5. The fourth search succeeds with the match of the subscript 3, yielding a down pointer to block 9, which is searched for the fifth subscript (2). When that search succeeds, there are no more subscripts in the program reference, so the global search has been successful. If any of the search attempts had failed, then the global search would have been unsuccessful.

Use of the naked reference can reduce the search time for a node. Whenever a global reference is made, a pointer to the head block of the lowest level referenced is saved in the user information table. Then, when a naked reference appears, the search begins immediately with this head block. Search time may be further reduced in partially ordered sibling levels by using the cyclic search technique described in subsection 3.3.2 for the $NEXT function.

Because of the time required for disc accessing, it is always the case that the disc block being searched is brought into primary memory in a buffer area in order to expedite searching. In some implementations, each user has a designated disc buffer area (one or two disc blocks in size). In other implementations, the entire system has a pool of buffers which can be allocated to individual user partitions as needed.

The structure of individual nodes and their disc blocks can be changed by the execution of the SET and KILL commands. In the SET command, it is possible to alter the value of existing nodes or to create new nodes and blocks. In

the KILL command, nodes and their descendants may be deleted, resulting in the return of certain blocks to a central pool for the entire data base (see below).

If the size requirements of a node contract, through a SET command, the new node may be placed in its old location, with subsequent nodes in the block relocated to prevent fragmentation (the existence of unused space between nodes). Figure 15 shows what happens when node i_j's space requirements are reduced by K words.

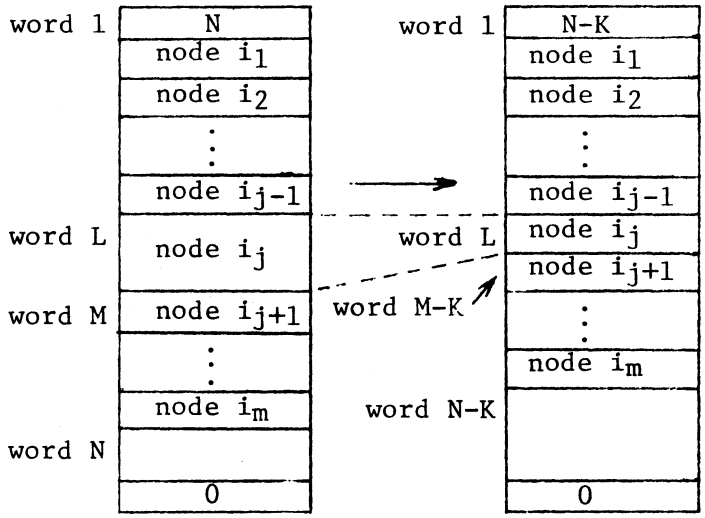


Figure 15 - Recompaction of global disc block

If the size requirements of a node expand, the opposite effect takes place. Nodes are pushed back within the block to accommodate the extra space requirement. However, sometimes this step is not possible, since the expansion may overflow the size of the block. In this case, several options are possible:

1) eliminate the enlarged node from the block and recompact the block without that node, placing the changed node in a new block with suitable continuation pointers;
2) leave the enlarged node in its place in the previous ordering, moving the nodes at the end of the block to a new block as necessary; and
3) perform a more global space optimization for the nodes in the block and its continuation blocks.

The third method is rarely used, however, since the processing time and the I/O time required are rather significant. When the space utilization of a global becomes particularly bad, however, it is possible to perform this type of compaction.

When a new block is required, either because of expansion of node size, creation of a new node with a value, or creation of a new node at a lower level which requires a pointer, blocks may be obtained from a central pool of blocks. The new block can be connected to the existing global structure by means of continuation pointers for new descendant nodes. Thus, if a node holding a string value obtains a descendant through a SET assignment, then a new block must be obtained to hold the descendant node and the node holding the string must be changed from type 3 (string) to type 7 (string with pointer), with the address of the down pointer being stored in the node.

Since MUMPS globals are treated as sparse arrays, all searching is explicit. Thus, there is no requirement for ordering the nodes in a block according to ascending or descending subscript values. As a result, the system may perform arbitrary reordering of nodes and their values in order to use storage most effectively. (The reader should not be misled by Figure 13, which may give the impression that some type of ordering exists.)

3.3.2 Tracing and existence functions

Because of the dynamic nature of MUMPS globals, it is sometimes impossible to know beforehand the node structure at a particular level of a global array, or whether a particular node is a terminal node or has a value. To aid in determining this information, the MUMPS language provides two functions: the $NEXT function and the $DATA function.

The $NEXT function provides a facility for tracing all siblings at a given level below the root node in ascending numeric subscript order. In Table I of Section 1, several examples are shown using this function on the global in Figure 5. The $NEXT function returns the value of the next numerically higher

subscript at the lowest level referenced in its global argument. A -1 is allowed as the lowest level subscript in $NEXT, so the value of the smallest numerical subscript on that level can be determined. If $NEXT returns a -1, no higher subscript exists at the level referenced.

In implementing the $NEXT function, the entire lowest level referenced is normally searched for the next higher subscript, beginning with the head block for the level. Some optimization can be achieved by beginning the search in the present block of the sibling chain, wrapping back to the head block when the end of the chain is reached, and then searching up to the present block. This requires that the system "remember" the starting block of the cyclic search. Also, note that this cyclic method is useful only when the $NEXT function argument is a naked reference, as otherwise the search will always begin in the head block of the level. Another technique would be to recognize that global subscripts are nonnegative integers; the search can stop if the next consecutive integer is encountered prior to the end of the chain. Thus, in Figure 5 of Section 1, $NEXT(^G(1)) would stop searching as soon as it found ^G(2), since there cannot be any intervening subscript values. This can save time with large sibling sets that are at least partially ordered. However, it does entail more overhead in the search algorithm, as this next integer test must be made each time a node is encountered.

The $DATA function provides a way of determining whether a particular global node exists, and if it does, whether it has descendants or data associated with it. Again, Table I of Section 1 illustrates the $DATA function. The $DATA function returns an integer number which can be viewed as a binary truth value. The units "bit" is on (one) if the node specified in the argument of $DATA has a value associated with it; if it does not the "bit" is off (zero). The tens "bit" is on if the specified node has descendants (i.e., contains a

pointer to lower levels); otherwise, the "bit" is off. Thus, the values returned by $DATA are interpreted as shown in Table I.

In implementing $DATA, a search of the lowest level is made until the specified node is encountered. On an unsuccessful search ($DATA returns 0), the entire level must be searched. The cyclic searching strategy discussed for $NEXT can also be introduced here to improve efficiency.

3.4 Allocation and de-allocation of globals

In subsection 3.3, reference was made to a "central pool" of disc blocks from which available blocks could be obtained and to which unneeded blocks could be returned. In order to manage the total disc space available for the storage of globals, programs, and directories within a MUMPS system, there must be a strategy for assigning a disc block to a particular global and making that block unavailable to other requests, as well as a strategy for making the block available again when it is no longer needed. These strategies are called allocation and de-allocation.

First, however, a description of the physical layout of a disc storage device will help clarify the terminology.

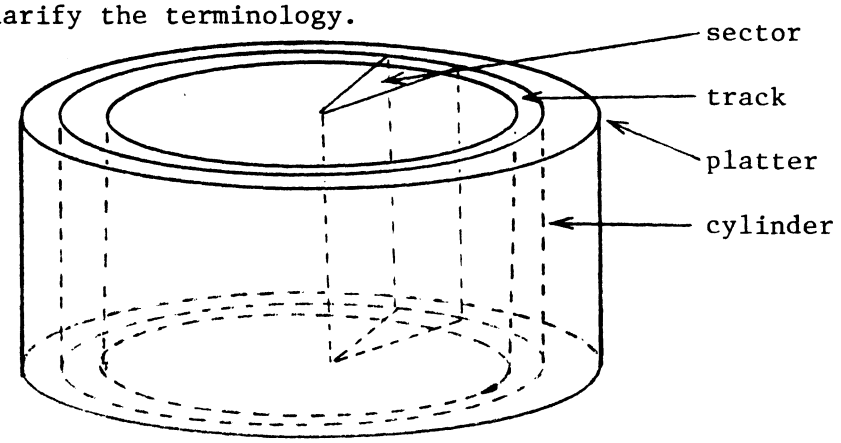


Figure 16 - Physical layout of a disc

Figure 16 shows two platters (each platter contains two read/write surfaces) of a removable pack-type disc. Each platter is divided into many tracks, which are concentric rings on the disc surface. Tracks which share the same concentric

ring on different platter surfaces make up a cylinder. A track is subdivided into a number of disc blocks, which is the unit of storage used in global management. Blocks which are aligned vertically in a cylinder make up a sector, which can be viewed as a slice of a multi-layer cake. Typically, discs have a movable unit with a read/write head for each platter surface, so that any block in a given sector can be accessed effectively simultaneously, although only one block can be read or written out at a time. Also, as long as all needed disc blocks fall within one cylinder, they can all be referenced or modified without moving the disc head mechanism. This improves access time substantially.

There are two techniques that have been widely used for the allocation of disc blocks, not only in MUMPS, but also in a variety of other operating systems. The simplest of these is called the bit map technique; the other method is a linked block technique.

In the bit map technique, each block on the disc is represented by a bit of information which is set to 1 if the block is in use and 0 if it is available for allocation. One bit is required for every block on the disc, so that for a typical large disc system, having 20 surfaces, each with 400 tracks and 32 blocks, a total of 256,000 bits of information is required. This information can be stored in 16,000 16 bit words. In computer systems which have extensive quantities of primary memory, it is not uncommon to store part or all of this information in the primary memory. However, it is more common to store the information on the disc itself, either in a single "file" or separated by track. As shall be shown later, it is often advantageous to allocate a block on a particular track. If the bit map for disc allocation of a track is kept on the track, even greater efficiency can be achieved.

In the linked list technique, the principle of chaining with continuation pointers is used. All of the unused blocks are linked together by continuation pointers; when a block is requested, the unused block can simply be removed from the end of the linked list. When the block is no longer needed, it can be linked back into the linked list. The techniques which are used are similar to those used for list processing [Knuth, 1974]. It is possible to maintain only one linked list for the entire disc, but it is more common to keep a linked list for each track of disc so that available blocks can be located on a given track.

For either of these techniques, there must exist operating system processes to accomplish the allocation and de-allocation. These processes must be treated as "critical regions" [Dijkstra, 1968], since various MUMPS programs trying to allocate and de-allocate disc blocks simultaneously could interfere with each other. If there are not proper provisions made in the operating system, two users could end up having the same block allocated to them. One user must be permitted to complete the allocation process before another is permitted to begin. This subject is discussed in somewhat greater detail in the companion paper on the structure of a MUMPS operating system. Readers wishing more familiarity with the issues involved are referred to [Shaw, 1974] or other recent operating systems books.

3.5 Programming considerations

Although it is not the purpose of this report to recommend effective programming techniques in MUMPS, it should be apparent that the subscript structure chosen for globals can have a significant effect upon the performance of user programs, when working with global implementations of this nature. Both the number of continuation blocks at a given level and the number of down pointers which must be traversed to reach a given node are important considerations in the design

of global structures and their application programs. The objective of the overall design effort should be to minimize the number of disc accesses that must be made.

The structure of the global itself is often of overriding importance in the design of MUMPS programs. A physical structure must occasionally be selected strictly on the basis of performance and efficiency with little or no consideration given to the "natural" representation of the global information. (It should be noted that similar problems exist in other programming systems and that development of effective data base specification languages with efficient storage and retrieval mechanisms is a major research problem [Date, 1975].)

The following example should indicate the nature of this problem. Suppose that one wishes to search a large, one-dimensional array (let us assume 4096 nodes) to locate a given value. Suppose further that the values can be arranged into ascending order. In some programming languages, then, it would be possible to declare a one-dimensional array, sort the elements, and apply binary searching methods [Knuth, 1973] to locate the desired value or to determine its absence, in 12 or fewer tries.

In MUMPS, however, the problem is quite difficult, first because all subscript searching is explicit rather than indexed, and second because a one-dimensional structure is especially inefficient for explicit searching. With existing MUMPS implementations, a one-level global of n nodes will require an average of n/2 comparisons for a successful search and n comparisons for an unsuccessful search. The volume of searching can be reduced only by partitioning the n nodes into m levels, in order to achieve the same effect as a binary search. In this case, though, MUMPS requires the explicit creation of a physical search structure rather than an arbitrary searching algorithm for a linear structure.

If the 4096 nodes are in a one-level array, then a successful search requires, on the average, 2048 comparisons. If the 4096 nodes are rearranged into hierarchical fashion (even though the nodes have a logically linear relationship with each other), considerable improvement can be achieved. Thus, by having 64 first level nodes, each with 64 children, the average search can be reduced to 64 (32 at each level) plus one down pointer traversed. At four levels, with 8 first level nodes, and 8 at each succeeding level, the average search can be reduced to 16 + three down pointers traversed. (Total search time is increased by the time required to compute the subscript values to four levels, but as disc searching time is the critical resource, then the computational time increase is acceptable.) In general at each of the m levels $\sqrt[m]{n}/2$ nodes per access will be searched; in order to find a given terminal node, a total search of m ($\sqrt[m]{n}/2$) nodes will be required. It should be noted that there is an extra disc access for each additional level, so that, for this case, 3 or 4 levels represents an optimal balance between address computation and disc access.

The intent of this rather lengthy example is to show that it is to the programmer's advantage to optimally structure a file, even when its nodes have no hierarchic relation. Considerable testing is required in order to identify the proper tradeoff between address computation and traversal of disc blocks. The difficulty of handling this apparently simple case in MUMPS provides motivation for the discussion of alternate global types (see subsection 5.2).

4. Optimization Considerations

4.1 Overview

Because MUMPS is used primarily for data base management applications, the performance of the various data base storage and retrieval functions is a major determinant in the overall performance of MUMPS systems. The efficiency with which these operations are carried out affects the total workload of the system, the amount of time spent on user programs as opposed to operating system overhead, and the response time which can be provided to the user at the terminal.

Because the MUMPS language and the MUMPS operating system were designed for rapid processing, it is usually the case that MUMPS programs and systems are bound by the speed at which the input/output operations of the data management function can be performed. Accordingly, improvements in system performance can be best achieved through improvements in the performance of the MUMPS global system.

The issues which are involved in trying to optimize the MUMPS global system include scheduling of disc requests, choosing between pre-allocation or dynamic allocation of disc storage to a global, and minimizing the effects of arm movement (rotational latency) upon storage and retrieval. This section will treat each of these considerations.

4.2 Scheduling of disc requests

Scheduling processes are concerned with determining the order in which a sequence of requests for use of a given resource will be serviced. In a multi-user environment such as MUMPS, it is common for the execution of several user programs to be blocked while awaiting the completion of a disc operation, possibly a global update or reference. The disc scheduler must determine which programs are to be served first and which must wait.

Most schedulers operate upon two primary considerations: the priority of the program making the request and the optimizing of placement of data on the disc.

In MUMPS, each program can execute at a different priority level. Most scheduling schemes which have been implemented for MUMPS penalize the user whose program makes heavy use of processor time in carrying out computations while giving a higher priority to those jobs which are more interactive.

This same priority scheme could be used in the disc scheduling algorithm, although it was designed for determining which programs are permitted to have the central processor. While it would be possible to apply the same technique to use of the disc subsystem, there are other considerations which can be taken into account in order to achieve improved service.

An important determinant of overall performance in MUMPS systems is disc access time, as opposed to disc transfer rates or central processor time. Disc access time has two components: arm movement and rotational latency. Arm movement (or seek time) refers to the interval of time required to move the disc read/write head to a given track position. Depending upon the distance which must be covered by the read/write head, the elapsed time for this operation may range up to 100 milliseconds (ms.). Rotational latency refers to the interval of time required for a particular disc block on a track to come underneath the read/write head. For a typical disc rotating at 1500 r.p.m., the average rotational latency is 20 ms. Thus, the time taken to retrieve a given disc block may range from close to zero to 120 ms., due to the effects of arm movement and rotational latency.

From the standpoint of the operating system, attempts must be made to minimize delays caused by arm movement and rotational latency. Optimization can be carried out at two points within the operating system:

1) in placement of the blocks of a global on the disc;

2) in selecting the order in which disc requests will be served.

This latter problem can be handled by the disc scheduler. (The former problem is handled partly by the allocation policies discussed below.)

Every global request maps onto a request to access a particular block of disc storage. In a multiprogramming environment, various user jobs will issue global requests concurrently. In addition, the nature of the hierarchical data base is such that any full global reference (not a naked reference), will require multiple disc accesses. The actual number of accesses required is equal to the level of depth of a node within the global (i.e., one access per subscript in the global or naked reference), plus an additional access for every continuation block searched in looking for a given subscript at each level.

For example, a MUMPS statement such as:

```
SET X=^G(3,1,4)
```

requires at least four global accesses, one for the global directory to locate the header block for G, and then one for each of the three subscripts. The existence of continuation blocks at each level may add to the total number of accesses.

As a result, a global reference generates not one, but a series of disc requests. Furthermore, the addresses of successive accesses are only obtained sequentially, each from the previous node. It is the overall time that is required to perform these multiple, chained accesses that determines the response time to each single global reference.

From the standpoint of scheduling, then, there is often a queue of disc requests waiting to be served. There are several types of scheduling algorithms which can be utilized in servicing this queue. The first method is first-come, first-served (FCFS), in which the disc requests are simply taken in the order in which they are generated by the user programs. Since this method may involve different users' globals, widely scattered on the disc, there is a tendency for this method to use considerable overhead in arm movement. Its primary advantage is the simplicity of the scheduling algorithms.

Another scheduling algorithm used in time-sharing systems is called SCAN, in which the disc arm moves from the outermost cylinder of the disc to the innermost one and then back to the outer, reading or writing records in both directions in the order in which they are encountered. A similar algorithm is known as C-SCAN [Teorey, 1972], in which the arm moves from the outer to the inner cylinder servicing requests, and is then immediately moved back to the outer cylinder without servicing requests.

With both SCAN and C-SCAN, the queue of disc requests must be maintained in sorted form according to cylinder number, with every disc request being inserted at the appropriate point in the ordering. These methods are normally implemented via a linked list arrangement, with new requests being linked into the appropriate place in this list and completed requests being deleted. FCFS is implemented more simply with a one-dimensional array used as a queue, along with pointers to the beginning and end of the queue [Knuth, 1974].

SCAN and C-SCAN have been shown to be more efficient in terms of disc retrievals for those systems where I/O requests can be satisfied by a single disc access. It is not clear that these results are applicable to MUMPS, since each I/O request (a global reference) can initiate several chained disc accesses. Conventional methods of simulating, measuring, and comparing different scheduling policies do not seem to apply as well to MUMPS.

It appears that placement of globals on the disc, rather than scheduling of accesses, is a more important factor in improving the performance of disc system storage and retrieval. The next subsections will treat these allocation issues in more detail.

4.3 Allocation strategies

4.3.1 Minimization of seek time and rotational latency

In the previous subsection, it was observed that both seek time and rotational latency are important factors in determining the number of disc requests that can be served in a given period of time. By reducing delays caused by these factors, the throughput of the system can be increased, making it possible to provide users with a better response time. Scheduling disc requests to minimize physical movement of the arm can help in this regard, but a larger contribution can be made by placing the blocks of a particular global on the disc in such a way as to eliminate arm movement and minimize rotational latency in making successive accesses to that global.

On a large disc with a capacity of 50 million characters or more, the number of characters which can be stored on a single cylinder is in excess of 100,000. Thus, it is possible to allocate storage for a global in such a way as to eliminate the need for significant head movement, unless the global grows quite large. For large globals, it is possible to store the information on contiguous cylinders so as to minimize the amount of movement needed. With the bit map or linked list approaches to disc allocation discussed above, it is possible to select blocks from a particular cylinder or area to hold global nodes as the size of a global expands. This technique can be extremely effective and easily implemented until the amount of storage on the disc exceeds 80% of the available disc space, at which point it becomes increasingly difficult to perform optimal allocation. As a result, the allocated sectors for a global on

a nearly full disc system may be widely scattered and may result in considerably degraded performance. Subsection 4.3.3 discusses some reallocation techniques which can be used to help alleviate this problem.

This nearest-cylinder approach is an effective solution to block allocation for expanding globals. The primary remaining problems are now minimizing the interference among the various users of a global, and managing the rotational latency delays.

Both multi-user interference and rotational latency are extremely difficult problems for which only the barest outline of solutions can be proposed. In the multi-user case, there are a number of user requests pending at any given instant. Some of these requests are for globals and others may be for programs accessed through the MUMPS program directory. Programs are stored on the same disc as are globals on many MUMPS implementations. As a result of this situation, the interleaving of disc requests among multiple users can counteract the optimal allocation of blocks for a global.

Rather than serving all of one user's global reference as a continuous sequence of disc requests, these references are interspersed with others in many instances. Thus, the information content of a block must be obtained before determining whether a continuation block must be searched. In the case of a down pointer, however, it is necessary for the MUMPS interpreter to carry out some computation to obtain the value of the next subscript in the global reference from the execution stack. It is normally the case that other disc requests are served during these times, often causing the read/write head to move from the cylinder which holds the global being referenced. This movement, designed to maximize the number of disc requests served, may actually be counterproductive if there is considerable arm movement.

A better method for minimizing multi-user interference may be to give an individual user a high priority throughout a complete global reference unless that user tries to access a global node which has been locked out by another user. This approach assumes that blocks have been allocated within a global so as to minimize arm movement and thus applies an overall optimization technique rather than a local optimization technique. Additional experiments are necessary to determine the utility of this approach. (Virtually, all techniques work for lightly loaded systems; they must be applied to those systems where the load is heavy.)

The problem of rotational latency deals with the placement of successive blocks in a global. If there are 32 blocks on each track of each cylinder on the disc, then it takes approximately 1 ms to go from one block to the next on the average moving-head disc system. If one block generates a continuation pointer, the optimal placement of the continuation block would be such that the read/write head is almost ready to pass over the desired block. If continuation blocks are not placed well, there may be a delay of up to 40 ms. before that block can be read or written simply as an effect of the rotational speed of the disc. The same argument can be applied to down pointers; they should ideally produce a disc address which will be reached within a few milliseconds.

In MUMPS, however, this is an especially difficult problem. When one block is accessed and read into a buffer area, it is difficult to determine the processing time that will be required to locate the continuation pointer or the down pointer. First, the number of nodes that must be searched is variable; it can range from one in the case of a single subscript producing a down pointer to forty or more in the case of a 256-word disc block having integer-valued global nodes. Second, the time-shared nature of the MUMPS system makes it uncertain as to when that search will be carried out relative to the time at which the disc request was completed. With the elimination of the second

possibility through the technique of serving a user's entire global request, the time variation of searching is large, but may still permit some optimization of rotational latency. If we assume that searching for a particular node and making a comparison can be done in less than 20 microseconds, then an entire block can be searched in less than 1 ms. Accordingly, if continuation blocks and down pointer blocks are placed two away from the block which points to them, it becomes possible to process an entire block and request the next block (whether across or down) before that block passes under the read/write head. If the transfer rate for the disc is particularly slow or the processing time is higher than the number suggested here, the same principles can be applied, but the separation between a block and its successors might have to be larger.

It is obviously impossible to place all of the continuation blocks and down pointer blocks in an ideal position from the standpoint of rotational latency. However, by using all of the blocks which are separated from a given block by a given number of blocks, a rather high efficiency can be achieved. It should be remembered that the disc block may be on any of the platters of the disc, as long as it is in the same cylinder. Thus, for a typical disc, twenty blocks may be placed at a given distance from a block.

The savings which can be achieved through use of this technique should not be underestimated. When compared to the rotational latency times of a random distribution of blocks on a cylinder, a fivefold improvement can be achieved by placing a successor block within four places of the optimal placement.

MUMPS systems which are "tuned" for efficient disc management apply this optimal allocation of blocks to minimize seek time and to minimize rotational latency. Furthermore, the simplicity of the solution to minimization of seek time makes rotational latency the prime determinant of service times for disc

block requests in most MUMPS systems. Readers who wish to develop a mathematical model for dealing with rotational latency and scheduling disc requests to accommodate rotational latency are directed to [Coffman and Denning, 1974]; it should be noted, however, that disc requests in MUMPS do not fit the Poisson distribution model of interarrival times which Coffman and Denning assume.

4.3.2 Pre-allocation vs. dynamic allocation

It was observed in the previous subsection that allocation, rather than scheduling, is the most important factor in achieving a layout of global data on the disc which leads to efficient handling of disc requests. An efficient allocation method, combined with a simple scheduling policy will result in better performance than an inefficient allocation scheme with a sophisticated scheduling policy, since the loss of efficiency caused by poor allocation will be directly proportional to the average number of disc accesses that are required for the average global reference.

One of the problems which can occur in performing space allocation is that the disc can become full, or nearly full. In this case, it becomes very difficult to achieve good placement of blocks relative to their predecessors. When the amount of global storage approaches the maximum capacity, it is likely that a serious degradation of performance will result.

One technique that has been suggested and used to handle this problem is called pre-allocation. Rather than assigning new blocks dynamically as they are required by individual globals and returning unused blocks to a central pool, it is possible to assign a number of blocks on one cylinder or on contiguous cylinders prior to the global actually being used. All blocks allocated for a global are kept in a pool for that global only. If a global expands beyond its preassigned limits, an error condition may be raised by the program, requiring rearrangement of the disc or a change in allocation boundaries.

Both pre-allocation and dynamic allocation affect the distribution of the disc blocks comprising a global. With pre-allocation, the scattering of blocks across many cylinders can be significantly reduced, thereby reducing the time required to access many nodes from one global. With dynamic allocation, the expansion of global sizes in a growing data base can have a deleterious effect upon system performance and upon response time.

The use of pre-allocation creates several new problems, however. First, it is somewhat dependent upon the ability of the user to accurately estimate the storage requirements for the global. Since the user need not and probably will not understand the details of the global implementation, this estimate can only be made in gross terms, such as the largest number of nodes the global will possess, with the system estimating actual disc space based upon average node size (a system parameter which can be measured and modified over time). Because of the wide distribution of node sizes which is possible in MUMPS (from a pointer to a long string with a pointer), the value for the average node size will have a large variance, which will tend to make estimates inaccurate. In one case, too much space will be allocated for a global's needs and be effectively lost to other globals; in the opposite case, too little space will be allocated and it will be necessary to perform secondary allocations.

The second problem with pre-allocation is that it is dependent upon the growth and decay of the individual globals. If most globals are relatively stable, with few deletions and with few value changes which result in a major change in the size of a node, then space within each block can be used quite effectively and the allocation parameters can provide for this stability or for the more normal case of slow growth. If globals are highly dynamic in space utilization, however, then sufficient space to store the maximum requirements for each global must be provided. If the size of two globals is complementary, for example one is small and the other is large, or vice-versa, the pre-

allocation scheme must allocate enough space for the maximum size of both, even though this situation never occurs. This example illustrates that pre-allocation can result in considerable amounts of lost space.

It appears that an intermediate scheme works best. Different globals can be "assigned" to a given cylinder or set of cylinders by simply locating their header on that cylinder. The normal dynamic allocation algorithm will then assign new blocks on the same or nearby cylinders. At the same time, however, no block is permanently bound to a particular global. Pre-allocation of headers achieves a separation of globals and allows more efficient dynamic allocation.

When new globals are created on an existing system, the bit map or linked list can be examined to locate the cylinder which has the largest number of free blocks. The new global can then be placed on that cylinder with the knowledge that the largest possible number of additional blocks for that global can be placed on the same cylinder.

4.3.3 Reallocation techniques

As has been noted, many of these allocation strategies are ineffective when there is little excess storage available. Available storage becomes badly fragmented, with the few remaining disc blocks scattered across a number of cylinders. It becomes increasingly difficult to allocate disc blocks for a global in locations which are optimal for future access. System parameters can be built into the MUMPS global system to provide this information. By keeping track of the number of free blocks, the frequency of head movement, and similar measures, the onset of this problem, with its effects upon users, can be rather easily detected. When this condition arises, the best solution is the complete reorganization of the entire disc or at least several of the most frequently used globals.

If the computer system has more than one disc drive, then the fragmented disc system should be loaded on one drive and a scratch disc should be loaded on a second drive. Then, globals should be copied, one at a time, in such a way that the number of cylinders which they occupy is minimized. The global directory can be rebuilt to show the new locations of headers. The free blocks on the disc should be distributed so that there are at least some adjacent to each global. This allocation of free blocks can help to lessen the effects of further disc block allocations.

In the absence of another disc drive, it is possible to perform the reallocation with the use of magnetic tape. Globals can be copied to an on-line tape, one global at a time, mounting enough tapes so that the entire status of the disc can be stored off-line. A scratch disc can then be mounted and the tapes can be reloaded using the contiguous allocation technique described in the disc-to-disc reallocation method.

Reallocation in this manner requires the absence of other disc accesses, since the state of the disc must be preserved until a copy is made. Thus, it must be performed when all other users are blocked from using the system. A disc-to-disc copy can be performed in approximately fifteen minutes with currently used disc systems; widespread use of 100 million character, high speed discs will reduce this time to approximately five minutes. Thus, for all but the most critical on-line activities, reallocation can be performed as part of a normally scheduled procedure, similar to the programs used to save the status of the disc as a protection against disc system failures.

Reallocation could be invoked in conjunction with pre-allocation methods when the system detects an overflow of the original allocation. It could be run for the entire disc or simply for a certain specified subset of globals. The overflowing global would then be locked out and copied to a larger area, with all of the continuation and down pointers suitably modified.

Of course, if the total allocation space desired or required by the existing globals is greater than that available, no amount of reallocation or optimization of allocation will help; more secondary storage (disc) space would have to be made available to the system. Most data base management systems seem to have a constantly growing data base and most MUMPS systems have behaved similarly; it is necessary to recognize the performance characteristics associated with nearly full systems and act accordingly. Otherwise, the reallocation program can perform reallocation repeatedly with diminishing success, much as the garbage collection routines which attempt to locate free space in a list processing system when almost all of the space is in use. The amount of productive work accomplished quickly approaches zero in both cases.

Thus, allocation to minimize seek time and rotational latency time are essential for good MUMPS global handling. Furthermore, it is necessary to try to reallocate disc blocks to improve seek time and latency characteristics when the performance begins to degrade. Finally, it is important to be able to recognize a heavily loaded disc so that unneeded information can be expunged or so that additional disc storage may be acquired.

It is also possible to reallocate space within a sequence of continuation blocks. First, in a dynamic allocation scheme, the continual appearance and disappearance of nodes, coupled with rapid growth and shrinkage of existing nodes, can result in considerable fragmentation of space. There may be a considerable amount of unused space within each disc block. It is possible to form a new chain of sibling node blocks. If there is a large amount of excess space in the disc blocks at the time of reallocation, then the sibling nodes can be compacted into a smaller number of blocks and the excess blocks can be returned to a pool of free blocks available for allocation.

Another more elaborate reallocation could be made by keeping track of the frequency of disc accesses for individual nodes, counting both direct accesses and access to descendants of a node. This information could be used to rearrange the ordering of nodes within blocks so that nodes which are accessed most frequently are those which are searched first during the execution of a global reference. In particular, nodes which are accessed often should be located in the first block, rather than in any continuation blocks, for a set of siblings. Optimal arrangement of nodes could then result in fewer blocks being accessed. The price for achieving this form of optimization is that each node must keep an access count to be used in performing the reorganization. Although there appear to be some rather attractive optimizing possibilities associated with the use of this method, there is a significant overhead in keeping the counts and in performing the rearrangement. For this reason, existing MUMPS systems have not used this approach on-line.

4.4 Direct mapping of traces to disc addresses

Rather than dealing with the placement of nodes within blocks and the placement of blocks on the disc, it is possible to consider a radically different approach to global accesses, one which reduces the total number of disc accesses. The traditional handling of globals, as discussed above, accesses a node by translating the sequence of subscripts in the global reference into a path by following the pointers associated with each node through its descendants until reaching the desired node. The access time will be further delayed by the time taken to search and match each subscript in order to locate the down pointer to the children of a node on the path from the header to the desired node. (The naked reference circumvents this delay by storing an intermediate pointer so as to bypass some number of levels of accessing.)

If it were possible to map a sequence of 1 or more subscripts into a unique disc address, it would be possible to compute this address directly from the global references and retrieve a global value with a single disc access, producing a considerable savings in I/O time and therefore user response time.

Bernstein and Tsichritzis [Bernstein, 1974] have investigated this technique, which they have termed multiple domain address calculation. They have developed an addressing scheme for a subset of the general problem. Their technique works when the nodes of a tree-structured file have been decomposed into nodes of specific types, with the traces of each node type containing a certain known number of indices. They require the specification of a definition tree to hold this information, which is then consulted by the sytem prior to each address calculation. These constraints make their approach unworkable in MUMPS, due primarily to the lack of a priori knowledge about the structure of globals.

However, a similar scheme is being studied by the authors which would compute unique integers and thus disc addresses from subscript sequences of varying lengths, without the requirement of any a priori knowledge of the tree structure. This scheme would be applicable to MUMPS. The fundamental notion involved in this approach is the computation of a function which assigns a unique positive integer to any trace of length n. (For the mathematically-minded, this idea is similar to the Gödel numbering concept). If small bounds can be placed on the value of a subscript and if the number of subscripts is restricted to three or four, this technique can be applied. However, in the general case, the computational requirements seem to outweigh the potential benefits of the use of the addressing scheme.

There are two other problems which must be overcome if this method is to prove useful. First, the computational algorithm must operate upon the subscript sequence plus the name of the global to produce a unique mapping. Second,

with very large subscripts, the integer produced by the function becomes so huge as to exceed the arithmetic capabilities of the computer system. A secondary solution would generate an integer within some large range (possibly by modular division), but there would then be a searching requirement to locate the proper subscript. However, the ability to jump directly to the address on the disc of a specific global node remains so attractive within the confines of a hierarchical data base system that continuing attempts are being made to modify the address calculation procedures and utilize this mechanism whenever possible.

PART III - DISCUSSION AND EVALUATION

5. Analysis of MUMPS Globals

5.1 Introduction

The original designers of MUMPS had several goals in mind when the concept of globals was introduced:

1) to provide a file structure that would handle the "several levels of structure of a medical record data base, and to support the rather complex updating and retrieval needs of such a system" [Greenes et. al., 1969]
2) to facilitate sharing of the data base among the various users of the system, with appropriate safeguards to avoid time-sharing conflicts
3) to make the syntax of global usage closely resemble the syntax of local array usage in the programming language.

It can be safely said that MUMPS has successfully achieved all of these goals. As with any set of design decisions in a programming language, though, the achievement of these goals is reflected in a series of tradeoffs which may limit the range of applications for which the language is the best possible programming tool. This tradeoff between specialization and generality is a natural phenomenon, which is not restricted to computers and programming languages, but which can be found throughout the world. When something is created or designed to satisfy a specialized purpose, its general utility is reduced proportionately to the degree of specialization involved. The counterpart of this phenomenon, which the designers of MUMPS have tried to avoid, is that things which are designed to be extremely general are not usually the best for the performance of a specific task.

In some sense, then, criticism which is directed at the notion of globals as it exists in MUMPS is based upon the desire to have a greater degree of flexibility in their usage or to have means for implementing globals more

efficiently. In the succeeding subsections, certain extensions to the present concept of globals are proposed, but the scope of proposals is largely to those ideas which can be easily integrated within the present structure of the language and within the original designers' goals for the language. In subsection 5.4, globals are considered in a broader sense in order to contrast them with other techniques for data base management.

5.2 New global types

Tree structures have been used in MUMPS because the flexibility of trees allows users to structure their data in a manner that conforms closely to a logical hierarchy. With environments which are organized hierarchically, such as a traditional medical record, this structure can be extremely attractive. Indeed, the correspondence between the external appearance of the data and its implementation in a programming system permits the programmer to develop an information system using MUMPS with minimal difficulties in transforming the external representation to an internal representation.

When information must be stored which is not hierarchically organized, however, the system overhead which must be paid for the flexibility of trees is not always desirable. For the case of the one-dimensional array illustrated in subsection 3.5, a simpler form of structure would permit more efficient referencing and access. Furthermore, if the user conceives of the data set in a one-dimensional way, it would be desirable to provide a mechanism which supports that concept in an efficient way, without the overhead of the general purpose tree.

Two different kinds of globals can be introduced to support these one-dimensional structures. The first of these is termed a "sequential" global and supports the accessing of nodes in a pre-determined order. The second of

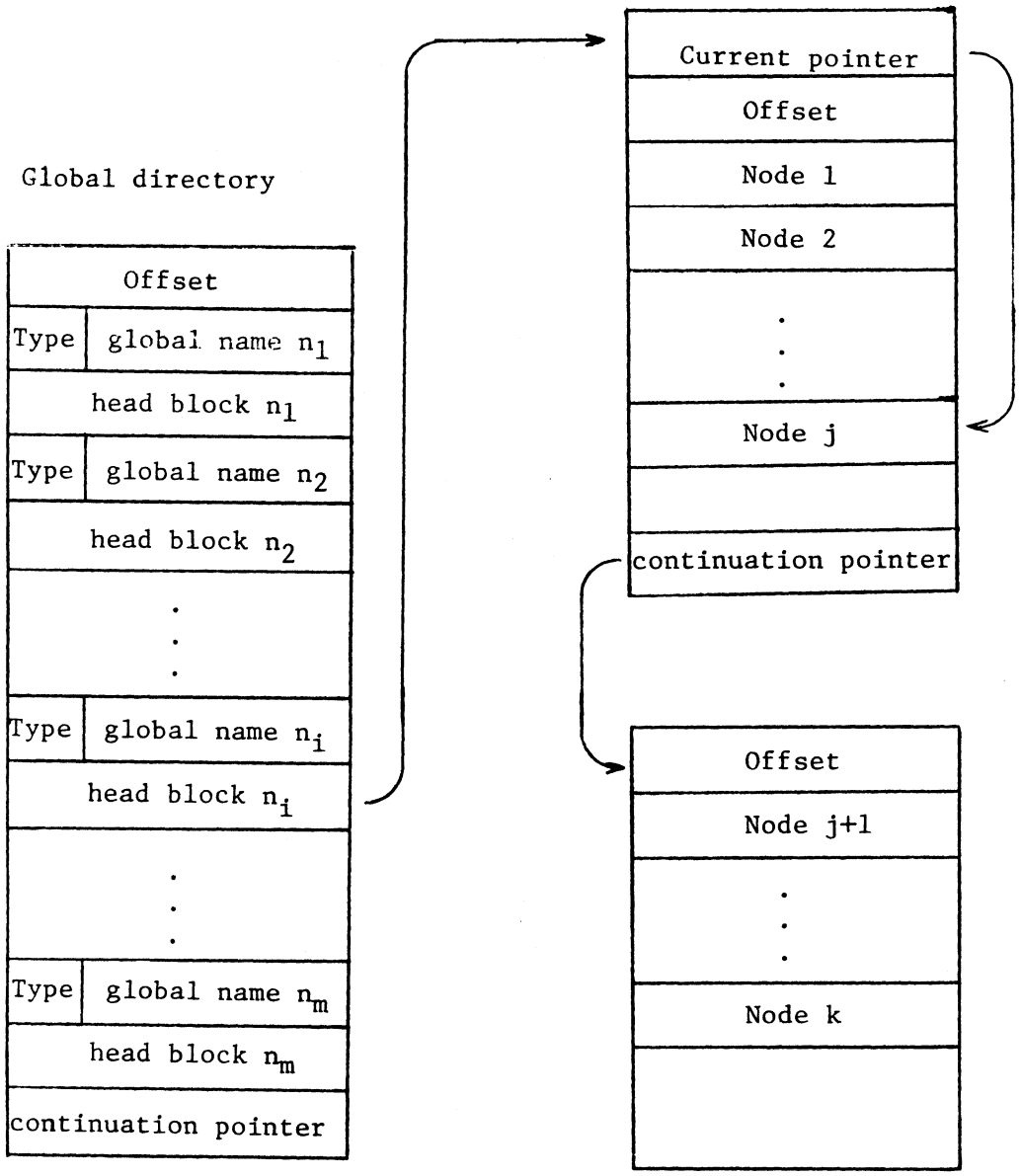
these is termed a "random" global and permits nodes to be accessed in an arbitrary (random) order. These globals can be cleanly introduced into the syntax of the MUMPS language and easily implemented.

5.2.1 "Sequential" globals

A sequential file is one in which the elements may be accessed in only one order, beginning with the first element in the file. In order to access the nth element in a file, it is necessary to access the preceeding n-1 elements. If information is written into a sequential file, elements are placed in the file in the order in which they are written, so that an attempt to read from that file will produce the same ordering.

Sequential files are extremely common in a number of data processing applications, dating from a variety of tape handling and card handling applications. In any application where each element must be served once and only once, a sequential file is the optimal processing structure. Examples of such applications are credit card billing systems and order processing systems. The string-processing functions of MUMPS would be extremely helpful in dealing with sequential records since the $PIECE and $EXTRACT functions can separate the fields of a record into their component parts quite simply.

A sequential global could be created which would read or write the next node in sequence. Such a global would have an associated pointer which would point to the next element in the sequence (see Figure 17). The current value of the pointer could be kept in the first word of the global's storage and consulted upon each access. For language and implementation simplicity, a sequential global could be opened for reading or for writing, but not both. In general, the updating of a sequential global should be done by copying from one global into a new one rather than by updating individual elements in the global. However, if unlimited updating is permitted, it is



The first word of the first block of a sequential global points to the address at which the next read or write operation occurs

Figure 17 - Implementation of sequential globals

possible to avoid serious recompaction problems by chaining the individual nodes together, making the actual storage ordering irrelevant.

The required language syntax to deal with sequential globals is quite simple. First, there is the need to set a sequential global to its initial position. This may be done with the LOCK command in the MUMPS language. Thus, if ^ABC is a sequential global, and the command

```
LOCK ^ABC
```

would reserve ^ABC for the user and would set the pointer to the first element of ^ABC. If there is the desire to "rewind" ^ABC and process it again, one could use a LOCK with no arguments, then a LOCK as above to accomplish the reinitialization.

Successive elements of the sequential global can simply be referenced by ^ABC, which unambiguously applies to the next element in the sequential global. Use of the sequential global on the left-hand side of a SET command implies output of an element to the sequential global, while each use of the sequential global on the right-hand side implies input of the next element. It should be noted that the MUMPS command

```
SET X=^ABC,Y=^ABC,Z=^ABC
```

is different from the command

```
SET (X,Y,Z)=^ABC
```

for sequential globals, since the first command involves the input of three elements, while the latter command has only one input.

Finally, processing sequential globals would be simplified by the introduction of a function to test for the end-of-file condition so that elements could be accessed until the last element was accessed. It is possible that the $DATA function could be applied to sequential globals for this purpose.

5.2.2 "Random" globals

The notion of a random global provides a mechanism which allows the case of a one-level global to be treated specially, since a linear structure is, as noted above, a fairly common occurrence. Rather than restricting the access to the global to be purely sequential, the random global allows nodes to be accessed in any order, and permits a given global to be read from or written into any ordering. In short, the capabilities are identical to those of the general purpose global, but a specialized storage structure is utilized to take advantage of the linear nature of the file.

By expanding the contents of the global directory to include an entry for the type of global (e.g. 0 for normal, 1 for sequential, 2 for random), it is possible to use exactly the same syntax for random globals as for normal tree-structured globals. When a global is first created, the protocol for creating globals must determine the type of global being created. (If one desires to permit creation of globals of various types during an executing MUMPS program, it would be necessary to include some special function or command to make a global directory entry indicating the type of global which is being created.)

The random global resembles an indexed sequential file in that the pointer from the global directory points to a disc block whose elements are a set of pointers. The value of the subscript calculates the offset from the beginning of the block, possibly carrying on to continuation blocks. When that element is found, it contains the address at which the node is stored. Since only the pointers must be ordered, the contents of the node may be stored in arbitrary order, permitting reallocation of space with corresponding readjustment of pointers if necessary. The node contains space for storing the subscript value so that it is possible to use a two-way pointer mechanism to simplify the

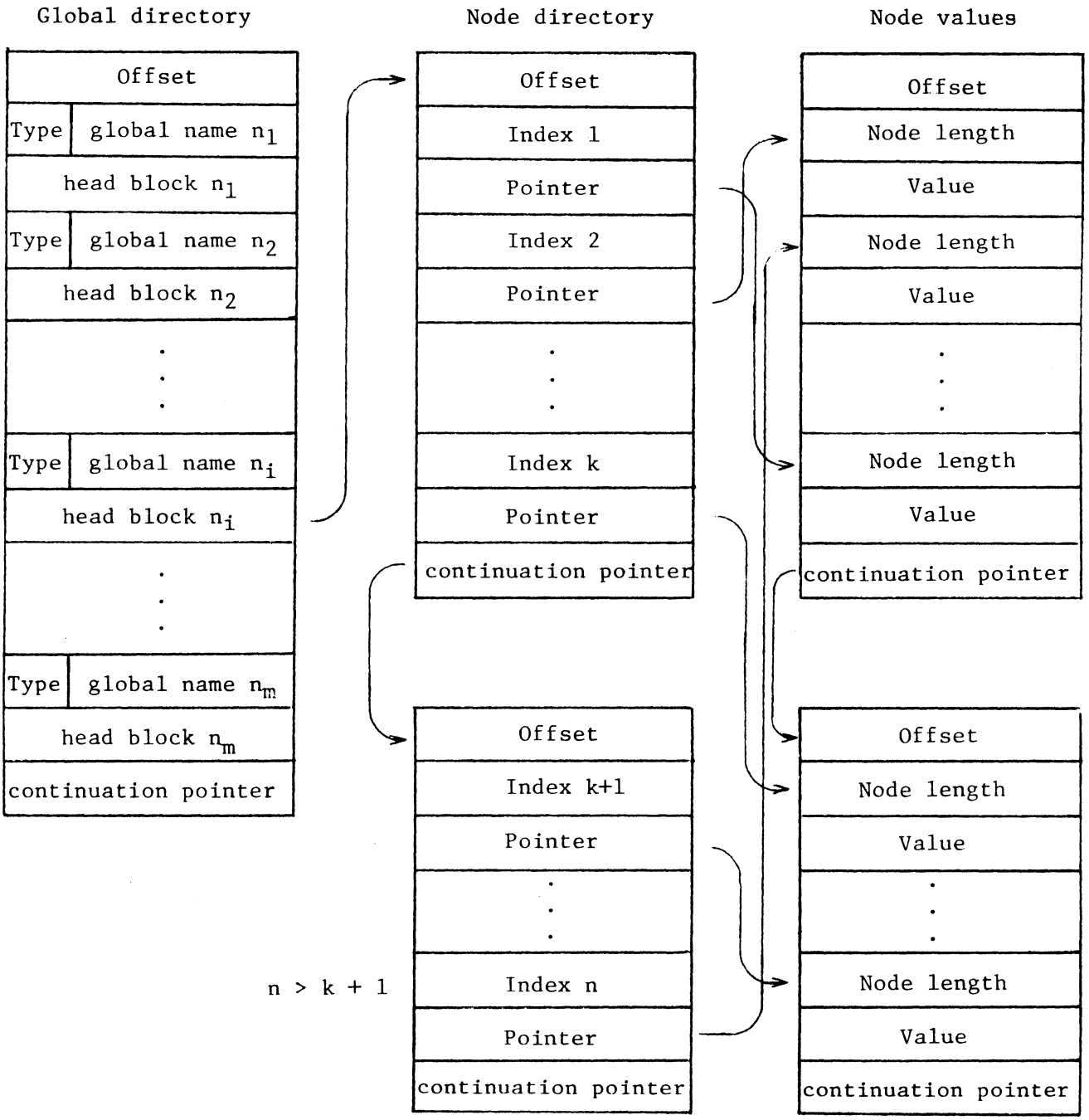


Figure 18 - Implementation of random globals

problems of rearranging the ordering of the nodes. Figure 18 shows the storage structure that can be used to accomplish the implementation of random globals.

5.2.3 "Declared" globals

The complete flexibility given by the general purpose tree-structure, combined with the automatic mode conversion features implicit in MUMPS, allows global nodes to change from integers to strings (for example) and allows new nodes to be arbitrarily introduced at any level with any subscript value. This generality has been shown to be extremely valuable for variable size hierarchical structures, but rather expensive in terms of implementation overhead. In discussing the algorithms to permit direct address calculation of node location from the sequence of subscripts, and in discussing the approaches to allocation and reallocation, it has been observed that the complete generality inherent in the concept of globals made it impossible to achieve these optimization ideals.

For some classes of hierarchical structures, though, one can clearly specify the precise structure of the hierarchy, including subscript ranges, the type of information to be stored in each node, and the maximum amount of information which will be stored in each node. In other words, one could specify to the system at global creation time the characteristics of the global. With this information, it is then possible to allocate storage with great efficiency and accuracy. Furthermore, if the global is to be largely static (with updates at daily, weekly, or monthly intervals), it would be feasible to build a table to perform address calculations. It would then be possible to write programs which could access this table for every reference to the global and jump directly to the desired node, without the intermediate disc accesses required for normal hierarchical access.

Although the notion of declaration of variables and storage structures is foreign to MUMPS, it is in extremely widespread use in other programming languages used for file handling application. For example, both PL/I and COBOL require the specific declaration of fixed size storage structures. Also, most commercially available, hierarchically organized data management systems place similar limitations upon their users and data base designers [Date, 1975].

In the MUMPS environment, the specification of a global structure would be done only to assist in optimization of global access and allocation for those cases where an a priori knowledge of the global contents exists. There would be no requirement to make such declarations; declaration-free usages of globals would proceed as always. Once again, incorporation of this facility would require no changes to the syntax of the command language, since the program would still be dealing with hierarchical shared files. If the description of the global existed, the entry for that global could be flagged in the global directory with the addition of a pointer to the description table.

The best technique for "declaring" globals is through the use of a conversational program which the user invokes to specify the structure of the global. This program would create the desired table. In that way, the notion of declarations would not be introduced into the MUMPS language.

This proposal is somewhat more complex than either the random global or the sequential global proposal, both in terms of its requirement on the user and its implementation within the existing framework of globals. Yet it remains consistent with the philosophy of MUMPS by keeping the file system invisible to most users and by improving the overall computational power of a MUMPS system.

These new global types offer a rather significant increase in the range of applicability of MUMPS. They also permit certain types of optimization to be performed which are impossible with the traditional MUMPS global structure. Furthermore, inclusion of these new types does not complicate the language in any way. All of these concepts should be investigated further for possible incorporation into MUMPS.

5.3 Global security

Because MUMPS was designed for sharing a data base among many users, no restrictions were built into MUMPS concerning access control of globals or global nodes. The right to access a MUMPS system automatically gave users the right to access the global data base. As MUMPS systems have developed, some small steps toward access control have been taken. The most significant of these is the development of separate global directories for different users (or classes of users), thereby preventing users from accessing globals which belong to other users.

As MUMPS systems have developed in complexity, however, there appears to be an increasing need for access control to globals and to global nodes. For example, one may wish to specify certain users who may change the values of global nodes or create new nodes (write access). One may wish to identify the "owner" of a global to decide who may delete all or part of the global. Finally, one may wish to permit certain users to reference nodes but not change them (read access). It is possible to incorporate all of this security information into tables within primary memory, directly within the global directories, or in the globals themselves.

Global security, however, is only part of a larger security problem involving access to the MUMPS system, access to specific programs, the ability to write new programs in addition to executing existing ones, and several other issues. Accordingly, this topic is being treated separately by the authors in a companion report, "Design of a Multiprogramming System for the MUMPS Language". Readers interested in security issues are referred to that report.

5.4 Other data management systems

Since the development of MUMPS, a number of rather significant innovations have been made concerning the management of large volumes of data. The urgent information management problems which led to the development of MUMPS have since spurred the development of a wide variety of tools for information management. Indeed, the problem of managing large quantities of data is one of the most challenging problems in computing today.

The past few years have seen a proliferation of data base management systems, with an extremely large number of commercially available products. The science of data structures and organization has made considerable advances as well. There are three major approaches to data management which have been taken other than the hierarchical approach of MUMPS, but it is still too early to determine which of these, if any, will prevail in future data base design and management.

The first approach is known as the "network" model. The approach was formulated by the Data Base Task Group of the CODASYL (Conference on Data Systems Languages) Programming Language Committee [CODASYL, 1971]. In a data base network, the nodes of data form a graph rather than a tree. In other words, any node may be connected to any number of other nodes without any

hierarchical relationship. The 1971 CODASYL report included a Data Management Language which provides for data specification, storage, and retrieval, with appropriate features for access control, concurrency (multi-user interlocks), and other necessary details.

The second approach is known as the "set-theoretic" approach [Childs, 1968]. Nodes in a data base are grouped together into sets and any number of sets may be related to one another. For data management purposes, one may think of sets being created around different properties possessed by items in a data base. A retrieval may then be specified, for example, as the intersection of a set of sets in order to identify those data objects which possess a specified number of properties.

The third approach is known as the "relational" model of data [Codd, 1970], which uses the algebra of relations to specify the correspondences between objects in a data base. Whereas hierarchical and network models of data assume an explicit relationship among the objects in a data base, the relational model makes no such assumption and creates the relationship only upon a specific retrieval request. The storage mechanism is hidden from the data management language and as many as five different storage techniques are commonly used within each relational system for information storage.

The attractiveness of the set-theoretic and relational approaches can be seen when one considers the variety of uses to which a data base is put. Suppose, for example, that the data base consists of abbreviated information about a set of patients visiting a clinic. For each patient, there exists a "mini medical record", which includes information on diagnoses, medications, hospitalizations, laboratory tests, etc. One application of the data base is for patient care; a provider wishes to retrieve part or all of the abbreviated record. Another application of the data is for research; a researcher wishes to identify all those patients having a given set of characteristics.

Within a hierarchical data base organization, the hierarchy will be designed to facilitate one application or the other, but not both, since one will involve utilizing the entire hierarchy while the other will involve accessing small pieces of information in an orthogonal way. With a relational or set-theoretic model, this built-in structure does not exist and the two types of access may be equally fast. Furthermore, the relational model does not presuppose any relationship among data objects and therefore would be most flexible for a broad range of research queries. A means for incorporating a relational model of data in a programming language has been proposed [Earley, 1973].

One of the fundamental underlying issues is that of data independence. One objective of the relational and set-theoretic approaches is to decouple the use of the data within a program from its physical representation on a secondary storage device. Most present computer applications (not just those in MUMPS) are data-dependent, in that the knowledge of the data organization and access technique is built into the application logic [Date, 1975]. If the storage structure were to be changed, it would necessitate changes in the application programs; achievement of data independence would eliminate that need. The price of data independence is a moderate decrease in the efficiency of execution of data management operations.

Each of these approaches to data base management has been implemented several times, using various sizes of computers. For each data management model, there exists a query language designed for nonprogramming users which carries out the storage and retrieval functions. These languages, of which SEQUEL is an example for relational systems [Chamberlin, 1974], provide extremely powerful, intuitive mechanisms for management, but have rather limited capabilities for computation. Furthermore, all of these languages are command-oriented, i.e., they require the user to formulate a query in the language rather than carrying out a dialogue. Some experimentation is underway [Codd, 1974] to develop programs which converse with the user and then automatically create the command in the data management language.

As yet, however, there does not exist a system which provides the high degree of data management capability and the computational power of MUMPS, and which can be implemented on a small-to-medium sized computer. However, the increasing use of networks of small computers, combined with a growing number of online application programs involving both computation and data management, has resulted in a number of research and development projects with similar goals [Wasserman, 1975].

5.5 Summary

MUMPS provides a data base management mechanism which is well integrated within the structure of a programming language and operating system. Use of that mechanism is performed in a symbolic manner so as to make most of the details of the file management system invisible to programmers. At the same time, however, the programmer must be intimately aware of the file structure for an application and must carefully build that data dependence into the programs.

The present MUMPS data management facilities are well-suited only for those data sets where a single explicit hierarchy exists and all implementations have been oriented to achieving maximum performance from that organization. When considered within the framework of general purpose data management systems, the view of data taken by MUMPS is quite narrow, in that only one type of data structure exists. If MUMPS is to be utilized effectively for a broader range of applications, it would seem that some thought should be given to methods for incorporating a broader view of data into MUMPS and to supporting more than one type of file organization. Although application programs would remain data-dependent, the programmer would have more options available for selecting data structures which would improve the exectuion efficiency of MUMPS programs.

6. Conclusion

This report has discussed the techniques used for implementing MUMPS globals on secondary storage devices. All existing versions implement globals as a set of linked, fixed length disc blocks. The number of pointers from each block is reduced to one by making use of the fact that all trees can be represented as binary trees (having two or fewer descendants per node). The down pointer, i.e., the left subtree, is stored explicitly, while the pointer to the right subtree is replaced by physical contiguity of the nodes in a disc block.

Considerable attention has been given to the problems of storage allocation and access time in a dynamic, hierarchical data base environment. Techniques for optimal allocation of secondary storage have been treated in general terms, since theoretical results for the hierarchical model of data base organization are based on a large number of assumptions which may not be true in MUMPS.

Finally, the strengths and weaknesses of MUMPS globals have been examined in the context of a wider range of data management systems. MUMPS is seen to be the only programming system which currently offers the combination of multi-programmed computation, string-processing, and data base management. However, there appear to be several types of modifications that can be made to the present MUMPS data management facilities, including addition of new global types, which can broaden their utility, thereby serving a greater range of data manipulation and management needs. Beyond that, additional research is required to develop mechanisms which help to achieve more flexible data structures and a larger measure of data independence within the MUMPS framework.

Bibliography

Bernstein, P.A., and Tsichritzis, D.C. "Allocating Storage in Hierarchical Data Base", Univ. of Toronto Computer Systems Research Group Technical Report CSRG-34, May 1974.

Bowie, J. "MUMPS", Proceedings 1973 MUMPS Users' Group Meeting, pp. 94-106.

Busacker, R.G., and Saaty, T.L., Finite Graphs and Networks: an Introduction with Applications, McGraw-Hill Book Company, New York, 1965.

Cardenas, A.F., "Evaluation and Selection of File Organization--A Model and System", Comm. ACM, Vol. 16, No. 9 (Sept., 1973), pp. 540-548.

Chamberlin, D.D. and Boyce, R.F. "SEQUEL: A Structured English Query Language," Proceedings of the 1974 ACM SIGFIDET Workshop on Data Definition, Access, and Control. New York: Association for Computing Machinery, 1975.

Childs, D.L. "Description of a Set-Theoretic Data Structure," Proc. AFIPS 1968 FJCC, pp. 557-564.

CODASYL, "Data Base Task Group Report" (April, 1971). Available from ACM, 1133 Avenue of the Americas, N.Y. 10036.

Codd, E.F. "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, Vol. 13, No. 6 (June, 1970), pp. 377-387.

Codd, E.F. "Seven Steps to Rendezvous with the Casual User," in Data Base Management Systems. Amsterdam: North-Holland Publishing Company, 1974.

Coffman, E.G. and Denning, P.J. Operating Systems Theory. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

Conway, M.E., "Design of a Separable Transition-Diagram Compiler," Comm. ACM, Vol. 6, No. 7 (July, 1963), pp. 396-408.

Date, C.J., An Introduction to Database Systems. Reading, Mass: Addison-Wesley, 1975.

Deo, N., Graph Theory with Applications to Engineering and Computer Science. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1974.

Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages, ed. F. Genuys. New York: Academic Press, 1968. pp. 43-112.

Earley, J., "Relational Level Data Structures for Programming Languages,", Acta Informatica, Vol. 2, No. 4 (1973), pp. 293-310.

Fuller, S.H., "Minimal-Total-Processing Time Drum and Disk Scheduling Disciplines," Comm. ACM, Vol. 17, No. 7 (July, 1974), pp. 376-381.

Greenes, R.A., Pappalardo, A.N., Marble, C.W., and Barnett, G.O., "Design and Implementation of a Clinical Data Management System," Computers and Biomedical Research, Vol. 2 (1969), pp. 469-485.

Johnson, M.E. and Dayhoff, R.E. "MUMPS Primer". MDC 1/6, 10/14/74, MUMPS Development Committee.

Knuth, D.E., The Art of Computer Programming: Vol. 1 Fundamental Algorithms second edition, Reading, Mass: Addison-Wesley, 1974.

Knuth, D.E., The Art of Computer Programming: Vol. 3 Sorting and Searching. Reading, Mass: Addison-Wesley, 1973.

Lum, V.Y., "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept," Comm. ACM, Vol. 16, No. 10 (Oct., 1973), pp. 603-612.

McCraith, D.L., and Pappalardo, A.N., "MUMPS-An Interpretative Information System for Minis," Digest of Papers, COMPCON Fall 1974, pp. 177-178.

Ore, O., Graphs and Their Uses. New York: Random House, 1963.

Peck, L.J. and Greenes, R.A., "Preliminary MUMPS Documentation Manual". MDC 3/4, 10/14/74, MUMPS Development Committee.

Shaw, A.C., The Logical Design of Operating Systems. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1974.

Sherertz, D.D., "MUMPS Transition Diagrams", MDC 1/9, 3/12/75, MUMPS Development Committee.

Sterbenz, P., Floating-Point Computation. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1974.

Teorey, T.J., "Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems," Proceedings 1972 FJCC, pp. 1-11.

Wasserman, A.I., et al, "A Formal Approach to the Implementation of the Proposed MUMPS Language Standard," Proceedings 1974 MUMPS Users' Group Meeting, pp. 159-169.

Wasserman, A.I., "Programming Language Requirements for Interactive Software," in preparation, 1975.

Cylinder--Conceptually, the set of all disc blocks which are equidistant from the edge of a disc storage device, when the device has more than two read/write surfaces. Alternatively, a cylinder consists of the collection of disc blocks that can be read or written on a disc without moving the read/write head.

Data Base--In a computing system, the set of data available to the users and processes of the system. All or part of the data base may be "shared" (available to more than one user or process) or "private" (restricted to one user or class of users)

Deadlock--(also "deadly embrace") A condition in a multiprogrammed system where two or more processes are blocked from doing productive work because none of the processes can obtain all of the resources needed to continue work. Example: if two processes A and B both require globals X and Y to complete their work and A has X and B has Y, then neither A nor B can finish. This condition is termed deadlock and must be prevented in multiprogrammed systems.

Descend--In a tree-structure, to move away from the root.

Disc block--A segment of data stored on a rotating secondary storage device which is transferred by the input/output control system. Disc blocks are typically from 200 to 512 characters in length.

Edge--In graph theory, the connection between any two nodes.

Graph--A set of objects V, called vertices, and a set of objects E, called edges, such that each element e_k of E is identified with an unordered pair (v_j,v_k) of elements of V. In other words, a collection of zero or more points with zero or more edges connecting some or all of those points. If it is possible to reach all of the vertices by traversing the edges from any given vertex, the graph is said to be connected.

Header--(also "head block") The disc block to which a global directory entry points; contains information about the nodes closest to the root.

Leaf--A node in a tree structure with no descendant nodes.

Node--A vertex; in MUMPS globals, an entity with a symbolic address with which information can be associated; a root node corresponds to the root of a tree, in that it has no ancestors; a terminal node is a leaf with no descendant nodes.

Pointer--A data element which contains the address of some location where additional data (possibly another pointer) is stored; a continuation pointer is used in MUMPS global allocation to point to blocks which contain nodes which are siblings of nodes in a given block; a down pointer is used in MUMPS global allocation to point to blocks which contain nodes which are the descendants of a given node.

Rotational latency--On a rotating secondary storage device, the period of time which elapses before a desired disc block passes under the read/write head.

Seek time--On a moving head rotating secondary storage device, the period of time which elapses before the read/write head can be positioned over a particular cylinder (or track) of the device.

Sibling set--In the theory of trees, the set of all vertices (nodes) which have the same parent.

Trace--In the theory of trees, the sequence of vertices which must be traversed in going from the root to a designated node.

Track--The set of disc blocks which can be read or written on a single platter disc without moving the read/write head (cylinder applies to multiple platter discs).

Tree--A graph which has no cycles and which has a designated node called the root; a binary tree is a tree in which no node has more than two immediate descendants.