To: The CGNS Steering Committee

Subject: Comment on *Proposal for the support of hierarchical data structures in the CGNS unstructured meshes format*. by M. Delanaye, E. Robin, and A. Patel. NUMECA Int. July 2000

Re: M. Aftosmis, NASA Ames Research Center, Moffett Field, CA 94404.

Date: November 2000

The proposal titled "Proposal for the support of hierarchical data structures in the CGNS unstructured meshes format" by M. Delanaye, E. Robin, and A. Patel of NUMECA Inc. states that it proposes modification of the CGNS unstructured mesh format to include hierarchical information. Direct hierarchical references are currently absent from the CGNS standard and this proposal suggests a modification to the standard to include such information. This is clearly an important addition and strategies for including this information should be reviewed by the steering committee. Unfortunately, about half of the proposal focuses not on hierarchical information, but upon proposed extensions to the basic storage of unstructured mesh connectivity. We show that if adopted, these changes would result in a 2 to 3.5 fold increase in storage for mesh connectivity. These modifications are not only unnecessary and difficult to support, but may have serious consequences detrimental to the success of the CGNS unstructured format. This letter offers comments upon both aspects of the NUMECA proposal, hierarchical storage and storage of unstructured element connectivity.

To avoid the appearance of any conflict of interest, members of the steering committee should understand clearly that *none* of the data structures discussed or proposed in this letter are used in the author's own research or codes.

Hierarchical Elements:

The proposal suggests adding an optional data array of type integer to the `Elements_t` data structure with two integers for each element. The first of these integers would be the index of the "first" child while the second points to a sibling element. This arrangement is a bi−directional structure and allows any cell to quickly point to both its children and its siblings. Finding all children of a given parent requires only looking to see who the first child is, and then traversing the linked list from sibling−to−sibling, until the list terminator (−1) is encountered.

Initially this structure appears both compact and flexible. For this reason, such structures are common within a running program where redundancy can often provide advantages in execution speed. Nevertheless, two aspects of this structure are unattractive in an archival storage format.

1. Non−uniqueness − Which child is pointed to directly by the parent? Why is this child preferred? In fig. 1 and table 1. of the proposal, parent cell *A* points only to child *B* and while *B, C, D,* and *E,* are all children of *A,* child *E* has no sibling pointer, while all other children of *A* have one. Clearly there is no obvious preference for one sibling over another, and we are just traversing a linked list, built in some order by some previous application program. This asymmetry and non−uniqueness of the format has permitted the *Originator* (the application which originally created the data) to an imprint of decisions made *within* that application to affect all downstream *Clients* (in this context, downstream applications which use this dataset).

2. Redundancy − The non−uniqueness pointed to in the preceding paragraph is indicative of redundancy within the data structure. In table 1 (of the proposal) this redundancy also manifests itself by the large number of "−1" flags that appear. Even in this simple example, only 6 of the allocated 14 storage locations point to something other than this list terminator (−1). Over half the storage allocated (paid for in disk space and file access time) is wasted. In fact, this waste is *mandated* by the format. Following the rules set down in the original proposal, a parent points to its first child and all cells may point to one sibling. Only parents will have a non−flag "child" pointer. Since the physical cells covering the domain in any computation are invariably "children" and not "parents" every fine cell in the domain will *always* waste this storage location. Additionally since the final sibling on the linked list will never point to additional siblings, at least one child of every parent will waste his sibling pointer.

Both of these unsavory features are a direct result of redundancy within the proposed data structure. Such redundancy has two other major consequences. Most obviously it costs the industry money as redundant information is stored in files across the entire industry. Perhaps less obviously, this information takes more time to retrieve, and as processor speeds continue to out−pace disk I/O, retrieval will take progressively more time (relative to processor speed). In addition, redundancy permits topological conflicts to be buried within datasets. If one can find out information via multiple methods, then self−inconsistent topological relationships may lie inadvertently buried within datasets. While within an executing program redundant data structures can mask latency and speed execution, overly redundant archival storage incurs costs in both storage media and access time (as well as network bandwidth).

A Counter Proposal

A simple modification to the proposed format avoids the consequences of the preceding paragraphs, while simultaneously reducing the required storage by half. Under this system every child simply points to its parent. Parents who are not children themselves point to a list terminator (−1 in the original proposal). The `hierarchicalData` array would only contain only one integer (not two) and the example in table 1 (of the original proposal) would contain only one list terminator. The fact that this "bottom−up" scheme requires exactly half the storage of that originally proposed is no accident, and the presence of such a scheme was alluded to by the fact that less than ½ of the locations in table 1 (of the proposal) are actually used.

1. Unique/Symmetric − all children with the same parent are treated identically, and point uniquely to the same entity. This makes it more difficult for the originator to artificially bias the data for downstream clients.

2. Minimal − requires ½ the storage of the scheme originally proposed, and since it does not store redundant information, it encourages storage of self−consistent datasets.

3. Flexible − The linked−list storage scheme in the original proposal can be recovered by a single sweep over the cells. In this sweep, each child puts its index at the tail of the linked−list of it's parent. On many systems this will be as fast, or faster than actually reading the redundant second integer of the original scheme from disk. Moreover as processor speeds increase, this one−time−cost decreases with processor speed and live memory bandwidth − which historically increase an order−of−magnitude faster than gains in bandwidth to disk.

Table 1: Illustration of a new `hierarchicalData` array for cells shown in Fig.1 of the original proposal.

Cell	A	B	C	D	E	F	G
Parent	−1	A	A	A	A	D	G

Finally, it is worth repeating the third point listed above. Hierarchical data in the form requested by the original NUMECA proposal can be obtained directly with a single pass down the cell list. There are many applications that will use custom data structures to traverse hierarchical data, and it is impossible for the standard to anticipate the implementations of all such clients. The format discussed here makes it possible to construct both list−based and tree−based hierarchical information with simple constant−time (per−cell) algorithms. Moreover it does not saddle downstream clients of a dataset with decisions made internally in the originating application.

Storage of Unstructured Element Connectivity

It can be argued that the preceding section discusses removal of a single redundant integer from the proposed format for storing hierarchical data. While this is a seemingly trivial change, it is recommended on the principle that the archival standard format should store minimal and generic information in a form that makes it possible for clients to inexpensively build their own custom data structures. The second half of the NUMECA proposal concerns a major change in the way that element connectivity is stored in a way that dramatically disregards this principle. If adopted, this scheme will incur massive additional storage costs, and penalize client applications with increasingly slow I/O and increased development costs as developers of client applications scramble to code for a whole host of potential storage schemes. The loss of standardization that will result will also unduly burden the CGNS development team since many internal functions with duplicate functionally will need to be written to support the plethora of possible options.

Currently the CGNS unstructured format supports Element−>node connectivity. Paragraph 2 of the proposal states that in the opinion of the authors, "this is too restrictive" and proposes an alternative based upon a dimensional cascade of element−>face, face−>edge, edge−>node. Contrary to the proposal's claim that the current format is "too restrictive", it can easily and quickly provide precisely the information requested by the NUMECA proposal using a simple 2 pass algorithm that runs with a constant time bound (per−cell). The algorithm is standard for constructing face and edge−lists from cell−based element storage.

1. (1 cell sweep) Each cell puts its own index on linked−lists of each of its node.
2. (1 node sweep) Each node traverses the cells on its own linked list and matches cells to create cell−>face, face−>edge, and edge−>node connectivity.

Since the work done at each cell in step 1, or each node in step 2 does not depend upon the total number of cells in the mesh, the work per−cell (or per−node) is clearly constant.

If a particular client chooses not to re−compute this connectivity, such clients always have the option of storing additional information separately, however such client customizations should not be forced upon all users of the CGNS standard. In this way, idiosyncrasies of a particular client do not get imprinted into all datasets for which it is the originator.

Including the proposed modifications would have a serious impact upon third−party developers of CGNS compliant software. Such developers would not know what to expect in the files and would therefore have no alternative but to support *all possible*

options in order to remain CGNS compliant. This dramatically increases development cost and inherently delays the appearance of such applications in the marketplace. Since any client who wishes to claim CGNS compliance would have to support *all possible* formats, this would have the effect of penalizing the entire industry with increased development costs or risk loss−of−standardization − negating the purpose of CGNS.

Finally, let us examine the storage requirements for the data structures in the proposal. We have already established that within a running application, it is common practice to use topologically consistent but redundant data to mask latency or to speed execution, and therefore it is not the intent of this section to critique various data structures, only to establish the storage requirements for archival purposes.

Consider an asymptotically large (no boundaries) data set with *N* nodes connected either with pure hexahedral or pure tetrahedral tessellations. These represent bounding cases with mixed element or hybrid meshes falling between these extrema.

Currently supported: Cell−>Node Connectivity (*N* nodes)

	Pure Hex mesh	Pure Tet mesh
Total No. of Cells	*N*	6*N*
Total Connectivity	(*N* cells)(8 nodes/cell) = **8*N***	(6*N* cells)(4 nodes/cell) = **24*N***

Proposed by NUMECA: Cell−>Face, Face−>Edge, Edge−>Node Connectivity (*N* nodes)

	Pure Hex mesh	Pure Tet mesh
Total No. of Cells	*N*	6*N*
No. of faces/cell	6	4
No. of faces in mesh	4*N*	12*N*
No. of edges/face	4	3
No. of edges in mesh	3*N*	7*N*
Cell−>face storage	(6 faces/cell)(*N* cells) = 6*N*	(4 faces/cell)(6*N* cells) = 24*N*
Face−>edge storage	(4 edges/face)(4*N* faces) = 16*N*	(3 edges/face)(4*N* faces) = 12*N*
Edge−>node storage	(2 nodes/edge)(3*N* edges) = 6*N*	(2 nodes/edge)(7*N* edges) = 14*N*
Total Connectivity	6*N* + 16*N* + 6*N* = **28*N***	24*N* + 12*N* + 14*N* = **50*N***

For either of the cell types considered its obvious that the scheme in the NUMECA proposal incurs an excessive penalty for connectivity storage. For pure hexahedral meshes the proposed scheme stores 3.5 times more data than the existing structure while for tetrahedral meshes it stores 2.08 times more data. Mixed meshes of pyramids, prisms, etc. will fall between these extremes.

In light of this simple analysis and when one considers that the proposed data structures can be created on−the−fly in constant time (per−cell) from those already supported by the CGNS unstructured format, there seems to be little motivation to adopt this proposed modification. In fact, all of the earlier arguments about processor speed and memory bandwidth increasing far faster than disk bandwidth take on new urgency. Clients that do not wish to re−compute their custom internal data structures should be encouraged to keep proprietary files outside of the standard. However, with such massive storage penalties associated with schemes such as these, it is likely that in the long run, re−computation will be substantially faster than storage on ever−more−distant disk. As the archival standard, CGNS should continue to strive to consume minimal resources while making it possible for clients to inexpensively recover their proprietary formats.