Defense Technical Information Center Compilation Part Notice

ADP023792

TITLE: Enhancements to the eXtensible Data Model and Format [XDMF]

DISTRIBUTION: Approved for public release, distribution unlimited

This paper is part of the following report:

TITLE: Proceedings of the HPCMP Users Group Conference 2007. High Performance Computing Modernization Program: A Bridge to Future Defense held 18-21 June 2007 in Pittsburgh, Pennsylvania

To order the complete compilation report, use: ADA488707

The component part is provided here to allow users access to individually authored sections of proceedings, annals, symposia, etc. However, the component should be considered within the context of the overall compilation report and not as a stand-alone technical report.

The following component part numbers comprise the compilation report:
ADP023728 thru ADP023803

Enhancements to the eXtensible Data Model and Format (XDMF)

Jerry A. Clarke and Eric R. Mark
US Army Research Laboratory (ARL), Aberdeen Proving Ground, MD
{clarke, emark}@arl.army.mil

Abstract

The eXtensible Data Model and Format (XDMF) is an active common data hub used to pass meta data and value between application modules in a standard fashion. XDMF views data as consisting of two basic types: Light data and Heavy data. Light data is both meta data and small amounts of values. Heavy data typically consists of large arrays of values. Support for XDMF is included in the widely used visualization package ParaView, computational codes like ALE3D from Lawrence Livermore National Laboratory (LLNL) and the SIERRA computational framework from Sandia National Laboratory (SNL). A major effort is now under way to enhance XDMF to more directly support concepts like Adaptive Mesh Refinement (AMR), mixed topologies, higher order elements and parallel datasets. Discussed here are the motivations for the enhancements and the benefits of the new version.

1. XDMF: The Need to Enhance

The eXtensible Data Model and Format (XDMF) were developed at a standardized way to exchange scientific data between high performance computing (HPC) codes and tools. Used as both a common format for visualization of many different codes and as a vehicle for developing coupled simulations, XDMF has been adapted to purposes beyond its original design.

Primarily driven by the need to efficiently support the HPC codes ALE3D from Lawrence Livermore National Laboratory (LLNL), CTH and the SIERRA framework code collection from Sandia National Laboratory (SNL), XDMF is undergoing a major redesign. Interested parties from Army Research Laboratory, LLNL, SNL, and Kitware Inc. are participating in the new effort.

Some of the major features required by the new design are:

- Be as backwards compatible as possible
- Remain easy to represent simple topologies
- Represent hierarchical relationships
- Handle higher order elements
- Be efficient in parallel situations
- Handle topologies of mixed element types
- Handle additional application specific information and description
- Map well to existing HPC codes, formats, and visualization tools

XDMF categorizes data by two main attributes; size and function. Data can be ***Light*** (typically less than about a thousand values) or ***Heavy*** (megabytes, terabytes, etc.). In addition to raw values, data can refer to ***Format*** (rank and dimensions of an array) or ***Model*** (how that data is to be used, i.e., XYZ coordinates vs. Vector components).

XDMF uses eXtensible Markup Language (XML) to store Light data and to describe the data Model. HDF5 is used to store Heavy data. The data Format is stored redundantly in both XML and hierarchical data format (HDF5). This allows tools to parse XML to determine the resources that will be required to access the Heavy data.

While not required, a C++ application programming interface (API) is provided to read and write **XDMF** data. This API has also been wrapped so it is available from popular languages like Python, Tcl, and Java. The API is not necessary in order to produce or consume XDMF data. Currently, several HPC codes that already produced HDF5 data use native text output to produce the XML necessary for valid XDMF.

1.1. XML

The XML format is widely used for many purposes and is well documented at many sites. There are numerous open source parsers available for XML. The XDMF API takes advantage of the libxml2 parser to provide the necessary functionality. Without going into too much detail, XDMF views XML as a "personalized HTML" with some special rules. It is case sensitive and is made of three major components: elements, entities, and processing information. In XDMF, we're primarily concerned with the elements. These elements follow the basic form:

```
<ElementTag
        AttributeName="AttributeValue"
        AttributeName="AttributeValue"
        ….. >
                CData
</ElementTag>
```

Each element begins with an <tag> and ends with a </tag>. Optionally there can be several "Name=Value" pairs which convey additional information. Between the <tag> and the </tag> there can be other <tag></tag> pairs and/or character data (CData). CData is typically where the values are stored; like the actual text in an HTML document. The XML parser in the XDMF API parses the XML file and builds a tree structure in memory to describe its contents. This tree can be queried, modified, and then "serialized" back into XML.

XML is said to be "well formed" if it is syntactically correct. This is if all of the quotes match, all elements have end elements, etc. XML is said to be "valid" if it conforms to the *Schema* or *DTD* defined at the head of the document. For example, the schema might specify that element type A can contain element B but not element C. Verifying that the provided XML is well formed and/or valid are functions typically performed by the XML parser. Additionally XDMF takes advantage of two major extensions to XML.

1.2. XInclude

As opposed to entity references in XML (which is a basic substitution mechanism), XInclude allows for the inclusion of files that are not well formed XML. This means that with XInclude the included file could be well formed XML or perhaps a flat text file of values. The syntax looks like this:

```
<Xdmf Version="2.0"
xmlns:xi="http://www.w3.org/2001/XInclude">
<xi:include href="Example3.xmf"/>
</Xdmf>
```

The xmlns:xi establishes a namespace xi. Then anywhere within the Xdmf element, xi:include will pull in the URL.

1.3. XPath

This allows for elements in the XML document and the API to reference specific elements in a document. For example:

The first Grid in the first Domain

/Xdmf/Domain/Grid

The tenth Grid .... XPath is one based.

/Xdmf/Domain/Grid[10]

The first grid with an attribute *Name* which has a value of *"Copper Plate"*

/Xdmf/Domain/Grid[@Name="Copper Plate"]

All valid XDMF must appear between the <Xdmf> and the </Xdmf>. So a minimal (empty) XDMF XML file would be:

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>
<Xdmf Version="2.0">
</Xdmf>
```

While there exists an Xdmf DTD and a Schema they are only necessary for validating parsers. For performance reasons, validation is typically disabled.

2. XDMF Elements

The organization of XDMF begins with the ***Xdmf*** element. So that parsers can distinguish from previous versions of XDMF, there exists a *Version* attribute (currently at 2.0). Any element in XDMF can have a *Name* attribute or have a *Reference* attribute. The Name attribute becomes important for grids while the Reference attribute is used to take advantage of the XPath facility (more detail on this later). Xdmf elements contain one or more ***Domain*** elements (computational domain). There is seldom motivation to have more than one Domain.

A Domain can have one or more ***Grid*** elements. Each Grid contains a ***Topology***, ***Geometry***, and zero or more ***Attribute*** elements. Topology specifies the connectivity of the grid while Geometry specifies the location of the grid nodes. Attribute elements are used to specify values such as scalars and vectors that are located at the node, edge, face, cell center, or grid center.

To specify actual values for connectivity, geometry, or attributes, XDMF defines a ***DataItem*** element. A DataItem can provide the actual values or provide the physical storage (which is typically an HDF5 file).

2.1. XdmfItem

There are six different types of DataItems:

- **Uniform** – this is the default. A single array of values.
- **Collection** – a one dimension array of DataItems.
- **Tree** – a hierarchical structure of DataItems
- **HyperSlab** – contains two data items. The first selects the start, stride and count indexes of the second DataItem.
- **Coordinates** – contains two DataItems. The first selects the parametric coordinates of the second DataItem.
- **Function** – calculates an expression.

2.1.1 Uniform

The simplest type is Uniform that specifies a single array. As with all XDMF elements, there are reasonable defaults wherever possible. So the simplest DataItem would be:

```
<DataItem Dimensions="3">
        1.0 2.0 3.0
</DataItem>
```

Since no *ItemType* has been specified, Uniform has been assumed. The default *Format* is XML and the default *NumberType* is a 32 bit floating point value. So the fully qualified DataItem for the same data would be:

```
<DataItem ItemType="Uniform"
        Format="XML"
        NumberType="Float" Precision="4"
        Rank="1" Dimensions="3">
        1.0 2.0 3.0
</DataItem>
```

Since it is only practical to store a small amount of data values in the XML, production codes typically write their data to HDF5 and specify the location in XML. HDF5 is a hierarchical, self describing data format. So an application can open an HDF5 file without any prior knowledge of the data and determine the dimensions and number type of all the arrays stored in the file. XDMF requires that this information also be stored redundantly in the XML so that applications need not have access to the actual heavy data in order to determine storage requirements.

For example, suppose an application stored a three dimensional array of pressure values at each iteration into an HDF5 file. The XML might be:

```
<DataItem ItemType="Uniform"
        Format="HDF"
        NumberType="Float" Precision="8"
        Dimensions="64 128 256">
        OutputData.h5:/Results/Iteration 100/Part
        2/Pressure
</DataItem>
```

Dimensions are specified with the slowest varying dimension first (i.e., KJI order). The HDF filename can be fully qualified, if it is not it is assumed to be located in the current directory or the same directory as the XML file.

2.1.2. Collection and Tree

Collections are Trees with only a single level. This is such a frequent occurrence that it was decided to make a Collection a separate type in case the application can optimize access. Collections and Trees have DataItem elements as children. The leaf nodes are Uniform DataItem elements:

```
<DataItem Name="Tree Example" ItemType="Tree">
        <DataItem ItemType="Tree">
                <DataItem Name="Collection 1"
                ItemType="Collection">
                        <DataItem Dimensions="3">
                                1.0 2.0 3.0
                        </DataItem>
                        <DataItem Dimensions="4">
                                4 5 6 7
                        </DataItem>
                </DataItem>
        </DataItem>
        <DataItem Name="Collection 2"
        ItemType="Collection">
                <DataItem Dimensions="3">
                        7 8 9
                </DataItem>
                <DataItem Dimensions="4">
                        10 11 12 13
                </DataItem>
        </DataItem>
        <DataItem ItemType="Uniform"
                Format="HDF"
                NumberType="Float" Precision="8"
                Dimensions="64 128 256">
                OutputData.h5:/Results/Iteration
                100/Part 2/Pressure
        </DataItem>
</DataItem>
```

This DataItem is a tree with three children. The first child is another tree that contains a collection of two uniform DataItem elements. The second child is a collection with two uniform DataItem elements. The third child is a uniform DataItem.

2.1.3. HyperSlab and Coordinate

A *HyperSlab* specifies a subset of some other DataItem. The slab is specified by giving the start, stride, and count of the vales in each of the target DataItem dimensions. For example, given a dataset MyData.h5:/XYZ that is 100×200×300×3, we could describe a region starting at [0,0,0,0], ending at [50, 100, 150, 2] that includes every other plane of data with the HyperSlab DataItem.

```
<DataItem ItemType="HyperSlab"
        Dimensions="25 50 75 3"
        Type="HyperSlab">
                <DataItem
                        Dimensions="3 4"
                        Format="XML">
                                0 0 0 0
                                2 2 2 1
                                25 50 75 3
                </DataItem>
                <DataItem
                        Name="Points"
                        Dimensions="100 200 300 3"
                        Format="HDF">
                                MyData.h5:/XYZ
                </DataItem>
</DataItem>
```

Notice that the first DataItem specified Start, Stride and Count for each dimension of the second DataItem. Suppose, instead that we only wish to specify the first Y data value from the DataItem and the last X value. This can be accomplished by providing the parametric coordinate of the desired values and using the *Coordinates* ItemType.

```
<DataItem ItemType="HyperSlab"
        Dimensions="2"
        Type="HyperSlab">
                <DataItem
                        Dimensions="2 4"
                        Format="XML">
                        0 0 0 1
                        99 199 299 0
```

```
                </DataItem>
                <DataItem
                        Name="Points"
                        Dimensions="100 200 300 3"
                        Format="HDF">
                                MyData.h5:/XYZ
                </DataItem>
</DataItem>
```

The first Y value is index 1 of item 0,0,0 while the last X value is index 0 of item 99, 199, 299. The dimensionality of the specified coordinates must match that of the target DataItem.

2.1.4. Function

Function ItemType specifies some operation on the children DataItem elements. The elements are referenced by $X where X is the zero based index of the child. For example, the following DataItem would add the two children DataItem elements together in a value by value operation resulting in the values 5.1, 7.2 and 9.3:

```
<DataItem ItemType="Function"
        Function="$0 + $1"
        Dimensions="3">
        <DataItem Dimensions="3">
                1.0 2.0 3.0
        </DataItem>
        <DataItem Dimensions="3">
                4.1 5.2 6.3
        </DataItem>
</DataItem>
```

The function description can be arbitrarily complex and contain SIN, COS, TAN, ACOS, ASIN, ATAN, LOG, EXP, ABS, and SQRT. In addition, there are the JOIN() and WHERE() expressions. JOIN can concat or interlace arrays while WHERE() can extract values where some condition is true. In the following examples we take advantage of the XPath facility to reference DataItem elements that have been previously specified:

Multiply two arrays (element by element) and take the absolute value

```
  <DataItem ItemType="Function"
    Function="ABS($0 * $1)">
    <DataItem Reference="/Xdmf/DataItem[1]" />
    <DataItem Reference="/Xdmf/DataItem[2]" />
  </DataItem>
```

Interlace 3 arrays (Useful for describing vectors from scalar data)

```
  <DataItem ItemType="Function"
    Function="JOIN($0 , $1, $2)">
    <DataItem Reference="/Xdmf/DataItem[1]" />
    <DataItem Reference="/Xdmf/DataItem[2]" />
    <DataItem Reference="/Xdmf/DataItem[3]" />
  </DataItem>
```

3. Grid

The DataItem element is used to define the data format portion of XDMF. It is sufficient to specify fairly complex data structures in a portable manner. The data model portion of XDMF begins with the *Grid* element. A Grid is a container for information related to two-dimensional (2D) and three-dimensional (3D) points, structured or unstructured connectivity, and assigned values.

The Grid element now has a GridType attribute. Valid GridTypes are:

- **Uniform** – a homogeneous single grid (i.e. a pile of triangles)
- **Collection** – an array of Uniform grids
- **Tree** – a hierarchical group
- **SubSet** – a portion of another Grid

Uniform Grid elements are the simplest type and must contain a *Topology* and *Geometry* element. Just like the DataItem element, Tree and Collection Grid elements contain other Grid elements as children:

```
<Grid Name="Car Wheel" GridType="Tree">
        <Grid Name="Tire" GridType="Uniform">
                <Topology ....
                <Geometry ...
        </Grid>
        <Grid Name="Lug Nuts" GridType="Collection">
                <Grid Name="Lug Nut 0"
                GridType="Uniform"
                        <Topology ....
                        <Geometry ...
                </Grid>
                <Grid Name="Lug Nut 1"
                GridType="Uniform"
                        <Topology ....
                        <Geometry ...
                </Grid>
                <Grid Name="Lug Nut 2"
                GridType="Uniform"
                        <Topology ....
                        <Geometry ...
                </Grid>
        </Grid>
        ....
```

A SubSet GridType is used to define a portion of another grid or define new attribute on grid. This only selects the geometry and topology of another grid, the attributes from the original grid are not assigned. The Section attribute of a SubSet can be *DataItem* or *All.*

3.1. Topology

The Topology element describes the general organization of the data. This is the part of the computational grid that is invariant with rotation, translation, and scale. For structured grids, the connectivity is implicit. For unstructured grids, if the connectivity differs from the standard, an Order may be specified. Currently, the following Topology cell types are defined:

Linear

- Polyvertex – a group of unconnected points
- Polyline – a group of line segments
- Polygon
- Triangle
- Quadrilateral
- Tetrahedron
- Pyramid
- Wedge
- Hexahedron

Quadratic

- Edge_3 – Quadratic line with 3 nodes
- Tri_6
- Quad_8
- Tet_10
- Pyramid_13
- Wedge_15
- Hex_20

Arbitrary

- Mixed – a mixture of unstructured cells

Structured

- 2DSMesh - Curvilinear
- 2DRectMesh – Axis are perpendicular
- 2DCoRectMesh – Axis are perpendicular and spacing is constant
- 3DSMesh
- 3DRectMesh
- 3DCoRectMesh

There is a *NodesPerElement* attribute for the cell types where it is not implicit. For example, to define a group of Octagons, set Type="Polygon" and NodesPerElement="8". For structured grid topologies, the connectivity is implicit. For unstructured topologies the Topology element must contain a DataItem that defines the connectivity:

```
<Topology Type="Quadrilateral" NumberOfElements="2"
>
        <DataItem Format="XML" DataType="Int"
        Dimensions="2 4">
                0 1 2 3
                1 6 7 2
        </DataItem>
</Topology>
```

The connectivity defines the indices into the XYZ geometry that define the cell. In this example, the two quads share an edge defined by the line from node 1 to node 2. A Topology element can define *Dimensions* or *NumberOfElements*; this is just added for clarity.

Mixed topologies must define the cell type of every element. If that cell type does have an implicit number of nodes that must also be specified. In this example, we define a topology of three cells consisting of a Tet (cell type 6) a Polygon (cell type 3) and a Hex (cell type 9):

```
<Topology Type="Mixed" NumberOfElements="3" >
        <DataItem Format="XML" DataType="Int"
        Dimensions="20">
                6     0 1 2 7
                3  4  4 5 6 7
                9     8 9 10 11 12 13 14 15
        </DataItem>
</Topology>
```

Notice that the Polygon must define the number of nodes (4) before its connectivity. The cell type numbers are defined in the API documentation.

3.2. Geometry

The Geometry element describes the XYZ values of the mesh. The important attribute here is the organization of the points. The default is XYZ; an X, Y, and Z for each point starting at parametric index 0. Possible organizations are:

- **XYZ** - Interlaced locations
- **XY** - Z is set to 0.0
- **X_Y_Z** - X,Y, and Z are separate arrays
- **VXVYVZ** - Three arrays, one for each axis
- **ORIGIN_DXDYDZ** - Six Values: Ox,Oy,Oz + Dx,Dy,Dz

The following Geometry element defines eight points:

```
<Geometry Type="XYZ">
        <DataItem Format="XML" Dimensions="2 4 3">
                0.0  0.0  0.0
                1.0  0.0  0.0
                1.0  1.0  0.0
                0.0  1.0  0.0

                0.0  0.0  2.0
                1.0  0.0  2.0
                1.0  1.0  2.0
                0.0  1.0  2.0
        </DataItem>
</Geometry>
```

Together with the Grid and Topology element we now have enough to make a full XDMF XML file that defines two quadrilaterals that share an edge (notice not all points are used):

```
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" []>

<Xdmf Version="2.0"
xmlns:xi="http://www.w3.org/2001/XInclude">
        <Domain>
                <Grid Name="Two Quads>
                        <Topology
                        Type="Quadrilateral"
                        NumberOfElements="2" >
                                <DataItem
```

```
                Format="XML"
                DataType="Int"
                Dimensions="2 4">
                        0 1 2 3
                        1 6 7 2
                </DataItem>
            </Topology>
            <Geometry Type="XYZ">
                <DataItem
                Format="XML"
                Dimensions="2 4 3">
                        0.0  0.0  0.0
                        1.0  0.0  0.0
                        1.0  1.0  0.0
                        0.0  1.0  0.0

                        0.0  0.0  2.0
                        1.0  0.0  2.0
                        1.0  1.0  2.0
                        0.0  1.0  2.0
                </DataItem>
            </Geometry>
        </Grid>
    </Domain>
</Xdmf>
```

4. Attribute

The Attribute element defines values associated with the mesh. Currently the supported types of values are:

- **Scalar**
- **Vector**
- **Tensor** – 9 values expected
- **Tensor6** – a symmetrical tensor
- **Matrix** - an arbitrary NxM matrix

These values can be centered on:

- **Node**
- **Edge**
- **Face**
- **Cell**
- **Grid**

A Grid centered Attribute might be something like "Material Type" where the value is constant everywhere in the grid. Edge and Face centered values are defined, but don't map well to many visualization systems. Typically Attributes are assigned on the Node:

```
<Attribute Name="Node Values" Center="Node">
        <DataItem Format="XML" Dimensions="6 4">
                100 200 300 400
                500 600 600 700
                800 900 1000 1100
                1200 1300 1400 1500
                1600 1700 1800 1900
                2000 2100 2200 2300
        </DataItem>
</Attribute>
```

Or assigned to the cell centers:

```
<Attribute Name="Cell Values" Center="Cell">
        <DataItem Format="XML" Dimensions="3">
                3000 2000 1000
        </DataItem>
</Attribute>
```

4. Information

There is regularly code or system specific information that needs to be stored with the data that does not map to the current data model. There is an *Information* element. This is intended for application specific information that can be ignored. A good example might be the bounds of a grid for use in visualization. Information elements have a Name and Value attribute. If Value is nonexistent the value is in the CDATA of the element:

```
<Information Name="XBounds" Value="0.0 10.0"/>
<Information Name="Bounds"> 0.0 10.0 100.0 110.0 200.0
210.0 </Information>
```

Several items can be addressed using the *Information* element like time, units, descriptions, etc. without polluting the XDMF schema. If some of these get used extensively they may be promoted to XDMF elements in the future.

References

1. Schoeder, Will, Ken Martin, and Bill Lorensen, *The Visualization Toolkit*, Third Edition, Kitware, Inc., 2002.

2. Keasler, J., "A New Vista in Scientific Data Management." Nuclear Explosives Code Developers' Conference, Lawrence Livermore National Laboratory, December 15, 2004.

3. Clarke, Jerry A. and Raju R. Namburu, "A distributed computing environment for interdisciplinary applications." *Concurrency and Computation: Practice and Experience*, Volume 14, Issue 13–15, pp. 1161–1174, Nov–Dec 2002.

4. Edwards, H.C. and J.R. Stewart, "SIERRA: a software environment for developing complex multiphysics applications." First MIT Conference on Computational Fluid and Solid Mechanics, Cambridge, MA, pp. 1147–1150, June 2001.

5. Clark, J. and S. DeRose, "XML Path Language (XPath) Version 1.0." W3C Recommendation, the World Wide Web Consortium, November 1999.

6. Marsh, J. and D. Orchard, "XML Inclusions (XInclude) Version 1.0." W3C Working Draft, 17-July-2000.