A Distributed Computing Environment for Interdisciplinary Applications

Jerry A. Clarke
US Army Research Laboratory
Aberdeen Proving Ground, MD
clarke@arl.army.mil

Raju R. Namburu
US Army Research Laboratory
Aberdeen Proving Ground, MD
raju@arl.army.mil

ABSTRACT

Practical applications are generally interdisciplinary in nature. The technology is well matured for addressing individual discipline applications and not for interdisciplinary applications. Hence, there is a need to couple the capabilities of several different computational disciplines to address these interdisciplinary practical applications. One approach is to use coupled or multi-physics software, which typically involves developing and validating the entire software spectrum for a specific application, which will be time consuming and may require more time to get to the end user. The other approach is to integrate individual well-matured computational technology discipline's software by taking advantage of the existing scalable software and validation investments, and tremendous developments in computer science and computational sciences. This integrated approach requires consistent data model, data format, data management, seamless data movement, and robust modular scalable including coupling algorithms. To address these requirements, we developed a new flexible data exchange mechanism for HPC codes and tools, known as the eXtensible Data Model and Format (***XDMF***). XDMF provides computational engines with the tools necessary to exist in a modern computing environment with minimal modification. Instead of imposing a new programming paradigm on HPC codes, XDMF uses the existing concept of file I/O for distributed coordination. XDMF incorporates Network Distributed Global Memory (***NDGM***), Hierarchical Data Format version 5 (***HDF5***), and eXtensible Markup Language (***XML***) to provide a flexible yet efficient data exchange mechanism. . This paper discusses development and implementation of distributed computing environment for interdisciplinary applications utilizing the concept of a common data hub. Also, the implementation of XDMF is demonstrated for a typical blast-structure interaction interdisciplinary application.

1.0 INTRODUCTION

Many challenging High Performance computing applications require the use of capabilities from separate computational disciplines. For example, simulating a blast interacting with a structure requires the integration of a Computational Fluid Dynamics (CFD) code with Computational Structural Mechanics (CSM) code, and Computational Chemistry and Material Science (CCM) code along with a mathematically consistent coupling algorithms at the interfaces. That is, an interface definition consistent with the numerical approaches used between each discipline's software, namely blast and structure in a mathematically consistent manner is required to address this application. On the other hand, multi-physics codes attempt to provide this functionality utilizing unified numerical algorithms or single software starting from partial differential equations. The computational approach discussed in this paper involves coupling existing validated individual codes from each discipline by taking advantage of tremendous developments in computer science, interface or coupling algorithms, and computational sciences. While individually impressive, these codes are typically not designed to be coupled. Additionally, attempting to modify the internal communications scheme of these scalable codes is not only difficult but could possibly affect their validity.

Existing systems like Globus[1] and Legion[2], CORBA[3], KeLP[4], and the Active Data Repository[5] provide data exchange mechanisms among their services. Systems like POLYLITH[6], Darwin[7], and Olan[8] provide a software module interconnection framework. While these systems have met with varying amounts of success, they are not currently sufficient for our purposes since implementation can require significant site wide coordination or significant modification to existing software. Emulating the concept of a common I/O in a parallel and efficient manner alleviates these problems. Individual codes can periodically read and write necessary data to a centrally accessible location in order to exchange necessary information. As long as individual codes can agree on a common data model and format, this approach can be extended to a wide variety of HPC codes. While not as efficient as single multi-physics coupling software, this approach has the potential to add significant functionality to current HPC software.

Hence, a common, active, data model and format suitable for the heavily used HPC codes is needed. HDF5 defines a feature rich data format for structured and unstructured datasets as well as groups of data. Hence, we opted for HDF5 for storage of enormous datasets. For description of the meaning of data or small amounts of data, we have chosen to use the eXtenisible Markup Language (XML). While HDF5 has an "attribute" facility for storing name=value pairs, the use of XML allows us to easily support other heavy data formats as necessary. Additionally, the enormous amount of software available to process XML makes it a natural candidate for storage and manipulation of data. Each individual discipline computational software is scalable. That is, each runs on a set of distributed processors. Hence, there is a need for field requests and data transfers while the software is being executed for these coupled applications. This requirement is fulfilled by Network Distributed Global Memory (NDGM) which provides a physically distributed, logically shared, unstructured memory buffer. Instead of handling the mapping and unmapping of memory pages automatically, NDGM is accessed through a subroutine interface. While less automatic, this allows applications to form a “cooperative shared memory” that is simple yet efficient. NDGM is a client-server layer that consists of multiple server processes and an Application Programmers Interface (API) for clients. Each server maintains a section of a virtual contiguous buffer and fields requests for data transfer and program synchronization. Clients use the API to transfer data in and out of the virtual buffer and to coordinate their activity.

Through the use of HDF5, XML, and NDGM we have defined the “eXtensible Data Model and Format “ (XDMF). It provides a level of abstraction for enormous distributed datasets. Computational codes, visualization, and user interface can all interface with the data in a well-defined method without severely limiting performance. As mentioned previously, XDMF is both a data model and format. It allows for a self-describing method of storing large data structures and the information necessary to tell how the data is to be used. This not only makes it possible to combine the capabilities of several codes, but also makes the development of reusable pre and post-processing tools possible.

The rest of the paper is organized as follows: first we discuss the main components of XDMF, namely, NDGM, XML, and HDF5. Next we discuss the data model and format through a simple example. Finally a simple numerical example utilizing the XDMF is provided before the conclusion.

2.0 eXtensible Data Model and Format (XDMF)

XDMF can best be described as an active data model and format. It is a self-describing data hub that can be dynamically updated by multiple clients. Data format (number type, array dimensions, etc.) is managed separately from data model (how data is to be used). This greatly enhances the flexibility of the system.

Data format refers to the raw data to be manipulated. Information like number type (float, integer, etc.), precision, location, rank, and dimensions completely describe any dataset regardless of its size. The description of the data is also separate from the values themselves. We refer to the description of the data as “Light” data and the values themselves as “Heavy” data. Light data is small and can be passed between modules easily. Heavy data may be potentially enormous; movement needs to be kept to a minimum. Due to the different nature of heavy and light data, they are stored using separate mechanisms. Light data is stored using XML, Heavy data is typically stored using HDF5. While we could have chosen to store the light data using HDF5 “attributes”, using XML does not require every tool to have access to the compiled HDF5 libraries in order to perform simple operations.

Data model refers to the intended use of the data. For example, a three dimensional array of floating point vales may be the X,Y,Z geometry for a grid or calculated vector values. Without a data model, it is impossible to tell the difference. Since the data model only describes the data, it is purely light data and thus stored using XML. It is targeted at scientific simulation data concentrating on scalars, vectors, and tensors defined on some type of computational grid. Structured and Unstructured grids are described via their topology and geometry. Calculated, time varying data values are described as “attributes” of the grid. The actual values for the grid geometry, connectivity, and attribute values are contained in the data format. This separation of data format and model allows HPC codes to efficiently produce and store vales in a convenient manner without being encumbered by our data model which may be different from their internal arrangement.

Utilizing the common data model and format, codes and tools can produce and consume data just as they would write and read any other data file. These data “files”, however, can exist in a distributed shared memory system (called NDGM) which has barriers and semaphores to help coordinate parallel activity. This is what makes XDMF more than just another file format.

2.1 Network Distributed Global Memory

At the heart of the data organization, is a unique heterogeneous shared memory system. Network Distributed Global Memory (NDGM) [9] provides ICE with a physically distributed, logically shared, unstructured memory buffer. But instead of handling the mapping and unmapping of memory pages automatically, NDGM is accessed through a subroutine interface. While less automatic, this allows applications to form a “cooperative shared memory” that is simple yet efficient.

NDGM is a client-server layer that consists of multiple server processes and an Application Programmers Interface (API) for clients. Each server maintains a section of a virtual contiguous buffer and fields requests for data transfer and program synchronization. Clients use the API to transfer data in and out of the virtual buffer and to coordinate their activity.

Calls to the API result in lower level messages being sent to the appropriate NDGM server that keeps track of its piece of the total virtual buffer. The API translates the global memory address into a local address that the server then transfers from its local memory.

Client programs use an API to access the virtual NDGM buffer as contiguous bytes. No structure is placed upon the NDGM buffer; the application can impose any structure on this buffer that is convenient. In addition, NDGM is designed to implement a system of applications in contrast to a single monolithic parallel application. The API includes facilities to get and put contiguous memory areas, get and put vectors of data, acquire and release semaphores, and to initialize and check into multiple barriers.

Each server maintains a local memory buffer that maps into the virtual buffer address space. This local buffer can be in one of three locations: local address space (obtained via malloc), system shared memory, or a local file. If system shared memory is used, a client executing on the same physical machine as the server accesses the shared memory instead of making requests to a server. This access is transparent to the NDGM client application and results in faster data transfers. Using a file as the server's local storage allows NDGM servers to restart with their local memory already initialized.

Clients and servers run on top of a layered message-passing interface. Similar in concept to well known message passing interfaces like PVM or MPI, this layer provides a level of abstraction, freeing the upper layers from the details of reading and writing data. The NDGM message-passing layer has fewer facilities than either PVM or MPI but is designed to pass NDGM data efficiently with minimal copying. This layer provides calls to establish connections, send messages, probe for incoming messages, read messages, and close connections.

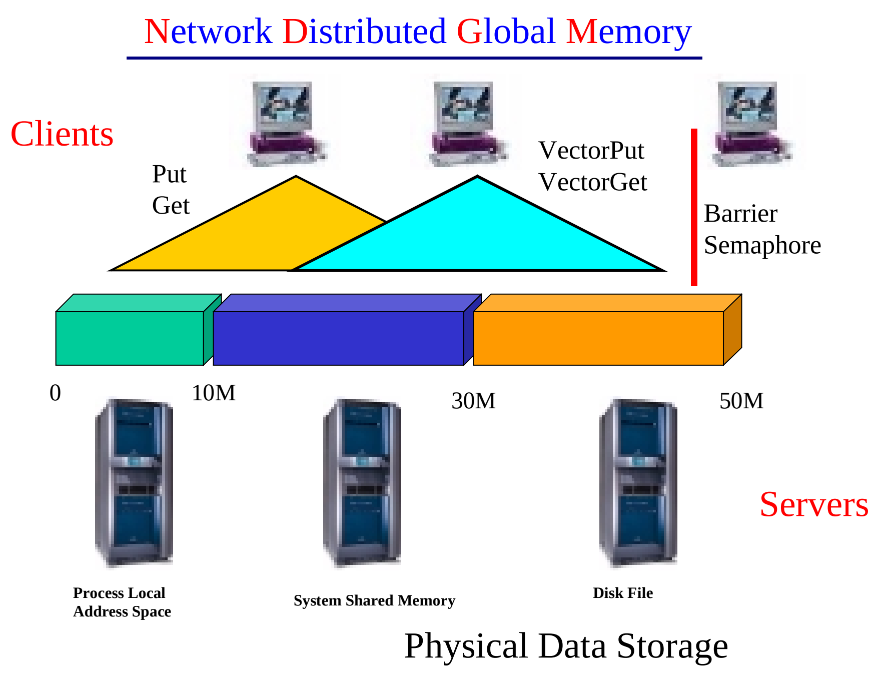

Figure 1. Emulating Shared Memory on Distributed Systems

NDGM has been used to develop parallel applications, but it is particularly useful as a “data rendezvous” for a collection of applications. A parallel, computationally intensive code can write a snapshot of data to NDGM then continue its’ processing. The data can then be visually inspected, through 2D plots and 3D surfaces, while not inhibiting the progress of the code.

2.2 Hierarchical Data Format

NDGM provides a distributed, heterogeneous unstructured buffer. To provide some structure to this buffer, ICE uses the “Hierarchical Data Format Version 5” (HDF5)[10] from the National Center for Supercomputing Applications (NCSA). HDF5, a well-known and widely used format, is designed to allow an orderly access to structured and unstructured datasets. All access is accomplished through a well-defined application programmer’s interface (API) .

The HDF5 API provides an efficient and powerful means of describing and accessing both data values and the associated meta-data (data about the data). HDF5 defines data types (chars, integers, and floats of various precisions and byte orders) and data space (rank and dimensions). There is a grouping facility that allows the construction of a directed acyclic graph description of the data arrangement and an attribute facility to store name=value pairs. In addition to allowing access to disk files, HDF5 provides a “Virtual File Layer”. This allows the user to add “drivers” for data access. We have written an HDF5 “driver” for NDGM that allows for NDGM access via the standard HDF5 API.

To simplify access to HDF5 files accessed via separate drivers, we have introduced the concept of “domains”. HDF5 uses filenames in several of its’ API routines. If these filenames are prepended by “NDGM:”, the HDF data refers to NDGM. For completeness, prepending “FILE:” to the filename refers to a disk file. If the domain is omitted, a disk file is assumed. This new HDF driver allows a single application to store data on disk, in distributed memory, or in both.

2.3 eXtensible Markup Language

HDF5 defines a feature rich data format for structured and unstructured datasets as well as groups of data. We primarily use HDF5 for storage of enormous datasets or “Heavy” data. For description of the meaning of data or small amounts of data, we have chosen to use the eXtenisible Markup Language (XML). While HDF5 has an “attribute” facility for storing name=value pairs, the use of XML allows us to easily support other heavy data formats as necessary. Additionally, the enormous amount of software available to process XML makes it a natural candidate for storage of this “Light” data.

XML is human readable text originally intended for use on the Web. XML looks like HTML but allows the definition of the *tags* and *attributes*. XML can be stored in a file or transmitted like any other character string. Through the use of freely available software, XML can be “parsed” into an in-memory tree structure [11]. Tree nodes can be added, modified, or deleted. The resulting tree can then be “serialized” back into an XML string. Since XML is just text, any program can produce it without the need for special libraries.

The data model in XDMF is stored in XML. This provides the knowledge of what is represented by the Heavy data. In this model, HPC data is viewed as a hierarchy of ***Domains***. A Domain may contain one or more sub-domains but must contain at least one ***Grid***. A Grid is the basic representation of both the geometric and computed/measured values. A Grid is considered to be a group of elements with homogeneous ***Topology*** and their associated values. If there is more than one type of topology, they are represented in separate Grids. In addition to the topology of the Grid, ***Geometry,*** specifying the X, Y, and Z positions of the Grid is required. Finally, a Grid may have one or more ***Attributes***.

Attributes are used to store any other value associated with the grid and may be referenced to the Grid or to individual cells that comprise the Grid.

The XML may be passed as an argument, stored in an external file or communicated via a socket mechanism. For customization purposes, tools can also augment the standard content with XML "processing instructions". This is useful for attaching peer level information to the standard XML content without modifying the base specification.

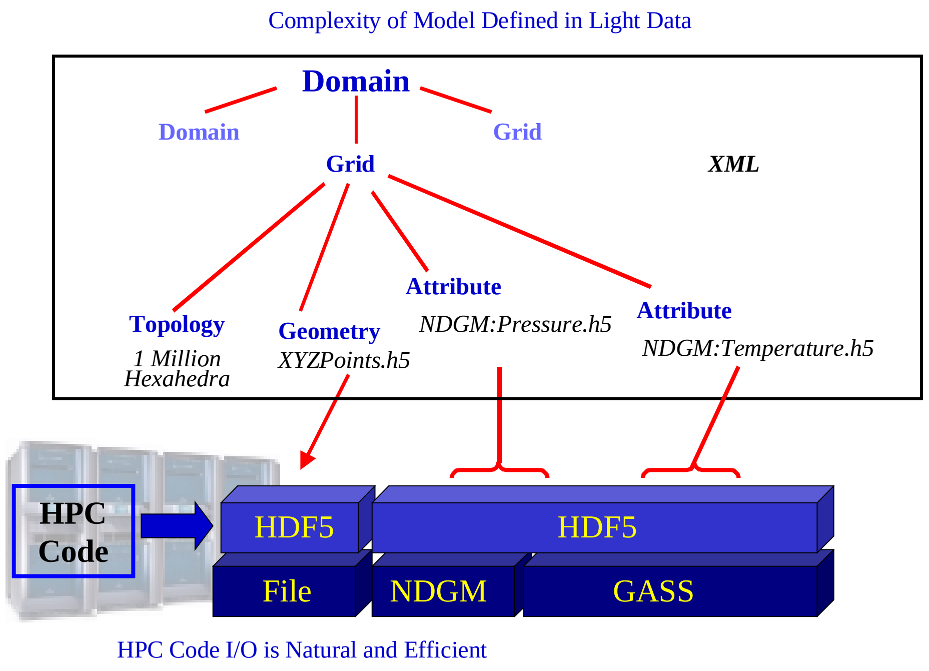

Figure 2. Separating Light and heavy data

The concept of separating the light data from the heavy data, as shown in Figure 2, is critical to the performance of this data model and format. HPC codes can read and write data in large, contiguous chunks that are natural to their internal data storage, to achieve optimal I/O performance. If codes were required to significantly re-arrange data prior to I/O operations, data locality, and thus performance, could be adversely affected, particularly on codes that attempt to make maximum use of memory cache. The complexity of the dataset is described in the light data portion, which is small and transportable. For example, the light data might specify a topology of one million hexaherda while the heavy data would contain the geometric XYZ values of the mesh and pressure values at the cell centers stored in large, contiguous arrays. This key feature will allow reusable tools to be built that do not put onerous requirements on HPC codes. Despite the complexity of the organization described in the XML below, the HPC code only needs to produce the three HDF5 datasets for geometry, connectivity, and pressure values.

Through the use of NDGM, HDF, and XML we have defined the "eXtensible Data Model and Format " (XDMF). It provides a level of abstraction for enormous distributed datasets. Computational codes, visualization, and user interface can all interface with the data in a well-defined method without severely limiting performance. As mentioned previously, XDMF is both a data model and format. It allows for a self-describing method of storing large data structures and the information necessary to tell how the data is to be used. We provide a C++ class library mainly as a convenience layer. Codes and tools use this layer from C++, C, or FORTRAN to easily access any XDMF functionality. In addition, this layer has been "wrapped" for access from Tcl, Python, and Java.

3.0 Accessing eXtensible Data Model and Format (XDMF)

The computationally intensive components of a large system are generally developed using system-programming languages like C, C++, or FORTRAN. Once these computationally intensive components have been developed however, they may be "glued" together in a number of ways to provide the overall required functionality. Using a system-programming language for this task is tedious, time-consuming, and inflexible.

Scripting languages are specifically designed for this purpose. They tend to be "weakly-typed" so that the output of one component can easily be used as the input to another with little concern over the "type" of the data. While one pays in runtime efficiency for this flexibility, scripting languages are intended to call large chunks of functionality and not be used for fine grain control.

Languages like Java, Python and Tcl, are supported on a variety of platforms and can be used to glue together existing programs. One can also extend the functionality of any of these scripting languages by adding additional commands. In addition to developing callable functions in the language itself, this is accomplished by adding a "wrapper" to interface the language command-calling interface with the argument list of a system-programming routines that implements the desired functionality efficiently. "*The Simplified Wrapper and Interface Generator*" (**SWIG**) is a widely used tool that generates these wrappers for code written in C or C++ for access via languages like Tcl, Python, and Java. SWIG allows commands to be added to these languages that access the custom code in an object-oriented fashion.

While significantly different in syntax, most scripting languages are similar in functionality. The choice of a scripting language can regularly be a matter of personal preference. XDMF has been wrapped to provide interfaces to Java, Tcl, and Python. Access to FORTRAN and 'C' is accomplished via a 'C' interface to C++. This flexibility is key to providing a useful data exchange mechanism to both HPC codes which are written in system programming languages, and analysis tools which may be written in scripting languages. While HPC codes can access XDMF via system programming languages, custom tools can be written in high level languages that maintain much of the performance of the system programming language components.

Scientific Visualization is a critical component of ICE. Both runtime and post-processing visualization are possible when using XDMF as the data hub. Because of the various capabilities of different visualization packages, we support both commercial and Open Source products.

The commercial product EnSight from CEI is supported via a “data reader”. EnSight loads the reader at runtime from a shared object that can then convert and external XDMF dataset to EnSight’s internal representation. EnSight is a full-featured product and is used extensively for presentation quality post-processing scientific visualization.

OpenDX is the open source software project based on IBM’s Visualization Data Explorer. OpenDX differs significantly from EnSight in that the user connects visualization “modules” via a data flow diagram. This makes OpenDX more configurable than EnSight at the cost of having a more complex user interface. XDMF data is converted to OpenDX format during runtime or during post processing. Users of OpenDX can build custom interfaces for XDMF data while hiding the complexities of the data flow.

The Visualization Toolkit (vtk) from Kitware is another open source visualization software product. It is a widely used C++ class library of visualization and graphics algorithms that includes interfaces for Tcl, Python, and Java. XDMF supports vtk natively, that is the XDMF grid class has a method that returns a native vtk grid. Methods also exist that assign scalars and vectors to the vtk grids. This interface allows for the development of custom visualization tools via a programming interface.

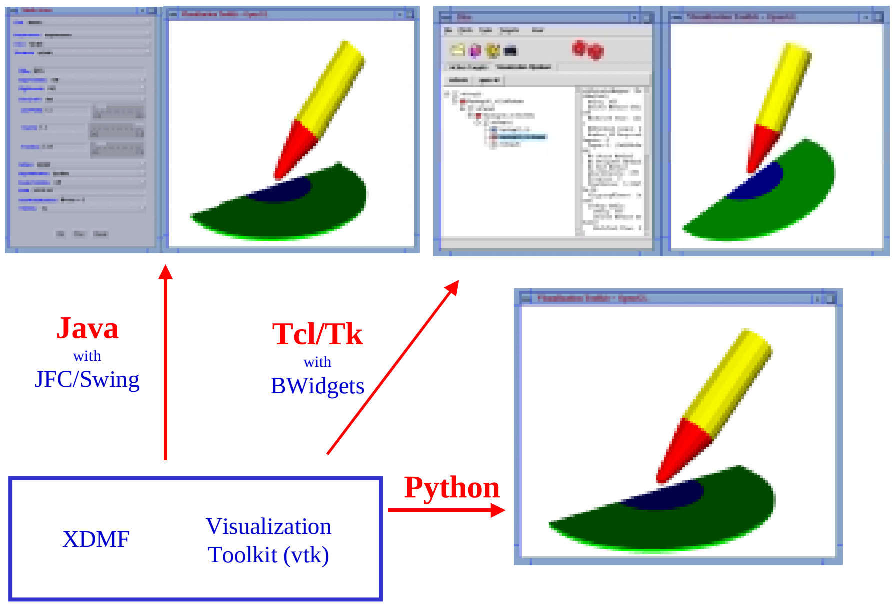

Figure 3. Data Access from High Level Languages

The ability to quickly prototype custom applications and tools in higher level languages allows for the rapid development of task-specific applications. By utilizing vtk, these applications can include visualization of enormous amounts of data in addition to graphical user interfaces.

4.0 Test Case

Naturally, the flexibility and functionality of the system sacrifice some performance when compared to a "hard-wired" solution. A balance between functionality and performance must be reached that allows for reusable tools that perform their function with acceptable efficiency. In addition, to be truly useful, existing HPC programs must be able to take advantage of the system without overly burdensome modifications.

To gauge the usefulness of the system we chose to attempt a one-way coupling of two heavily used HPC codes to solve a relevant computational problem. An explosive charge was simulated using the CTH code from Sandia National Laboratory[12]. A solid steel wall was placed in the simulation that was "loaded" by the explosive charge. Next, a concrete block wall was simulated using the ParaDyn program from Lawrence Livermore National Laboratory[13]. Utilizing XDMF, the loading of the wall was transferred from the structured CTH domain to the unstructured ParaDyn domain via a third "interpolator" process. Since this problem has already been accomplished using CTH "tracer" points and disk files[14], it gave us a method of validating the results of our new approach.

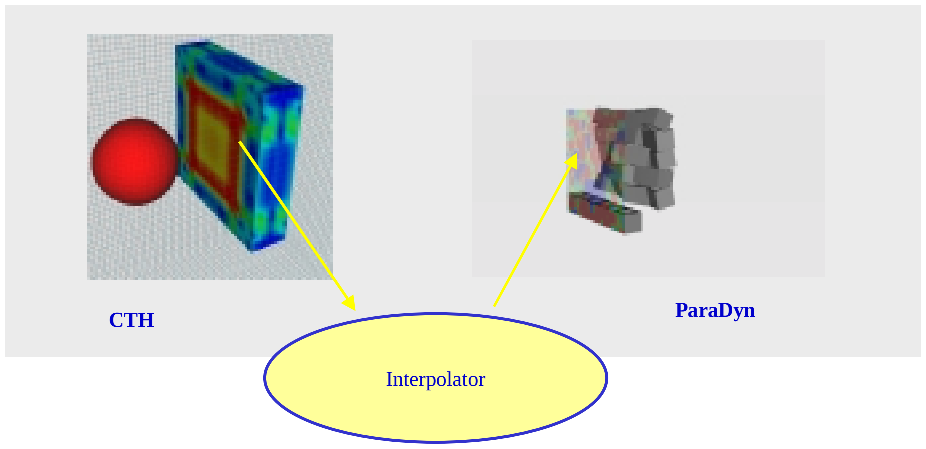

Figure 4. Coupling CTH and ParaDyn with XDMF

ParaDyn, from Lawrence Livermore National Laboratory, is a parallel version of the widely used, finite element based structural dynamics program ***Dyna3D*** . ParaDyn, like many currently used HPC codes, is written primarily in FORTRAN and uses MPI to achieve parallelism. We felt that adding runtime visualization capability to ParaDyn, via ICE, would demonstrate the steps required to integrate existing components and also result in a useful distributed application at ARL.

ParaDyn, like many HPC simulation codes, follows this basic execution:

- Read in computational grid and input parameters from the file system
- Initialize internal variables
- Iterate over the core physics routines of the code until final solution is reached, periodically writing intermediate solutions to the file system
- Write final solution, cleanup, and exit

The additional ICE calls map well into this execution flow. Since the code is mainly FORTRAN and ICE access is accomplished via C++, FORTAN wrapper functions are needed to encapsulate the required ICE functionality.

For example, we write a new FORTRAN subroutine PARAINITICE( ), called when ParaDyn initializes its internal variables, to initialize the necessary ICE C++ objects and store their addresses in static variables. When the nodes need to update XDMF, they have access to the appropriate C++ objects.

The internal structure of ParaDyn is complex enough to place it beyond the scope of our discussion. Suffice it to say that internal variables are accessed through an internal database API. For simplicity, let us assume that there exists such FORTRAN subroutines as PDGETXYZ(), PDGETNODEVAR(), and PDGETCELLVAR() to return XYZ location and scalar values. We add a subroutine call to the main ParaDyn loop to call a new PARACHECKICE() subroutine every iteration. This is where the majority of the ICE functionality is accessed.

The concrete wall in ParaDyn consisted of a computational grid of about 110,000 hexahedral elements. The following XDMF XML file describes the computational grid and a cell centered scalar. This is used as input to a visualization program or some other post-processing analysis tool.

```
<XDMF>
        <Domain>
                <Grid
                        Name=”Concrete Wall”>
                        <Topology
                                Type=”Hexahrdron”
                                NumberOfElements=”110520”>
                                <DataStructure
                                        Dimensions=”110520 8”
                                        Format=”HDF”
                                        DataType=”Int”
                                        Precision=”4”>
                                NDGM:Wall.h5:/Initial/Connections
                                </DataStructure>
                        </Topology>
                        <Geometry
                                Type=”XYZ” >
                                <DataStructure
                                        Dimensions=”179685 3”
                                        Format=”HDF”
                                        DataType=”Float”
                                        Precison=”8”>
                                NDGM:Wall.h5:/Initial/XYZ
                                </DataStructure>
                        </Geometry>
                        <Attribute
                                Name=”Effective Plastic Strain”
                                Type=”Scalar”
                                Center=”Cell”>
                                <DataStructure
                                        Dimensions=”110520”
                                        Format=”HDF”
                                        DataType=”Float”
                                        Precision=”8”>
                                NDGM:Wall.h5:/Results/EffPlaStr
                                </DataStructure>
                        </Attribute>
                </Grid>
        </Domain>
</XDMF>
```

The same overall strategy was used to outfit CTH to update XDMF. But since CTH uses structured, rectilinear grids, the updates tend to be more efficient due to the contiguous nature of the date. Since this is a one-way coupling, only CTH need to write information to NDGM in order to accomplish the simulation. For visualization purposes, however, both CTH and ParaDyn periodically output data. The visualization below shows the initial loading of the wall represented by the colors on a mesh at the original concrete block locations. At that point in the simulation, the initial loading, gravity, and contact with other blocks effect the block’s displacement. The lowest row of concrete block is fixed to the ground and not allowed to move. The simulation is small enough (about 110,000 hexahedral elements) to complete in a reasonable amount of time for benchmarking purposes. In fact, the relatively small amount of data being sent to the NDGM buffer and the noncontiguous nature of the unstructured mesh magnifies communication latency effects. In addition, in order to get a “worst case” idea of the overhead involved, we ran the problem on the IBM NH-2 (375 MHz Power3, the jobs were submitted in a manner to distribute the work across different SMP nodes), with the

NDGM buffer on one node. In this scenario, all of the MPI nodes must funnel their runtime data to one designated "collection" node that holds the HDF5 data.

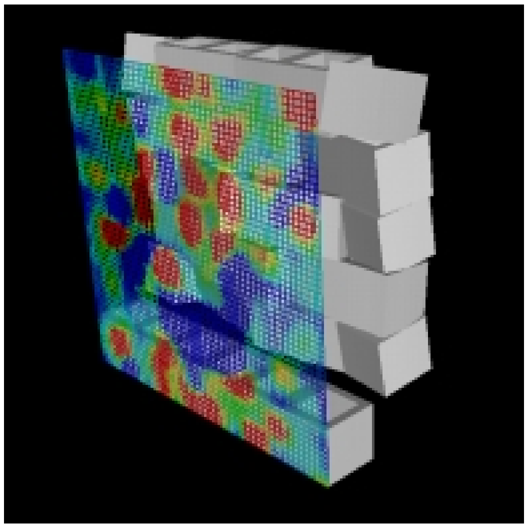

Figure 5. Loading of Concrete Wall from CTH

Primarily due to the noncontiguous nature of the finite element data, ParaDyn on the IBM sees only about 12MB/sec effective throughput. Effective throughput is being defined as the total amount of data being transferred over the entire run divided by the total runtime with updates minus the total runtime without updates. To give an idea of the other end of the performance spectrum, CTH on an Origin 2000 can see over 200MB/sec effective throughput since large chunks of contiguous data is being transferred via system shared memory.

With both ParaDyn and CTH, most of the interface code deals with accessing the internal database APIs. To provide a more straightforward example, we provide a complete code on our WEB site (http://www.arl.hpc.mil/ice) that has pre-processing, computation, and file I/O confined to a single FORTRAN source file. The code needed to interface this code to ICE is then added in the previously mentioned fashion. This interface is about 150 lines of C++ code. Much of this interface is reusable for other applications; the main difference is the access of the HPC code's internal variables. This may result in a new convenience object to encapsulate the functionality thus reducing many interfaces to significantly less code.

5.0 Conclusion

We developed a new eXtensible Data Model and Format for addressing interdisciplinary applications. In addition to providing a common data model and format that allows for the development of powerful and reusable tools, XDMF allows efficient active updates to facilitate runtime visualization and HPC code coupling. The applicability of the approach is demonstrated for an interdisciplinary blast-structure interaction application. For this demonstration purpose, the coupled application used widely accepted individual discipline software, namely, CTH for blast simulations and ParaDyn for structural simulations. The approach is promising for a wide class of applications and we are currently extending this approach for a different set of software. In addition, we are developing a suite of reusable tools written in Java and Python to provide a complete environment for the computational scientist. This ICE will provide a modern environment for existing and future numerical simulations.

6.0 ACKNOWLEDGEMENTS

This work is part of the DOD High-performance Computing Modernization program Common High-performance computing Scalable Software Initiative (CHSSI) project. Use of SGI O3K and IBM SP at Army Research Laboratory Major Shared Resource Center is acknowledged.

7.0 References

1. Foster, I., Antonio, J., “The Globus project: a status report”, proceedings of the Seventh heterogeneous Computing workshop, pp 4-18, march 1998

2. Grimshaw, A., Ferrari, A., Knabe, F., Humphrey, M., “Wide area computing: resource sharing on a large scale”, Computer , volume 32, issue 5, pp 29-37, May 1999

3. Object Management Group, “The Common Object Request Broker: Architecture and Specification”, num. 91.12.1, December 1991

4. Baden, S.B., Fink, S.J., “A programming methodology for dual-tier multicomputers “, IEEE Transactions on software engineering, Volume 26, Issue 3, pp 212-226, March 2000

5. Chialin Chang, Kurc, T., Sussman, A., Saltz, J., “ Optimizing retrieval and processing of multi-dimensional scientific datasets”, Proceedings of 14th international Parallel and Distributed Symposium, 2000. pp. 405-410, May 2000

6. Purtilo, J.M., “The POLYLITH Software Bus”, ACM TOPLAS, Vol 16, Number 1, pp 151-174, Jan. 1994

7. Magee, J., Dulay, N., Kramer, J., “A Constructive Development Environment for Parallel and Distributed programs”, Proceedings of the International Workshop on Configurable Distributed Systems, Pittsburgh, March 1994

8. Bellissard L., Boyer, F., Riveill, M., Vion-Dury, J., “System Services for Distributed Application Configuration”, Proceedings of Fourth international conference on Configurable Distributed Systems. Pp 53-60, May 1998

9. Clarke J., “Emulating Shared Memory to Simplify Distributed-Memory Programming”, IEEE Computational Science & Engineering, Vol 4, No. 1, pp 55-62, January-March 1997

10. Folk, M., McGrath, R., Yeager, N., “HDF: an update and future directions”, Proceedings of IEEE 1999 international Geoscience and Remote Sensing Symposium, Volume 1, pp 273-275, july 1999

11. “Document Object Model (DOM) Level 1 Specification.”, World wide Web Consortium, http://www/w3.org/TR/REC-DOM-Level-1

12. McGlaun, J.M., Thompson, S.L., and Elrick, M.G., "CTH: A Three-Dimensional Shock Wave Physics Code," *International Journal of Impact Engineering*, Vol. 10, pp. 351-360, 1990.

13. Hoover, C.G., DeGroot, A.J., Maltby, J.D., and Procassini, R.J., "*ParaDyn: DYNA3D for Massively Parallel Computers*," UCRL 53868-94, Lawrence Livermore National Laboratory, Livermore, CA, 1995.

14. Raju R. Namburu, Jimmy O. Balsara, Tommy L. Bevins, and Photios P. Papados, “Large-Scale Explicit Simulations on Scalable Computers,” Advances in Engineering Software, Vol. 29, pp. 187-196, 1998.