ANL/MCS-TM-357

SIGMA release v1.2 – Capabilities, enhancements and fixes

Mathematics and Computer Science Division

About Argonne National Laboratory
Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via DOE's SciTech Connect (http://www.osti.gov/scitech/)

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: **orders@ntis.gov**

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: **reports@osti.gov**

Disclaimer
This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

ANL/MCS-TM-357

SIGMA release v1.2 – Capabilities, enhancements and fixes

Vijay Mahadevan, Iulian Grindeanu, Navamita Ray, Rajeev Jain, Danqing Wu

Mathematics and Computer Science Division,
Argonne National Laboratory

September 30, 2015

EXECUTIVE SUMMARY

Computational physics solvers take a continuous domain of interest, the problem geometry, and apply specialized discretizations based on the Partial Differential Equation (PDE) describing the relevant physical model. The discretization involves two components: computing a discrete mesh of the domain that accurately models the continuous geometry and defining the solution fields as degrees-of-freedom on the mesh, that needs to be computed by appropriately solving the discretized operators. The resulting single/multi component physics solutions need to be serialized back to a format amenable for visualization.

The Scalable Interfaces for Geometry and Mesh based Applications (SIGMA) toolkit provides interfaces and tools to access geometry data, create high quality unstructured meshes along with unified data-structures to load and manipulate parallel computational meshes for various applications to enable efficient physics solver implementations. Mesh generation is a complex problem since most problem geometries involve complicated curved surfaces that require physics imposed spatial resolution and optimized elements for good quality. These tools simplify the process of generation and handling of discrete meshes with scalable algorithms to leverage efficient usage from desktop to petascale architectures.

SIGMA contains several components including CGM for the geometry handling, MOAB to represent the unstructured mesh data-structure, MeshKit containing several mesh generation algorithms and CouPE for driving coupled multi-physics solvers. These components are used tightly within the SHARP toolkit that is part of the Integrated Product Line for fast reactor simulations. The goal of the SHARP framework is to perform fully resolved coupled physics analysis of a reactor on heterogeneous geometry, in order to reduce the overall numerical uncertainty while leveraging available computational resources.

In this report, we present details on SIGMA toolkit along with its component structure, capabilities, and feature additions in FY15, release cycles, and continuous integration process. These software processes along with updated documentation are imperative to successfully integrate and utilize in several applications including the SHARP coupled analysis toolkit for reactor core systems funded under the NEAMS DOE-NE program.

TABLE OF CONTENTS

EXECUTIVE SUMMARY i
Table of Contents ii
List of Figures iv
List of Tables iv
1 Introduction 5
2 SIGMA toolkit 6
3 Verdict mesh-quality library integration 8
3.1 Background 9
3.2 Description 9
3.3 Current Uses within SIGMA and Future Plans 14
4 Parallel hierarchical mesh generation capability 14
5 Analysis of the OBB tree implementation in MOAB 19
6 Conformal local Hexahedral refinement 21
7 Integration of Lasso component to MOAB 22
8 iMOAB – A language agnostic MOAB interface 23
8.1 iMOAB: In-built types 24
8.2 iMOAB: Enumeration types 24
8.3 iMOAB: Function specifications 24
8.3.1 iMOAB_Initialize 24
8.3.2 iMOAB_InitializeFortran 25
8.3.3 iMOAB_Finalize 25
8.3.4 iMOAB_RegisterApplication 25
8.3.5 iMOAB_DeregisterApplication 26
8.3.6 iMOAB_ReadHeaderInfo 26
8.3.7 iMOAB_LoadMesh 27
8.3.8 iMOAB_WriteMesh 27
8.3.9 iMOAB_GetMeshInfo 28
8.3.10 iMOAB_GetVertexID 29
8.3.11 iMOAB_GetVertexOwnership 29
8.3.12 iMOAB_GetVisibleVerticesCoordinates 30
8.3.13 iMOAB_GetBlockInfo 30
8.3.14 iMOAB_GetBlockID 31
8.3.15 iMOAB_GetVisibleElementsInfo 31
8.3.16 iMOAB_GetBlockElementConnectivities 32
8.3.17 iMOAB_GetElementID 32
8.3.18 iMOAB_GetPointerToSurfaceBC (Neumann B.C) 33
8.3.19 iMOAB_GetPointerToVertexBC (Dirichlet B.C) 34
8.3.20 iMOAB_DefineTagStorage 34

8.3.21 iMOAB_GetDoubleTagStorage 35
8.3.22 iMOAB_SetDoubleTagStorage 36
8.3.23 iMOAB_SynchronizeTags 36
8.4 Fortran and C drivers 37

9 Software infrastructure and collaborative development 37

9.1 SIGMA software processes 37
9.1.1 Build System 37
9.1.2 Documentation updates 38
9.1.3 Continuous integration 38
9.2 Release schedules 40
9.2.1 SIGMA v1.1 Release and notes (Jan 2015) 40
9.2.2 SIGMA v1.2 Release and notes (October 2015) 40
9.2.3 SIGMA v1.2 Release and notes (October 2015) 40
9.3 ATPESC participation (August 2015) 40

10 Publications and talks (FY15) 41

11 Summary 42

Acknowledgments 43

REFERENCES 43

LIST OF FIGURES

Figure 1. Traditional computational workflow 5
Figure 2. SIGMA toolkit 8
Figure 3. The scaled Jacobian metric for elements in sahex mesh. 13
Figure 4. Condition number of elements in sahex mesh. 14
Figure 5. The refinement templates defined for degree 2 and 3 refinements over a reference triangle and quadrilateral entity. The local vertex order of the reference entities follow MOAB conventions[18]. The new vertices to be introduced during refinement are assigned local ids wrt to the reference. Similarly, the children entities are assigned local order. The half-facet maps are created for the refined entities in terms of their local ids. 15
Figure 6. Standard template refinement patters for simple geometries. 16
Figure 7. Updating the AHF maps for the refined mesh takes place in two stages. First, maps for the children of each triangle are updated. After, both the triangles have been refined, the maps are now updated to connect the children entities sharing a parent facet. 17
Figure 8. Scalability studies for the mesh hierarchy generation algorithm. The top figure shows the initial coarse hexahedral mesh of a single assembly in a reactor core with fuel pins and central control rod with 32 partitions. The bottom figure shows the serial O(n) refinement performance of the UMR algorithm. 18
Figure 9. Parallel efficiency of the hierarchical mesh generation algorithm (Blues ALCF machine). 19
Figure 10. Hierarchical Oriented Bounding Box (h-OBB) (a) Leaf-level OBBs layed on top of the previous mesh (b) Artificially generated high-valence mesh replacing the surface of a cube. 20
Figure 11. Plot of ray fire times (averaged over 100k rays) various valencies and area with respect to the total surface size for the high-valence region. 20
Figure 12. Plot of ray fire times (averaged over 100k rays) various valencies and area with respect to the total surface size for the high-valence region. Here, the OBB splitting parameter is adjusted to better handle stark contrast of the high-valence regions. 21
Figure 14. SIGMA Buildbot test suite pages (a) The main buildbot web status page (b) Sample waterfall page with pending builds. 39

LIST OF TABLES

Table 1. *Output from mbquality tool for sahex mesh* file (`sahex1_unic_128p.h5m`) 12

1 Introduction

Computational modeling and simulation of complex physical phenomena presents several challenges and it is imperative to converge on scalable, flexible workflows to improve scientific productivity. Typically, computational physics solvers take a continuous domain of interest, describe the detailed problem geometry, and apply specialized discretizations on a discrete mesh to solve multiple Partial Differential Equation (PDE) systems to model interactions between the physical scales of interest. The spatial discretization alone involves multiple components: capturing the geometry complexity accurately, generating a discrete mesh representation of the domain so as not to introduce geometry errors and defining the solution fields as degrees-of-freedom (DoFs) on the mesh, which needs to be computed by appropriately solving the discretized operators. The resulting single/multi component physics solutions are usually serialized back to a format amenable for post-processing and visualization as shown in Fig. 1.

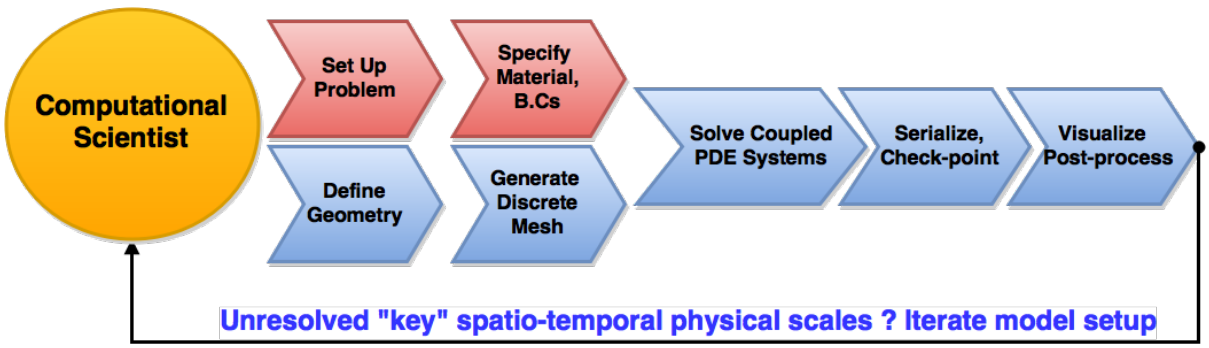


Figure 1. Traditional computational workflow

Additional difficulties in performing predictable simulations require quantification of numerical errors including those that are introduced due to geometry modeling and mesh resolution errors. Such intensive modeling needs require automated computational tools to simplify the iterative refinement and optimization in the workflow for complex problems that typically take several modifications to get the right problem configurations when matching experimental setups.

In this report, we present details on the Scalable Interfaces for Geometry and Mesh based Applications (SIGMA) toolkit [1] along with its component structure, capabilities, and feature additions in FY15, release cycles, and continuous integration process. These software processes along with updated documentation are imperative to successfully integrate and utilize in several applications including the SHARP coupled analysis toolkit for reactor core systems funded under the NEAMS DOE-NE program.

The organization of the report is as follows: In §2, the background on the SIGMA toolkit and its associated components are proved followed by §3 where details on the new mesh quality quantification library integration into MOAB is detailed. In section §4, the major research and development of a parallel uniform mesh refinement hierarchy

methodology along with results are provided in addition to potential implications to applications in the SHARP toolkit. Next, the design of a new language-agnostic mesh API is described that can potentially replace the existing ITAPS based wrappers in the SHARP physics modules that have proven to be cumbersome to maintain and extend. Finally, the software engineering processes used in SIGMA are given in order to verify usability and to enhance confidence in the overall SHARP coupled multi-physics simulation toolkit.

2 SIGMA toolkit

SIGMA provides interfaces and tools to access geometry data, create high quality unstructured meshes along with unified data-structures to load and manipulate parallel computational meshes for various applications to enable efficient physics solver implementations. Mesh generation is a complex problem since most problem geometries involve complicated curved surfaces that require physics imposed spatial resolution and optimized elements for good quality. These tools simplify the process of generation and handling of discrete meshes with scalable algorithms to leverage efficient usage from desktop to petascale architectures.

The SIGMA toolkit has been designed with the aim to enable and expedite modeling scientific problems by computational scientists and converge on simulation workflow without investing significant effort in learning an array of tools. It provides interfaces and tools to understand geometry models, create high quality unstructured meshes along with unified data-structures to load and manipulate parallel computational meshes for various applications to enable efficient physics solver implementations. Additionally, optimal quality mesh generation is a complex process for complicated curved problem geometries, with physics imposed spatial resolution requirements. These tools simplify the process of generation and handling of discrete meshes with scalable algorithms to leverage efficient usage from desktop to petascale architectures. The interaction of the various SIGMA components along with several key dependent applications and tools are shown in Fig. 2.

CGM: The Common Geometry Module (CGM) [2], is a library with a common API for querying and modifying the topological model for geometry represented in `ACIS` and `OpenCASCADE` solid modeling engines. CGM (`ITAPS iGeom`) also provides the common geometry infrastructure for the CUBIT mesh generation toolkit [3].

In collaboration with researchers in Sandia and our contributors in University of Wisconsin, over the past several years, we have been maintaining compatibility with the CUBIT releases. Currently, Cubit v14.0 support has been verified in CGM and it is expected that Cubit 14.9/15.0 support will be available in the coming months for both mesh generation and geometry model creation for use in nuclear engineering and fusion problems.

MeshKit: One of the unique capabilities provided by SIGMA toolkit is the inclusion of an array of generic mesh generation algorithms that seamlessly integrates into several different external applications. MeshKit serves as a library of advanced mesh-generation

algorithms implemented natively and provides an extensible design to perform R&D on new meshing techniques.

The MeshKit library [4, 5] contains several mesh-generation algorithms that utilize CGM and MOAB descriptions of geometry and mesh to discretize complex geometric domains. Nuclear reactor specific geometries can be easily meshed through a GUI tool from Kitware built on top of Reactor Geometry Generator written in MeshKit whose overall goal is to develop a complete package for generating reactor core models.

MOAB: The Mesh-Oriented datABase (MOAB) [6] (`ITAPS: iMesh, iMeshP`), is a library for uniformly representing both structured and unstructured meshes, along with associated field data defined on mesh entities, using an efficient array-based data model. MOAB provides query, construction, and modification of finite-element meshes, plus polygons and polyhedra. Various options are available for writing and visualizing the final meshes produced by meshing algorithms. MOAB uses an HDF5-based file format, which can be visualized by using a ParaView plugin that is implemented by the MOAB library. The Visit visualization tool can also be configured and built with MOAB to provide a similar import capability.

MOAB has shown demonstrated scalability to at least 512K processors [7] using its efficient parallel mesh handling algorithms, which includes arbitrary point location, and interpolation of field data during multi-mesh solution transfers.

The relational mapping between MOAB and CGM is also implemented in Lasso (`ITAPS iRel`), which is utilized for mesh generation and adaptive mesh refinements, where associations between mesh and geometry must be queried and maintained. Recently, in a move to reduce the dependency chain for SIGMA, a decision was taken to integrate the Lasso source base into MOAB. The description of this effort is provided in one of the subsections.

Solver Interoperability: High-resolution computational physics solvers require assembling the discrete operators on unstructured meshes and strong software coupling between scalable solvers in PETSc and MOAB provide abstractions (**DMMoab**) to manage field data, DoFs and traversal semantics to fully enable efficient solution techniques. In coupled phenomena, aggregating the interaction between such solvers to capture the physical behavior during solution evolution of dependent models require consistent, global nonlinear solvers with advanced block-preconditioning strategies (in addition to robust multi-mesh transfer projection algorithms) [8]. These algorithms have been implemented within the Coupled Physics Environment (**CouPE**) [9] that utilizes the software abstractions exposed through DMMoab and native MOAB-based wrappers.

CouPE is implemented based on MOAB and makes use of PETSc solvers to drive the global nonlinear problem to convergence and provides a flexible infrastructure to abstract coupling needs without intrusively modifying existing physics solvers. Several real-world problems in nuclear reactor analysis [10] have been analyzed using this framework to

accurately converge to coupled solutions of interest while quantifying numerical errors introduced due to both coupling and discretization using SIGMA tools.

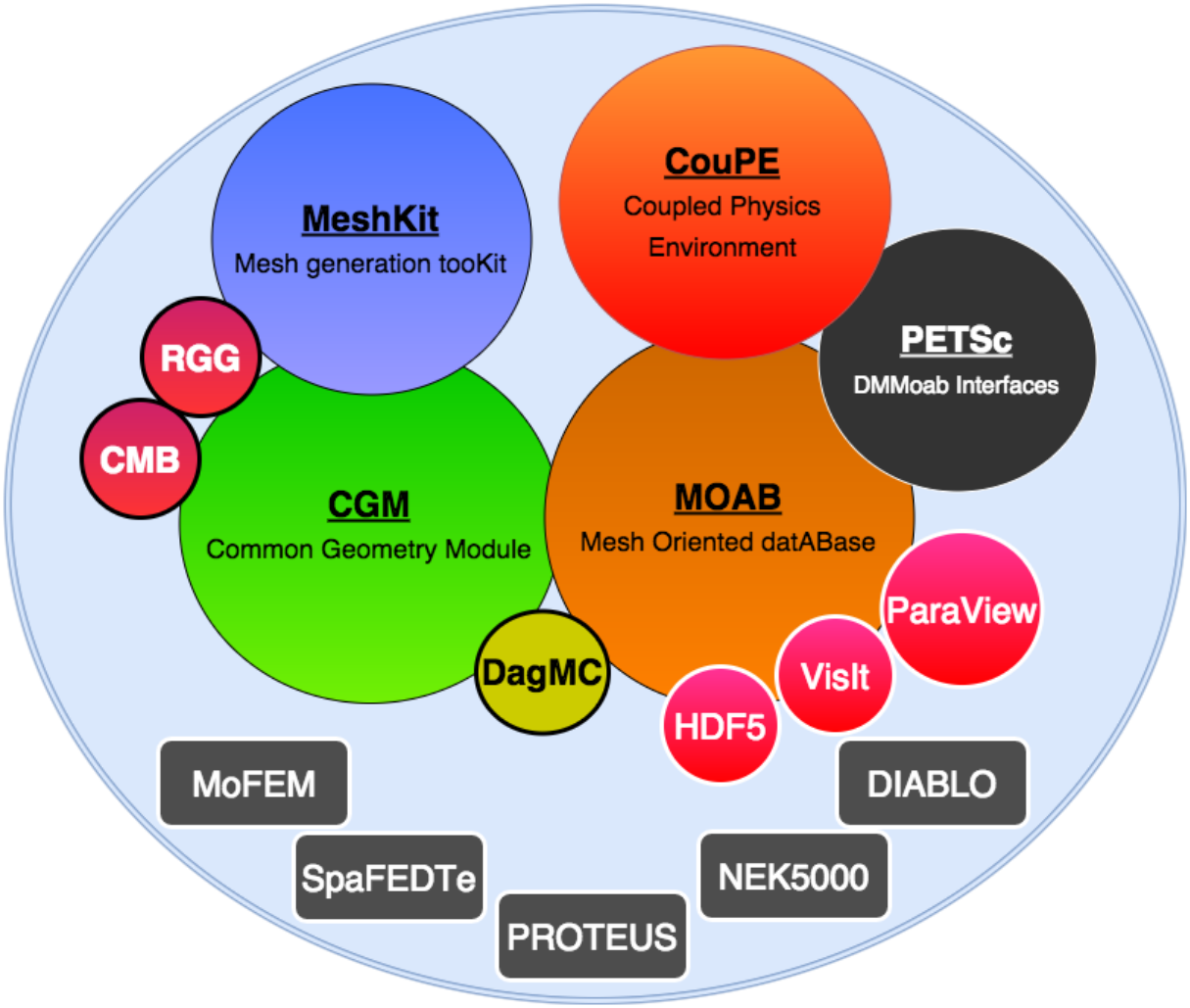


Figure 2. SIGMA toolkit

The above SIGMA components provide the tools to seamlessly traverse the entire computational workflow from problem design to analysis, improving scientific productivity. The following sections in this paper describe details about each of the components along with dependent applications developed by collaborators. The details on the software architecture of SIGMA and the continuous integration process enforced to maintain rigorous code standards will be described in the following sections.

3 Verdict mesh-quality library integration

One fundamental aspect of any mesh generation toolkit is the quality of the elements generated. As a result, significant research has gone into defining quality metrics for which elements may be judged. Element quality criteria are generally agreed upon standards for acceptance of a mesh for simulation purposes.

One concern is that the errors introduced by interpolation be as small as possible. In the finite element method, another important requirement is that the condition numbers of the assembled stiffness matrices be small. Sixty years after the invention of the finite element method, the relationship between mesh quality measures, numerical accuracy, and stiffness matrix conditioning is still an area of research, especially for anisotropic cases.

3.1 Background

Verdict [11] is a comprehensive suite of mesh quality metrics for evaluating the quality of hexahedral, tetrahedral, quadrilateral and triangular finite elements. Verdict has been developed to help standardize the quality metrics. It is included in CUBIT, and it is also integrated in VTK, maintained by KitWare. In VTK integration, the filter vtkMeshQuality is a wrapper around Verdict library.

This library was originally developed at Sandia and released under LGPL licensing. Due to the need to measure and perform a posteriori operations during the process of mesh generation, a decision was made to include the Verdict library directly in MOAB as a ready-to-use quality measure tool.

3.2 Description

Verdict is a collection of methods for evaluating the geometric qualities of standard finite elements, using a variety of quality metrics [12, 13]. All quality metrics are real numbers, and they are computed as a function of vertex coordinates for the respective cell. The methods are written in C++ and have a simple "*C*" interface. Each metric can be computed individually or in combination with other metrics. When multiple metrics are evaluated at once, if possible, common calculations are shared, to increase efficiency.

For example, the function to compute the Jacobian of a hex element is

```
double v_hex_jacobian( int num_nodes, double coordinates[][3] );
```

It can be used as follows:

```
double coords[8][3];
double jac_value =  v_hex_jacobian(8, coords);
```

There are functions like these for each individual quality metric [14, 15], and they are all defined in verdict.h, which is the only header a user would have to include to use the functionality. Another way of directly using verdict is to compute with one call multiple qualities for an element, with a method like this:

```
//! Calculates quality metrics for hexahedral
elements.
void v_hex_quality( int num_nodes,
                    double coordinates[][3],
                    unsigned int
```

In this method, metrics_request_flag is a mask for desired quality metrics, and qualities are returned in a structure defined also in verdict.h, HexMetricVals.

```
struct HexMetricVals
{
  /** \sa v_hex_edge_ratio */
  double edge_ratio ;
  /** \sa v_hex_max_edge_ratio */
  double max_edge_ratio ;
  /** \sa v_hex_skew */
  double skew ;
  /** \sa v_hex_taper */
  double taper ;
  /** \sa v_hex_volume */
  double volume ;
  /** \sa v_hex_stretch */
  double stretch ;
  /** \sa v_hex_diagonal */
  double diagonal ;
  /** \sa v_hex_dimension */
  double dimension ;
  /** \sa v_hex_oddy */
  double oddy ;
  /** \sa v_hex_med_aspect_frobenius */
  double med_aspect_frobenius ;
  /** \sa v_hex_condition */
  double condition ;
  /** \sa v_hex_jacobian */
  double jacobian ;
  /** \sa v_hex_scaled_jacobian */
  double scaled_jacobian ;
  /** \sa v_hex_shear */
  double shear ;
  /** \sa v_hex_shape */
  double shape ;
  /** \sa v_hex_relative_size */
  double relative_size_squared;
  /** \sa v_hex_shape_and_size */
  double shape_and_size ;
  /** \sa v_hex_shear_and_size */
  double shear_and_size ;
  /** \sa v_hex_distortion */
  double distortion;
};
```

Internally, a matrix with possible quality metrics for each element type is maintained, and could be extended if new measures become available. In the matrix, rows correspond to MOAB entity types, and columns correspond to QualityType enum.

```
MB_EDGE_RATIO = 0                    // 0
MB_MAX_EDGE_RATIO                    // 1
MB_SKEW                              // 2
MB_TAPER                             // 3
MB_VOLUME                            // 4
MB_STRETCH                           // 5
MB_DIAGONAL                          // 6
MB_DIMENSION                         // 7
MB_ODDY                              // 8
MB_MED_ASPECT_FROBENIUS              // 9
MB_MAX_ASPECT_FROBENIUS              // 10
MB_CONDITION                         // 11
MB_JACOBIAN                          // 12
MB_SCALED_JACOBIAN                   // 13
MB_SHEAR                             // 14
MB_SHAPE                             // 15
MB_RELATIVE_SIZE_SQUARED             // 16
MB_SHAPE_AND_SIZE                    // 17
MB_SHEAR_AND_SIZE                    // 18
MB_DISTORTION                        // 19
MB_LENGTH                            // 20 edge
MB_RADIUS_RATIO                      // 21 tet
MB_ASPECT_BETA                       // 22 tet
MB_ASPECT_RATIO                      // 23 tet
MB_ASPECT_GAMMA                      // 24 tet
MB_MINIMUM_ANGLE                     // 25 tet
MB_COLLAPSE_RATIO                    // 26 tet
MB_WARPAGE                           // 27 quad
MB_AREA                              // 28 quad
MB_MAXIMUM_ANGLE                     // 29 quad
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBVERTEX
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	MBEDGE
1	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1	1	1	0	1	0	1	0	1	0	1	0	0	1	1	MBTRI
1	1	1	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	1	1	1	MBQUAD
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBPOLYGON
1	0	0	0	1	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	MBTET
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBPYRAMID
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBPRISM
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBKNIFE
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	MBHEX
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBPOLYHEDRON
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	MBENTITYSET

To compute all qualities available for one element type, `VerdictWrapper::all_quality_measures` returns a map from the quality type to the corresponding value.

```
VerdictWrapper vw(&mb);
EntityHandle hex;
std::map<QualityType, double> qualities;
vw.all_quality_measures(hex, qualities);
```

For simpler use in MOAB, a verdict wrapper is implemented. A MOAB user can include VerdictWrapper.hpp in the application, and utilize it like this:

```
VerdictWrapper vw(&mb);
EntityHandle hex;
double jacobian;
rval = vw.quality_measure(hex, MB_JACOBIAN, jacobian);
```

The user has to pass moab::QualityType (MB_JACOBIAN) above, which is an enum type with supported qualities. If a specific quality metric is not supported for a particular element, MB_NOT_IMPLEMENTED error flag is returned.

A quality tool was added too, to report minimum and maximum quality values for all entities in a mesh file. Example of output, for sahex1 model:

Table 1. ***Output from mbquality tool for sahex mesh*** **file (`sahex1_unic_128p.h5m`)**

15180 entities of type MBHEX		
Quality Name	**MIN**	**MAX**
edge ratio	1.05258	3.685
maximum edge ratio	2.29601	12.8944
skew	8.40E-005	0.868193
taper	0.00152519	1.03726
volume	0.0332479	0.364545
stretch	0.127674	0.564097
diagonal	0.807002	0.999996
characteristic length	0.0763946	0.345867
oddy	4.85347	227.634
average Frobenius aspect	1.44291	5.84735
maximum Frobenius aspect	1.48698	6.74519
condition number	1.48698	6.74519
jacobian	0.0237652	0.33081
scaled jacobian	0.399842	0.999988
shear	0.399842	0.999988
shape	0.151717	0.669321
relative size squared	0.00110542	0.132893
shape and size	0.000167711	0.0889482
shear and size	0.000441994	0.113245
distortion	0.424985	0.998294
47259 entities of type MBQUAD		
Quality Name	**MIN**	**MAX**
edge ratio	1.05258	12.8887
maximum edge ratio	1	12.8887
skew	0	0.868193
taper	0	1.03726
stretch	0.109396	0.949672
oddy	0.0131246	82.0621
average Frobenius aspect	1.00284	6.48314
maximum Frobenius aspect	1.00328	6.48314
condition number	1.00328	6.48314
jacobian	0.0190122	1.11875
scaled jacobian	0.399842	1
shear	0.399842	1
shape	0.154246	0.996735
relative size squared	1	1
shape and size	0.154246	0.996735

shear and size	0.399842	1
distortion	0.424985	1
radius ratio	1.05632	6.48314
aspect ratio	1.02676	6.94434
warpage	1	1
area	0.0265983	1.11875
maximum angle	90	154.438
49046 entities of type MBEDGE		
Quality Name	**MIN**	**MAX**
length (volume)	0.0969843	1.25

Plots with 2 quality measures, scaled jacobian and condition number. (range is the same as computed by VisIt) are shown in Fig. 3 and Fig. 4 respectively.

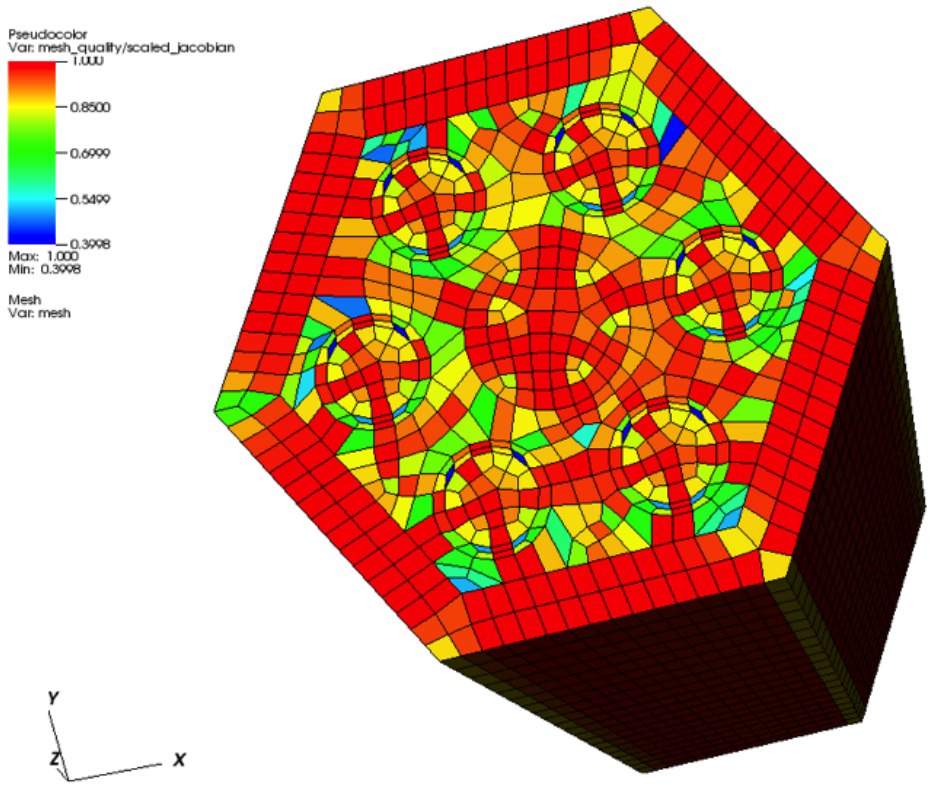


Figure 3. The scaled Jacobian metric for elements in sahex mesh.

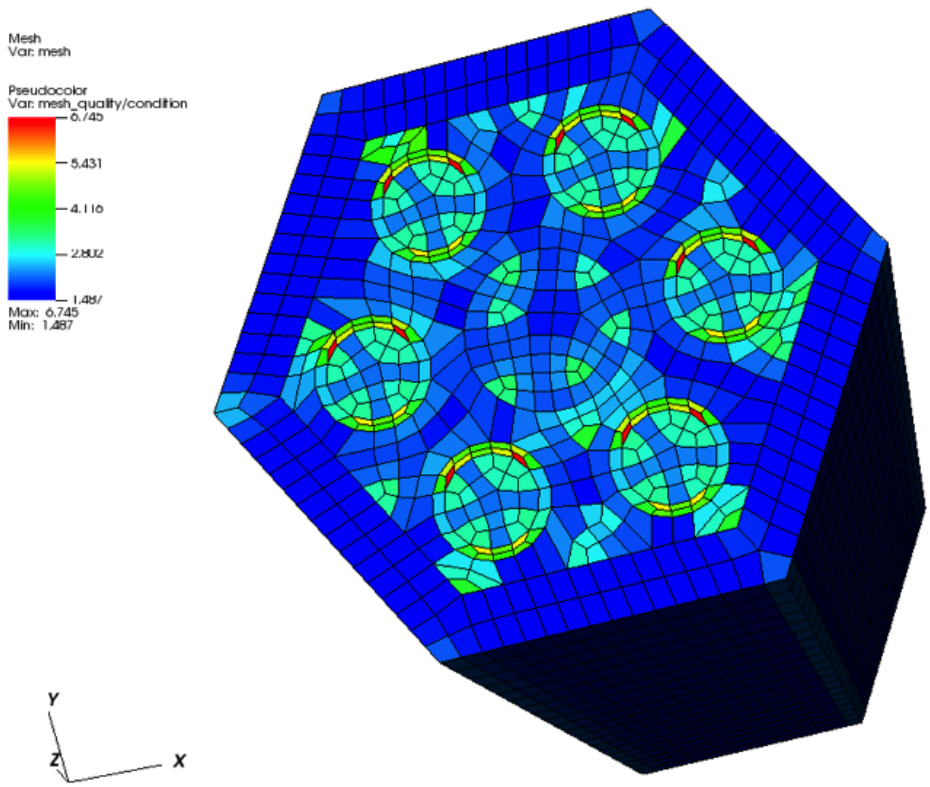


Figure 4. Condition number of elements in sahex mesh.

3.3 Current Uses within SIGMA and Future Plans

It is currently used for immediate feedback during uniform mesh refinement, and for checking the quality of the thin layers of hexahedra generated in MeshKit by the PostBL tool (anisotropic boundary layer generation). Future plans include using it for adaptive mesh refinement (AMR) in MOAB and overall quality checking for all meshes generated by MeshKit.

4 Parallel hierarchical mesh generation capability

In the numerical solution of complex partial differential equations (PDE's) using finite element methods for unstructured meshes, the two most computationally intensive steps are mesh generation and linear solvers. An initial coarse mesh representing the computational domain might not be of sufficient resolution to get meaningful results out of the discretizations for physical scales that might be present. As a result, the capability to refine a mesh is an essential part of any simulation process. On the other hand, it is well known that multi-level methods such as geometric multigrid methods (GMG) can theoretically deliver optimal time complexity for solving sparse linear systems from PDE discretizations. Thus, it would be advantageous to use nested multi-level i.e., hierarchical meshes to achieve accuracy and computational efficiency, especially in the context of large-scale parallel computing, as both the number of processors and the mesh resolution

increase. Uniform mesh refinement(UMR) provides a simple and efficient way to generate such hierarchies via successive refinement of the mesh at a previous level. It also provides a natural hierarchy via parent and child type of relationship between entities of meshes at different levels. While UMR is a relatively simple process, it is by no means trivial. Notable issues include performing multi-level and multi-degree refinement efficiently, and data structure and software design. We developed a parallel uniform refinement-based algorithm [16] under the mesh framework MOAB [6] to generate multi-degree, multi-dimensional and multi-level meshes from coarse unstructured meshes. The developed capability is currently being used for a variety of purposes such as convergence studies, mutlilevel methods, generating large meshes in parallel to overcome IO bottlenecks, etc.

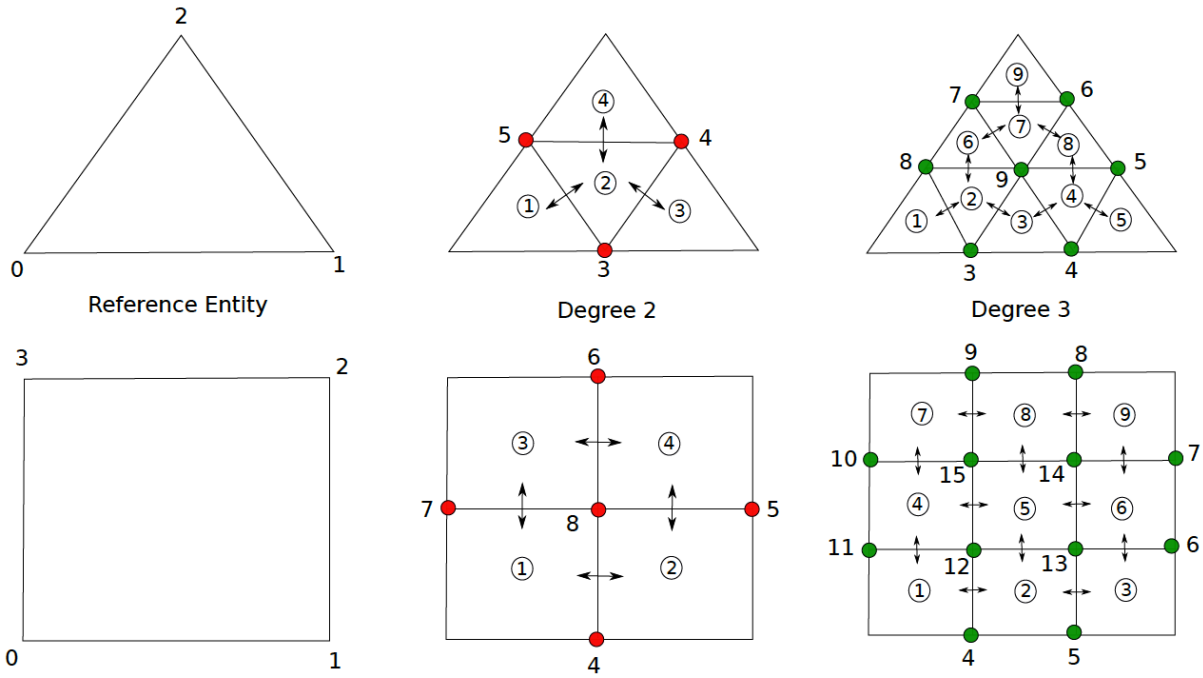


Figure 5. The refinement templates defined for degree 2 and 3 refinements over a reference triangle and quadrilateral entity. The local vertex order of the reference entities follow MOAB conventions[18]. The new vertices to be introduced during refinement are assigned local ids wrt to the reference. Similarly, the children entities are assigned local order. The half-facet maps are created for the refined entities in terms of their local ids.

The uniform refinement based mesh hierarchy generation is build on top of the array-based parallel mesh framework of MOAB and utilizes its various functionalities such as compact memory storage, parallel communication algorithms to resolve shared entities on partition boundaries, etc. The refinement algorithm has three key components: (a) template refinement patterns, (b) an array-based half-facet mesh data structure and (c) level-wise compact mesh storage. Entity type and degree-specific refinement templates store several semantic data such as the local connectivity of the new refined entities and other mesh data structural information with respect to the reference entity [18]. The templates are integral not only to make decisions on subdividing an entity to its children but also to update the underlying mesh data structures for all the new children. The multi-degree refinement

patterns follow the vertex positions analogous to high-order (degree p where $p >= 2$) Lagrangian elements and the associated templates stores degree specific data for a supported entity type. Fig. 5 illustrates the templates for 2D entity types. A considerably general refinement pattern for both volume and manifold meshes using templates is also shown in Fig. 6.

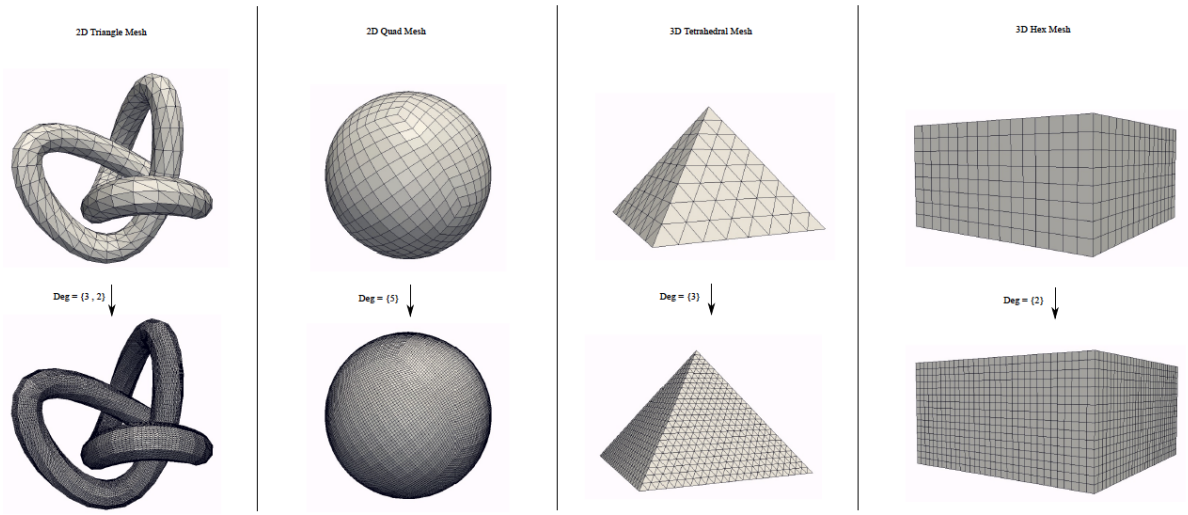


Figure 6. Standard template refinement patters for simple geometries.

During refinement, one of the key tasks is to avoid introducing duplicate vertices for shared boundaries between entities. This requires frequent calls to adjacency routines to get entities connected through entity boundaries. We use an array-based half-facet mesh data (AHF [17]) structure that stores the mapping between all sibling half-facets corresponding to a facet of a mesh entity and has compact memory footprint. As each level of the mesh hierarchy is generated by refinement of the previous level, the appropriate AHF maps are updates for the new level in a two-step process. First, during refinement of a single entity, the local maps only for the children of the working entity are updated via the templates. Second, after all entities are refined, the maps between children sharing a parent boundary are updated. Fig. 7 shows this two-step mesh data structural update process.

Array based mesh storage leads to increased efficiency, however it requires careful consideration to maintain it for operations involving a change in the contiguity of the memory space. By virtue of the uniform refinement of the mesh, it is possible to estimate the total number of entities that will be created for a given degree of refinement. Once the storage requirement for a new level is estimated, the memory is allocated in contiguous blocks. During refinement, as each entity is subdivided, the new entities are stored according to the local order specified in the refinement template. Thus, children of the first entity in previous level are stored first, then the children of the second and so on. This data layout supports straightforward index based inter-level (i.e., parent-to-child or child-to-parent) queries, which are vital to multi-level methods. To provide mesh independence at each level, we duplicate vertices from the previous level to create the new mesh along with new

vertices. As a result, the mesh hierarchy generation would return a sequence of meshes that are independent of each other while providing inter- and intra-level mesh access.

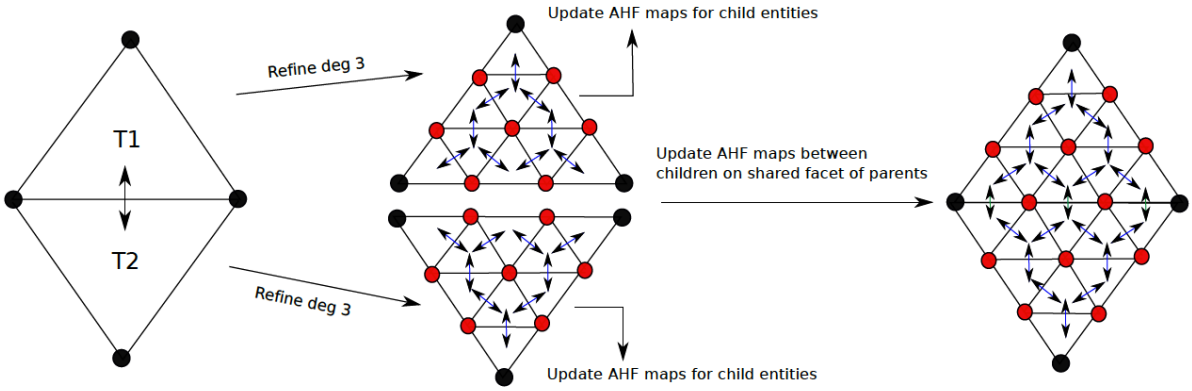
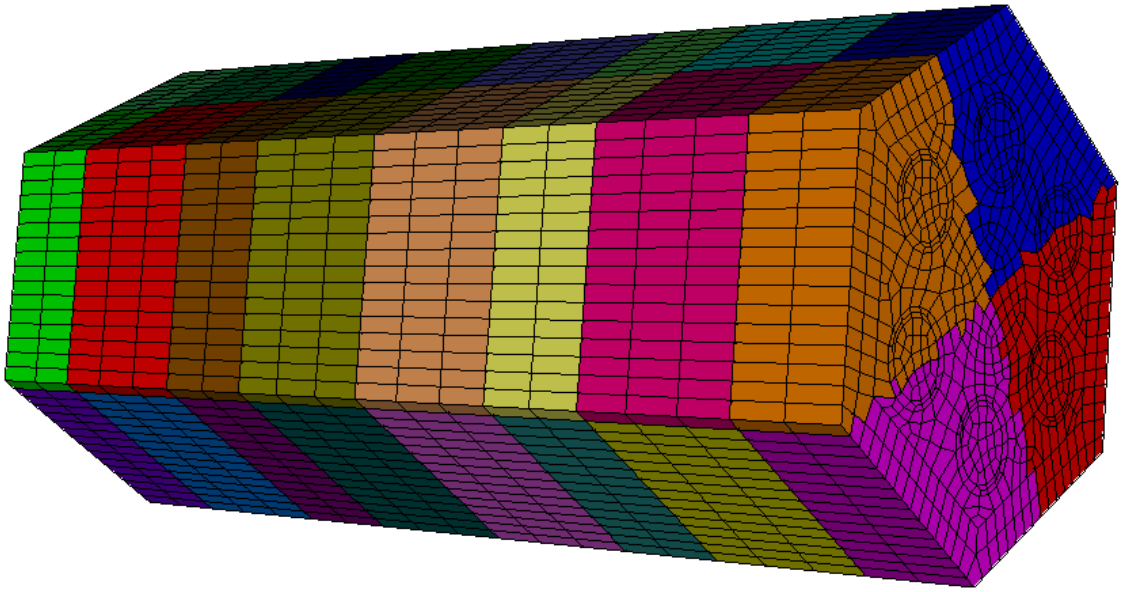


Figure 7. Updating the AHF maps for the refined mesh takes place in two stages. First, maps for the children of each triangle are updated. After, both the triangles have been refined, the maps are now updated to connect the children entities sharing a parent facet.

The MOAB library implemented with array-based data-structures has been designed to be scalable in memory access and there are several optimized one-sided communication algorithms that make use of strategies to minimize total data transferred between processors. Using this parallel framework, the mesh hierarchy generation is performed in a series of optimal steps. Once each processor loads a part of the distributed coarse mesh, local refinement for all the entities can be performed. Since the refinement is based on pre-defined templates, the co-ordinates for the new vertices and entities on the shared interface between processors will be the same. We utilize this information to design the communication algorithm to resolve the newly created entities in preparation for synchronization of ghost layers and exchange of meta-data (MOAB tags). This parallel merge of the interface mesh algorithm [7, 4] first matches mesh vertices based on geometric proximity and then uses connectivity matching algorithm to decipher the corresponding entity in the local mesh.

During preliminary strong scalability studies, the parallel mesh hierarchy generation perform extremely well as evident in Fig. 8. However, more rigorous testing on larger number of processors are needed to conclude about the performance of the algorithm. The parallel UMR capability is useful in generating hierarchy of meshes to perform convergence studies and for creating optimal multigrid preconditioners for elliptic PDE solvers. Tests are currently being conducted using the PETSc-MOAB interfaces to employ Geometric MultiGrid (GMG) preconditioned solvers to ensure optimal convergence for Poisson problems with variable coefficients.



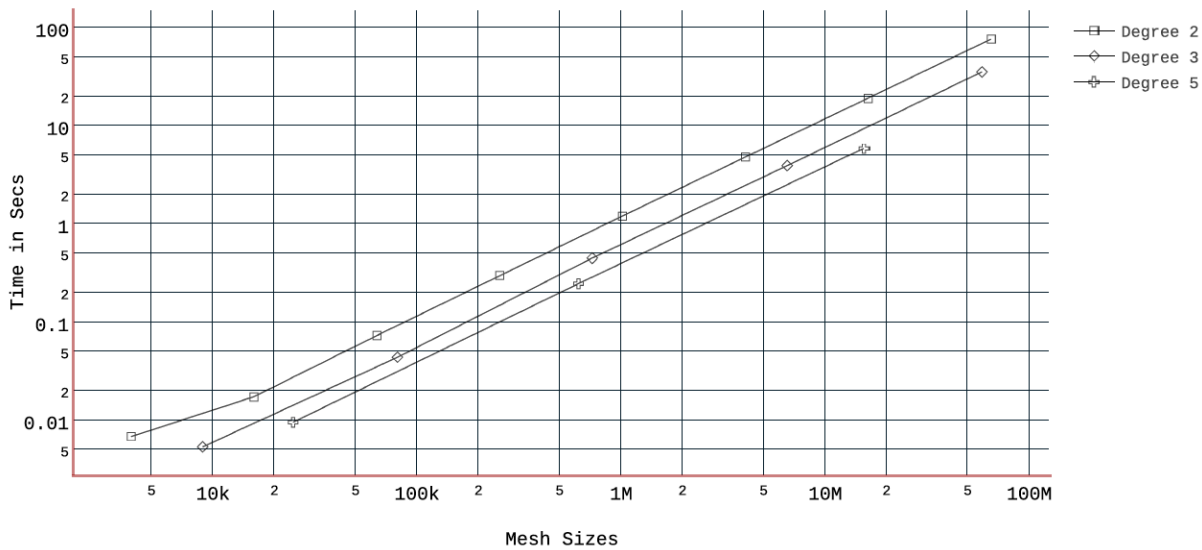


Figure 8. Scalability studies for the mesh hierarchy generation algorithm. The top figure shows the initial coarse hexahedral mesh of a single assembly in a reactor core with fuel pins and central control rod with 32 partitions. The bottom figure shows the serial O(n) refinement performance of the UMR algorithm.

The parallel strong scalability of the AHF based hierarchy generation has also been measured recently and the results will be presented at the upcoming International Meshing Roundtable conference in Austin, TX (Oct 2015). In this paper, we present the measured weak scalability results by splitting the timing into primarily two main components 1) the local refinement of the domain decomposed mesh pieces and 2) the vertex based merge utilizing parallel advanced MOAB communication algorithm. The preliminary results for parallel efficiency of the refinement algorithm and the parallel mesh merge implementation based on coordinates of shared vertices are shown in Fig. 9. These performance studies were

performed on the Blues LCF machine (310 nodes, 16 cores/node with Intel Sandy Bridge processor and 64GB RAM per node) cluster.

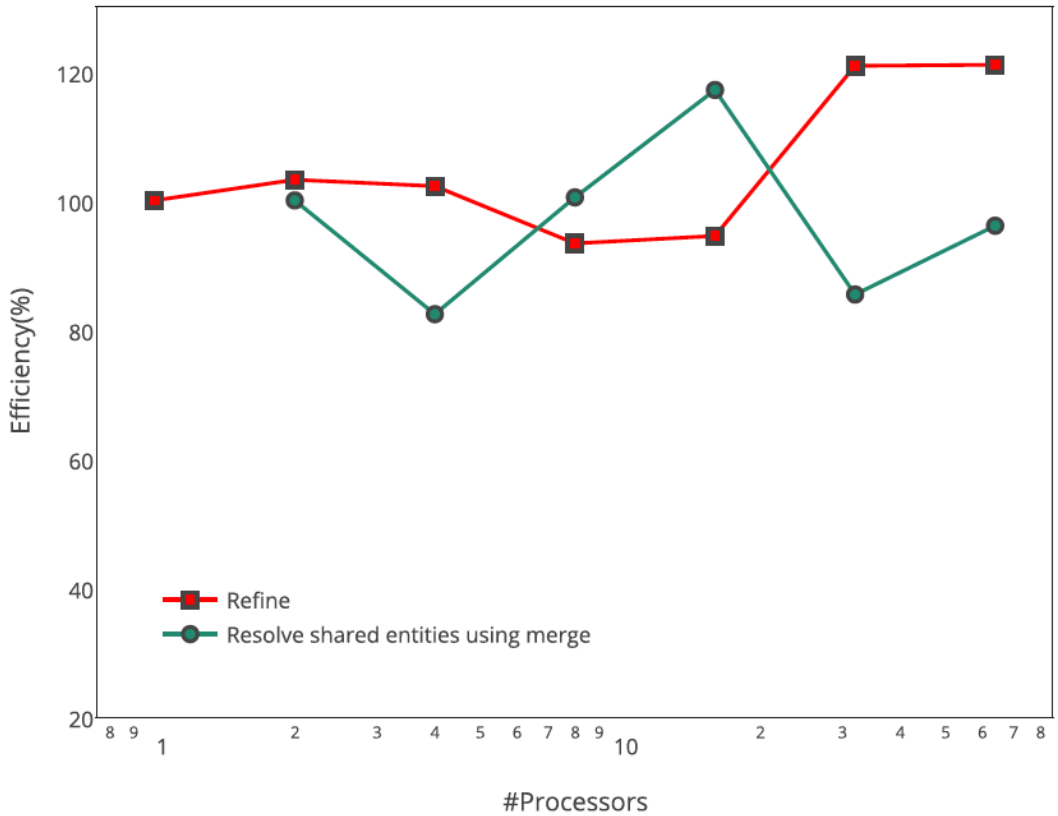


Figure 9. Parallel efficiency of the hierarchical mesh generation algorithm (Blues ALCF machine).

5 Analysis of the OBB tree implementation in MOAB

Analysis of complex surface meshes generated in MOAB has shown lower than expected performance in the past for Monte-Carlo based applications that rely on optimal ray tracing performance. It is believed that this is partially caused by regions referred to as high-valence. A single vertex being shared by a large number of triangles characterizes high-valence regions, which can occur when using faceted geometry representations. As an example, the region shown in Fig. 10 with high triangle density contains a variable number of triangles that share a single vertex. Manual generations of high valence regions such as this allow us to characterize this behavior based on the region's size, valency, and interaction with surrounding mesh in the context of OBB generation.

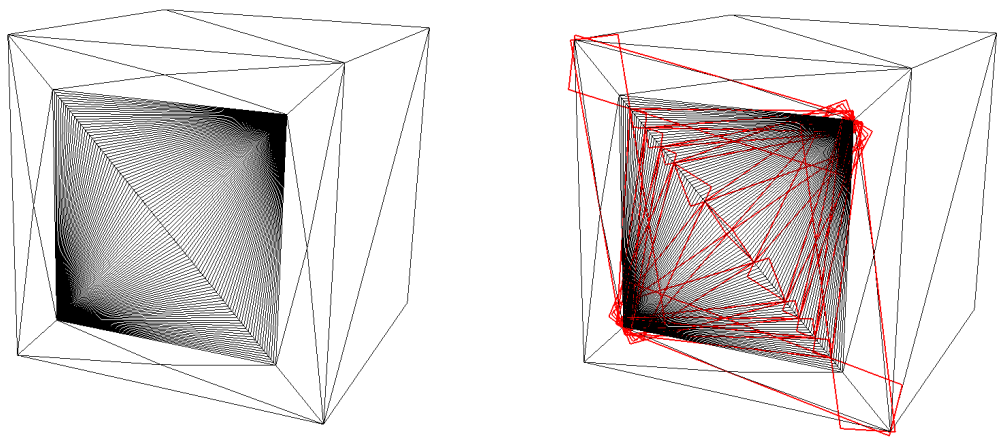


Figure 10. Hierarchical Oriented Bounding Box (h-OBB) (a) Leaf-level OBBs layed on top of the previous mesh (b) Artificially generated high-valence mesh replacing the surface of a cube.

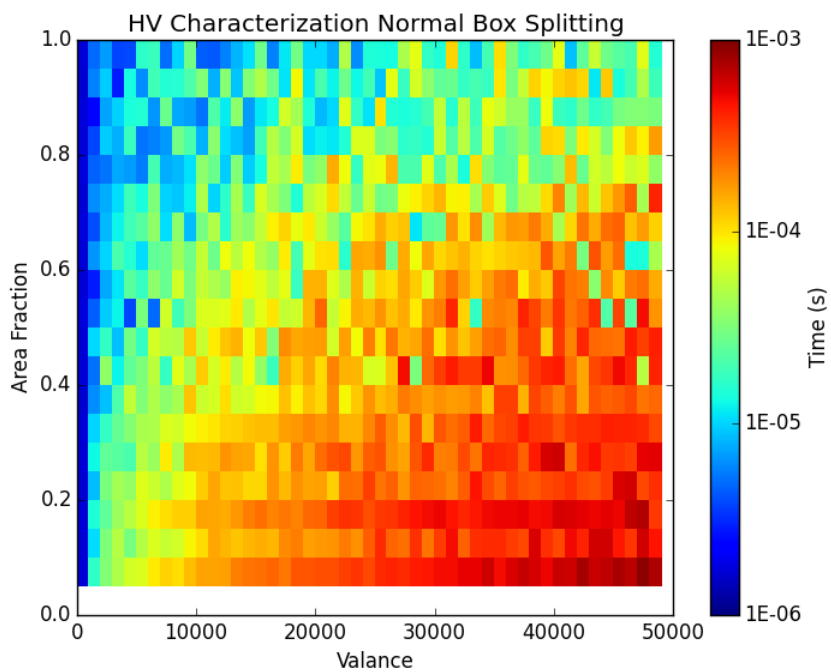


Figure 11. Plot of ray fire times (averaged over 100k rays) various valencies and area with respect to the total surface size for the high-valence region.

Adjusting the parameters of the recursive OBB tree building algorithm shows a vast improvement for the case of high-valence regions. By performing optimizations on the parameters, the ray-tracing computational time was considerably reduced as shown in Fig. 11 and Fig. 12. This demonstrates the ability to improve spatial query performance for this acceleration structure based on examination of mesh features. In the future, automated detection of these features may allow for increased performance on generalize meshes by adapting OBB tree parameters and heuristics during its construction. The implication of this speedup will be directly evident when performing Monte-Carlo ray-tracing calculations using

the geometry and mesh infrastructure provided natively by the SIGMA toolkit. These demonstrations were performed using the DagMC tool developed on top of MOAB library.

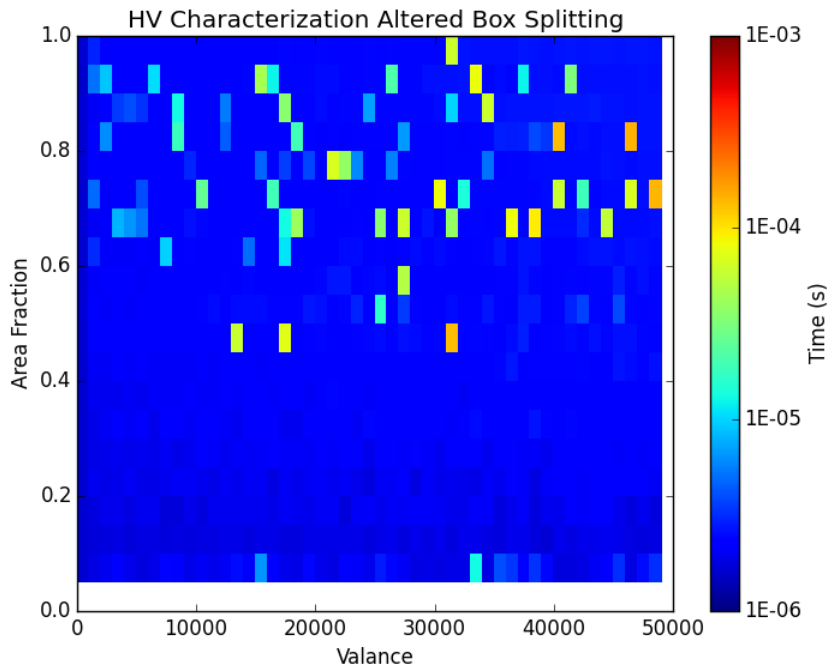


Figure 12. Plot of ray fire times (averaged over 100k rays) various valencies and area with respect to the total surface size for the high-valence region. Here, the OBB splitting parameter is adjusted to better handle stark contrast of the high-valence regions.

6 Conformal local Hexahedral refinement

In FY15, Scott Mitchell (SNL) collaboratively working with the SIGMA team pursued development of local hexahedra mesh refinement capabilities in MOAB and Meshkit. MOAB currently support uniform refinement and is being extended to support mesh adaptivity. The uniform refinement capabilities can be utilized in parallel for generation of a fine mesh from an initial coarse mesh. As is well-known, refining hexes is inherently more difficult than subdividing tetrahedra, because requiring hexes to meet face-to-face introduces global structural dependencies, such as layers of hexes. Our main technical advance is discovering a technique that holds the promise of being flexible enough to provide better quality elements in certain regimes, in particular when refining unstructured regions of about tens of elements thick in size as shown in Fig. 13. (Larger and smaller patches, and structured meshes, are better handled using other techniques.) The key idea is automatically defining hybrid layers of hexes called "slabs." Slabs are two elements thick, but only one layer is required to be structured and meet face to face. By breaking the refinement region into slabs, we are able to select geometric configurations that are likely to result in good quality elements. Slabs generalize prior methods that only worked for structured meshes.

The identification of slabs and defining the refinement operations is implemented; remaining tasks include efficiently updating the mesh database, linking to smoothing, and fine-tuning the selection algorithm to produce better quality hexes. We expect to complete the basic operations of the remaining tasks by December 2015. Future follow-on efforts will be invested to ensure good output quality and publish the results in a journal article. The development is an ongoing partnership between Sandia and ANL.

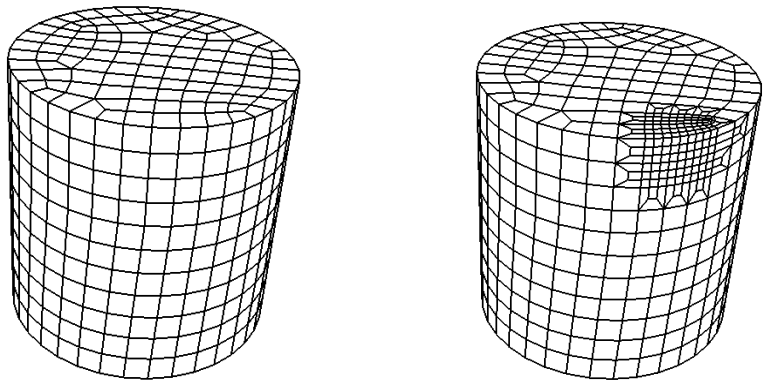


Figure 13. (a) Left – Original coarse HEX mesh (b) Right – Locally refined HEX mesh

7 Integration of Lasso component to MOAB

As mentioned previously, CGM offers a complete ITAPS (iGeom) implementation for standardized access to geometry entities, as needed during mesh generation or other geometry queries, through interfaces to two geometry engines: ACIS and OpenCascade.

MOAB also supports a partial iGeom implementation, namely FBiGeom, which is built based on MOAB topology conventions. FBiGeom is a “facet-based” boundary geometry representation in contrast to the continuous boundary representation (Brep) in CGM. Internally, MOAB represents these geometric entities through mesh sets; a GEOM_DIMENSION tag is specified for each geometry entity, dim=[0,3]. These mesh sets have a hierarchy defined by parent/children relations inside MOAB along with additional tags that define senses and orientation between adjacent corresponding geometry entities. Each face in such a model can then be decomposed in to quadrangles or triangles, and each curve is represented as mesh edges. Hence, a completely discrete representation of the geometry can be created using standard topological entities and used within MOAB for geometric relational queries.

This feature was originally being provided by the Lasso component in SIGMA. A decision was taken to merge this source base into MOAB directly so as to provide all downstream codes the capability to query geometry – mesh relations within applications. This has several potential applications, including and not limited to uniform or adaptive refinement conforming to geometric features.

For mesh refinement (uniform or adaptive), it is important to have access to accurate model geometry so as not to introduce meshing errors. The geometry data can be used effectively to maintain model integrity during mesh discretization and adaptation. Due to the integration of Lasso, MOAB can now identify, maintain and determine associativity between mesh entities and geometry entities, for all geometrical base dimensions, vertex, edge, face and volume. The current implementation uses the iMesh and iGeom API, and optionally with FBiGeom. Efforts are underway to move away from the ITAPS interfaces for this functionality to avoid double indirections when calling relational queries and to leverage the array-based data structure provided by MOAB directly. In FY16, we expect to invest effort into creating a unified adaptivity framework based on MOAB where the geometry-mesh relational queries will become important.

8 iMOAB – A language agnostic MOAB interface

The iMOAB implementation is intended to provide a language agnostic (C/C++/Fortran 90/77, Python) API for applications and drivers to load, manipulate, query, handle and traverse through an unstructured MOAB mesh in parallel. This implementation takes its inspiration from the existing ITAPS iMesh interfaces, which were found to be too verbose and cumbersome for maintenance and support within the NEAMS applications (PROTEUS, Nek5000, Diablo). After several rounds of iterative discussions with the physics code development teams, a suitable interface that supports all solver related queries was transcribed and finalized. The first draft of the implementation along with unit tests and a driver written in both Fortran 77 and C has been verified.

Several key characteristics and principles were rigorously captured during the design of the iMOAB interfaces. Some of these are listed here.

1) All data in the interface are exposed via POD (Plain Old Data) types (int, double, long, char, etc),
2) Pass all parameters by reference, so we do not have to use VAL() when calling from Fortran routines;
3) All array memory are allocated by the client calling code, which simplifies the memory management from the interface standpoint. This is a key difference between the ITAPS approach and iMOAB where only the driver will take up concerns about de-allocation of the data.
4) Always pass the pointer to the start of array along with the total allocated size for the array. The allocated size is used to validate whether the array storage can sufficiently handle the necessary query.
5) Return the filled array requested by client along with optionally the actual length of the array that was filled. (for typical cases, should be the allocated length)

The headers, sources and tests for iMOAB are in a git repository, hosted by BitBucket (https://bitbucket.org/fathomteam/imoab.git). The details on the iMOAB API and links to the driver implementations in both C and Fortran are given in the next section.

8.1 iMOAB: In-built types

In order to abstract out the differences in configuration and architecture, several measures to describe the local, global IDs and error return codes have been defined as plain old data types. These will be replaced by MOAB specific type signatures when iMOAB is merged into MOAB repository.

```
#define ErrCode          int
#define iMOAB_AppID      int
#define iMOAB_GlobalID   int
#define iMOAB_LocalID    int
#define iMOAB_String     char*
```

8.2 iMOAB: Enumeration types

```
enum MOAB_TAG_TYPE
Enumerator:
    DENSE_INTEGER
    DENSE_DOUBLE
    DENSE_ENTITYHANDLE
    SPARSE_INTEGER
    SPARSE_DOUBLE
    SPARSE_ENTITYHANDLE
```

8.3 iMOAB: Function specifications

There are several inherent operations supported by iMOAB such as loading and writing a mesh file along with performing in-memory queries to get the hierarchy of material blocks, the elements within the block, the element connectivities, the vertex IDs and their coordinates. Additionally, routines to help define and get/set tag related storage for use in coupled multiphysics applications is also provided. These routines and interfaces will be consumed by physics applications and CouPE respectively.

8.3.1 iMOAB_Initialize

```
ErrCode iMOAB_Initialize ( int            argc,
                           iMOAB_String * argv
                         )
```

Initialize the iMOAB interface implementation.

Will create the MOAB instance, if not created already (reference counted).

Operations: Collective

Parameters:

[in] **argc** (int) Number of command line arguments

[in] **argv** (iMOAB_String*) Command line arguments

8.3.2 iMOAB_InitializeFortran

```
ErrCode iMOAB_InitializeFortran ()
```

Initialize the iMOAB interface implementation from Fortran driver.

It will create the MOAB instance, if not created already (reference counted).

Operations: Collective

8.3.3 iMOAB_Finalize

```
ErrCode iMOAB_Finalize ( )
```

Finalize the iMOAB interface implementation.

It will delete the internally reference counted MOAB instance maintained in iMOAB, if the reference count reaches 0.

Operations: Collective

8.3.4 iMOAB_RegisterApplication

```
ErrCode iMOAB_RegisterApplication ( iMOAB_String  app_name,
                                    MPI_Comm *    comm,
                                    iMOAB_AppID   pid
                                  )
```

Register application - Create a unique application ID and bootstrap interfaces for further queries.

Note:

Internally, a mesh set will be associated with the application ID and all subsequent queries on the MOAB instance will be directed to this mesh/file set.

Operations: Collective

Parameters:

[in] **app_name** (iMOAB_String) Application name (PROTEUS, NEK5000, etc)

[in] **comm** (MPI_Comm*) MPI communicator to be used for all mesh-related queries originating from this application

[out] **pid** (iMOAB_AppID) The unique pointer to the application ID

8.3.5 iMOAB_DeregisterApplication

```
ErrCode iMOAB_DeregisterApplication ( iMOAB_AppID  pid )
```

De-Register application: delete mesh (set) associated with the application ID.

The associated communicator will be released, and all associated mesh entities and sets will be deleted from the mesh data structure. Associated tag storage data will be freed too.

Operations: Collective

Parameters:

[in] **pid** (iMOAB_AppID) The unique pointer to the application ID

8.3.6 iMOAB_ReadHeaderInfo

```
ErrCode iMOAB_ReadHeaderInfo ( iMOAB_String  filename,
                               int *         num_global_vertices,
                               int *         num_global_elements,
                               int *         num_dimension,
                               int *         num_parts,
                               int           filename_length
                              )
```

Get global information from the file.

It should be called on master task only, and information obtained could be broadcasted by the user. It is a fast lookup in the header of the file.

Operations: Not collective

Parameters:

[in]	**filename**	(iMOAB_String) The MOAB mesh file (H5M) to probe for header information
[out]	**num_global_vertices**	(int*) The total number of vertices in the mesh file
[out]	**num_global_elements**	(int*) The total number of elements (of highest dimension only)
[out]	**num_dimension**	(int*) The highest dimension of elements in the mesh (Edge=1, Tri/Quad=2, Tet/Hex/Prism/Pyramid=3)
[out]	**num_parts**	(int*) The total number of partitions available in the mesh file, typically partitioned with mbpart during pre-processing
[in]	**filename_length**	(int) Length of the file name string

8.3.7 iMOAB_LoadMesh

```
ErrCode iMOAB_LoadMesh ( iMOAB_AppID   pid,
                         iMOAB_String  filename,
                         iMOAB_String  read_options,
                         int *         num_ghost_layers,
                         int           filename_length,
                         int           read_options_length
                       )
```

Load a MOAB mesh file in parallel and exchange ghost layers as requested.

All communication is MPI-based, and read options include parallel loading information, resolving shared entities. Local MOAB instance is populated with mesh cells and vertices in the corresponding local partitions.

Note:

This will also exchange ghost cells and vertices, as requested. The default bridge dimension is 0 (vertices), and all additional lower dimensional sub-entities are exchanged (mesh edges and faces). The tags in the file are not exchanged by default. Default tag information for GLOBAL_ID, MATERIAL_SET, NEUMANN_SET and DIRICHLET_SET is exchanged. Global ID tag is exchanged for all cells and vertices. Material sets, Neumann sets and Dirichlet sets are all augmented with the ghost entities.

Operations: Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**filename**	(iMOAB_String) The MOAB mesh file (H5M) to load onto the internal application mesh set
[in]	**read_options**	(iMOAB_String) Additional options for reading the MOAB mesh file in parallel
[in]	**num_ghost_layers**	(int*) The total number of ghost layers to exchange during mesh loading
[in]	**filename_length**	(int) Length of the filename string
[in]	**read_options_length**	(int) Length of the read options string

8.3.8 iMOAB_WriteMesh

```
ErrCode iMOAB_WriteMesh ( iMOAB_AppID   pid,
                          iMOAB_String  filename,
                          iMOAB_String  write_options,
                          int           filename_length,
                          int           write_options_length
                        )
```

Write a MOAB mesh along with the solution tags to a file.

Note:

> The interface will write one single file (H5M) and for serial files (VTK/Exodus), it will write one file per task. Write options include parallel write options, if needed. Only the mesh set and solution data associated to the application will be written to the file.

Operations: Collective for parallel write, non collective for serial write.

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**filename**	(iMOAB_String) The MOAB mesh file (H5M) to write all the entities contained in the internal application mesh set
[in]	**write_options**	(iMOAB_String) Additional options for writing the MOAB mesh in parallel
[in]	**filename_length**	(int) Length of the filename string
[in]	**write_options_length**	(int) Length of the write options string

8.3.9 iMOAB_GetMeshInfo

```
ErrCode iMOAB_GetMeshInfo ( iMOAB_AppID  pid,
                            int *        num_visible_vertices,
                            int *        num_visible_elements,
                            int *        num_visible_blocks,
                            int *        num_visible_surfaceBC,
                            int *        num_visible_vertexBC
                          )
```

Obtain local mesh information based on the loaded file.

Number of visible vertices and cells include ghost entities. All arrays returned have size 3. Local entities are first, then ghost entities are next. Shared vertices can be owned in MOAB sense by different tasks. Ghost vertices and cells are always owned by other tasks.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[out]	**num_visible_vertices**	(int*) The number of vertices in the current partition/process arranged as: owned/shared only, ghosted, total_visible (array allocated by client, `size := 3`)
[out]	**num_visible_elements**	(int*) The number of elements in current partition/process arranged as: owned only, ghosted/shared, total_visible (array allocated by client, `size := 3`)
[out]	**num_visible_blocks**	(int*) The number of material sets in local mesh in current partition/process arranged as: owned only,

		ghosted/shared, total_visible (array allocated by client, `size := 3`)
[out]	**num_visible_surfaceBC**	(int*) The number of mesh surfaces that have a NEUMANN_SET BC defined in local mesh in current partition/process arranged as: owned only, ghosted/shared, total_visible (array allocated by client, `size := 3`)
[out]	**num_visible_vertexBC**	(int*) The number of vertices that have a DIRICHLET_SET BC defined in local mesh in current partition/process arranged as: owned only, ghosted/shared, total_visible (array allocated by client, `size := 3`)

8.3.10 iMOAB_GetVertexID

```
ErrCode iMOAB_GetVertexID ( iMOAB_AppID       pid,
                            int *             vertices_length,
                            iMOAB_GlobalID *  global_vertex_ID
                          )
```

Get the global vertex IDs for all locally visible (owned and shared/ghosted) vertices.

The array should be allocated by the client, sized with the total number of visible vertices from iMOAB_GetMeshInfo method.

Operations: Not collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**vertices_length**	(int*) The allocated size of array (typical `size := num_visible_vertices`)
[out]	**global_vertex_ID**	(iMOAB_GlobalID*) The global IDs for all locally visible vertices (array allocated by client)

8.3.11 iMOAB_GetVertexOwnership

```
ErrCode iMOAB_GetVertexOwnership ( iMOAB_AppID  pid,
                                   int *        vertices_length,
                                   int *        visible_global_rank_ID
                                 )
```

Get vertex ownership information.

For each vertex based on the local ID, return the process that owns the vertex (local, shared or ghost)

Note:

> Shared vertices could be owned by different tasks. Local and shared vertices are first, ghost vertices are next in the array. Ghost vertices are always owned by a different process ID. Array allocated by the client with total size of visible vertices.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**vertices_length**	(int*) The allocated size of array (typically `size := num_visible_vertices`)
[out]	**visible_global_rank_ID**	(int*) The processor rank owning each of the local vertices

8.3.12 iMOAB_GetVisibleVerticesCoordinates

```
ErrCode iMOAB_GetVisibleVerticesCoordinates ( iMOAB_AppID  pid,
                                              int *        coords_length,
                                              double *     coordinates
                                            )
```

Get vertex coordinates for all local (owned and ghosted) vertices.

Note:

> Coordinates are returned in an array allocated by client, interleaved. Size of the array is dimension times number of visible vertices. The local ordering is implicit, owned/shared vertices are first, then ghosts.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**coords_length**	(int*) The size of the allocated coordinate array (array allocated by client, `size := 3*num_visible_vertices`)
[out]	**coordinates**	(double*) The pointer to client allocated memory that will be filled with interleaved coordinates

8.3.13 iMOAB_GetBlockInfo

```
ErrCode iMOAB_GetBlockInfo ( iMOAB_AppID       pid,
                             iMOAB_GlobalID *  global_block_ID,
                             int *             vertices_per_element,
                             int *             num_elements_in_block
                           )
```

Get the global block information and number of visible elements of belonging to a block (MATERIAL SET).

A block has to be homogeneous, it can contain elements of a single type

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**global_block_ID**	(iMOAB_GlobalID) The global block ID of the set to be queried
[out]	**vertices_per_element**	(int*) The number of vertices per element
[out]	**num_elements_in_block**	(int*) The number of elements in block

8.3.14 iMOAB_GetBlockID

```
ErrCode iMOAB_GetBlockID ( iMOAB_AppID       pid,
                           int *             block_length,
                           iMOAB_GlobalID *  global_block_IDs
                         )
```

Get the global block IDs for all locally visible (owned and shared/ghosted) blocks.

Block IDs are corresponding to MATERIAL_SET tags for material sets. Usually the block ID is exported from Cubit as a unique integer value. First blocks are local, and next blocks are fully ghosted. First blocks have at least one owned cell/element, ghost blocks have only ghost cells. Internally, a block corresponds to a mesh set with a MATERIAL_SET tag value equal to the block ID.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**block_length**	(int*) The allocated size of array (typical `size := num_visible_blocks`)
[out]	**global_block_IDs**	(iMOAB_GlobalID*) The global IDs for all locally visible blocks (array allocated by client)

8.3.15 iMOAB_GetVisibleElementsInfo

```
ErrCode
iMOAB_GetVisibleElementsInfo ( iMOAB_AppID       pid,
                               int *             num_visible_elements,
                               iMOAB_GlobalID *  element_global_IDs,
                               int *             ranks,
                               iMOAB_GlobalID *  block_IDs
                             )
```

Get the visible elements information.

Return for all visible elements the global IDs, ranks they belong to, block ids they belong to.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**num_visible_elements**	(int*) The number of visible elements (returned by GetMeshInfo)
[out]	**element_global_IDs**	(iMOAB_GlobalID*) element global ids
[out]	**ranks**	(int*) The owning ranks of elements
[out]	**block_IDs**	(iMOAB_GlobalID*) The block ids the elements belong to

8.3.16 iMOAB_GetBlockElementConnectivities

```
ErrCode
iMOAB_GetBlockElementConnectivities ( iMOAB_AppID        pid,
                                      iMOAB_GlobalID *   global_block_ID,
                                      int *              connectivity_length,
                                      int *              element_connectivity
                                    )
```

Get the connectivities for elements within a certain block.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**global_block_ID**	(iMOAB_GlobalID*) The global block ID of the set being queried
[in]	**connectivity_length**	(int*) The allocated size of array (typical `size := vertices_per_element*num_elements_in_block`)
[out]	**element_connectivity**	(int*) The connectivity array to store element ordering in MOAB canonical numbering scheme (array allocated by client); array contains vertex indices in the local numbering order for vertices elements are in the same order as provided by GetElementOwnership and GetElementID

8.3.17 iMOAB_GetElementID

```
ErrCode iMOAB_GetElementID ( iMOAB_AppID        pid,
                             iMOAB_GlobalID *   global_block_ID,
                             int *              num_elements_in_block,
```

```
                                iMOAB_GlobalID *  global_element_ID,
                                iMOAB_LocalID *   local_element_ID
                              )
```

Get the global IDs for all locally visible elements belonging to a particular block.

The method will return also the local index of each element, in the local range that contains all visible elements

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**global_block_ID**	(iMOAB_GlobalID*) The global block ID of the set being queried
[in]	**num_elements_in_block**	(int*) The allocated size of global element ID array, same as `num_elements_in_block` returned from GetBlockInfo()
[out]	**global_element_ID**	(iMOAB_GlobalID*) The global IDs for all locally visible elements (array allocated by client)
[out]	**local_element_ID**	(iMOAB_LocalID*) (*`Optional`*) The local IDs for all locally visible elements (index in the range of all primary elements in the rank)

8.3.18 iMOAB_GetPointerToSurfaceBC (Neumann B.C)

```
ErrCode
iMOAB_GetPointerToSurfaceBC ( iMOAB_AppID      pid,
                              int *            surface_BC_length,
                              iMOAB_LocalID *  local_element_ID,
                              int *            reference_surface_ID,
                              int *            boundary_condition_value
                            )
```

Get the surface boundary condition information.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**surface_BC_length**	(int*) The allocated size of surface boundary condition array, same as `num_visible_surfaceBC` returned by GetMeshInfo()
[out]	**local_element_ID**	(iMOAB_LocalID*) The local element IDs that contains the side with the surface BC

[out]	**reference_surface_ID**	(int*) The surface number with the BC in the canonical reference element (e.g., 1 to 6 for HEX, 1-4 for TET)
[out]	**boundary_condition_value**	(int*) The boundary condition type as obtained from the mesh description (value of the NeumannSet defined on the element)

8.3.19 iMOAB_GetPointerToVertexBC (Dirichlet B.C)

```
ErrCode
iMOAB_GetPointerToVertexBC ( iMOAB_AppID      pid,
                             int *            vertex_BC_length,
                             iMOAB_LocalID *  local_vertex_ID,
                             int *            boundary_condition_value
                           )
```

Get the vertex boundary condition information.

Operations: Not Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**vertex_BC_length**	(int) The allocated size of vertex boundary condition array, same as `num_visible_vertexBC` returned by GetMeshInfo()
[out]	**local_vertex_ID**	(iMOAB_LocalID*) The local vertex ID that has Dirichlet BC defined
[out]	**boundary_condition_value**	(int*) The boundary condition type as obtained from the mesh description (value of the Dirichlet_Set tag defined on the vertex)

8.3.20 iMOAB_DefineTagStorage

```
ErrCode iMOAB_DefineTagStorage ( iMOAB_AppID    pid,
                                 iMOAB_String   tag_storage_name,
                                 int *          tag_type,
                                 int *          components_per_entity,
                                 int *          tag_index,
                                 int            tag_storage_name_length
                               )
```

Define a MOAB Tag corresponding to the application depending on requested types.

Note:

In MOAB, for most solution vectors, we only need to create a "Dense", "Double" Tag. A sparse tag can be created too. If the tag is already existing in the file, it will not be

created. If it is a new tag, memory will be allocated when setting the values. Default values are 0 for for integer tags, 0.0 for double tags, 0 for entity handle tags.

Operations: Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**tag_storage_name**	(iMOAB_String) The tag name to store/retrieve the data in MOAB
[in]	**tag_type**	(int*) The type of MOAB tag (Dense/Sparse, Double/Int/EntityHandle), enum MOAB_TAG_TYPE
[in]	**components_per_entity**	(int*) The total size of vector dimension per entity for the tag (e.g., number of doubles per entity)
[out]	**tag_index**	(int*) The tag index which can be used as identifier in synchronize methods
[in]	**tag_storage_name_length**	(int) The length of the tag_storage_name string

8.3.21 iMOAB_GetDoubleTagStorage

```
ErrCode iMOAB_GetDoubleTagStorage ( iMOAB_AppID    pid,
                                    iMOAB_String   tag_storage_name,
                                    int *          num_tag_storage_length,
                                    int *          entity_type,
                                    double *       tag_storage_data,
                                    int            tag_storage_name_length
                                  )
```

Retrieve the specified values in a MOAB double Tag.

Operations: Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**tag_storage_name**	(iMOAB_String) The tag name to store/retreive the data in MOAB
[in]	**num_tag_storage_length**	(int) The size of tag storage data (e.g., num_visible_vertices*components_per_entity or num_visible_elements*components_per_entity)
[in]	**entity_type**	(int*) type 0 for vertices, 1 for primary elements
[out]	**tag_storage_data**	(double*) The array data of type *double* to be copied from the internal tag memory; The data is assumed to be contiguous over the local set of visible entities (either vertices or elements)
[in]	**tag_storage_name_length**	(int) The length of the tag_storage_name string

8.3.22 iMOAB_SetDoubleTagStorage

```
ErrCode iMOAB_SetDoubleTagStorage ( iMOAB_AppID   pid,
                                    iMOAB_String  tag_storage_name,
                                    int *         num_tag_storage_length,
                                    int *         entity_type,
                                    double *      tag_storage_data,
                                    int           tag_storage_name_length
                                  )
```

Store the specified values in a MOAB double Tag.

Operations: Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**tag_storage_name**	(iMOAB_String) The tag name to store/retreive the data in MOAB
[in]	**num_tag_storage_length**	(int*) The size of tag storage data (e.g., num_visible_vertices*components_per_entity or num_visible_elements*components_per_entity)
[in]	**entity_type**	(int*) type 0 for vertices, 1 for primary elements
[out]	**tag_storage_data**	(double*) The array data of type *double* to replace the internal tag memory; The data is assumed to be contiguous over the local set of visible entities (either vertices or elements)
[in]	**tag_storage_name_length**	(iMOAB_String) The length of the tag_storage_name string

8.3.23 iMOAB_SynchronizeTags

```
ErrCode iMOAB_SynchronizeTags ( iMOAB_AppID  pid,
                                int *        num_tags,
                                int *        tag_indices,
                                int *        ent_type
                              )
```

Exchange tag values for given tags.

Operations: Collective

Parameters:

[in]	**pid**	(iMOAB_AppID) The unique pointer to the application ID
[in]	**num_tags**	(int*) Number of tags to exchange
[in]	**tag_indices**	(int*) Array with tag indices of interest (size = *num_tags)
[in]	**ent_type**	(int*) type of entity for tag exchange

8.4 Fortran and C drivers

The links to Fortran and C drivers can be obtained by visiting the up to date documentation website for iMOAB.

ftp://ftp.mcs.anl.gov/pub/fathom/imoab-docs/fex_8F_source.html

ftp://ftp.mcs.anl.gov/pub/fathom/imoab-docs/driver_8c.html

It is expected that the convergence to the API after iterative discussions with the team members will happen by December 2015. This API will then be integrated as part of the core MOAB implementation that can then be configured and utilized in all applications.

9 Software infrastructure and collaborative development

The SIGMA toolkit is completely open-source and encourages a collaborative development process to satisfy the needs of applications in computational science. Due to the large user base in both academia and industry, strong processes to maintain code quality without inhibiting user contributions were necessary. SIGMA uses a continuous integration and build process for all its components and details on different parts of the toolkit are given in the following sections.

9.1 SIGMA software processes

The SHARP toolkit uniformly uses the Git version control system and all the repositories are hosted at the Bitbucket website. Git is a relatively new distributed revision control and source code management system. The flexibility and the seamless workflow that Git provides is making it a widely adopted version control system for software development and all the SIGMA tools utilize the best practices put forth to enable collaborative code developement. The use of pull requests through the Bitbucket interface also brings in contributions from users willing to be involved with the development of SHARP components and examples based on the libraries. This system also encourages peer review on commits and requests thereby combining some SQA process with CI to ensure the robustness and efficiency of the code. Several combinations of builds with different compilers including the GNU, PGI, Intel and Clang have been added to the suite of builds that are run nightly.

One of the complex parts of the SIGMA toolkit, due to the sheer nature of its dependencies, is the configuration and build system. Efforts to simplify the overall build system has been undertaken as a parallel effort to writing the solver pieces to ensure seamless integration of new physics components to SHARP and other applications. Some details on this build system are provided next.

9.1.1 Build System

The build system of Due to the number of required and optional dependencies (MOAB, HDF5, Zoltan, Metis, Parmetis, NetCDF, Zlib, PETSc, Blas/Lapack, Blacs), the

packaging and build system for applications using CouPE can be quite complex. There are several concurrent efforts to ameliorate the difficulties encountered and to make it flexible for the user to configure based on –download-PACKAGE and –with-PACKAGE primitives. There are two main paths a user could take:

1. Autotools
2. CMake

All SIGMA tools from their inception have used autotools build system and it continues to be well supported. Alternatively, the CMake-based build system allows a user to build several components in the SIGMA tools: CGM, MOAB, Lasso and MeshKit using CMake. There are options provided to download and install these dependencies. Part of this effort is being implemented in collaboration with Kitware using the SuperBuild system.

9.1.2 Documentation updates

Some details on updates to different aspects of manuals, documentation and code repositories are provided below.

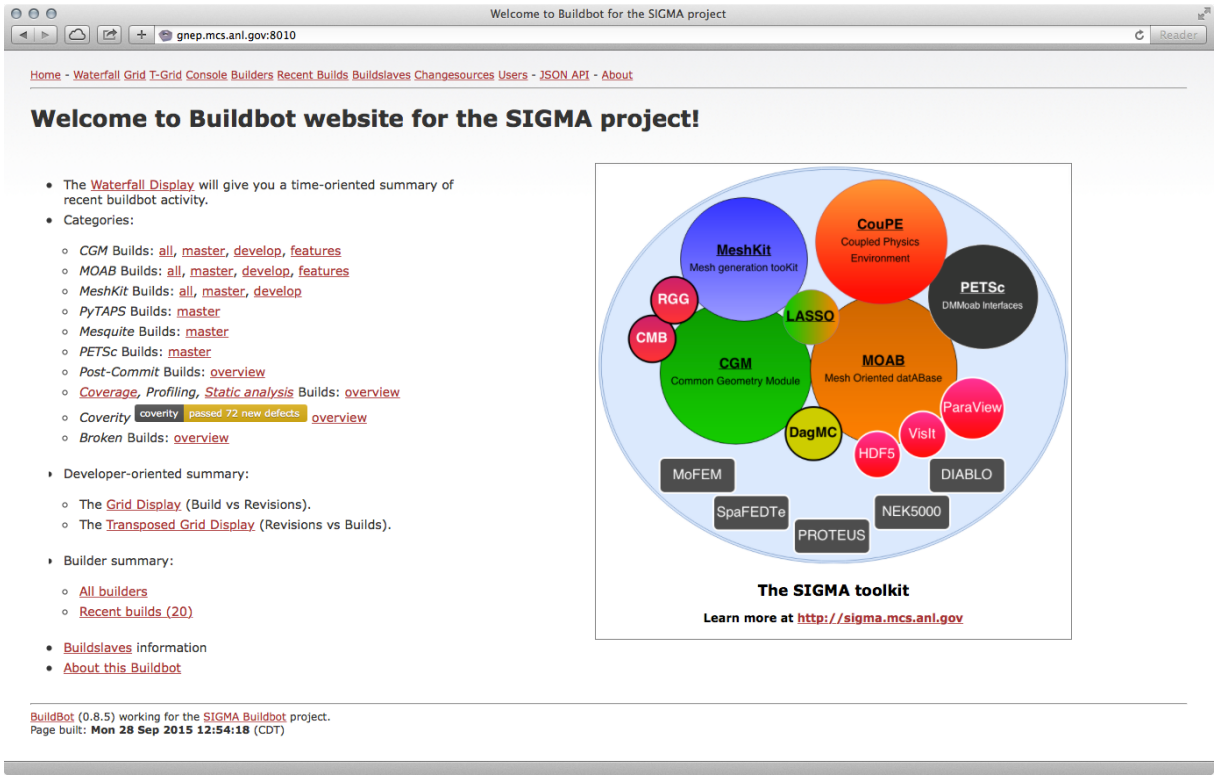
1. **SIGMA Website:** Contains the latest news and developments about CGM, MOAB, MeshKit and CouPE. It contains links to other sources of documentation mentioned in this section. Introduction, download and build instructions for all the SIGMA components is also available here.

2. **Documentation:** All SIGMA components contain sources and documentation that conform to the *doxygen* documentation guidelines along with generation of user and developers guide to be showcased in the reference pages. These artifacts are updated nightly, as part of the continuous integration support (Buildbot) for all library components.

3. **Repository:** All SIGMA components are hosted on Bitbucket as open-source repositories. There are detailed build instructions along with information on branches and other developer related information available in the website and using mailing lists.

9.1.3 Continuous integration

The SIGMA Buildbot system [39] is currently setup to perform both nightly and continuous branch and integration tests for all of the component libraries. The tests are run for different configuration options, combinations of dependencies, and variations in compiler types, versions, MPI implementations and machine architectures. The buildbot test suite infrastructure tests all major branches including master (primary branch), develop (integration branch) and feature branches (user submitted PRs, fixes, feature additions). Fig. 14 shows the waterfall page for SIGMA buildbot system that currently tests over 200 configuration combinations every single day for CGM, MOAB, MeshKit, PETSc (DMMoab) and PyTAPS components.



SIGMA buildbot server: https://gnep.mcs.anl.gov:8010

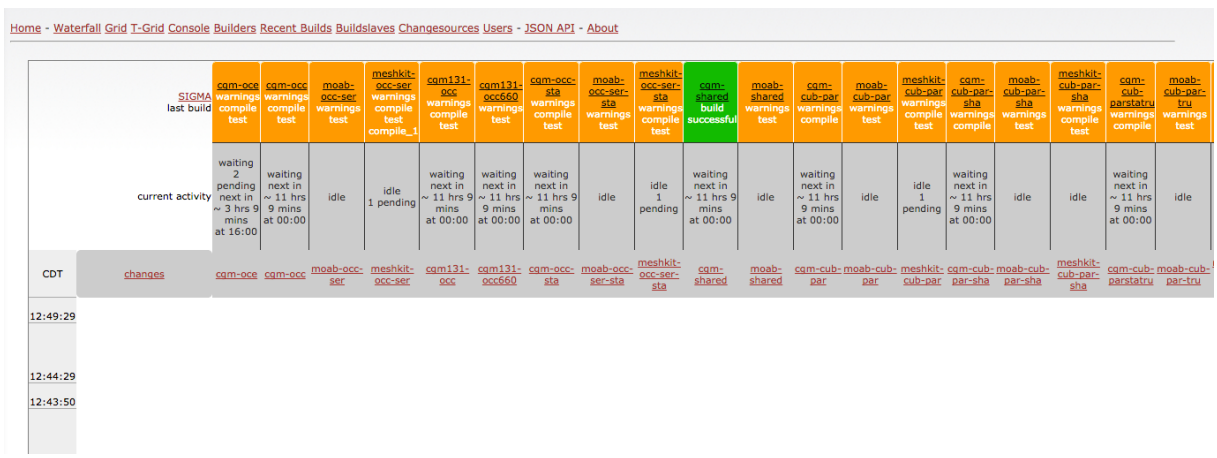


Figure 14. SIGMA Buildbot test suite pages (a) The main buildbot web status page (b) Sample waterfall page with pending builds.

Strong focus on improving code standards and robustness in the libraries are continually being pursued by reducing compiler warnings (GNU/Intel/PGI/clang), memory leaks (valgrind checks) and static code analysis (cppcheck, coverity). These fixes become part of the ongoing release patches after being thoroughly verified on various build combinations.

9.2 Release schedules

The SIGMA toolkit has been undergoing a constant release cycle of 6-8 months with several advanced capabilities being introduced in every release. Several important bug fixes and issues have also been resolved, in parallel I/O, tag handling, solution transfer capabilities etc. The notable feature additions and fixes for the past two releases are given below.

9.2.1 SIGMA v1.1 Release and notes (Jan 2015)

Notable feature additions and fixes for the SIGMA v1.1 release during the first quarter of FY15 is available at https://sigma.mcs.anl.gov/news/sigma-release-1-1.

9.2.2 SIGMA v1.2 Release and notes (October 2015)

We expect to release SIGMA v1.2 during the first week of October 2015 with the most notable features listed in this report for MOAB along with corresponding change sets in MeshKit containing several enhancements and fixes for native scalable mesh generation algorithms. Detailed changeset will be available at https://sigma.mcs.anl.gov/news/sigma-release-1-2.

9.2.3 SIGMA v1.3 Release (expected date: Apr 2016)

The next release of SIGMA (v1.3) is expected to occur in April 2016 and will contain the first release of Mesquite mesh optimization package support natively inside MOAB. A unified design to support adaptive mesh refinement strategies is also being pursued and will be implemented in this context for usage in applications.

9.3 ATPESC participation (August 2015)

The Argonne Training Program on Extreme-Scale Computing covers the key topics that computational science and engineering researchers must master to develop and use leading-edge applications. The program fills a gap in the training that most computational scientists receive; hence it will provide a more comprehensive program than typical short courses. Individuals with expertise and leadership responsibilities in each of these key topic areas will serve as the respective lecturers and the hands-on laboratory session guides.

The core of the program will focus on programming methodologies that are effective across a variety of supercomputers and that are expected to be applicable to exascale systems. Multiple approaches will be covered but the emphasis will be on unifying concepts and levels of abstraction that provide migration paths and performance portability among current and future architectures. Additional topics to be covered include computer architectures, mathematical models and numerical algorithms, approaches to building community codes for HPC systems, and methodologies and tools relevant for Big Data applications.

The SIGMA team conducted several tutorial sessions (Vijay Mahadevan) at the ATPESC training program hosted by Argonne along with hands-on sessions (Vijay Mahadevan, Iulian Grindeanu) as part of the FastMath section of the tutorial. Rajeev Jain also participated in the ATPESC training program this year and has gained valuable background that should prove useful for his research in mesh generation algorithms for nuclear engineering problems.

Additional details on the tutorial sessions conducted by the SIGMA team are available in the link below.

https://press3.mcs.anl.gov/sigma/atpesc2015.

10 Publications and talks (FY15)

1) X. Zhao, R. Conley, N. Ray, V. S. Mahadevan, X. Jiao, "Conformal and Non-conformal Adaptive Mesh Refinement with Hierarchical Array-based Half-Facet Data Structures", In proceedings of 24th International Meshing Roundtable (IMR24), Austin, TX, Oct 2015.

2) N. Ray, I. Grindeanu, X. Zhao, V. S. Mahadevan, X. Jiao, "Array-Based Hierarchical Mesh Generation in Parallel", In proceedings of 24th International Meshing Roundtable (IMR24), Austin, TX, Oct 2015.

3) V. S. Mahadevan, E. Merzari, I. Grindeanu, "A glass box coupling methodology for rigorously solving strongly coupled phenomena in nuclear reactors", Invited, In proceedings of VI International Conference on Coupled Problems in Science and Engineering, San Servolo, Venice, Italy, May 2015.

4) R. Jain, V. S. Mahadevan, "Simplying Workflow for Reactor Assembly and Full-Core Modeling", In proceedings of Joint International Conference on Mathematics and Computation, Nashville, April 2015.

5) R. Jain, N. Ray, I. Grindeanu, D. Wu, V. S. Mahadevan, "Scalable Mesh Generation for HPC Applications", Poster In Proceedings of 27th SuperComputing conference, Austin, TX, Nov 15-20, 2015.

6) J. Becker, R. Jain, V. S. Mahadevan, R. O'Bara, P. O'Leary, "Meshing Sodium-cooled Fast Reactor Cores for Nuclear Energy Advanced Modeling and Simulation", Poster In Proceedings of 27th SuperComputing conference, Austin, TX, Nov 15-20, 2015.

7) X. Jiao, N. Ray, C. Lu, X. Zhao, “High-Order Surface Reconstruction with Applications in Parallel Meshing and Finite Element Solvers”, SIAM CS&E, Salt Lake City, March 2015.

8) N. Ray, X. Zhao, C. Lu, “MOAB based unstructured meshing and solvers”, LANS seminar series, Argonne National Lab, Aug 2015.

9) N. Ray, “Array-Based Hierarchical Mesh Generation in Parallel”, LANS seminar series, Argonne National Lab, Aug 2015.

10) X. Zhao, “Reconstructing High-order Surface for Meshing in MOAB”, LANS seminar series, Argonne National Lab, Aug 2015.

11) C. Lu, “2-D Parallel PDE Solvers for Structured and Unstructured Meshes with DMMoab”, LANS seminar series, Aug 2015.

12) M. S. Shephard, V. S. Mahadevan, G. Hansen, C. W. Smith, "FASTMath Unstructured Mesh Technologies and hands-on sessions", Argonne Training Program on Extreme-Scale Computing (ATPESC) workshop, St. Charles, IL, Invited talk, August 2015.

13) V. S. Mahadevan, I. Grindeanu, R. Jain, N. Ray, D. Wu, P. Wilson, P. Shriwise, "SIGMA: Scalable Interface for Geometry and Mesh Based Applications", U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC-3) PI Meeting, Washington DC, July 2015.

14) V. S. Mahadevan, "Easing computational workflows through flexible and scalable tools", 3rd Joint Lab for Extreme Scale Computing Workshop, Barcelona, Spain, Jun 29 -- Jul 1, 2015.

15) V. S. Mahadevan, N. Ray, I. Grindeanu, B. F. Smith, "FASTMath Unstructured Mesh (MOAB) Solver (PETSc) Interactions", In Proceedings of SIAM Computational Science and Engineering (SIAM CS\&E15), Salt lake City, Utah, Mar 2015.

16) N. Ray, X. Zhao, V. S. Mahadevan, X. Jiao, "Efficient Unstructured Mesh Traversal Methods Based on Array-Based Half Facets", In Proceedings of SIAM Computational Science and Engineering (SIAM CS\&E15), Salt lake City, Utah, Mar 2015.

17) R. Jain, J. Becker, V. S. Mahadevan, P. Shriwise, R. O'Bara, I. Grindeanu, "OCC-Based Meshing for RGG Applications Using MeshKit", In Proceedings of SIAM Computational Science and Engineering (SIAM CS\&E15), Salt lake City, Utah, Mar 2015.

18) V. S. Mahadevan, I. Grindeanu, R. Jain, N. Ray, D. Wu, P. Wilson, "SIGMA: Scalable Interface for Geometry and Mesh Based Applications", In Proceedings of SIAM Computational Science and Engineering (SIAM CS&E15), poster in CS&E Software mini-symposterium, Salt lake City, Utah, Mar 2015. (**Won the best symposterium award**)

11 Summary

The SIGMA toolkit contains several components that have been designed with the motivation to improve scientific productivity of computational scientists dealing with issues in complex unstructured mesh based solvers. The geometry model support through CGM, the advanced mesh generation capabilities provided by MeshKit, the scalable unstructured mesh handling features provided by MOAB and robust coupled multi-physics solvers

provided by CouPE enable high-fidelity applications to focus on the physics representations and discretizations. The use of SIGMA tools in the SHARP framework has also enabled us to seamlessly tackle explicitly heterogeneous reactor assembly and full core geometry configurations and with demonstrations that showcase accurate convergence to true coupled solutions. Several enhancements for the components in SIGMA are planned for FY16 and the direct impact on the computational workflow for SHARP will be significant.

Acknowledgments

We thank the contributions from various users to the different components in SIGMA (CGM, MOAB, MeshKit, CouPE). We would also like to extend our appreciation to our collaborators, Dr. Scott Mitchell (SNL) for his work on conformal Hex refinement, Patrick Shriwise (UWisc-Madison) for OBB-tree optimizations and Dr. Jim Jiao (SUNY) along with his students Cao Lu, Xinglin Zhao for helping test the parallel UMR strategies. This work was supported in part by the U.S. Department of Energy Office of Nuclear Energy Nuclear Energy Advanced Modeling and Simulation (NEAMS) Program; by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research; and by the U.S. Department of Energy's Scientific Discovery through Advanced Computing program, under Contract DE-AC02-06CH11357.

REFERENCES

1. SIGMA (Scalable Interfaces for Geometry and Mesh-Based Applications).
   URL: http://sigma.mcs.anl.gov/
2. Tautges, T. J. (2005). CGM: a geometry interface for mesh generation, analysis and other applications. Eng. Comput. 17, 486–490.
3. G. D. Sjaardema, T. J. Tautges, T. J. Wilson, S. J. Owen, T. D. Blacker, W. J. Bohnhoff, T. L. Edwards, J. R. Hipp, R. R. Lober, and S. A. Mitchell, CUBIT mesh generation environment volume 1: Users manual, Tech. Report SAND-94-1100, Sandia National Laboratories, 1994.
4. Rajeev Jain and Timothy J. Tautges. Generating unstructured nuclear reactor core meshes in parallel. Procedia Engineering, 82(0):351 – 363, 2014. 23rd International Meshing Roundtable (IMR23).
5. Jain, Rajeev, Tautges, T. J., (2013). MeshKit. Report of US DOE, Reactor Campaign. ANL/MCS-TM-336.
6. Tautges, T. J., Meyers, R., Merkley, K., Stimpson, C., and Ernst, C. (2004). MOAB: A mesh-oriented database, SAND2004-1592. Sandia National Laboratories, Albuquerque, NM.
7. Timothy J. Tautges, Jason A. Kraftcheck, Nathan Bertram, Vipin Sachdeva, and John Magerlein. Mesh interface resolution and ghost exchange in a parallel mesh representation. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW '12, pages 1670–1679, Washington, DC, USA, 2012. IEEE Computer Society.

8. Keyes DE et al. 2013 Multiphysics simulations: challenges and opportunities. Int. J. High Perform. Comput. Appl. 27, 4–83.
9. V. S. Mahadevan, "Coupled Physics Environment (CouPE) library – Design, implementation and release", prepared for the U.S. Department of Energy, Office of Nuclear Energy, Milestone Technical Report, ANL/MCS-TM-345, January 2014.
10. Mahadevan VS, Merzari E, Tautges T, Jain R, Obabko A, Smith M, Fischer P., High-resolution coupled physics solvers for analysing fine-scale nuclear reactor design problems, Phil. Trans. R. Soc. A, 2021, 372:20130381, June 2014.
11. C J Stimpson, Corey Ernst, P Knupp, P P Pébay, D Thompson, The Verdict Library Reference Manual, Sandia National Laboratory and Elemental Technologies, 2007.
12. P. Knupp. “Algebraic mesh quality metrics for unstructured initial meshes.” Finite Elements in Analysis and Design, 39:217–241, 2003
13. Shewchuk, Jonathan, “What Is a Good Linear Finite Element? Interpolation, Conditioning, Anisotropy, and Quality Measures”, In Proc. of the 11th International Meshing Roundtable, 2002.
14. V. N. Parthasarathy et al. “A comparison of tetrahedron quality measures.” Finite Elements in Analysis and Design, 15:255–261, 1993.
15. P. P. Pebay and T.J. Baker, “Analysis of Triangle Quality Measures”, AMS Mathematics of Computation, 72(244):1817–1839, 2003.
16. Navamita Ray, Iulian Grindeanu, Xinglin Zhao, Vijay Mahadevan, and Xiangmin Jiao. Array-based hierarchical mesh generation in parallel. In Proceedings of the 24nd International Meshing Roundtable, To appear, 2015.
17. Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and TimothyJ. Tautges. AHF: Array-based half-facet data structure for mixed-dimensional and non-manifold meshes. In Josep Sarrate and Matthew Staten, editors, Proceedings of the 22nd International Meshing Roundtable, pages 445–464. Springer International Publishing, 2014.
18. Timothy J. Tautges. Canonical numbering systems for finite-element codes. International Journal for Numerical Methods in Biomedical Engineering, 26(12), 1559–1572, 2010.
19. Tautges TJ, Kim H, Caceres A, Jain R. 2011 Coupled multi-physics simulation frameworks for reactor simulation: a bottom-up approach. In Int. Conf. on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2011), Rio de Janiero, Brazil, 8–12 May 2011. La Grange Park, IL: American Nuclear Society.

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC