On-Demand Unstructured Mesh Translation for Reducing Memory Pressure during In Situ Analysis

Jonathan Woodring and
James Ahrens
CCS-7 Division
Los Alamos Natl. Laboratory
Los Alamos, NM 87544, USA
{woodring|ahrens}
@lanl.gov

Timothy J. Tautges,
Tom Peterka, and
Venkatram Vishwanath
MCS Division
Argonne National Laboratory
Lemont, IL 60439, USA
{tautges|tpeterka|venkatv}
@mcs.anl.gov

Berk Geveci
Kitware, Inc.
Clifton Park, NY 12065, USA
berk.geveci@kitware.com

ABSTRACT

When coupling two different mesh-based codes, for example with in situ analytics, the typical strategy is to explicitly copy data (deep copy) from one implementation to another, doing translation in the process. This is necessary because codes usually do not share data model interfaces or implementations. The drawback is that data duplication results in an increased memory footprint for the coupled code. An alternative strategy, which we study in this paper, is to share mesh data through on-demand, fine-grained, run-time data model translation. This saves memory, which is an increasingly scarce resource at exascale, for the increased use of in situ analysis and decreasing memory per core. We study the performance of our method compared against a deep copy with in situ analysis at scale.

Categories and Subject Descriptors

D.1.m [**Programming Techniques**]: Miscellaneous

General Terms

Performance

Keywords

in situ analysis, lazy evaluation, unstructured grids, computational meshes

1. INTRODUCTION

Explicit data duplication (deep copying) is the usual solution to share meshes between coupled codes (linked at the memory interface), but there is an increased memory footprint overhead. For example with in situ (run-time) analysis, simulation mesh data are typically duplicated and translated to an analysis mesh copy. This is because different codes, especially libraries, do not share the same data models, interfaces, and/or implementations. We investigate mesh sharing without data duplication (shallow copy) through on-demand, fine-grained, run-time data model translation, for two reasons: 1) Saving memory is important for exascale architectures as memory is decreasing per core. 2) Our method allows two codes to interpret data in their native interface without refactoring algorithms.

We show this on-demand, shallow copy method is a feasible solution for sharing meshes with in situ analysis. We couple two different mesh models, MOAB (Mesh-Oriented datABase) [26] and VTK (Visualization ToolKit) [24], for our study. VTK and MOAB do not share implementations or interfaces, therefore it is not as simple as pointer sharing. With our on-demand translation, we show that VTK algorithms (filters) are able to run directly on a MOAB mesh, without rewriting the algorithms and saving precious memory. Our methodology and performance study provides guidance to be able to replicate this process for on-demand shallow-copy-sharing in other mesh sharing codes.

UltraVis'13, November 17-21 2013, Denver, CO, USA
Copyright is held by the owner/author(s).
Publication rights licensed to ACM.
ACM 978-1-4503-2500-4/13/11...$15.00
http://dx.doi.org/10.1145/2535571.2535592

2. BACKGROUND

Code coupling is necessary for many large-scale simulations, ranging from coupled multi-physics, in situ analysis [28], and I/O libraries. Coupling via memory avoids performing I/O to storage or over the network, avoiding the one of slowest bottlenecks. ParaView Catalyst [10] and VisIt Libsim [31] are in situ libraries based on VTK [24] that allow analysis to run in simulations. DIY [21] is a library for constructing visualization and analysis algorithms, using a data model similar to MPI and MPI-IO.

Unfortunately, a problem arises for memory sharing that different codes usually do not share data models. Many mesh-based codes (simulations) grow their own custom mesh models. Sandia National Laboratories have originated several: phdMesh [12], Exodus II [23], and CUBIT [3]. ITAPS [5] is an effort to standardize the data model interface for meshing tools, and MOAB [26] implements the iMesh interface. GAMBIT [2] is a popular mesh implementation used for CFD and other physical simulations. In visualization and analysis, many of the data models are based on OpenDX and IBM Data Explorer [1], such as VTK for example. More recently, there has been research into new analysis data models suitable for next generation architectures: PISTON [16], DAX [19], and EAVL [18].

There are even different data models used in I/O libraries [15, 27], as most adapt an agnostic, byte-based data model, losing most of the semantic mesh information. Applications have to share data model semantics, either verbally or through metadata, to be able to interpret data. More sophisticated I/O libraries, such as HDF5 [11], pNetCDF [15], sciDB [4], NetCDF4 [22], and XDMF [6], provide schema for storing and retrieving mesh semantics. ADIOS [17], DataSpaces [8], GLEAN [29], HIO [7], DAMSEL [25], Panda [14] and DRepl [13] are advanced I/O and communication libraries for coupling codes and translating between data formats, memory models, and serialized data.

Since data models can vary greatly, shared data are usually copied from one mesh implementation to another, using extra memory. If two mesh data models have similar implementations, it may be possible to pass the implementation by reference. However, best practices in code design discourage the direct sharing of *data structures* between codes. This propagates implementation dependencies and makes code development much more difficult and fragile. In our work, coupled codes can continue to use their own data model, without reference sharing that results in fragility and forces algorithm implementation changes. Our work achieves this through on-demand, lazy evaluation, only converting mesh data as necessary. This saves memory by avoiding explicit mesh duplication. Many data flow pipeline implementations follow the lazy evaluation model, but at a coarse granularity [1, 24, 31]. Our work is more similar to fine-grain, lazy evaluation model (generators) found in functional languages like Haskell, OCaml, and Scheme and other languages like Python. There have been several efforts to integrate fine-grain, lazy evaluation into visualization execution [9, 30].

3. UNSTRUCTURED MESHES

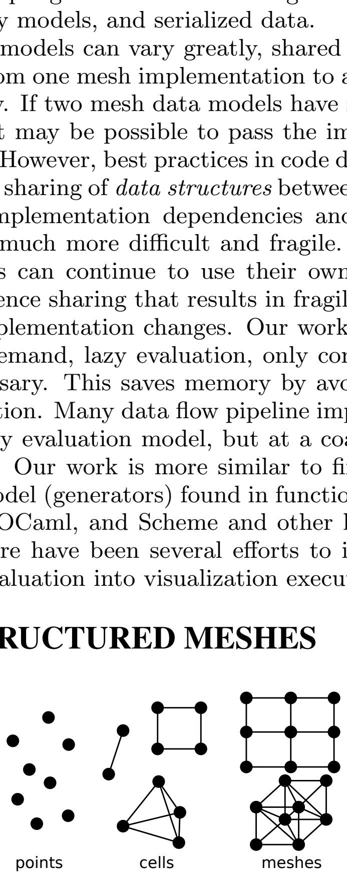

Figure 1: Examples of points, cells, and meshes.

To be able to transparently share meshes between two coupled codes, one must look into the differences and similarities in data models and implementations. We will be primarily concerned with the sharing and translation of two different unstructured grids (meshes): MOAB and VTK. A mesh is a graph consisting of points (vertices) and cells (zones) (see Figure 1). Points are 0 dimensional entities. A cell is a set of connected points, where they form n dimensional entities enclosing a space, e.g., edges (1D), faces (2D), and volumes (3D). Unstructured grids are arbitrary meshes that can contain any number of cell types and are able to represent all other mesh types. Therefore, they provide a good base for our studies. Other specialized meshes (like image data or structured meshes) have constraints that can be used to optimize memory usage and/or accelerate computation.

The information that is required for an unstructured mesh (and what needs to be shared) is a point list, cell connectivity list, and cell type list. Also, there are domain-specific quantities associated with points and cells that need to be shared, typically called attribute data (the "variables"). For example, points will have position attribute data, for its x, y, z coordinates. Attribute data can be arbitrary, depending on the usage domain, such as different scalar, tuple (vector), and tensor (matrix) quantities. Adjacency and neighborhood information, such as shared points between cells, can be explicitly stored as attribute data in an unstructured mesh implementation that are implicit in other meshes. There can be field data associated with an entire mesh or a subset of a mesh, such as time step or block identifier. Finally, large-scale meshes in supercomputing applications have attribute data to describe parallel partitioning and sharing, such as ghost cell ownership.

Exemplar Differences: MOAB and VTK.

MOAB (Mesh Oriented datABase) is a library for representing, storing, reading, and evaluating mesh data for both structured and unstructured meshes. It implements the ITAPS iMesh interface [5] allowing it to freely operate with other tools using ITAPS interfaces. MOAB's data model consists of four basic types: entities, entity sets, the interface instance, and tags. Entities are points and cells. Tags are attribute or field data. Entities (points and cells) in MOAB are referenced using "handles," which represent both the type (with the high-order bits) and the address (unique identifier). MOAB stores "sequences" of handles, represented by a pair of handles (the upper and lower bound) for a contiguous range of addresses.

A key data structure in MOAB is the Range, which can store arbitrary lists of dense handles (addresses) and it uses C arrays to store dense attributes associated with a Range. We refer to "dense" or "sparse" as the difference between using every address in a consecutive address range (a "dense" array where every address is used) or selected addresses in a consecutive address range (a "sparse" associative map where there are gaps in the addresses used). For example, point coordinate attributes are represented using three dense arrays, associated with the consecutive point handle addresses. Attributes can be "sparse," as well in MOAB, with data stored by a (handle, value) associative map.

VTK (Visualization ToolKit) is a general purpose visualization and analysis library that supports many different mesh types, including unstructured grids, which is used by several visualization and analysis software packages. A VTK unstructured grid (vtkUnstructuredGrid) is represented by several VTK classes and C arrays: vtkPoints for point identifiers and coordinates, vtkCellArray for cell identifiers and connectivities, an array for cell types, an array for random access into the vtkCellArray, and vtkDataSetAttributes (a container of vtkDataArrays) for holding field and attribute data. The points and cells are identified by dense addresses, starting at 0. Point and cell addresses are contained in separate address namespaces, due to being stored in separate containers. Attribute data, which are stored in vtkDataArrays, are always dense.

The primary data model difference between the two unstructured grid implementations is how cells and points are addressed and the sparsity of addresses. In VTK, points and cells are referenced by 0-based indices, while in MOAB addressing can start at any base. MOAB mixes point and cell identifiers in the same address space, while VTK uses separate address spaces. Point and cell types are implicit in the MOAB address (the high bits) while VTK uses a separate data array to indicate cell type. All VTK addresses are dense, where point and cell addresses are in the $[0, n-1]$ range, while MOAB addresses use a combination of sparse and dense address ranges. Likewise, attribute data in MOAB may be sparse and only defined for certain points or cells, while VTK allocates data for all points and/or cells, for a particular attribute.

Internally MOAB and VTK both use C arrays for their data structures, but they use them in different ways. For example, MOAB uses a column store for the x, y, z point coordinates, while VTK uses an interleaved (row) store for point coordinates. MOAB is flexible with its definitions of point connectivities per type. For example, it does not enforce that tetrahedra only have 4 points per cell in the cell connectivity array. In contrast, VTK has strict definitions of cell types, and algorithms expect that a tetrahedron only has 4 points per cell. Likewise, VTK can mix different cell types in the connectivity array, while MOAB will separate cells by types into contiguous sequences. What this means is that it is not possible to directly share references of data structures between MOAB and VTK. They both use C arrays, but have very different internal implementations of the unstructured grid data model.

4. COPYING UNSTRUCTURED MESHES

We will discuss the similarities and differences of a traditional "deep copy" (duplication of source mesh data to destination mesh) and our on-demand, run-time translation (on-demand "shallow copy"). We will use MOAB and VTK as our example source and destination meshes. Our methodology should be applicable to most unstructured grid implementations and provide lessons for performance. To perform a deep copy, for every point in a MOAB mesh, we get the MOAB coordinates, look up the VTK address for that point, and set the corresponding VTK point coordinates. Likewise, for every cell in MOAB, we acquire the MOAB connectivity list for a cell, convert the point addresses in that list to VTK point addresses, look up the VTK address for that cell, and insert the connectivity list into VTK with a VTK ordering. Simultaneously, we convert the MOAB cell type into a cooresponding VTK cell type when inserting the cell. For attribute data, the data are copied from MOAB C arrays to VTK vtkDataArrays, making sure to convert addresses.

The main difference between a deep copy and shallow copy is the point in time when the conversion is done. For a deep copy, the MOAB mesh conversion is done all at once, storing the results in an VTK copy of the data. An on-demand, shallow copy defers the MOAB to VTK conversion to any VTK data access. For example if VTK requests point data from a deep copy of a MOAB mesh, the data are already converted (memoized) and read back from VTK memory. In an on-demand, shallow copy version, point data are read from MOAB memory, and converted to VTK point data format, at that point in time and no sooner than that. While this has a drawback of extra computation, there is a huge memory savings, due to not explicitly storing an extra copy of data. Our later results will show that the computational overhead (for our in situ analysis use cases) is small and insignificant.

Address Translation for Points and Cells.

The first hurdle for copying points and cells from MOAB to VTK (and any unstructured grid implementation) is accounting for the address differences. Therefore, we need an address translating function M (Figure 2) that converts VTK addresses to MOAB addresses: $M(src) \rightarrow dest$, where src is the source address (MOAB) and $dest$ is the destination address (VTK). Due to the potential sparsity of MOAB addresses, we utilize a map and address arithmetic to convert MOAB source addresses into VTK destination addresses. The naive solution would use a map consisting of (key, value) pairs for every single source to destination mapping. That particular solution drastically increases the memory footprint and slows the address translation speed to $O(log(n))$.

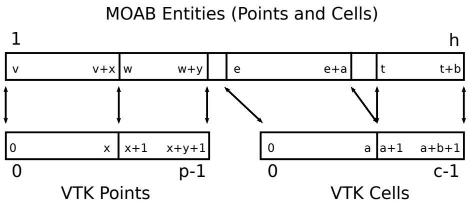

Figure 2: An example mapping of MOAB addresses to VTK addresses.

Instead, we map ranges of MOAB source addresses (consecutive addresses in a Range) to ranges of destination VTK addresses (some segment between $[0, n-1]$), by $M : [lower_s, upper_s] \mapsto [lower_d, upper_d]$, where s denotes a MOAB source range and d is a VTK destination range. MOAB addresses consist of dense addresses sequences, $[lower_i, upper_i]$ for the ith sequence. VTK address range is implicitly $[0, p-1]$ for p points and $[0, c-1]$ for c cells. To do the mapping, we break the VTK destination addresses into equal sized subsequences to matching MOAB source sequences.

The lower_bound operation is used to find the range mapping during address translation. The key in the map is the right hand end of the source address range ($upper_s$ in a $[lower_s, upper_s]$) range. The value is not the destination address, but the difference between the left hand end of the mapped MOAB source and VTK destination ($lower_d - lower_s$ where the two mapped ranges are $[lower_s, upper_s]$ and $[lower_d, upper_d]$). In C++ map notation, this address mapping is stored as: $m[upper_s] = lower_d - upper_d$, which is only as large as the number of MOAB address subsequences. Therefore, the address translating function M is implemented by $dest = m.lower\_bound(src) + src$. The address translation time, comparatively, is on the order of lower_bound or $O(log(i))$, where i is the number of MOAB address subsequences, a small constant. An inverse address mapping function M^{-1} can be created similarly, as well, which is necessary for converting cell connectivity lists.

Copying Attribute and Array Data.

Due to the sparsity of MOAB attributes, we query the MOAB attributes to determine if a particular attribute is defined. If an attribute is defined for at least one point or cell during a deep copy, we create a copy of the MOAB attributes, for all points and/or cells, into a VTK vtkDataArray. If the attribute is not defined for a particular point or cell, a default value is copied into the vtkDataArray. This is due to the fact that VTK uses dense point and cell attributes. Therefore, if at least one MOAB point has a particular attribute, a new VTK array will be created in the

VTK copy of the mesh for that attribute. In an on-demand shallow copy version, attribute data are looked up in MOAB on every VTK attribute access. This requires doing a reverse address lookup for the requested VTK point to translate it to MOAB address space, and then retrieve the attribute data from MOAB, returning it to VTK.

Additionally, VTK has other array data that are implicit or unnecessary in MOAB. For example, cell type attributes are stored in an array in VTK which are implicit to MOAB cell addresses. In MOAB, the high-bits of a cell address are used to indicate the cell type. When deep copying cells, MOAB cell types are converted to VTK cell types and stored in the VTK cell type array. In the shallow copy version, cell types are determined through converting a VTK cell address to MOAB cell address, stripping the high-bits, and converting those high-bits back to a VTK cell type.

One additional array that VTK requires in a deep copy, which MOAB doesn't have, is a cell address offset into the vtkCellArray. In VTK, vtkCellArray is a list of all cell connectivities. To provide random access to a cell's connectivity by cell address, VTK uses an address offset lookup into the cell connectivity list. During a deep copy, this array will be updated as MOAB cells are copied to VTK. In a shallow copy, this extra array is not necessary. A VTK cell address is converted to a MOAB address, connectivity is retrieved from MOAB, and point addresses are converted from MOAB addresses to VTK addresses.

Automatic Shallow Copy via Implementation Hiding.

The final point of discussion is the code refactoring required to implement deep copy vs. on-demand shallow copy. In a deep copy, the translation code is isolated to one point of a coupled code. A source mesh is copied to a destination mesh, applying translation functions only once. From that point forward, the copied mesh can be used in algorithms. In a naive on-demand shallow copy solution, translation functions can be embedded at the point needed to convert from one mesh model to another for algorithms attempting to share a mesh. Again, this scatters implementation dependent code, leading to fragility and maintenance issues.

A better solution is to hide on-demand translation code in an implementation of a copied mesh data model, assuming that algorithms use *data model interfaces* and not *data model implementations*. It requires that coupled codes do not use implementation level details. For example with MOAB and VTK, we created an implementation of vtkUnstructuredGrid that is a shallow copy of a MOAB mesh. We created new implementations of vtkPoints, vtkCellArray and vtkDataArray, which reference an existing MOAB mesh, doing translation from MOAB to VTK on-demand. This translation is hidden, as it becomes part of the mesh implementation. This methodology for on-demand translation is just as seamless as a deep copy, from the perspective of VTK filters.

In C++, this corresponds to creating pure virtual classes (interfaces) for the data structures used by vtkUnstructuredGrid, with two explicit implementations for each of the data structures. The first implementation is the default VTK implementation, while the second implementation contains a MOAB mesh that does on-demand translation to the VTK data model. This allows all existing VTK unstructured grid algorithms to work as-is, regardless if it is a native VTK implementation or a MOAB shallow copy. This also means there is very little to no change to be made to the VTK algorithm codes, either.

The pseudocode for most implemented data access methods on the MOAB shallow copy becomes: 1) convert VTK id into MOAB handle, 2) retrieve data from MOAB via handle, 3) optionally, convert data to VTK data model as necessary (e.g., point addresses), and 4) return data to VTK. This type of translating code is sometimes referred to as a "thunk." A thunk is transparent to both sides of a coupled code, as it avoids any direct implementation dependencies, which has already been prototyped in most cases by a deep copy implementation.

One of the side effects of creating MOAB shallow copy implementations in VTK (and one of the reasons we have been stressing the importance of interface vs. implementation separation in data models) was that several algorithms (filters) did require alterations to the data access methods. VTK algorithms assume that there is only one implementation of a VTK data structure. Therefore, many VTK algorithms will attempt to get direct access to the underlying data structures of a data set, utilizing implementation level details. To fix those filters, it was a minor change that required an algorithm to iterate over a data set through its interface and not its implementation. This did not change the logic of the algorithms, but was tedious from man-hours perspective to fix the code. As we can see, changes in an implementation of a data model have far reaching effects on other code if they rely on implementation details.

5. PERFORMANCE RESULTS

To test the performance of our on-demand mesh translation vs. deep copy, we created a proxy application to simulate in situ workloads (to be released, open-source at a later date). The proxy application can shallow copy or deep copy a MOAB mesh to a VTK mesh with the following operations: copy only, touch (read) all mesh data, slice, clip, isosurface, threshold, surface rendering to PNG, and write to VTK file. It can run in serial or parallel, and link against unmodified VTK for baseline testing. Shallow copies do all mesh access through interface (get and set) operations, rather than pointer (implementation) access found in default VTK.

For our two small-scale tests, we used an HP SL230 (Intel Xeon E5-2650L: 2 sockets, 8 cores per socket @ 1.8 GHz) and an HP DL980 (Intel Xeon X6550: 8 sockets, 8 cores per socket @ 2.0 GHz), each with 128GB of memory. For our large-scale tests, we ran on cluster named "Moonlight," listed as "ML" in our plots. Moonlight nodes have two 8-core Intel Xeon E5-2670 running at 2.6 GHz each, for total of 512 processes in our largest tests, with 32GB memory per node. Small-scale tests used 1 to 8 million tetrahedra meshes, while large-scale tests used 16 to 512 million quadrilateral meshes, both with only one scalar attribute (fewer attributes will be in favor of a deep copy performance). The performance study measured the average time and maximum memory of the Cartesian product of: [refactored VTK, unmodified VTK] x [shallow copy, deep copy] x [copy only, touch (read) all data, slice, clip, threshold, isosurface, surface rendering] x 5 times. MOAB, VTK, and the proxy application were compiled with standard gcc -O2 options. Timing was done through clock_gettime (MONOTONIC_CLOCK) and memory was measured as incremental, high-water differences using the Linux /proc/smaps interface.

Due to lack of space, we provide a common key for all of the plots: *Dashed lines* are shallow copy and/or refactored versions of VTK. *Solid lines* are deep copy and/or default versions of VTK. For "SL230" and "DL980" single machine tests, *magenta lines* are 8 million tets (tetrahedra), *blue lines* are 4 million tets, *green lines* are 2 million tets, and *red lines* are 1 million tets. For "ML" cluster tests, *grey lines* are 512 million quads (quadrilaterals), *orange lines* are 256 million quads, *magenta lines* are 128 million quads, *blue lines* are 64 million quads, *green lines* are 32 million quads, and *red lines* are 16 million quads.

Performance of Deep Copy Implementations.

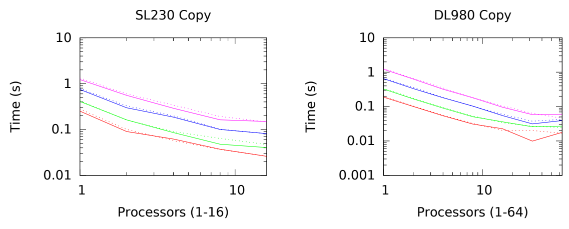

Figure 3: Refactored deep copy compared to default VTK deep copy. SL230 is 1.07 times slower and DL980 is 1.04 times slower.

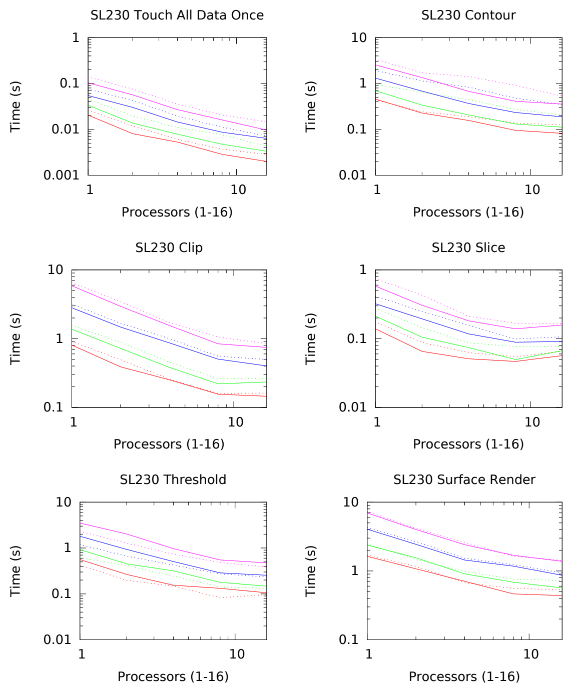

Figure 4: SL230 refactored compared to default VTK filtering. Touching is 1.35 times slower, isocontouring is 1.53 times slower, clipping is 1.14 times slower, slicing is 1.26 times slower, thresholding is 1.39 times faster, and rendering is 1.05 times slower.

Figures 3, 4, and 5 show the differences between deep copies of our refactored VTK (virtualized functions and data access through get-set operations) vs. deep copies of the default VTK implementation (non-virtual functions and data

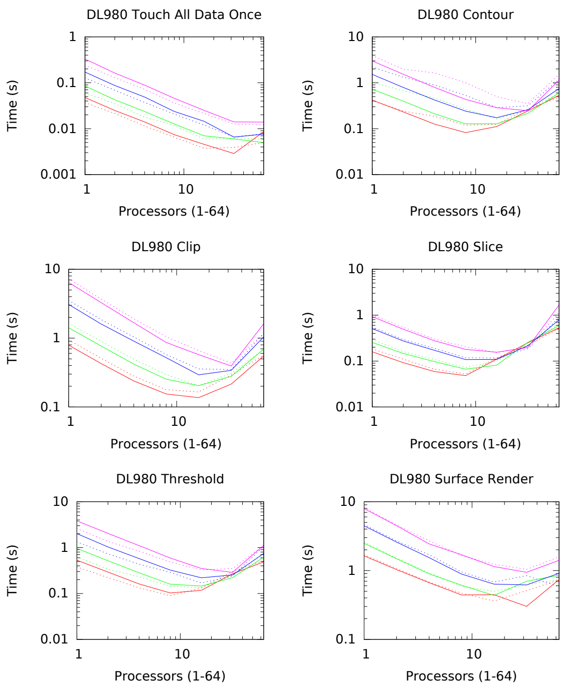

Figure 5: DL980 refactored compared to default VTK filtering. Touching is 1.28 times faster, isocontouring is 1.46 times slower, clipping is 1.12 times slower, slicing is 1.01 times faster, thresholding is 1.24 times faster, and rendering is 1.04 times slower.

access through pointers). A MOAB mesh was deep copied to a VTK mesh, in both cases. This exposes the performance overhead of using virtualized point-wise/cell-wise data model interface access, rather than implementation-direct data structure access (pointers). We show this because there are performance concerns for using pointers vs. interface accessor functions. As we can see, there is an overhead, but this time overhead is insignificant in relationship to the typical time of in situ analysis taken in simulations (around 1% or seconds vs. minutes) [20] (i.e., Amdahl's Law). Minor performance gains come at the cost of having tightly coupled code that is dependent on the data set implementation. In certain circumstances, our refactored version was faster that we suspect is due to caching and memory access patterns and would require further study to determine the cause.

Performance of Deep vs. Shallow.

Figures 6 and 7 show the memory and time savings of an on-demand shallow copy. A shallow copy of a MOAB mesh to VTK mesh is faster and uses less memory than the deep copy solution. It approximately saves 5 to 9 times memory, minimally, for a tetrahedral mesh that only has one attribute. With only one attribute, this test shows the worst memory and time savings for a shallow copy. As more attributes are added, a shallow copy will save more time and memory. Also, a shallow copy only has to be performed once, compared to a deep copy (unless the address mapping function needs to be changed), saving additional time for a copy. A deep copy has to be performed every time in situ analysis takes place, if an attribute changes, to make sure

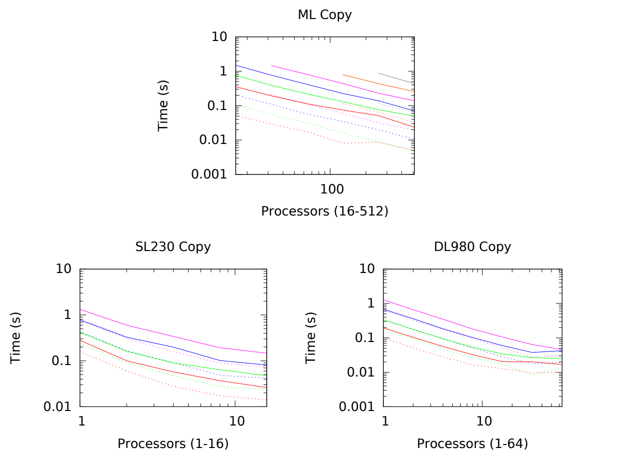

Figure 6: Moonlight shallow copy compared to deep copy is 7.11 times faster, SL230 is 1.88 times faster, and DL980 is 1.98 times faster.

that data are updated in VTK.

Performance of Algorithms on Deep vs. Shallow.

Figures 8, 9, and 10 show the performance of VTK algorithms running on shallow copy vs. deep copy versions of MOAB meshes. As was expected, algorithms using an on-demand shallow copy are slower than native, deep copied data, but not overly so. The worst case is shown by the touch (read) all data test, which is 4.12 times slower on average for Moonlight, 4.54 times slower on average for SL230, and 2.65 times slower on average for DL980. For tests that produce data products (isocontouring, clipping, slicing, thresholding, and surface rendering), this ranges from 1.02 to 2.16 times slower on Moonlight, 1.08 to 2.08 times slower on SL230, and 1.06 to 1.65 times slower on DL980.

This performance loss is acceptable for many reasons. 1) It trades memory gain for loss in algorithm speed, which is becoming a rare commodity as exascale and increased in situ analysis. 2) There are future potential optimizations, as these results are unoptimized implementations of the thunk. 3) In a typical projected exascale supercomputer, "computation will be free," as there will be more compute cycles than any other resource. 4) As noted earlier, the time taken for in situ analysis is insignificant in many cases, compared to the overall simulation time, and the performance loss here is minor relative to that. 5) On-demand translation hidden in the shallow copy mesh implementation saves programming time by allowing existing algorithms to execute as-is.

6. CONCLUSION

We have demonstrated that it is viable to couple simulations and in situ analysis through on-demand translation of meshes. A simulation mesh can be shallow copied to in situ code, without refactoring analysis algorithms and saving the memory cost of a copy of the mesh. Our study has several avenues that have been left unexplored. There may be optimizations that might be useful to attempt to couple multi-physics codes in this way. For coupled physics, the computational overhead may be large, but possibly worthwhile if it allows the codes to run, that wouldn't otherwise

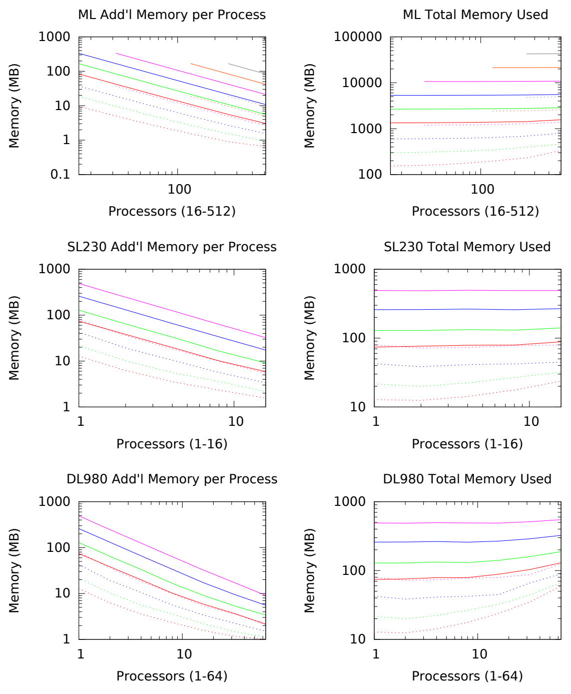

Figure 7: Moonlight shallow copy compared to deep uses 8.45 times less memory, for a quadrilateral mesh. SL230 uses 6.07 times less memory and DL980 uses 5.17 times less memory, for a tetrahedral mesh.

due to memory overhead. We can look into compiler optimizations that overlap mesh translation with mesh computation. Another optimization lies within the analysis algorithms, by specializing filters based on the implementation of data sets. For example, if an algorithm detects that it is running on a MOAB shallow copy implementation, it could use an optimized algorithm for the MOAB memory layout. These could be compiler level optimizations, as well. Also, this method would be useful in combination with I/O libraries, to seamlessly serialize meshes to and from storage, without copying data to the I/O library.

Acknowledgments

This research was funded by the Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) and Scientific Discovery through Advanced Computing (SciDAC) through the CESAR (Center for Exascale Simulation of Advanced Reactors) Co-design Center. Computing resources for the performance study were provided by LANL institutional computing and CCS-7 and HPC divisions. This paper is published under LA-UR-13-27909.

7. REFERENCES

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of the 6th conference on Visualization '95*, VIS '95, page 263, Washington, DC, USA, 1995. IEEE Computer Society.

[2] ANSYS, Inc. ANSYS fluent. http://www.ansys.com/Products/

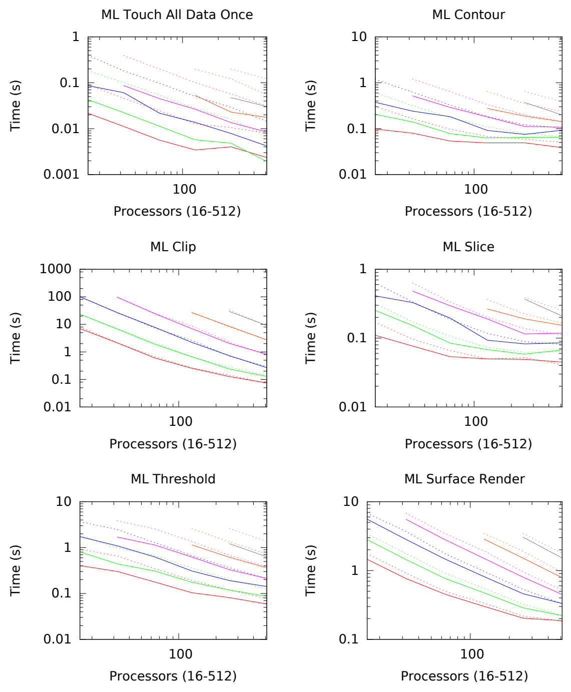

Figure 8: ML shallow copy filter timings compared to deep: touching is 4.12 times slower, isocontouring is 2.13 slower, clipping is 1.02 times slower, slicing is 1.19 times slower, thresholding is 2.16 times slower, and rendering is 1.21 times slower.

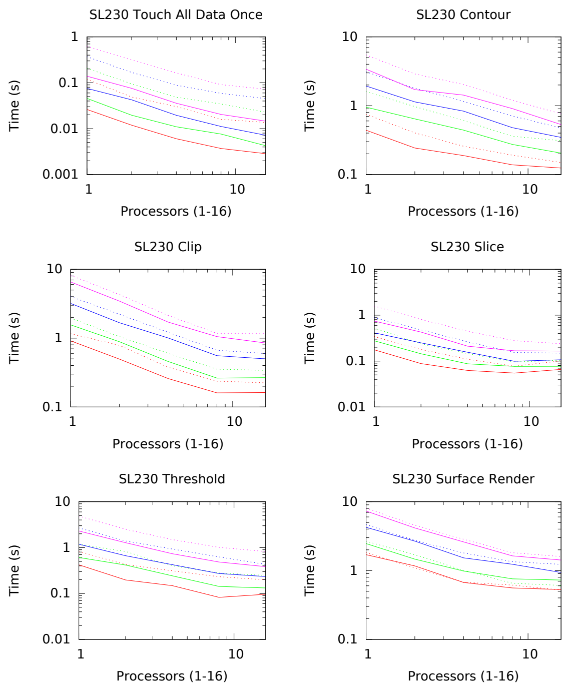

Figure 9: SL230 shallow copy filter timings compared to deep: touching is 4.54 times slower, isocontouring is 1.54 times slower, clipping is 1.26 times slower, slicing is 1.87 times slower, thresholding is 2.08 times slower, and rendering is 1.08 times slower.

Simulation+Technology/Fluid+Dynamics/Fluid+Dynamics+Products/ANSYS+Fluent, Apr. 2013.

[3] T. D. Blacker, W. J. Bohnhoff, T. L. Edwards, J. R. Hipp, R. R. Lober, S. A. Mitchell, G. D. Sjaardema, T. J. Tautges, T. J. Wilson, W. J. Oakes, S. Benzley, J. C. Clements, L. Lopez-Buriek, S. Parker, M. Whitely, D. White, and E. Trimble. CUBIT mesh generation environment volume 1: Users manual. Technical Report SAND94-1100, Sandia National Laboratories, May 1994.

[4] P. G. Brown. Overview of sciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[5] K. Chand, L. Diachin, X. Li, C. Ollivier-Gooch, E. Seol, M. Shephard, T. Tautges, and H. Trease. Toward interoperable mesh, geometry and field components for PDE simulation development. *Engineering with Computers*, 24(2):165–182, 2008.

[6] J. A. Clarke and R. R. Namburu. A distributed computing environment for interdisciplinary applications. *Concurrency and Computation: Practice and Experience*, 14(13-15):1161–1174, 2002.

[7] W. W. Dai. HIO: a library for high performance I/O and data management. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1759–1766, 2011.

[8] C. Docan, M. Parashar, and S. Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[9] D. Duke, M. Wallace, R. Borgo, and C. Runciman. Fine-grained visualization pipelines and lazy functional languages. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):973–980, 2006.

[10] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen. The ParaView coprocessing library: A scalable, general purpose in situ visualization library. In *2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 89–96. IEEE, Oct. 2011.

[11] M. Folk, A. Cheng, and K. Yates. HDF5: a file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing 1999*, Portland, OR, 1999.

[12] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, Sept. 2005.

[13] L. Ionkov, M. Lang, and C. Maltzahn. DRepl: optimizing access to application data for analysis and visualization. In *To appear in IEEE MSST Conference on Massive Storage 2013*, 2013.

[14] S. Kuo, M. Winslett, Y. Cho, J. Lee, and Y. Chen.

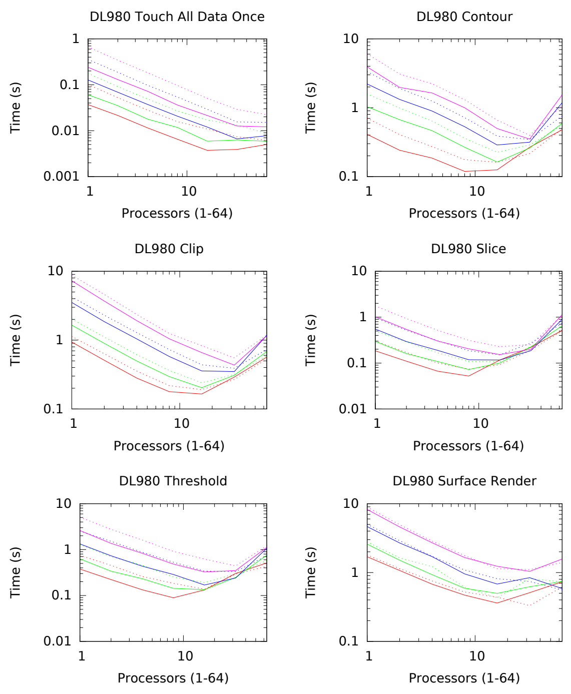

Figure 10: DL980 shallow copy filter timings compared to deep: touching is 2.65 times slower, isocontouring is 1.33 times slower, clipping is 1.17 times slower, slicing is 1.32 times slower, thresholding is 1.65 times slower, and rendering is 1.06 times slower.

Efficient input and output for scientific simulations. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 33–44, 1999.

[15] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: a high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39, 2003.

[16] L.-T. Lo, C. Sewell, and J. Ahrens. PISTON: a portable cross-platform framework for data-parallel visualization operators. Eurographics Symposium on Parallel Graphics and Visualization, 2012.

[17] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

[18] J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30, 2012.

[19] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 97–104. IEEE, Oct. 2011.

[20] J. M. Patchett, J. P. Ahrens, B. Nouanesengsy, P. K. Fasel, P. W. Oleary, C. M. Sewell, J. L. Woodring, C. J. Mitchell, L.-T. Lo, K. L. Myers, J. R. Wendelberger, C. V. Canada, M. G. Daniels, H. M. Abhold, and G. M. Rockefeller. LANL CSSE L2: Case Study of In Situ Data Analysis in ASC Integrated Codes. Technical Report LA-UR-13-26599, Los Alamos National Laboratory, Aug. 2013.

[21] T. Peterka, R. Ross, A. Gyulassy, V. Pascucci, W. Kendall, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri. Scalable parallel building blocks for custom data analysis. In *2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 105–112. IEEE, Oct. 2011.

[22] R. Rew, E. Hartnett, J. Caron, et al. NetCDF-4: software implementing an enhanced data model for the geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, 2006.

[23] L. A. Schoof and V. R. Yarberry. EXODUS II: a finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Sept. 1994.

[24] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[25] S. Sehrish, R. Latham, T. Tautges, N. Samatova, B. Clifford, Q. Koziol, W.-k. Liao, R. Ross, and A. Choudhary. DAMSEL - a data model based storage library. Technical report, Arlington, VA, USA, Oct. 2012.

[26] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: a mesh-oriented database. SAND2004-1592, Sandia National Laboratories, Apr. 2004. Report.

[27] R. Thakur, E. Lusk, and W. Gropp. Users guide for ROMIO: a high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Argonne National Laboratory, Lemont, IL, Oct. 1997.

[28] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing. In *Supercomputing, ACM/IEEE 2006*, page 91, Tampa, Florida, 2006.

[29] V. Vishwanath, M. Hereld, and M. E. Papka. Toward simulation-time data analysis and I/O acceleration on leadership-class systems. In *2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 9–14. IEEE, Oct. 2011.

[30] H. T. Vo, D. K. Osmari, B. Summa, J. L. D. Comba, V. Pascucci, and C. T. Silva. Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum*, 29(3):1073–1082, 2010.

[31] B. Whitlock, J. Favre, and J. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 100–109, 2011.