Mesh Interface Resolution and Ghost Exchange in a Parallel Mesh Representation

Timothy J. Tautges
Mathematics and Computer Science Division
Argonne National Laboratory
Chicago, IL
tautges@mcs.anl.gov

Jason A. Kraftcheck, Nathan Bertram
University of Wisconsin-Madison
Madison, WI
[kraftcheck@gmail.com, nbertram@wisc.edu]

Vipin Sachdeva, John Magerlein
International Business Machines
[vsache, mager]@us.ibm.com

***Abstract*— Algorithms are described for the resolution of shared vertices and higher-dimensional interfaces on domain-decomposed parallel mesh, and for ghost exchange between neighboring processors. Performance data is given for large (up to 64M tet and 32M hex element) meshes on up to 16k processors. Shared interface resolution for structured mesh is also described. Small modifications are required to enable the algorithm to match vertices based on geometric location, useful for joining multi-piece meshes; this capability is also demonstrated.**

Keywords—parallel I/O; unstructured mesh; structured mesh

I. INTRODUCTION

Most parallel simulation codes solving systems of PDEs use a domain decomposition approach, where the spatial domain (i.e. the mesh) is split into P "parts", one part solved on each processor. The initialization of mesh-based models on parallel computers requires more than simply reading the elements in a given part. For element-based unstructured mesh partitions, the vertices required on a part are indirectly indicated by the connectivity of the elements in a part. Most unstructured meshes also contain mesh edges and/or faces that are also part of the model definition, e.g. for boundary condition or geometric model definitions. Other metadata may be represented by collections of entities, with the initialization of a given collection on a processor required only if the collection contains entities also resident on the processor. After initialization of the mesh entities and collections, most codes also require information about vertices and other entities shared between processors, in order to properly formulate the equations being solved. Such codes may also require one or more layers of ghost or halo elements around the part. For adaptive codes, this information must be preserved or restored after mesh migration that usually accompanies mesh refinement.

There has been much work published on the subject of parallel output of solution and checkpoint data; however, less has been reported on parallel read and initialization of mesh-based data, probably because read times are amortized over the (usually longer) solution phase of a computation. However, read times are not so small as to be negligible, and for applications like post-processing can dominate. For adaptive codes, some parts of the initialization process must be repeated after each adaptive step.

In this paper, we describe the reading and initialization of mesh data. We describe the various parts of the read process for unstructured mesh, and several optimizations that have been done to improve read times. We also describe algorithms for resolving shared mesh interfaces between processors, and for exchanging ghost or halo layers between processors. An extension of the shared interface resolution algorithm that matches mesh vertices based on geometric proximity is described, that enables the assembly of multi-part meshes into contiguous mesh. For structured mesh, we also describe techniques for resolving shared interfaces that are mostly communication-free.

A. *Background*

The algorithms in this paper are implemented in MOAB. MOAB is a library for query and modification of structured and unstructured mesh, and field data associated with the mesh [1]. MOAB can represent all entities typically found in the finite element zoo, as well as polygons and polyhedra. Structured mesh is supported as well, with a special interface providing parametric block information [2]. The data model implemented by MOAB references four distinct data types:

- *Entity:* vertices, triangles, quads, etc.
- *Entity Set:* arbitrary collection of entities and other sets
- *Interface:* object through which all other functions are called, i.e. the database
- *Tag:* information stored on Entity, Entity Set, and Interface objects

This data model has proven remarkably versatile, able to represent most semantic information associated with

typical meshes, including boundary conditions, solution fields, geometric associativity, and parallel partitions. Internally, MOAB uses an array-based storage model. The *entity handle* data type used to reference mesh entities is an integral type, with the four high-order bits representing the entity type (MBVERTEX, MBEDGE, …, MBMAX), and the remaining bits representing an integer id. Entities created in sequence are given contiguous entity handles, which can be stored as range pairs *(begin_handle, end_handle)*. The TYPE(handle) and ID(handle) functions return the type and id embedded in a handle, respectively.

On parallel machines, mesh is represented and queried in MOAB the same way a serial mesh is; information about the parallel nature of the model is stored in the form of sets and tags, and can be queried as such. For convenience, MOAB's *ParallelComm* class also provides functions for providing this data, and for performing commonly needed parallel functions. For any entity shared with other processor(s), MOAB stores both the remote processor rank(s) and the handle(s) of the entity on those processor(s), *on all processors sharing the entity.* MOAB uses the HDF5 library for its native save/restore format [3]. Mesh models are initialized in parallel by reading mesh from a single file in parallel, using a partition stored as entity sets in the file. A partitioning tool has been implemented by interfacing with the Zoltan partitioning library [4].

B. Previous Work

Parallel I/O has been discussed extensively in the literature, e.g. see [5][6][7]. Overall, the vast majority of applications seem to have implemented application-side 2-phase I/O, where only a subset of processors interact directly with the file system and communication is used to read/write data. Fu et. al report read/write times of 1-7 GB/s and .25-2.4GB/s, respectively, on 16k processors. For these cases, the data being written consists of one or more solution fields defined at each grid point or cell across all processors. Typically, the best performance is obtained when writing 64-1024 different files and when interacting directly with MPI-IO rather than a high-level I/O library like parallel-netcdf[8] or HDF5[3]. Kimpe et. al [9] report I/O read/write bandwidths of no more than 0.4 MB/sec when using parallel netcdf or HDF5 on up to 8 processors of a computer cluser with a PVFS-based filesystem.

For the overall process of initializing large, unstructured meshes, Fournier et. al report the execution time to initialize 107M element unstructured meshes on an IBM BG/P machine [10]. Mesh initialization accounts for approximately 8% of the execution time of a 50-timestep problem, or approximately the cost of 5 timesteps.

On the subject of post-input mesh initialization, relatively few details have been reported in the literature. The most detailed description of this subject for unstructured meshes is reported by Seol et al [11]. In their work, mesh is migrated between processors in the context of a partition model, which is maintained before and after the migration. The method described in [11] requires at least four message exchanges between each sending and receiving processor pair: synchronizing the partition model before migration, sending entities, receiving back entity references, and forwarding references to other sharing processors. The described method packs entities of each topological dimension in separate messages; to avoid the high latency cost of sending many separate messages, a message aggregator is used. However, this aggregator may inhibit the use of asynchronous messages, and the latency hiding enabled by such messages.

For structured meshes, Falgout describes an algorithm to find matching interface mesh based on physical position [12]. Although this method works for cases with structured Cartesian mesh, a proximity-based criterion may fail in cases where the distance tolerance is not well-matched to the mesh spacing, and, as we show later in this paper, is not necessary.

C. Test Cases

Throughout the paper, performance measurements are quoted for two meshes: one with 32M hexahedral elements, and the other with 64M tetrahedral elements. Both meshes were generated from a 64-volume geometric model, and contain entity sets for materials, Neumann boundary conditions, and geometric model entity groupings. Material and Neumann sets would typically be needed by any simulation application, while geometric model grouping sets would be necessary for applications performing adaptive mesh refinement. Performance measurements were made on two parallel computing platforms. The first, Intrepid, is an IBM BG/P at Argonne National Labortory; tests described in this paper use up to 16k processors on this machine, in "virtual node" mode, with each core having access to 512MB of memory. The second platform, Fusion, is a cluster of 320 nodes, each containing two 4-core Xeon processors, communicating over an Infiniband network. Each node has either 36 or 96 GB of memory.

D. Organization

This paper begins by describing optimizations of the parallel mesh read process implemented in MOAB in Section II. Section III describes two data structures important in the implementation of the algorithms in this paper. The resolution of inter-processor interface mesh, and exchange of ghost layers between processors, is described in Section IV. In Section V we describe modifications necessary to change the inter-processor interface resolution algorithm into one finding matching vertices based on geometric proximity instead of global id. Shared interface resolution for structured meshes is discussed in Section VI. Conclusions and items for future work are described in Section VII.

II. Unstructured Mesh Parallel I/O

We implement parallel I/O using the parallel HDF5 library[3], using a one-phase approach where all processors interact in parallel with a single file. The

specific layout of data in MOAB's HDF5 file is outside the scope of this paper, but is described in detail elsewhere [13].

Parallel input and output of unstructured mesh models requires more than simply reading and writing mesh nodes and element connectivity lists. Complicating factors include:

Identifying processor-resident entities: domain-decomposed codes typically assign highest-dimensional entities to processors, and store the partition as either an element-based field or as sets of those entities. In either case, the connectivity lists of those entities must be read to identify the vertices to be read on a processor.

Fragmentation and ordering: If a partitioning of elements over processors is stored as an annotation of the original mesh, with no mesh reordering, there is usually significant fragmentation of the element numbering space over processors; that is, a given processor will have in its partition many small groups of contiguously-numbered elements. Even if the elements and vertices are reordered to follow the partition, for unstructured meshes, the vertex numbering will still be somewhat fragmented. Specifying many small selections of HDF5 datasets to read and write has been identified as a problem by Kimpe et. al [9], and our experience bears that out as well. This would not be the case if mesh parts were stored one per file (though that approach would complicate the workflow of handling such models).

Boundary conditions and other groupings: Besides the vertices and elements, codes also need various lower-dimensional entities (i.e. edges and faces) and sets containing them for the purpose of defining boundary conditions, geometric model groupings, and other purposes.

These factors degrade the raw performance of mesh input when measured in terms of bandwidth, either because they involve work that is not I/O, or because they imply multiple read calls, some of which are smaller in size and therefore don't perform as well as large, single-call reads.

The parallel HDF5-based read process implemented in MOAB is broken down into the following steps:

1. Read header information
2. Read partition sets
3. Read element connectivity
4. Read vertex coordinates
5. Read/process lower-dimensional elements
6. Read/process sets and tags

In our efforts to optimize the parallel HDF5-based read operation, we first focused on ensuring that concurrent I/O, rather than independent I/O, was used at all times. This required ensuring that the various conditions which would cause HDF5 to fall back to independent I/O did not happen. These conditions, described in more detail in Ref.[14], include requiring data type conversion or transformation, existence of point selections for the read, and non-contiguous and non-chunked reads. After satisfying these conditions, the measured read times shown in Figure 1. (orig-hex and orig-tet) were obtained on Intrepid, for processor counts ranging between 16 and 16k. Next, two optimizations were implemented. First, the HDF5 library was modified to detect appending selections and avoid searching the selection list in those cases. This removed N^2 behavior in the selection code in HDF5. Second, the partitioner implemented with MOAB was modified to reorder elements and vertices according to their order in the partition using a simple greedy method. The times after these optimizations are indicated by the "select" and "reord" curves, respectively, in Figure 1.

The selection optimization had much greater influence at low processor counts, and was qualitatively of greater benefit for tetrahedral meshes than for hexahedral meshes. Both these factors can be explained by the degree of fragmentation in the selection list, which is greater at lower processor counts (due to the greater number of elements per processor in those cases) and for tetrahedral meshes in general (which typically have a factor of 4-6 greater number of elements than hexahedral meshes, for a given number of vertices).

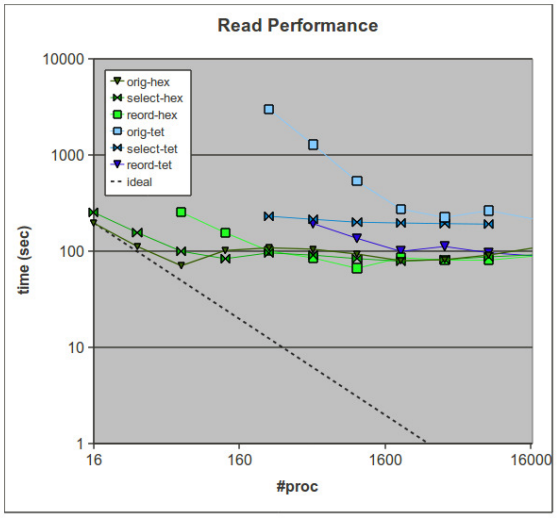

Figure 1. Read performance for 32M hex and 64M tet meshes. Original (no optimizations), select (improved selection mechanism), reord (after reordering).

The reordering optimization significantly improved read time for tet meshes, especially on larger numbers of processors, while for hex meshes resulted in relatively small improvements. We believe this difference is due mostly to the difference in number of elements sharing each vertex in tet versus hex meshes.

Overall, read times tend to flatten out above 128 and 2048 processors for hex and tet meshes, respectively, both to approximately 100 seconds. To assess further opportunities for optimization, it is useful to look at the distribution of time over the various read steps, shown in Figure 2. For datasets with the full complement of geometric topology sets (32m tet case), we observe that over 40% of the execution time is spent on reading and processing lower-dimensional entities, i.e. edges and triangles. These entities are the ones resolving various edges and faces in the original geometric model from

which the mesh was generated. If these entities and the corresponding sets are removed, we observe a dramatic reduction across all phases of the read operation (64m tet case). It is not clear why all phases of the read are improved, though it may have to do with synchronization required at the start of each dataset read, with fewer datasets in the file containing no geometric information. We are still investigating the reasons for this behavior.

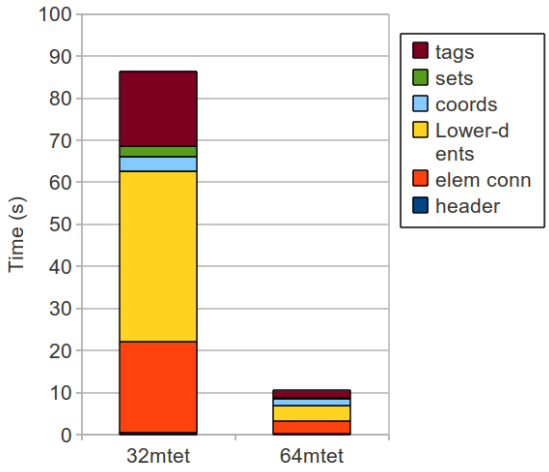

Figure 2. Distribution of read times, with (32m tet) and without (64m tet) geometry sets.

Both the 32M hex and 64M tet mesh files measure approximately 3GB in size, therefore the I/O throughput for reading and writing these meshes peaks at about 30MB/s on Intrepid. Removing the face and edge elements from the mesh results in a 2.7GB mesh file; the throughput for reading this mesh on Fusion is about 250MB/s. Again, we stress that these figures include work not normally included with I/O times (e.g. ghost exchange).

III. TUPLE LISTS AND CRYSTAL ROUTER

The algorithms described in this paper make frequent use of the two classes TupleList and CrystalRouter. TupleList stores *n* tuples, each tuple consisting of zero or more integer, long integer, unsigned long integer, and double datums, numbered zero to *(n-1)*. TupleList provides a function which sorts the tuples based on a specified datum index, using radix and quick sort algorithms for integer and double precision fields, respectively. The collective tuple_transfer() function communicates a TupleList to destination processors stored in a specified index of each tuple. When this function returns on the destination processor, that field holds the source processor for the tuple. This function greatly simplifies the routing of messages between processors.

CrystalRouter is a class for parallel communication first described by Fox [15]. At its core, CrystalRouter implements an optimized all-to-all, where message packets sent from *src* to *dest* are combined and routed across progressively smaller planes of processors, analogous to Gray code-based communication in hypercubes. CrystalRouter also provides the `gs_init()` function, which implements the capability to find shared elements of an integer space. On each processor, the CrystalRouter is given a collection of tuples *T*, and the index of the tuple datum representing the sort key. After `gs_init()` is called, a list of nonlocal compounds nl*(ind, np, T[np])* is returned, one compound per integer occurring on one or more other processors. For each compound, *ind* stores the index of the shared integer value in the original tuple list input to `gs_init`, *np* the number of other processors the integer is shared with, and *T[np]* the tuples from the sharing processors. `gs_init()` works by partitioning the range of integers over processors such that each processor is responsible for a portion of the range. One tuple transfer is done to send the input tuples to the responsible processors; the sort is done for a processor's range; then information on the shared tuples is sent back to the originating processors. Because the original tuples are returned in the *nonlocal* compound, the application can retrieve information about the remote copies of entities.

IV. SHARED INTERFACE RESOLUTION AND GHOST EXCHANGE

The starting point for shared interface resolution is after each processor has read and initialized its portion of the mesh (see Figure 7.)[1]. The overall strategy for initializing sharing data is to first identify vertices shared with other processors based on matching their global ids, then identifying shared non-vertex entities by matching their connectivity lists. Ghost entities are exchanged only after shared vertices and other entities have been identified. Although this algorithm is described with a 2D example mesh, we emphasize that it is implemented for 3D meshes, and 3D meshes are used to generate timing data reported below.

The algorithm for finding shared vertices is shown in Figure 4. When this algorithm completes, all vertices shared with all other processors are known, along with the handles of those vertices on those other processors. Also, based on the vertex sharing information, the processors communicating with the local processor are known. This list is kept in a data structure that also holds send and receive buffers for each of the communicating processors. Subsequent communication, for identifying shared non-vertex entities and exchanging ghost entities, consists only of point-to-point messages.

Sharing information is stored on entities in two tag types. If an entity is shared with only one other processor, the remote processor and handle are each stored in single-valued tags on the entity. For entities shared with more than one other processor, the remote *and* local processors/handles are stored on tags on the entity, with the owning processor/handle in the first position of these

[1] The tools for partitioning a mesh and reading it in parallel are not described here, but are available (see the MOAB wiki [13] for further information).

lists. Each entity is also marked with a single-byte tag, which stores its parallel status (e.g. if an entity is shared by one or multiple processors, if the entity is owned by the local processor, etc.)

The algorithm for finding shared non-vertex entities is shown in **Error! Reference source not found.** For each entity on the skin of the local mesh (step 1), the processors which might share the entity can be found by intersecting the sharing list for all the entity's vertices (step 2) After packing the connectivity for an entity into a message (step 4bi), those (local) vertex handles are replaced with their handles on the remote processor (using the sharing information resolved in the previous algorithm, step 4bii). Messages are exchanged asynchronously with the communicating processors.

An entity's vertices shared with another processor does not guarantee that entity is shared too. Also, we require the remote handles for entities actually shared by other processors. One way to resolve this issue would be to send potentially-shared non-vertex entities to the neithbor processor, and have that processor send back remote handles for the entities actually shared. However, that would require a sequence of two messages, which could not be passed concurrently. Instead, each processor sends the possible shared entities to the other. Those messages are received, and entity sharing information determined based on them. Since by definition each shared entity will be sent from each processor sharing that entity, every such processor is guaranteed to receive the shared entity from all other sharing processors. Furthermore, these messages can all be sent concurrently. In many cases these messages will be relatively small, and as stated earlier, they are point-to-point and involve a relatively small set of processors. We believe this to be the reason this algorithm shows good scaling behavior.

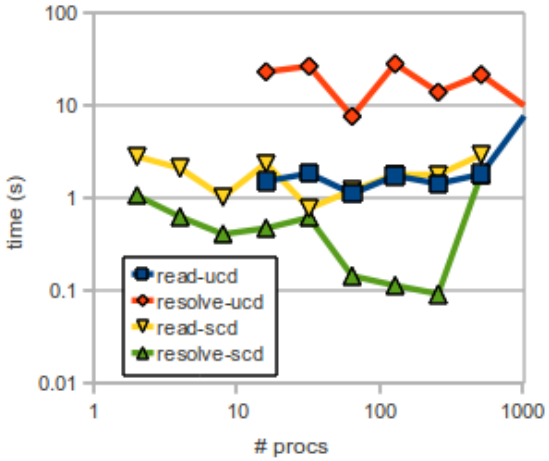

Figure 3. Execution time for parallel read and shared vertex resolution for sturctured mesh, using unstructured (ucd) and structured (scd) algorithms.

For brevity, we describe the algorithm in **Error! Reference source not found.**as sending each message using a single MPI function. However, this form of messaging may result in deadlock, if the receiving processor(s) have not allocated enough space for incoming messages. The algorithm as implemented in MOAB sends a first message of a known length, with the length of the total message as the first item in that message, then posts an asynchronous receive. The remote processor, after receiving that message and reading the overall length of the message, allocates the required buffer size, posts an asynchronous read for the remaining part of the message, then sends an acknowledgement to the sending processor. The sending processor, after receiving that acknowledgement, sends the remainder of the message, again using an asynchronous message. Currently, the length of the first fixed-size message is set to 1024 bytes. The result of resolving shared vertices and non-vertices is shown in Figure 9, with sharing lists indicated there.

The ghost exchange process is depicted in Figure 10. The algorithm for exchanging ghost elements is shown in Figure 5. . A key part of the algorithm, reading incoming ghost entities from another processor, is shown in Figure 6.

Several complications can be noted from Figure 6. First, for triangle *t2* sent from *P1* to *P0* and *P2*, some of the vertices supporting that entity must also be sent, while some are already present on the destination processor. For vertices already on the destination, the remote handle is substituted in the connectivity list in the message. New vertices from that message are referenced by entities later in the message using a handle consisting of the type MBMAXTYPE, and the id corresponding to the index of that new vertex in the entities contained in the message. This is depicted in step 4dii of Figure 5. and 3bi of Figure 6. Second, since we require every processor to know remote handles on all sharing processors, in the case of ghosting, the receiving processors must send the remote handles of entities to not only the sending processor, but to all other processors to which those same entities are sent. For example (see Figure 10), *P2* must send the local handles for *v6* and *t2* to both *P1* (the sending processor) and *P0*. Furthermore, in some cases the sending processor does not know the handle by which a new entity can be referenced on the receiving processor (e.g. *t2* in the *P2* to *P0* remote handle message). Instead, the handle on the owning processor is used (which is sent in the sharing list in the original message), with a negative processor number indicating this. Entities received as part of ghost exchange, and for which remote handles must be returned, are kept in the lists L1[p] (step 6).

The third complication in the ghost exchange process is that in some cases, ghost exchange results in some processors sharing entities (and therefore communicating together) that formerly did not. For example, in Figure 10, *P2* sends triangle *t1* to *P3*, and *P0* and *P1* must be informed of the remote handles for *t1* and its vertices on *P3*. This is accomplished as follows. The processor sending entities knows the other processors to which it sends a given entity, and includes all those processors in the sharing list as part of the message, with a zero handle for remote handles that are not yet known (step 4bi). By definition, these entities will be shared on more than two

processors, therefore the sharing list contains all remote processors/handles, the first of which is the owner. The receiving processor keeps a list of such entities (L2 in steps 3-4), to resolve the fourth complication (described next). The fourth complication is that a given processor may receive new entities from multiple processors, and should use the same (local) entity for them. Before creating a new entity, for entities whose eventual sharing list is longer than two processors, the owner handle is first checked against those in L2 (step 3c).

V. MESH JOINING

Large mesh models are often generated in pieces which are subsequently merged to form the overall model. The mesh merging step requires matching mesh vertices in the assembled model based on geometric proximity. A serial merge process requires that the whole assembled mesh be represented in a single process, which limits the overall size of the assembled mesh to what can be held in memory on a given machine. Parallel mesh merging removes this constraint, and in our implementation also significantly reduces the wall clock time for assembling the mesh.

Parallel mesh merging is easily supported by making a small modification to the vertex matching algorithm in Figure 4. First, instead of partitioning the global id space over processors, instead the geometric bounding box of all vertices is partitioned over processors, with each processor responsible for a distinct geometric region (plus a small epsilon layer whose thickness is twice the distance tolerance of the merge). The spatial extent covered by each processor can be computed deterministically based on the bounding box extent, the number of processors, and the merge tolerance. Each processor retrieves vertices on the skin of the local mesh, and performs a local merge on those vertices. Next, for the remaining skin vertices, each processor assembles a tuple list similar to that described in **Error! Reference source not found.**, except that instead of global ids, the tuples hold the (x, y, z) position of the vertex, and the destination processor is assigned according to the processor(s) responsible for that position in space[2].

The remaining parts of the algorithm in Figure 4. are the same, save for the matching process, which here works on spatial proximity rather than matched global ids. Note also that in contrast to the serial merge, merging a vertex with another on another processor does not result in one of those vertices being deleted; rather, the parallel sharing information is modified to indicate that they are the same logical vertex. The actual merging is done as part of the parallel write process described in Section II.

TABLE I. CPU TIMES (IN MINUTES) FOR MESH JOINING, COMPARED TO MESH COPY/MOVE TIMES, FOR TWO PROBLEMS.

	#procs	11M hex VHTR	58M hex VHTR
Copy/move	1	10.4	141.8
	8	5.8	59.6
	16	0.27	1.4
	32	0.036	0.12
	56	0.004	0.001
Join	1	3.81	70.48
	8	7.54	28.29
	16	6.7	17.62
	32	0.92	2.34
	56	0.25	0.8

The parallel mesh merge described above has been implemented in the Reactor Geometry Generator (RGG). A reactor core is constructed from fuel and other assemblies arranged in a rectangular or hexagonal lattice; typical cores use 10-20 different assembly types, with total assembly counts in the 100-1000 range. RGG generates mesh models for each assembly type, then uses a copy/move/merge process to assemble the overall core mesh. Meshes of over 50M hexahedral and tetrahedral elements have been produced by RGG. Clearly, the copy/move/merge process is a good candidate for parallel solution.

Timings for RGG are shown in TABLE I. for a Very High Temperature Reactor problem of two sizes, 11M and 58M hexahedral elements. Times for assembly mesh copy/move are given along with mesh join times, for comparison purposes. We note first that both the copy/move and join operations see super-linear speedups in some cases; memory usage data (not reported here) indicates that these steps are where the application goes from swapping to a state where the job fits in available memory. This indicates one important reason for parallelizing RGG, i.e. so the application can fit in memory without swapping. Next, mesh joining is observed to actually slow down going from one to eight processors; this is probably due to the communication overhead required in the parallel algorithm. However, at larger numbers of processors, the joining time is reduced far below the serial time.

VI. SHARED VERTEX RESOLUTION FOR STRUCTURED MESH

The algorithm in Figure 4. for finding vertices shared between processors applies equally well to structured meshes; however, a more efficient means for matching vertices is available that takes advantage of the structured mesh parameterization and requires far less communication.

First, if the vertices in a structured mesh block were created all at the same time for that block, the vertices on the skin of the block can be computed directly, given the block parameterization and the starting vertex handle for the block (these data are provided by MOAB's structured mesh interface [2]). Furthermore, if the parallel decomposition of the structured mesh is known on all processors, then the remote processors with whom a given processor shares mesh can be inferred from the parameterization. A processor need only find out the starting vertex handle on each communicating processor,

[2] Note that more than one processor may be responsible for a given point, if the point is located in the 2*epsilon-wide region on the box boundary.

after which it can compute the remote vertex handles directly, without further communication. Since handle computations are integer operations, they are quite fast, and the required communication (starting handles on each of the sharing processors) are short and point-to-point. In practice, directly computing sharing data is substantially faster than obtaining those data using the unstructured mesh-based methods described earlier.

MOAB reads parallel structured mesh using the pnetcdf library [8]. Four different decomposition strategies are supported, one of them 1D, and the remaining ones 2D. Execution times for shared vertex resolution using the unstructured and structured algorithms are shown in Figure 3. compared to read times. The structured algorithm improves resolve times by at least an order of magnitude in most cases.

VII. Conclusions

Input/output operations for structured and unstructured mesh models requires not only I/O for mesh vertices and elements, but also for intermediate edges, faces, and other metadata information. Initialization of a mesh model into its parallel domain-decomposed representation also typically requires identifying vertex and non-vertex entities on inter-processor interfaces, along with the other processors sharing those entities and their "handles" on remote processors. Exchanging one or more layers of ghost elements is also often required for simulation on a mesh. We describe the various algorithms used by the MOAB mesh library for performing parallel mesh I/O. Strong scaling results are given for shared interface resolution and ghost exchange which show good scalability out to 16k processors of an IBM BG/P. I/O times for large mesh models (32M hex and 64M tet mesh models are used) are around 100 seconds at 128 processors and above. We give timing data which show that these times may be drastically reduced by removing unused intermediate faces and edges, and unused sets containing them, from the data. We also describe how the shared vertex resolution algorithm can be modified to produce a mesh merging function, and show how this function supports the assembly of large lattice-based reactor core meshes.

Based on an informal survey of parallel application developers, we can say that the reported I/O times for these mesh models is adequately small, in an absolute sense. However, when measured in terms of bandwidth, we obtain I/O rates far lower than peak rates for this machine. Some of this is due to the non-I/O work required to initialize a parallel mesh. Still, we believe these I/O times could be further reduced. Two specific optimizations seem promising. First, as we have shown, simply removing unused intermediate faces and edges, and sets which contain them, will likely reduce I/O times substantially. We are in the process of testing this hypothesis for larger numbers of processors. Furthermore, we also conjecture that explicitly identifying these edges, faces, and sets, in the partition sets (which currently only store 3D elements in each part) will eliminate much of the non-I/O work currently performed in this process. We plan to implement this option into the partitioning tool currently used with MOAB.

More generally, we are also working with the developers of the HDF5 library to further improve concurrency in the HDF5 functions called by MOAB.

Acknowledgments

This work was supported in part by the U.S. Dept. of Energy Office of Nuclear Energy Nuclear Energy Advanced Modeling & Simulation (NEAMS) Program; by the U.S. Dept. of Energy Office of Scientific Computing Research, Office of Science; and by the US Department of Energy's Scientific Discovery through Advanced Computing program, under Contract DE-AC02-06CH11357.

References

[1] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst, "MOAB: A Mesh-Oriented Database," Sandia National Laboratories, SAND2004-1592, Apr. 2004.

[2] T. J. Tautges, "MOAB structured mesh interface," *MOAB structured mesh interface*. [Online]. Available: http://svn.mcs.anl.gov/repos/ITAPS/MOAB/trunk/src/moab/ScdInterface.hpp. [Accessed: 15-Sep-2011].

[3] "Hierarchical data format version 5," *The HDF Group*, 15-Sep-2011. [Online]. Available: http://www.hdfgroup.org/HDF5. [Accessed: 15-Sep-2011].

[4] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan Data Management Services for Parallel Dynamic Applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

[5] J. Fu, N. Liu, O. Sahni, K. E. Jansen, M. S. Shephard, and C. D. Carothers, "Scalable parallel I/O alternatives for massively parallel partitioned solver systems," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1 –8.

[6] W. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, p. 3.

[7] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, p. 40.

[8] J. Li, W. 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 *Proceedings of the 2003*

ACM/IEEE conference on Supercomputing, New York, NY, USA, 2003, p. 39–.

[9] D. Kimpe, A. Lani, T. Quintino, S. Vandewalle, S. Poedts, and H. Deconinck, "A study of real world I/O performance in parallel scientific computing," *Applied Parallel Computing. State of the Art in Scientific Computing*, pp. 871–881, 2010.

[10] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang, A. G. Sunderland, and J. C. Uribe, "Optimizing Code_Saturne computations on Petascale systems," *Computers & Fluids*, vol. 45, no. 1, pp. 103 – 108, 2011.

[11] E. Seegyoung Seol and M. Shephard, "Efficient distributed mesh data structure for parallel automated adaptive analysis," *Engineering with Computers*, vol. 22, no. 3, pp. 197–213, Dec. 2006.

[12] A. Baker, R. Falgout, and U. Yang, "An assumed partition algorithm for determining processor inter-communication," *Parallel Computing*, vol. 32, no. 5–6, pp. 394–414, Jun. 2006.

[13] T. J. Tautges, "MOAB Wiki." [Online]. Available: http://trac.mcs.anl.gov/projects/ITAPS/wiki/MOAB. [Accessed: 24-May-2009].

[14] J. Gruber, "RFC: Actual I/O Mode," The HDF Group, RFC THG 2011-08-04.v1, Aug. 2011.

[15] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, "Solving Problems on Concurrent Computers," *Prentice-Hall, Englewood Cliffs, New Jersey*, vol. 19, p. 88.

1. Find skin entities *se[d]*, *d* = 1..3
2. For each entity *e in se[d]*:
   2a. Get connectivity of *e* = *ce[j]* and sharing processor lists *p[c[j]][]*
   2b. Get intersection of all lists in *p*, *pe[]*
   2c. If *pe[]* is empty, continue to next entity
   2d. For each *p* in *pe[]*, add *e* to communication list *P[p]*
3. *m* = 0
4. For each sharing processor *p*
   4a. Post iRecv[*p*], *m*++
   4b. For each *e* in *P[p]*
      i. Add *e*, entity type *tp(e)*, *J*, (*ce[j]*, *j*=0..*J-1*) to *p* buffer *buff[p]*
      ii. Replace *ce[j]* with remote handles for those vertices on processor *p*
   4c. iSend message to *p*
5. While (*m*)
   5a. Get message, *m*—
   5b. Get *e*, *tp(e)*, *J*, *ce[j]*
   5c. Get entity *e'* of type *tp(e)* adjacent to *ce[j]*
   5d. If *e'*, add *p* to sharing data for *e'*

Figure 4. Algorithm for finding shared non-vertex entities.

1. For each communicating proc p, get vertices and entities to be sent to that proc, E[p]
2. m = 0
3. For each communicating partner p, post iRecv[p], m++
4. For each communicating partner p:
   4a. Pack number of entities E[p].size()
   4b. For e in E[p]:
      i. Pack sharing procs (including new procs), remote handles (zero for new procs), with owning proc/handle 1st entry
   4c. Pack # vertices in E[p], MBVERTEX, vertex coordinates
   4d. For each entity type of entities ET[p] in E[p]:
      i. Pack entity type
      ii. For each entity e in E[p]:
      - Get connectivity for ET[p], v[e]
      - For each v in v[e]:
        - If v is shared with p, substitute rh(v, p) for v
        - Else substitute HANDLE(MAX, i), i = index of v in E[p], and for procs p' sharing v, add p to extra procs pe[p']
5. For each communicating partner p:
   5a. Pack pe[p] onto sbuff[p]
   5b. iSend sbuff[p] to p
6. While (m):
   6a. Receive message from p; m—
   6b. Call recv_ents(p, rbuff[p], ep[], lh[p])
7. Add extra processors ep[] to communicating partners
8. For each communicating partner p:
   8a. Pack size of lh[p] in sbuff[p]
   8b. Post iRecv[p], m++
   8c. For e in lh[p], pack remote handle rh(e,p), e in sbuff[p]
   8d. iSend sbuff[p] to p
9. While (m):
   9a. Receive message from p; m—
   9b. Get number of entities i; for each i:
      i. Get local handle e, remote handle rh(e,p)
      ii. Add rh, p to sharing list for e

Figure 5. Agorithm for exchanging ghost entities.

For each entity type, number of vertices for that type, each entity of that type/nverts::
1. Unpack remote procs ps[ ], handles hs[ ]
2. If a vertex, unpack coordinates
3. Else
   3a. Unpack connectivity vs[ ]
   3b. For each handle vh in vs[ ]
      i. If type(vh) = MAX, then
      • idx = id(vh)
      • vh[ ] = new_ents[idx]
   3c. if hs[ ].size > 2 && L2[i].rh = hs[0] && L2[i].rp = ps[0], then
      i. new_h = L2[i].lh
   3d. else if there exists entity e with v[e] = vs[ ]
      i. new_h = e
4. If !new_h
   4a. Create new entity new_h
   4b. If hs[ ].size > 2
      i. L2[ ].rh = hs[0], L2[ ].ps = ps[0], L2[ ].lh = new_h
5. Update hs, ps tags on new_h
6. Foreach j in ps[j]:
   6a. If ps[j] = rank, continue
   6b. If idx of ps[j] in buf[ ] = L1.size, L1.resize[idx]
   6c. If new_h = L1[idx].lh[i]
      i. If L1[idx].rp[i] != -1 && hs[j]
      • L1[idx].rh[i] = hs[j]
      • L1[idx].rp[i] = -1
   6d. Else
      i. If !hs[j]
      • L1[idx].ps.append(ps[0]),

Figure 6. Algorithm for unpacking entities from a ghost exchange message.

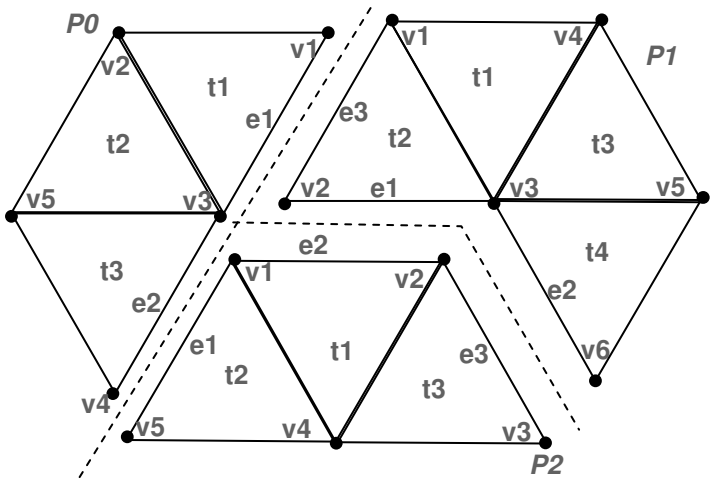

Figure 7. Mesh after initial partitioned read.

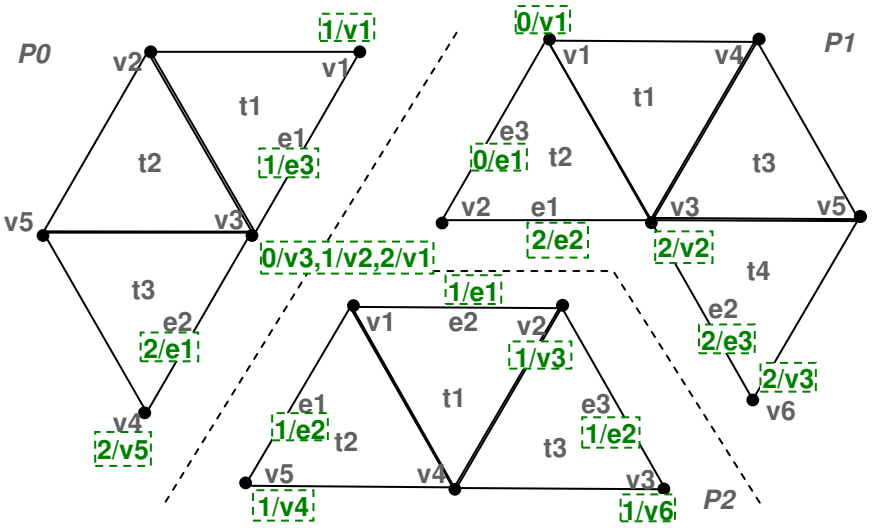

Figure 9. After shared interface resolution, with shared vertices/edges marked. Most shared entities are shared with only 1 other processor, and are marked only with the remote processor/handle. Vertex v3 on P0 is shared by all processors, and therefore local handle/processor appears in its sharing list.

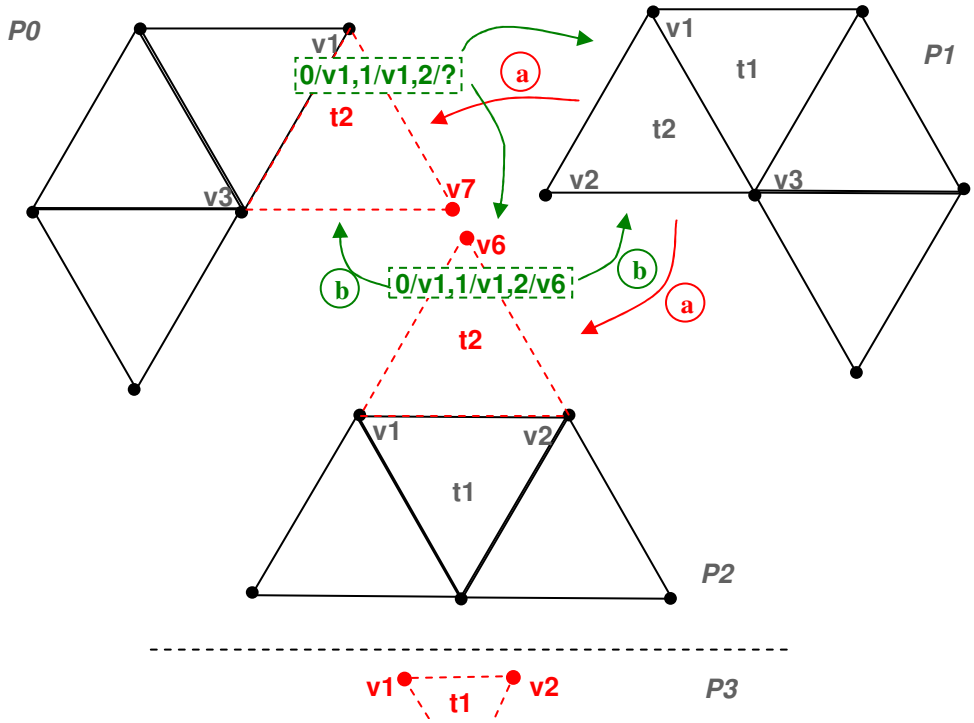

Figure 10. Ghost exchange, showing specific information for triangle t2 and its support. a) P1 sends quad t2 and vertex v1, along with sharing information 0/v1,1/v1,2/? for v1, to P0 and P2. b) P2 sending remote handles for v6 to P0 and P1. When P0 receives remote handles for v1, it replaces 2/? in sharing list with 2/v6 from message.