SANDIA REPORT

SAND2008-7881
Unlimited Release
Printed December 2008

Parallel Tetrahedral Mesh Refinement with MOAB

David Thompson, Philippe Pébay

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-7881
Unlimited Release
Printed December 2008

Parallel Tetrahedral Mesh Refinement with MOAB

David Thompson
Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
dcthomp@sandia.gov

Philippe Pébay
Sandia National Laboratories
M.S. 9159, P.O. Box 969
Livermore, CA 94551, U.S.A.
pppebay@sandia.gov

Abstract

In this report, we present the novel functionality of parallel tetrahedral mesh refinement which we have implemented in MOAB.

Acknowledgments

This work was supported by the ICO/MPO contract number T-00681-2 with Argonne National Laboratory (ANL). The authors thank Timothy Tautges (ANL) for having provided them with the opportunity to implement parallel tetrahedral mesh refinement in MOAB.

Contents

Appendix

This page intentionally left blank

1 Introduction

This report details work done to implement parallel, edge-based, tetrahedral refinement into MOAB. The theoretical basis for this work is contained in [PT04, PT05, TP06] while information on design, performance, and operation specific to MOAB are contained herein. As MOAB is intended mainly for use in pre-processing and simulation (as opposed to the post-processing bent of previous papers), the primary use case is different: rather than refining elements with non-linear basis functions, the goal is to increase the number of degrees of freedom in some region in order to more accurately represent the solution to some system of equations that cannot be solved analytically. Also, MOAB has a unique mesh representation which impacts the algorithm.

This introduction contains a brief review of streaming edge-based tetrahedral refinement. The remainder of the report is broken into three sections: design and implementation (§2), performance (§3), and conclusions (§4). Appendix A contains instructions for end users (simulation authors) on how to employ the refiner.

1.1 Streaming edge-based tetrahedral refinement

Consider a mesh M composed of an array $P = \{x_1, x_2, \ldots, x_p\}, x_i \in \mathbb{R}^3$ of nodal coordinates and an array of elements E, where each entry is some specification of the element type plus an array of integer indices into P that define the topological corners of an element. Here we are only interested in meshes that are simplicial complexes, so the element type is implied by the number of indices into P used to define the corners of the element. Table 1 contains an example set of nodal coordinates P while Table 2 contains an example array of simplicial elements E.

Table 1. An example of a nodal coordinate array P for a 3-dimensional mesh.

ID_P	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0	1	0
4	0	0	1
5	1	0	1
6	1	1	1
7	0	1	1

In addition to an input mesh M_i, a mesh refinement scheme based on subdividing elements (as opposed to excising a region from M_i and remeshing it) requires an indicator function. An *indicator function* (or *characteristic function*) is an injective map χ from some set of interest S to the set $\{0,1\}$. As explained in [PT05], we will be performing edge-based subdivision, so the domain of this map will be the set of edges of M_i. Entries of S that χ maps to 1 will be considered "marked" for

Table 2. An example of an element array E defined on the array P as specified in Table 1. Elements 0–4 are tetrahedra while elements 5–8 are triangles.

ID_E	i	j	k	ℓ
0	0	1	2	5
1	0	2	3	7
2	0	2	7	5
3	0	5	7	4
4	2	7	5	6
5	0	3	7	
6	0	7	4	
7	1	5	6	
8	1	6	2	

subdivision, while entries of S that χ maps to 0 must remain whole as the input mesh is partitioned into output simplices.

It is important for mesh refiners to produce conforming output meshes when given a conforming input M_i. A conforming mesh M is one where no topological corner points of M are interior to any element of M or any of the element's lower-dimensional boundaries. Whatever scheme an edge-based subdivision technique uses to generate an array T_o of output tetrahedra from an input tetrahedron T_i plus the values of χ on the edges of T_i, it must produce triangular boundaries that match results from input tetrahedra that neighbor T_i or the result will not be a conforming output mesh M_o. This can be accomplished in several ways, which include:

1. by matching boundaries on faces of T_i that have already been refined and imposing constraints on neighbors of T_i that have not been processed;
2. by performing multiple passes, imposing new constraints from neighbors and adding new constraints to neighbors until the boundaries are matched; or
3. by choosing a scheme that will automatically produce conforming outputs given conforming inputs.

The scheme outlined in [PT04] is of the latter class and uses comparisons of edge lengths on boundaries to choose conforming triangulations of the boundaries. When edge lengths cannot produce conclusive results, an additional topological corner node is added to the output point set P_o such that the boundary triangulation is symmetric and thus identical to neighbors.

Note that in order for a scheme to automatically produce conforming outputs given conforming inputs in a streaming fashion, the evaluation of χ on some edge e must not vary depending on what elements in M_i have boundary e. Otherwise, the edges of M_i would have to be enumerated and the values of χ on each edge would have to be stored before refinement took place. This would violate the reduced memory and/or communication cost that streaming provides.

2 Design & Implementation

MOAB is a mesh-oriented database aimed at mesh representation throughout the lifecycle of a mesh: generation, pre-processing, simulation, and even post-processing [TMM+04]. It aims to leave a small memory and algorithm-complexity footprint and operate on parallel, distributed-memory meshes as required by modern high performance computing platforms. To implement streaming, edge-based refinement in MOAB, we need to accommodate these goals.

2.1 Input and output

Access to a MOAB mesh is provided through the `MBInterface` class, which is abstract. A concrete implementation is provided by `MBCore`. In MOAB, no distinction is made between nodes, elements, and sets. All of these things are simply *entities* to MOAB, although the storage for each particular type of entity is separate and the integer *handles* used to identify entities of a single type are largely contiguous. Because entity handles assigned to consecutively created elements of the same type are themselves consecutive, one way that MOAB reduces storage cost is through *ranges*. Instead of storing a sequence of handles directly, only the first and last handles need to be stored. Where entries of an otherwise contiguous sequence are missing, an array of intervals is created. For example, the sequence of handles $\{1,2,\ldots,500,502,505,506,507\}$ can be represented as a range $R = \{[1,500],[502,502],[505,507]\}$. An `MBRange` R can be thought of as an array of intervals which is kept ordered and does not allow duplicate entries.

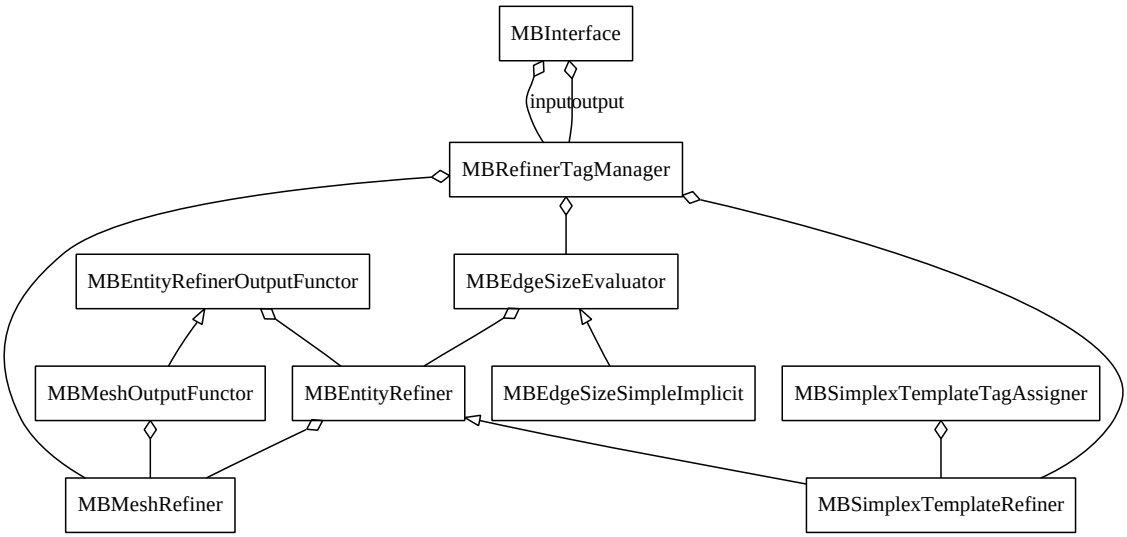


Figure 1. Classes and interactions for the MOAB simplicial refiner.

The introduction treated refinement as a process that takes a mesh M_i and indicator function χ as input and generates an output mesh M_o as a result. In MOAB things are slightly more complex as

some simulations may wish to use the same mesh for both input and output. To accommodate this use case, mesh refinement will take as input M_i, R_i, M_o, and χ (where R_i is an `MBRange` of input simplices) and produce as output R_o (an `MBRange` of output simplices)[1]. M_i and M_o may be identical. An abstract refiner class `MBMeshRefiner` abstracts the process that maps $(M_i, R_i, \chi, M_o) \rightarrow (M_o, R_o)$. The meshes M_i and M_o are passed as pointers to `MBInterface` objects, the ranges are `MBRange` objects, and χ must be a pointer to a concrete subclass of `MBEdgeSizeEvaluator` (which is itself an abstract class). `MBEdgeSizeSimpleImplicit` is an example implementation of an edge size evaluator. The `MBMeshOutputFunctor` class is used to insert new elements into the output mesh independent of any particular refinement scheme in use.

Figure 1 is a diagram of the refiner classes. Users will mostly interact with the `MBMeshRefiner` class which iterates over all of the requested entities in R_i and handles the parallel communication required after a refinement pass. As each entity is encountered during this iteration, a subclass of `MBEntityRefiner` is invoked to evaluate edges and subdivide elements as required. The `MBSimplexTemplateRefiner` is a concrete implementation that embodies the logic described in [PT04]. As edges and elements are subdivided, an `MBEntityRefinerOutputFunctor` is used to process the results. The concrete subclass `MBMeshOutputFunctor` stores the results in M_o. This set of classes should provide flexibility to implement other types of refinement while maintaining a single interface for the user and meeting the goals set forth for MOAB at the beginning of this section.

2.2 Tags

Any entity in MOAB may be *tagged* with scalar or vector values. The tag values may be any primitive C++ type (e.g., `double`, `int`, or `char`). Because mesh refinement introduces new nodes along existing edges of the input mesh and new elements to replace input tetrahedra with edges marked for subdivision, some way to interpolate existing tag values to these new nodes and elements is required. The refinement process uses a `MBRefinerTagManager` class to

- map tag identifiers from the input mesh to the output mesh,
- maintain a list of tags the user has requested be transferred from the input to the output, and
- maintain tags required for representing distributed-memory meshes.

2.3 Partitions and interfaces

Distributed-memory meshes in MOAB use tagged sets to represent portions of mesh owned by or shared with other processes. Specifically, MOAB has a concept of *partitions* and *interfaces*. MOAB partitioning draws upon the concept of covering in mathematics. Recall that a cover $C = \{c_i\}$ of set X is a set of subsets of X such that $\bigcup_i c_i \supset X$. Each c_i represents the portion of a mesh

[1] New nodes (0-simplices) are not added to the output.

stored on some process. A MOAB *interface* between processes i and j ($i \neq j$) is the intersection of their entries in the cover: $I_{ij} = c_i \bigcap c_j$. An interface may be shared with any number of processes: e.g., $I_{ijk\ell}$ is an interface shared by 4 processes; note that, in particular, $I_{ij} \supseteq I_{ijk\ell}$. Later we will be interested in non-empty interfaces that do not contain any subsets that are also interfaces; let us call these interfaces *minimal interfaces*. For some mesh on N_p processes a minimal interface is denoted $\tilde{I}_\alpha$ where $\alpha \in A \subseteq 2^{N_p}$. The set of all minimal interfaces $\tilde{I}_A$ is an exact proper cover of the union of all interfaces.

A MOAB *partition* p_i on process i is the subset of c_i which process i is said to own. Only one process may own a given entity and every entity must be owned, so that $P = \{p_i\}_i$ is an exact proper minimal cover of X. Note that if the concept of an interface above is extended to include elements shared with no other processes, then each process i that owns elements not shared with other processes will have $I_i = \tilde{I}_i \neq$. We may then write

$$p_i = \bigcup_{\alpha \in A_i} \tilde{I}_\alpha.$$

where A_i is the subset of A whose entries contain i. Distributed-memory meshes must have – on each process – a set for each minimal interface. Each of those sets must be tagged with the list α of processes share the entities it contains. The `MBRefinerTagManager::set_sharing` member is used to mark new nodes and elements with the list of sharing processes. When an edge is marked for subdivision, a new node will be created. The new node will be shared with the intersection of the list of processes sharing its endpoints. Similarly, a new node placed internal to an input tetrahedron's face to will be shared with the intersection of the processes sharings each of its topological corners. Elements (edges, triangles, and tetrahedra) inherit their list of sharing processes from the input element which they subdivide. Ownership is always set to be the lowest-numbered process in the list of sharing processes.

2.4 Global IDs

As simulations and other tools add or set new tag values on entities in a distributed mesh, it is important for entities shared with other processes to have consistent tag values. This requires a global numbering scheme so that individual entities represented on multiple processes may be named. MOAB defines a special integer tag for global IDs. No two entities of the same type may have the same global ID tag value[2]. Global IDs need not be contiguous. The refiner is responsible for assigning global IDs to new entities. This is accomplished by maintaining a set of newly-created entities contained in each minimal interface $\tilde{I}_\alpha$ during streaming subdivision. By the definition of a minimal interface, no entry in $\tilde{I}_\alpha$ is shared with any process other than those in α. Thus the owner of all entities in $\tilde{I}_\alpha$ is $\min\{\alpha\}$. After subdivision, `MBMeshOutputFunctor::assign_global_ids` is called. An all-gather of the number of minimal interfaces present on each process is performed, followed by an all-gather of the α for each of those interfaces and the number of elements contained in each $\tilde{I}_\alpha$. While theoretically this could incur a significant communication overhead (since $A \subseteq 2^{N_p}$ grows very large with only modest increases in N_p), we rely on the fact that parallel mesh

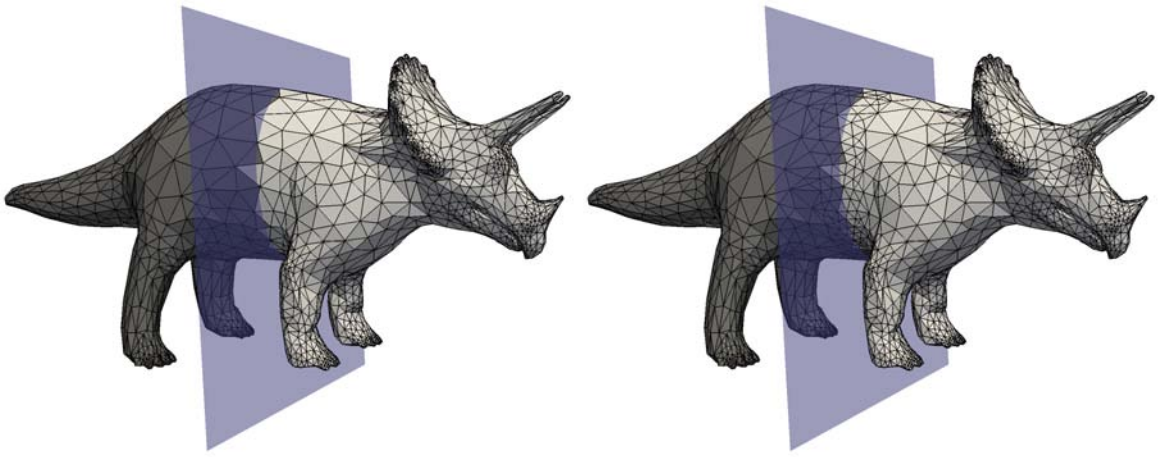
[2]MOAB allows entities of different types (e.g., vertices and hexahedra) to have the same global ID.

distributions typically minimize the number of processes that must communicate and thus keep $|A|$ small. Global IDs for new entities are assigned starting with the largest global ID present in the mesh before refinement plus 1, $G_{max}+1$. The lowest process ($i=0$) assigns $[G_{max}+1, G_{max}+|\tilde{I}_0|]$ to the entries of $\tilde{I}_0$ followed by all the other non-empty interface $\tilde{I}_\alpha, \alpha \in A_0$. The α are sorted so that the order of assignment of global IDs is unique. Then, the next lowest process ($i=1$) assigns global IDs starting with $G_{max}+1+|\bigcup_{\alpha\in A_0}\tilde{I}_\alpha|$. No communication of global IDs is required since each process knows the cardinality of each minimal interface across all processes and the ordering of entities within each minimal interface on every process is deterministically sorted by the global IDs of its vertices (or of the vertex's "parent" endpoint global IDs when the vertex is created during subdivision).

3 Performance

A true measure of the mesh refiner's performance cannot be made without some driving application. Instead, what this section attempts to characterize is the rate at which single-pass refinement can be accomplished in MOAB (where overheads due to entity creation and tag storage differ from previous implementations of the refinement scheme, such as in [TP06] where optimal parallel scale-up was demonstrated for load-balanced inputs) as well as some measure of the communication overhead required for global ID assignment.

The mesh used to generate timings in this section is shown in Figure 2. The indicator function χ used evaluates a simple implicit distance function that returns 1 when an edge midpoint is near (compared to the edge length) to a plane. In this case, the plane is $x = 0$.



(a) Coarse (b) Refined

Figure 2. The input mesh M_i used for the timing study. The transparent blue plane shows the implicit surface used to mark edges for subdivision. The light tetrahedra are on process 0; the dark tetrahedra are on process 1.

The input mesh M_i is partitioned so that 1, 2, 4, or 8 processes own some elements. No attempt was made to balance the number of elements per process or the number of refined elements per process; the number of input tetrahedra for each of the 4 runs is shown in Figure 3. The number of output tetrahedra per process is similar to the number of input tetrahedra and so no figure is presented. The raw data is included in Appendix B, along with the timing results for 5 replicates of each run.

The speed at which processes in the various runs were able to process tetrahedra was highly variable, ranging from less than 1 tetrahedron per millisecond to hundreds of tetrahedra per millisecond. However, when plotted against the total number of input tetrahedra visited by the process (Figure 4) or the total number of output tetrahedra produced by the process (Figure 5), a pattern

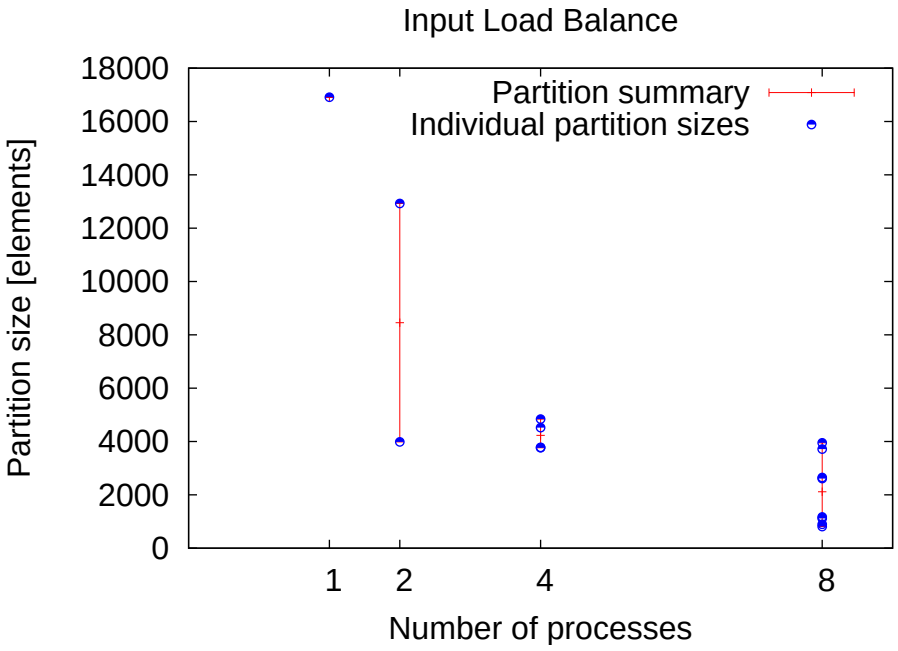


Figure 3. This plot shows the size of each process' input partition for runs with 1, 2, 4, and 8 processes. Figure 2 illustrates the subdivision for the 2-process run. The $x = 0$, $y = 0$, and $z = 0$ half-spaces are used to partition the mesh.

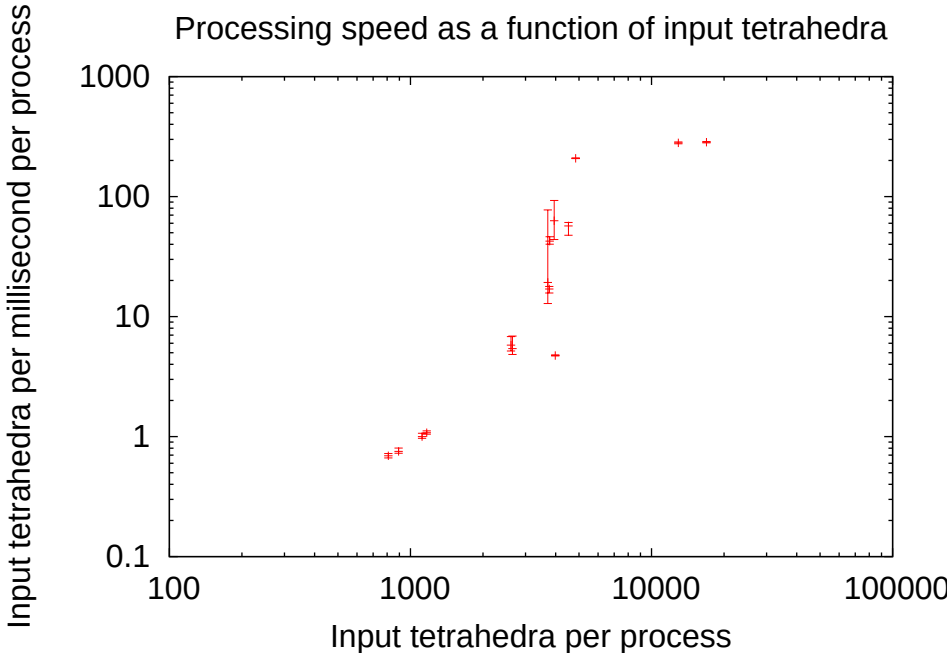


Figure 4. This plot shows the rate at which each process in the various jobs was able to perform subdivision on its input tetrahedra.

becomes evident. The overhead involved in filling instruction and data caches, interprocess communication, etc. is significant when there are less than approximately 6,000 input tetrahedra or 11,000 output tetrahedra per process.

There are many other factors that should be studied if time permits, including:

- the number of element and vertex tag values being copied and interpolated to the output,
- the number of minimal interfaces present on each process, and
- the effect of different edge size evaluators with a range of compute intensities and memory access patterns.

The test data here represent runs with minimal tag values, a small number of interfaces present, and a simple calculation for edge size evaluation.

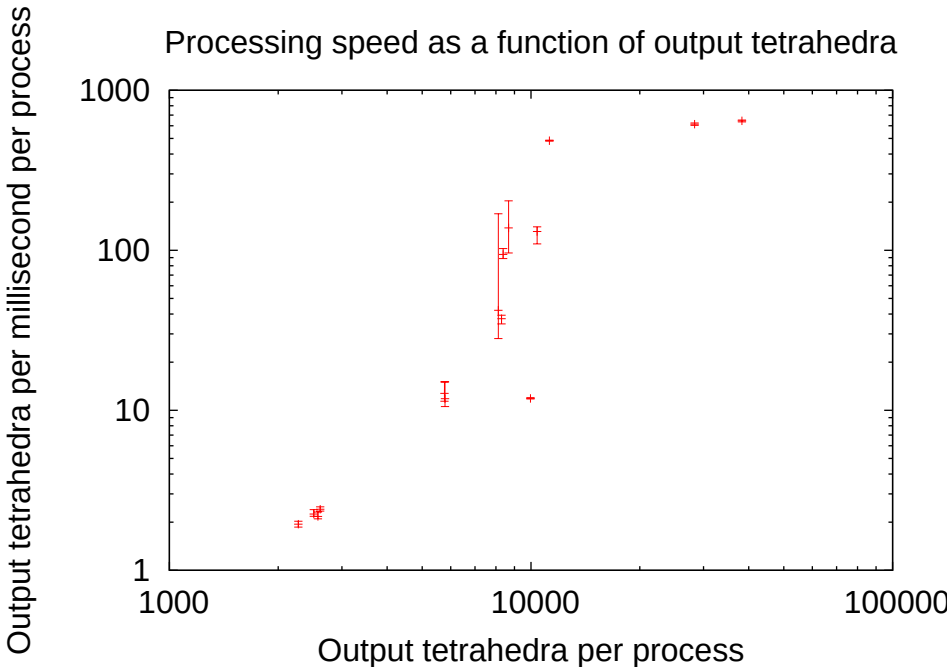


Figure 5. This plot shows the rate at which each process in the various jobs was able to produce output tetrahedra.

4 Conclusion

We have presented a framework for parallel mesh refinement and an embodiment that performs edge-based subdivision of simplicial complexes. The performance on small datasets has been characterized. Efficiency in parallel will clearly depend on the number of entities in R_i on each process, the number of entities in R_o on each process, and the particular edge size evaluator chosen. Other factors may affect performance as the scale is increased. In particular, the effect of the number of minimal interfaces present on each process has not been characterized. However, we expect this to vary from application to application and any scaling studies should include a particular simulation or mesh processing task or risk irrelevance. Any mesh redistribution performed by the simulation will significantly impact the performance of the refiner and this is something best left to a specific application rather than included in refinement.

4.1 Known issues and further development

While the refiner works, there are some known issues that we hope to resolve.

1. Occasionally the simplicial subdivision produces elements that are non-conforming. We have not been able to reliably reproduce the conditions where this occurs on a small test mesh but it does not appear related to inter-process mesh boundaries.
2. The remote handles of entities in mesh interfaces are not currently exchanged among sharing processes.
3. The interface for supporting multiple refinement passes needs testing.

Beyond these issues there are opportunities for further development that we would like to explore:

1. Implementing streaming hexahedral refinement using the `MBEntityRefiner` interface.
2. Exploring the use of a single, extremely coarse mesh that is refined each time step (or every few time steps) in order to provide a mesh of suitable resolution. This would be a way to effectively coarsen a mesh without the overhead of a non-local scheme (a typical requirement of mesh coarsening).
3. Investigating alternative edge size evaluation techniques in tandem with a real-world simulation in order to develop techniques reduce the frequency at which refinement must occur and still provide both speedy time step advances (i.e., few superfluous degrees of freedom) and the required accuracy (i.e., adequate degrees of freedom near features of interest). We expect that inference techniques employing information from current and previous time steps as well as boundary/initial conditions and geometric mesh quality could provide performance gains over other adaptive techniques.

References

[PT04] Philippe P. Pébay and David Thompson. Parallel mesh refinement without communication. In *IMR13: Proceedings of the 13th International Meshing Roundtable*, pages 437–448, September 2004.

[PT05] Philippe P. Pébay and David Thompson. Communication-free streaming mesh refinement. *Journal of Computing and Information Science in Engineering*, 5(4):309–316, 2005. Special Issue on Mesh-based Geometry.

[TMM+04] Timothy J. Tautges, Ray Meyers, Kerl Merkley, Clint Stimpson, and Corey Ernst. MOAB: A mesh-oriented database. Technical Report SAND2004-1592, Sandia National Laboratories, April 2004.

[TP06] David Thompson and Philippe P. Pébay. Embarrassingly parallel streaming mesh refinement. *Engineering With Computers*, 22(2):75–93, August 2006. DOI 10.1007/s00366-006-0020-3.

A A Brief User's Manual

The code fragment in Table A.1 provides an example of how to use the refiner.

Table A.1. A small example of mesh refinement.

```
#include "refiner/MBMeshRefiner.hpp"

                                    ⋮

// We assume you have mesh pointers...
MBInterface* Mi, Mo;
// ... and that the meshes have some tags you
// would like to interpolate to new nodes/elements.
MBTag VT0, VT1, ET;
// Create a refiner:
MBMeshRefiner refiner( Mi, Mo );
// Pass tag handles obtained from the INPUT mesh:
if ( Mi->tag_get_handle( "example0", VT0 ) == MB_SUCCESS )
  refiner.add_vertex_tag( VT0 );
if ( Mi->tag_get_handle( "example1", VT1 ) == MB_SUCCESS )
  refiner.add_vertex_tag( VT1 );
if ( Mi->tag_get_handle( "example2", ET ) == MB_SUCCESS )
  refiner.add_element_tag( ET );
// If the tags do not exist on Mo, they will be created.
// Now, perform the refinement:
refiner.refine( Ri, Ro );
```

Note that by default, the `MBMeshRefiner` will use an `MBSimplexTemplateRefiner` instance for the entity refiner and an `MBEdgeSizeSimpleImplicit` instance for the edge size evaluator. You do not need to create new instances if these are adequate for your purposes. Most likely, you will wish to create your own subclass of the `MBEdgeSizeEvaluator` class used to decide which edges of the mesh should be subdivided (by implementing an evaluation of some indicator function χ). To do this, you need only implement a single pure-virtual method:

```
virtual bool evaluate_edge(
  const double* p0, const void* t0,
  double* p1, void* t1,
  const double* p2, const void* t2 );
```

where `p0` and `p2` are pointers to nodal coordinates of the edge endpoints, `p1` points to the coordinates of a new node that may be created (set to the edge midpoint by default but may be modified

inside `evaluate_edge`), `t0` and `t2` are pointers to nodal tag values at the edge endpoints, and `t1` points to tag values at the new node that may be created (initialized with the average value of the endpoint tag values as appropriate and writable should `p1` be modified). The new node specified by `p1` and `t1` will only be created if `evaluate_edge` returns true. The pointers to node coordinates `p0`, `p1`, and `p2` *always* refer to an array of 6 double values. The first 3 are parametric coordinates and the following 3 are nodal coordinates of the nodes in question. The parametric coordinates are with respect to the element currently being refined but you should not rely on the element as a source of information for the return value of `evaluate_edge` since χ must return the same value for all elements sharing a given edge. When the MOAB mesh problem dimension is set to 2, only the first 2 values of the parametric and nodal coordinates are significant but the offset to nodal coordinates will always be 3 (for simplicity). When implementing `evaluate_edge`, you may use the `MBEdgeSizeEvaluator`'s `tag_manager` to determine the tag handles, their sizes, and their offsets in order to modify `t1`.

B Raw Timing Data

Table B.2. The number of input tetrahedra per process.

Process ID							
0	1	2	3	4	5	6	7
16910							
3987	12923						
3766	4522	3780	4842				
1117	2649	809	3713	1168	2612	893	3949

Table B.3. The number of output tetrahedra per process.

Process ID							
0	1	2	3	4	5	6	7
38307							
9970	28337						
8290	10395	8376	11246				
2509	5781	2272	8123	2613	5763	2576	8670

Table B.4. Average refinement time per process in milliseconds.
5 runs were performed for each of the four partitions.

Process ID							
0	1	2	3	4	5	6	7
59.46							
60.36							
59.56							
59.26							
58.74							
847.02	45.301						
845.77	46.789						
830.29	46.975						
836.01	45.970						
844.32	45.645						
216.52	76.540	94.29	23.194				
239.24	94.855	93.57	23.269				
228.38	76.041	91.77	23.064				
211.21	75.150	83.06	23.377				
214.52	74.201	81.74	23.217				
1152.88	547.892	1219.78	289.286	1114.87	505.890	1176.29	42.564
1075.79	506.019	1120.75	192.514	1047.49	456.977	1113.61	49.088
1151.10	513.554	1181.29	261.896	1110.65	461.521	1204.61	61.390
1153.69	384.380	1178.12	47.981	1080.95	452.016	1228.26	90.079
1047.78	495.517	1145.22	172.338	1067.17	383.234	1207.90	71.014

DISTRIBUTION:

2 Tim Tautges
1500 Engineering Dr
Madison, WI 53706, U.S.A.

1 MS 9159 Philippe P. Pébay, 8963

1 MS 9159 David Thompson, 8963

1 MS 9159 Neal Fornaciari, 8963

2 MS 9018 Central Technical Files, 8944

1 MS 0899 Technical Library, 9536


Sandia National Laboratories