Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 00 (2016) 000–000

Procedia Engineering

www.elsevier.com/locate/procedia

25th International Meshing Roundtable (IMR25)

Mesh adaptation for moving objects on shared memory hardware

Dan Ibanez[a,b,*], Mark Shephard[a]

[a] *Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY, United States*
[b] *Sandia National Laboratories, Albuquerque, NM, United States*

Abstract

We present a general algorithm for scheduling the application of cavity-based mesh modifications which enables parallel mesh adaptation on shared-memory parallel hardware including accelerators such as GPUs. We demonstrate this method on a mesh with objects moving in the fluid, and performance is studied on hardware including an Intel Knights Landing processor and an NVidia K80 GPU. Our code ports seamlessly between these architectures and has no architecture-specific algorithms.

Peer-review under responsibility of organizing committee of the 25th International Meshing Roundtable (IMR25).

Keywords: Mesh Adaptation; Cavity-based primitives; Parallel meshing; GPU; Shared memory

1. Introduction

Current leadership-class supercomputer acquisitions have accelerator hardware providing a significant portion of their computational power [1]. The nature of mesh adaptation involves random memory accesses and more integer arithmetic than floating point, which makes it somewhat misaligned with the principal design goals of recent architectures. Despite this, we have developed a method of applying mesh modifications that makes complete use of array-based shared-memory parallelism on-node, and still couples to MPI-based parallelism. This research note will focus on the on-node parallelism, while future publications may address the hybrid coupling with MPI.

We achieve a fully portable single-source code with the help of the Kokkos framework [2]. Our code is open-source software available at `https://github.com/ibaned/omega_h2`

2. Modification operators

Mesh adaptation in our work refers to applying local mesh modifications which maintain a conformal mesh but alter its connectivity. In our work we choose a minimal subset consisting of three operators:

1. Edge split: a new vertex is placed at the center of an old edge, all entities adjacent to the old edge are split in half.

* Corresponding author. Tel.: +1-518-227-8857
E-mail address: daibane@sandia.gov

Peer-review under responsibility of organizing committee of the 25th International Meshing Roundtable (IMR25).

2. Edge collapse: an old vertex (a) is collapsed across an old edge onto the other endpoint (b). All entities adjacent to the old edge are destroyed, while all other entities adjacent to a are deformed (they connect to b now instead of a).
3. Edge swap: all entities adjacent to an old edge are removed, and the cavity is re-meshed such that this old edge is not recreated.

There are a couple fundamental properties we require from a mesh modification operator in this particular implementation:

1. It must not modify the boundary of its cavity at all. This property is the basis for parallelism, allowing two adjacent but non-overlapping cavities to be operated on simultaneously by different threads.
2. Its cavity must be the elements adjacent to a central "key" entity. Having this property allows straightforward parallelization.

3. Overall adaptation steps

The modification operators described in Section 2 are applied in a steps, where each step consists of applying the same kind of operator many times to the mesh with a focus on a particular goal. The steps, in turn, are composed of mesh rebuilds or passes that each apply a set of non-overlapping mesh modifications. A step repeatedly executes the same pass until that pass does not modify the mesh. Here we describe, for each step, the logic involved in one pass of that step:

1. Refinement: Edges greater than $3/2$ their desired length are marked as candidates for splitting. All candidates have their cavity qualities predicted and an independent set of edges is selected for actual splitting.
2. Coarsening: Edges less than $1/2$ their desired length are marked as available to collapse in both directions. At each vertex adjacent to a marked edge, the best quality edge collapse involving that vertex is chosen, others are discarded. Finally, an independent set of these vertices is selected to collapse.
3. Swapping: Existing elements with low quality are identified, and four layers of elements around them are also marked. Then, all edges of marked elements become candidates for swapping. Any proposed swaps which do not *locally* improve quality are discarded. Following this, and independent set of edges is selected for swapping.
4. Coarsening for Shape Correction: Similar to swapping, elements around slivers are identified and all vertices of those elements are candidates for collapsing. Execution continues as for coarsening, with the added requirement that quality must be locally improved.

Each pass computes an independent set of cavities to modify as described in Section 4, and then rebuilds the mesh accounting for those modifications as will be described in Section 5

The second goal of adaptation (besides satisfying element size requirements) is to output all elements with quality greater than or equal to a user-provided "good quality" parameter, as described in the Swapping and Coarsening for Shape Correction steps. We use the mean ratio quality measure [3,4] for tetrahedra:

$$\left(15552 \cdot V^2 \cdot \left(\sum_{i=1}^{6} (l_i)^2\right)^{-3}\right)^{\frac{1}{3}} \tag{1}$$

Where V is the tetrahedron's volume and l_i is the length of its ith edge.

4. Independent set selection

At each pass during mesh adaptation, we select a set of mesh modifications whose affected cavities do not overlap (do not share elements) to apply. This allows us to execute each modification using fine-grained parallelism. Such an approach was suggested for GPU use by Pande et al. [5], although their implementation computed the independent

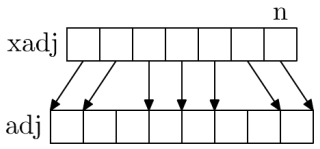

Fig. 1. Compressed row storage for graphs and upward adjacencies

set on the CPU. It is also already used in MPI-only adaptation codes [6] during coarsening to prevent a chain of overlapping edge collapses from removing too many mesh elements.

This is a graph independent set problem, with a graph where graph nodes are possible mesh modifications and graph edges represent an overlap between their cavities.

In 1986, Luby presented a highly parallelizable algorithm for finding maximal independent sets of graphs [7]. The structure of Luby's algorithm is iterative, and at each iteration graph nodes which are local maxima of some function are added to the independent set. In Luby's original algorithm, the function is actually random numbers at each graph node (whose values change at each iteration). We would like to resolve conflicts in a way that prefers "better" mesh modifications, which in this case is defined by output quality. So, instead of local maxima of random numbers, we find local maxima of output quality.

The proof for the time complexity of Luby's original algorithm relied on probability theory and changing the graph node numbers at each iteration [7]. In our algorithm, the number of iterations is bounded by the length of the longest path in the conflict graph whose nodes have monotonic quality values. In all the meshes we have encountered, the algorithm converges in no more than ten iterations.

We have either vertices or edges as the "key" entities, and either triangles or tetrahedra for elements. We can quickly construct the graph of keys that are adjacent to a common element, which represents all potential conflicts. At the beginning of each pass, each key entity is annotated as either being a candidate or not, and candidates are annotated with their output cavity quality.

All of our graphs are stored in compressed row format, which is used by many sparse linear solvers and by the METIS partitioner [8]. Figure 1 shows a compressed row graph with `n` graph nodes. The top array (`xadj`) maps each graph node to a spot in the bottom array (`adj`) where its adjacent graph nodes are listed. This deals with the fact that the amount of data per graph node is not constant.

5. Mesh rebuilding

We accept as input to each pass in adaptation a mesh defined by the simple topological arrays. Our implementation uses array structures similar to those of the original MOAB mesh database [9], and also of several solvers which implement their own structures. We assume that either all elements are tetrahedra or they are all triangles. This allows us to store downward adjacencies in a single array that can be thought of as two-dimensional. Upward adjacencies (e.g. vertices to elements) are stored in the compressed row format used for graphs, see Figure 1. This format deals with the fact that upward adjacency degrees are not constant for all entities.

A rebuild pass does the following:

1. Derive additional information about the input mesh, augmenting it with arrays that describe adjacencies, entities of interest, element qualities, etc.
2. Cavities to modify are selected, numbered, and output mesh size is computed.
3. Allocate and fill the output mesh arrays.
4. Deallocate the input mesh arrays.

The first arrays rebuilt are entity topology, listing the adjacent vertices of each output entity. After the new topology is built, entity information including simulation field data, boundary classification, and coordinates can also be rebuilt. Each of these transfers follows a similar pattern of copying the data for entities that exist in both meshes (in parallel) and then deriving the new cavity interior data (in parallel).

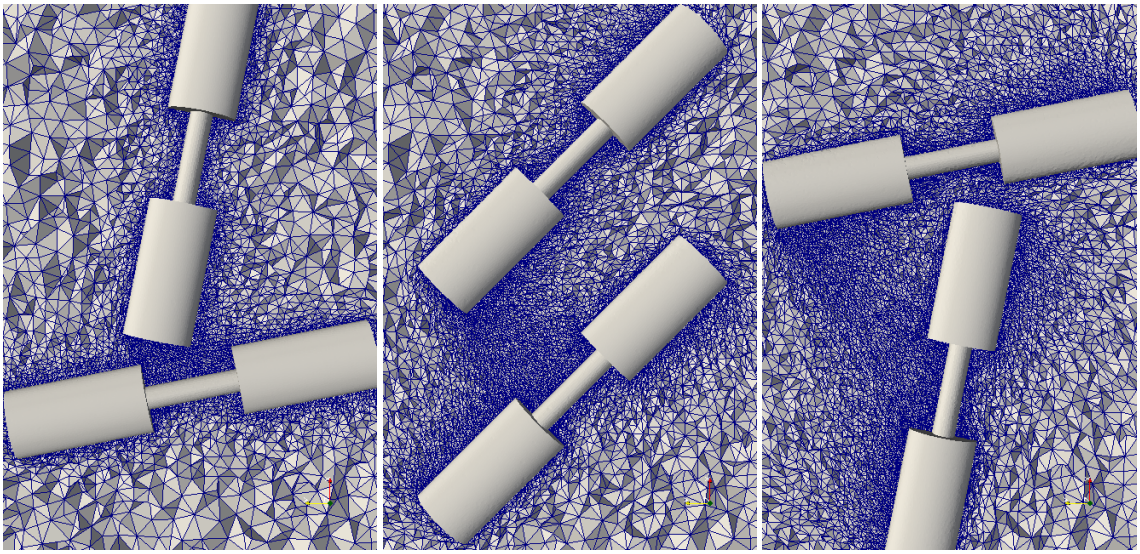

Fig. 2. Cutaway mesh views at steps 2, 8, and 14 of 16

Table 1. Runtime in minutes on different hardware

	number of OpenMP threads							
Hardware	N/A	1	2	4	8	16	32	64
Intel Xeon 2620 v4		203	115	76	55			
Intel Knights Landing		1196	598	299	153	79	42	24
NVidia K80	35							
NVidia GTX 980 Ti	15							

Of key interest is the number of mesh rebuilds required in practice, which determines how acceptable this approach is in terms of performance. Typically, a pass such as coarsening or refinement will handle most of its target cavities in the first rebuild, and subsequent rebuilds handle fewer leftover cavities. In all of our tests (see Section 6), we see ten to twenty rebuilds for each mesh adaptation call.

6. Results

In our demonstration case, we focus on the potential for mesh adaptation to support simulations that have 3D solid bodies moving through a fluid. Mesh adaptation can be used to adjust connectivity between iterations of mesh motion, preventing tangling and inversion. Several other researchers have made good progress in applying general adaptation for these purposes [10–13]. In our case we created a geometry consisting of two “rotors” whose ranges of motion overlap. This geometry is meant to be similar to simulations of interest such as helicopter rotors and manufacturing processes in which a fluid product is stirred.

The case is executed in a series of 16 time steps which each do the following:

1. A velocity field is prescribed for each object as vectors at mesh nodes.
2. This velocity field is spread onto the surrounding fluid mesh nodes by solving Laplace’s equation using the objects and domain boundary as Dirichlet conditions.
3. The mesh is moved according to the velocity field and stops right before any element goes below 20% mean ratio quality.
4. Mesh adaptation is applied to the deformed mesh to recover edge lengths and ensure all elements are above 30% mean ratio quality. Mesh adaptation will rebuild the mesh about ten times when applied.

5. Steps 2 to 4 are repeated for any remaining motion that would previously have inverted elements. This repetition occurs ten to twenty times per time step.

We generated the initial mesh using Gmsh [14] with optimization by Netgen to remove sliver elements. All subsequent motion and adaptation is handled by our code. Figure 2 shows cutaway views of the mesh after certain time steps. The number of elements increases from one million to two million from start to finish.

Table 1 shows the runtime performance across different hardware. We first run on an Intel Xeon processor typical of current servers and clusters. Then, we run the same case on two pieces of hardware found on current supercomputers: the Intel Knights Landing CPU and the NVidia Tesla K80 GPU. Finally, we also run on a more consumer-market GPU, the NVidia GTX 980 Ti. Unlike CPUs, GPUs do not offer clear controls for using a subset of threads, so it is typical to simply show GPU speedup versus serial (in this case, 7X for the GTX 980 Ti) as opposed to some kind of scaling on the GPU. The Xeon 2620 has 8 cores and the Knights Landing CPU has 64 cores, and we see decent scaling until the number of OpenMP threads equals the number of cores, on both CPUs.

7. Closing Remarks

The acquisition of heterogeneous supercomputers, especially those with GPUs, motivated a re-evaluation of how mesh adaptation is programmed. We have presented an alternative basis for ordering mesh modifications for parallelism, which is implemented portably for shared-memory and accelerator environments (one source code for all on-node hardware). We present an evolving-geometry demonstration of performance using a variety of representative node hardware.

Acknowledgements

This research was supported U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC00066117 (FASTMath SciDAC Institute).

References

[1] F. Wang, C.-Q. Yang, Y.-F. Du, J. Chen, H.-Z. Yi, W.-X. Xu, Optimizing linpack benchmark on gpu-accelerated petascale supercomputer, Journal of Computer Science and Technology 26 (2011) 854–865.

[2] H. C. Edwards, C. R. Trott, Kokkos: Enabling performance portability across manycore architectures, in: Extreme Scaling Workshop (XSW), 2013, IEEE, 2013, pp. 18–24.

[3] A. Liu, B. Joe, Relationship between tetrahedron shape measures, BIT Numerical Mathematics 34 (1994) 268–287.

[4] A. Loseille, V. Menier, F. Alauzet, Parallel generation of large-size adapted meshes, Procedia Engineering 124 (2015) 57–69.

[5] S. Pande, S. Biswas, A. De, Gpu-based parallel algorithms for delaunay mesh refinement, in: Proceedings of the 24th international meshing roundtable, 2015.

[6] H. De Cougny, M. S. Shephard, Parallel refinement and coarsening of tetrahedral meshes, International Journal for Numerical Methods in Engineering 46 (1999) 1101–1125.

[7] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM journal on computing 15 (1986) 1036–1053.

[8] G. Karypis, V. Kumar, MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, `http://www.cs.umn.edu/~metis`, 2009.

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

[10] G. Compere, J.-F. Remacle, J. Jansson, J. Hoffman, A mesh adaptation framework for dealing with large deforming meshes, International journal for numerical methods in engineering 82 (2010) 843–867.

[11] M. Wicke, D. Ritchie, B. M. Klingner, S. Burke, J. R. Shewchuk, J. F. O'Brien, Dynamic local remeshing for elastoplastic simulation, in: ACM Transactions on graphics (TOG), volume 29, ACM, 2010, p. 49.

[12] P. Clausen, M. Wicke, J. R. Shewchuk, J. F. O'brien, Simulating liquids and solid-liquid interactions with lagrangian meshes, ACM Transactions on Graphics (TOG) 32 (2013) 17.

[13] J. Chen, S. Li, J. Zheng, Y. Zheng, Parallel local remeshing for moving body applications, in: Proceedings of the 24th international meshing roundtable, 2015.

[14] C. Geuzaine, J.-F. Remacle, Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities, International Journal for Numerical Methods in Engineering 79 (2009) 1309–1331.