Efficient Finite Element Mesh Mapping Using Octree Indexing

Omar Adalat and Daniele Scrimieri

Department of Computer Science, University of Bradford, Bradford BD7 1DP, UK
{o.j.adalat,d.scrimieri}@bradford.ac.uk

Abstract. Modern manufacturing involves multiple stages of complex process chains where Finite Element Analysis is frequently used as a simulation method on a discretized mesh to provide an accurate estimation of factors such as stresses, strains, and displacements. The choice of the most suitable element type and density is dependent on the individual manufacturing process or treatment applied at each stage of the process chain. To map between unalike Finite Element meshes, differing in density and/or element type, an Octree spatial index was evaluated as a solution for highly performant mapping. Compared to existing solutions, the Octree spatial index introduces parallelism within index creation and provides a strategy to perform the most complex interpolation technique, Element Shape Function, in a more computationally efficient manner.

Keywords: octree, spatial indexing, finite element analysis, mesh mapping, process chain, parallel processing

1 Introduction

Present-day manufacturing involves multiple intricate successive stages of process chains. Finite Element Analysis (FEA) provides an accurate simulation for problems such as structural analysis, heat conduction, and fluid flow [13], where these simulations can be applied during the design phase of the product, allowing product designers to avoid incurring costs for defective prototypes. From one stage to the next, the most appropriate mesh density and element type is determined by the particular manufacturing process or treatment the product undergoes. Therefore, a need for mapping Finite Element data between differing meshes arises.

Afazov et al. developed the open-source software FEDES (the Finite Element Data Exchange System) [2] which is used to transfer Finite Element data across meshes and integrates with multiple different commercial Finite Element solvers such as ABAQUS, ANSYS, and MORFEO. Four mapping techniques are introduced within FEDES, namely: Nearest Point Method, Distance Method using Field of Points, Distance Method using Elements, and Element Shape Function. However, without a spatial index to accompany these interpolations, the time taken becomes impractical for any form of realistic usage, particularly where non-elementary and larger meshes are concerned.

Previously, grid-based indexing has been implemented for Finite Element mesh mapping within FEDES where buckets for a given point can be accessed in constant time [10], which was later adapted into a scale-based approach to tackle excessive grid refinements [11]. FEDES has seen applications in various areas, such as additive manufacturing [1].

Our approach in this paper explores the usage of an Octree for transferring Finite Element data. Octrees were first proposed by Meagher [7], an Octree is an 8-ary tree for recursive spatial subdivisions of a 3-dimensional space using Z-ordering [8]. An Octree is an extension of a Quadtree, a 4-ary tree, and its 2-dimensional analogue. More generally, Octrees have seen prior use to perform Finite Element mesh generation and simulations [4], but not for transferring Finite Element data. We begin with introducing the Octree index and the interpolations. Then, we provide experimental evaluation against grid-based indexing, where we find that the Octree index significantly outperforms the grid index for element indexing and mapping using the most accurate interpolation technique, Element Shape Function.

2 Octree construction

2.1 Node indexing

To index points from a point cloud $\rho = \{p_1, p_2, ..., p_n\}$ where $p_i \in \mathbb{R}^3$ using an Octree, the index is recursively traversed using Z-Ordering to determine which tree leaf node contains the node (hereafter all nodes are referred to as **octants**). Z-Ordering, also known as Morton Encoding [8], transforms a 3-dimensional value, a point, into a 1-dimensional value (the child octant index to insert into) using the space that the node in the Octree represents. For example, if p_i has x, y, and z values that are all greater than the space's origin, the index returned is 7. In the inverse case, 0 is retrieved.

Our variant of a pointer Octree observes the following properties:

1. Any data pertaining to points is only stored within leaf octants. Branch or interior octants are only used during tree traversal.
2. An octant either has no children or 8 children, allowing a constant-time check to determine whether a given node is a leaf or not.

Each octant stores its minimum bounding hexahedron (MBH) or axis-aligned bounding box (AABB) as two vectors, **aabbMin**, and **aabbMax**. The root node of the tree is calculated by determining the minimum and maximum values for each dimension of the mesh. At the initial time of index creation, before points are inserted, the root octant uniformly represents the entire space within the mesh since no subdivisions have occurred.

The minimum and maximum dimension values of the mesh can also be optionally located in parallel within our implementation. Each thread is designated a subset of points to process and returns its local minimum and maximum. The main thread then determines the global minimum and maximum values from the local results.

After the root octant is constructed, points are inserted by recursively locating the leaf node that contains the point using Z-Ordering. Points themselves are not inserted, but references (indexes that map to points inside of the nodes array) are stored. This saves memory usage and speeds up the subdivision process as no large objects are shifted around.

A naïve implementation of an Octree may recursively subdivide and split each leaf octant as soon as there is more than one point within the octant. This proves problematic primarily for two reasons. Firstly, since an octant implicitly represents a minimum bounding hexahedron, eventually the machine floating-point epsilon is hit. Secondly, boundless recursion without any termination condition will typically cause a stack overflow once a large enough mesh is used, as the stack has limited space.

As a solution to the aforementioned problems, we introduce two parameters for the Octree index:

- **Maximum depth**. Starting from the root octant (which is at $depth = 0$), no further subdivisions will occur once the maximum depth is reached.
- **Leaf split threshold**. A subdivision at a leaf will be prevented from occurring until a certain number of points are within the leaf, preventing the index from becoming sparsely distributed.

Since the Octree is depth-limited, the number of nodes is now maximally bound by $\sum_{i=0}^{Depth} 8^i$ although the leaf split threshold also contributes to controlling the number of nodes. Destructing the index and deallocating memory uses iterative post-order traversal with two stacks.

2.2 Element indexing

Creating an index on the elements of a mesh is accomplished by first indexing the mesh's nodes. After this, the array in FEDES containing the element-to-node mapping is reversed, which then allows a node's elements to be looked up when processing a node during searches. This differs from the approach of grid-based indexing where all of each element's nodes are indexed and a reference is added to the element, resulting in nodes being processed multiple times since nodes have a many-to-many relationship with elements, as a node may belong to multiple elements.

3 Interpolations

Four different mapping techniques are provided that act upon the **nodal** or **integration points** in the target mesh. The mapping methods are executed in parallel by distributing the total number of nodal or integration points amongst a number of threads.

Integration points are calculated by finding the center or average of each element within the mesh. Whether nodal or integration points are used throughout

the mapping is dependent on the type of Finite Element data being mapped. Displacements and forces are mapped at nodal points whereas stresses and strains are mapped via integration points.

3.1 Nearest Point Method

Mapping using the Nearest Point Method is the simplest approach and provides accurate results when mapping between fine to coarse meshes. Each nodal or integration point from the target mesh searches for the node in the source mesh with the minimum Euclidean distance, and then completely copies the Finite Element data associated with that node.

Algorithm 1: NearestNode

```
Data: octree, point
Result: node index
best distance squared ← ∞;
best node;
Enqueue root octant into priority queue pq;
while MinDist(pq's top node, point) < best distance squared and pq ¬empty
 do
    octant ← Dequeue top of pq;
    if octant is a leaf then
        forall points ∈ octant do
            if distance squared < best distance squared then
                best node ← point;
                best distance squared ← distance squared;
            end
        end
    else
        Enqueue all of octant's children;
end
return best node
```

Algorithm 2: MinDist

Data: *octant*, *point p*
Result: *minimum distance squared*
$dx \leftarrow max((p_x - aabbMin_x),\ 0,\ (aabbMax_x - p_x))$;
$dy \leftarrow max((p_y - aabbMin_y),\ 0,\ (aabbMax_y - p_y))$;
$dz \leftarrow max((p_z - aabbMin_z),\ 0,\ (aabbMax_z - p_z))$;
return $dx \cdot dx + dy \cdot dy + dz \cdot dz$

Algorithm 1 uses a best-first search search [9] to find the nearest node and Algorithm 2 calculates the minimum distance between a point and an AABB. Performing this interpolation technique using an Octree is carried out by placing the root octant into a priority queue and the top octant of the priority queue is

accessed within a loop so long as the associated top node may replace the current best distance. The priority queue implementation uses a min-heap, therefore the top octant of the priority queue can be accessed in constant time, and pushing can be performed in $O(log(n))$. The ordering of the octants within the priority queue uses Algorithm 2.

The octant that corresponds to where a target nodal or integration point would be contained will have a minimum distance of 0 using Algorithm 2, hence it will be the first leaf searched.

Note that distance squared is used in Algorithm 1 because the resulting Euclidean distance is not used elsewhere throughout this interpolation. Omitting costly square root operations allows a speedup to be gained.

The average time complexity to find the nearest point is $O(log_8(n))$ and in the worst case is $O(n)$ although the datasets for which the latter case exists are theoretical.

3.2 Distance Method using Field of Points

The Distance Method using Field of Points involves extending the search from finding the single nearest node to finding the nearest node in each of a number of quadrants. In 2D, these quadrants are north-west, north-east, south-east, and south-west, where the target mesh's nodal or integration point is used as the origin. In 3D, there are 8 quadrants (similar to the extension between Quadtrees and Octrees).

For each of the 8 quadrants located, a proportional distance coefficient is calculated using the following formula per quadrant:

$$k_i = \frac{\sum_{j=1}^{8} d_j}{d_i} \tag{1}$$

where d_i and d_j denote distances between the node and nodal or integration point. Thereafter, the mapped data at the node or integration point using each quadrant is given by

$$\sum_{i=1}^{8} F_i \cdot \frac{k_i}{\sum_{j=1}^{8} k_j} \tag{2}$$

where F_i represents the Finite Element data at that given quadrant's node.

To perform this interpolation using an Octree, a radius search is used, where all octants within a specified radius are scanned. For each node within the query sphere, it is determined which quadrant the node belongs to using Z-Ordering and the quadrant's best distance and best node are updated if applicable.

Algorithm 3 uses an approach based on calculating the Minkowski sum [3] and seeks to quickly eliminate octants that are outside the specified radius. Starting from the root octant, octants are processed recursively to determine whether

they are within the query sphere and if so their points are added to the result set.

Whilst using a radius search means that it is possible that some quadrants are not found during the search, these quadrants are always located at a significant distance. As a result, such quadrants would be assigned a low weighting using the proportional distance coefficient regardless.

It is also possible that if a target mesh nodal or integration point is located near the outer boundaries of the source mesh, certain quadrants do not exist at all. In any case, whenever a quadrant is not found, mapping is performed only using the quadrants that are found. Another special case that exists is where a node in the source mesh coincides with a target mesh nodal or integration point, in which case one of the distance values will be 0, causing a *NaN* value to emanate when divisions occur to calculate coefficients. To resolve this coincident case, the data associated with the coinciding node is copied instead.

Algorithm 3: WithinSphere

Data: *octant*, *point p*, *radius*
Result: whether the given *octant* is inside the search sphere
Vector *center* ← *octant*'s center;
Vector *extent* ← half of *octant*'s AABB (length, width, height);
Vector *relative* ← $(|\, p_x - center_x|, |\, p_y - center_y|, |\, p_z - center_z|)$;
Vector *max* ← $(radius + extent_x,\ radius + extent_y,\ radius + extent_z)$;
if $(relative_x > max_x \ or\ relative_y > max_y \ or\ relative_z > max_z)$ **then**
  return *False*
end
if $(relative_x < extent_x \ or\ relative_y < extent_y \ or\ relative_z < extent_z)$ **then**
  return *True*
end
distance squared ← $EuclideanDistanceSquared(extent,\ relative)$;
return *distance squared* < *radius squared*

3.3 Distance Method using Elements

To perform this interpolation, for each nodal or integration point in the target mesh, the source mesh element that minimizes average distance is selected. The minimum average distance is defined as $\frac{\sum_{i=1}^{k} d_i}{k}$ where k is the number of nodes in the element, d_i is the Euclidean distance to the target nodal or integration point. In the case where a node coincides between two meshes, that node's Finite Element data is transferred instead.

For each node in the selected element, a proportional distance is calculated

$$k_i = \frac{\sum_{j=1}^{m} d_j}{d_i} \tag{3}$$

where m denotes the number of elements in the node and d_i and d_j represent Euclidean distances from the nodal or integration point to the element node.

After calculating the coefficients for the element, the mapping is performed via the following equation:

$$\sum_{i=1}^{m} F_i \cdot \frac{k_i}{\sum_{j=1}^{m} k_j} \tag{4}$$

where F_i represents the Finite Element data at that given element's node and m is the number of elements.

Using an Octree, Distance Method using Elements is similar to the Nearest Point Method where the priority queue approach is used. However, the loop is terminated once a number of elements, *minimum element scans*, have been scanned or the priority queue is empty. It is likely that the first leaf scanned by the priority queue contains the element that minimizes average distance, but adjacent octants may have an element with a lower average distance, therefore they are also searched. The parameter *minimum element scans* should be set such that it accounts for the difference in density between the meshes.

3.4 Element Shape Function

The Element Shape Function is the most accurate mapping method, including being suitable for mapping between coarse to refined meshes. Mapping results are visualised for an aero-engine vane component in Figure 1. For each target mesh nodal or integration node, it involves identifying the element from the source mesh that satisfies a number of linear equations dependent on the element types. In essence, the equations initially test for the element that contains them. These equations are all solved using Newton–Raphson method. An element with m nodes has its values v_n mapped by [2]:

$$v_n = \sum_{i=1}^{m} s_{m_i} F_{m_i} \tag{5}$$

Where s_{m_i} is the element shape function, determined by source mesh element type for node m_i and F_{m_i} is the Finite Element field variable value at node m_i.

Performing this search using an Octree involves enqueueing the root node into a priority queue and terminating the search once the element that satisfies the linear equations is found. Due to the fact that the geometries between the two meshes may differ, the target mesh may have a node or integration point that falls outside the source mesh. In this case, the search tolerance is incrementally modified and the search is restarted.

In the grid-based index, when a nodal or integration point falls outside the source mesh, a search occurs using the entire index to attempt to identify the correct element prior to adjusting the tolerance. These cases occupy the majority of the runtime. Searching the entire index is substantially inefficient and unnecessary since the nature of spatial indexes and expansive searches dictate that it becomes increasingly unlikely to find the target element as the search progresses. Therefore, we define a parameter, *max leaf scans* which stops and

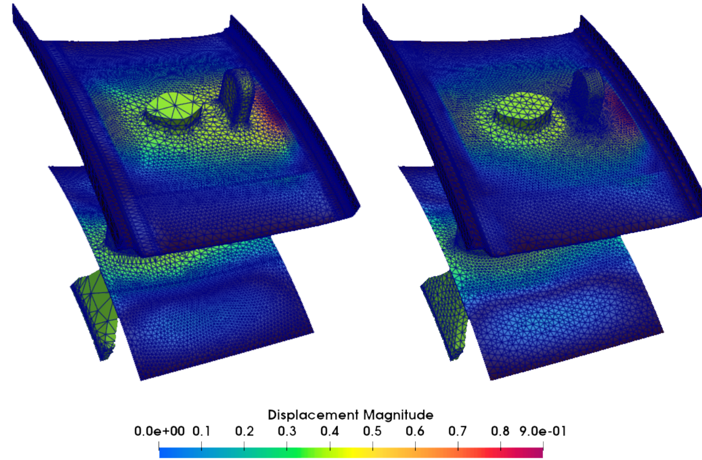


Fig. 1: Mapping displacement data from a source mesh (left) with 171,188 tetrahedron elements to a target mesh (right) with 125,361 tetrahedron elements using Element Shape Function.

readjusts the search tolerance earlier, after a certain number of leaves have been searched instead of a full exhaustive search.

The parameter *max leaf scans* must be selected with diligence. Predominantly, it is determined by the two other Octree parameters that have been selected, *maximum depth* and the *leaf split threshold*, but the maximum number of leaves to scan can also vary based on the type of element and the mesh. If a value that is too low is chosen, the search will be stopped abruptly when furthering the search would have found the desired element at the correct tolerance. Principally, the parameter *max leaf scans* is exceptionally useful when a prototype of a mesh is undergoing multiple iterations for quick and accurate mesh mapping.

4 Experimental evaluation

Experiments were written and run for the following:

- Index creation and mapping using the Octree index
- Index creation and mapping using the grid-based indexing from FEDES version 2

Empirical results were gathered using an AMD Ryzen 7 5800H which has 8 cores and 16 threads, with a base clock speed of 3.2GHz and maximum boost clock speed of 4.4GHz, and 16GB of RAM. All mapping was performed in parallel using 16 threads.

In this section, evaluation is performed using two examples where displacement and stress are the Finite Element field variables mapped.

- Mesh A of 54,128 nodes and 171,188 elements with an element type of linear tetrahedron to Mesh A' with 35,617 nodes and 125,361 elements with an element type of linear tetrahedron.
- Mesh B of 24,543 nodes and 83,316 elements with an element type of linear tetrahedron to Mesh B' with 41,637 nodes and 154,407 elements with an element type of linear tetrahedron.

All times were measured using wall clock time in terms of milliseconds. For the Octree index, the C++ Google Benchmark library was used with MSVC x64 v19.32.31332 using wall clock times averaged over 256 iterations. For the grid-based indexing and sequential searching, the function *Now* was used from the Pascal standard library with Free Pascal Compiler x64 v3.2.2 using wall clock times averaged over 10 iterations. The source code for the Octree index can be found at https://github.com/nightly/fedes and the grid-based index is available at https://github.com/dscrimieri/FEDES.

4.1 Parameters

The parameters used for the Octree were maximum depth of 15 and leaf split threshold of 5 as this set of parameters generalized well to all meshes. The grid-based indexing used an initial grid size of $50 \times 50 \times 50$ and bucket size of 60.

4.2 Indexing

Consistent with other solutions, the indexing time is ultimately negligible when compared to search times. Since the Octree doesn't process nodes multiple times when indexing elements unlike the grid-based indexing, the time taken for indexing is faster and eliminates redundant $O(log_8(n))$ operations that would otherwise be incurred if an approach similar to the grid-based indexing was used. Indexing as a whole is faster using an Octree due to not having to consider grid refinements nor computationally expensive array reallocations, as can be seen from Table 1.

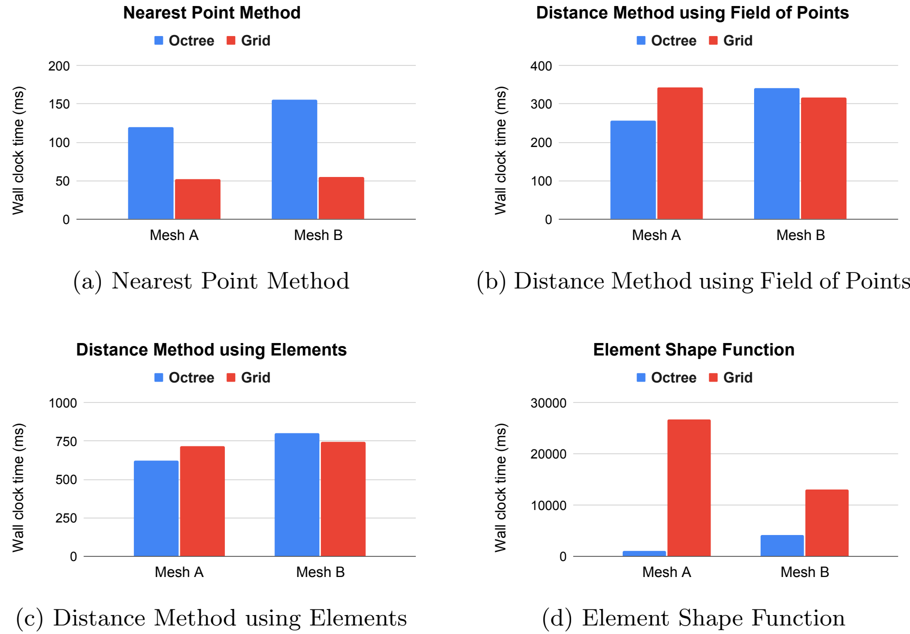
Table 1: Evaluation of wall clock times in milliseconds using the three implemented indexing techniques.

	Mesh A → Mesh A'		**Mesh B → Mesh B'**	
	Nodes	Elements	Nodes	Elements
Octree	16.3	57.7	6.47	25.2
Octree with Parallel Root Construction	15.6	57.1	5.56	24
Grid	22	920	15	302

Parallelization of the root node construction is marginally faster than sequential root construction. The majority of index creation time is occupied by insertions and traversals. However, larger meshes may observe greater speedups with parallelization of root node construction.

4.3 Interpolations

Figure 2 shows the timing results of interpolations using the Octree spatial index and the grid-based spatial index. For Distance Method using Field of Points, the radius was set to 10.00. Distance Method using Elements using an Octree index achieves similar performance to grid-based indexing, using the parameter *minimum element scans* = 50. Element Shape Function is vastly quicker using the Octree indexing for mesh A, as mesh A' contains a large number of nodal and integration points that fall outside the source mesh. The parameter *maximum leaf scans* were set to 1000 for both meshes A and B, with the accuracy and output remaining the exact same as *maximum leaf scans* = ∞. Therefore, we can observe that the parameter *maximum leaf scans* generalises well to various meshes and is largely dependent on the Octree's *maximum depth* and *leaf split threshold* values.



(a) Nearest Point Method

(b) Distance Method using Field of Points

(c) Distance Method using Elements

(d) Element Shape Function

Fig. 2: Comparison of wall clock times for the four interpolation techniques using Octree (blue) and Grid indexing (red) for Mesh A and A' (left) and Mesh B and B' (right).

In some cases, with certain interpolation types, parameters used, and depending on the mesh, the Octree index developed outperforms the grid-based indexing. Whilst the Octree index is slower for the Nearest Point Method, it is the fastest and least accurate interpolation. Other approaches to performing these interpolations may also be considered: for example, Distance Method using Field of Points could selectively expand a priority queue and search based on the quadrants an octant overlaps.

5 Conclusion

In this paper, we have presented an Octree spatial index that implements four different mesh mapping techniques: Nearest Point Method, Distance Method using Field of Points, Distance Method using Elements, and Element Shape Function. Compared to previous approaches, the time taken to perform the Element Shape Function interpolation has been considerably reduced by introducing an additional parameter for the search. Indexing, specifically element indexing, is also faster using the Octree index.

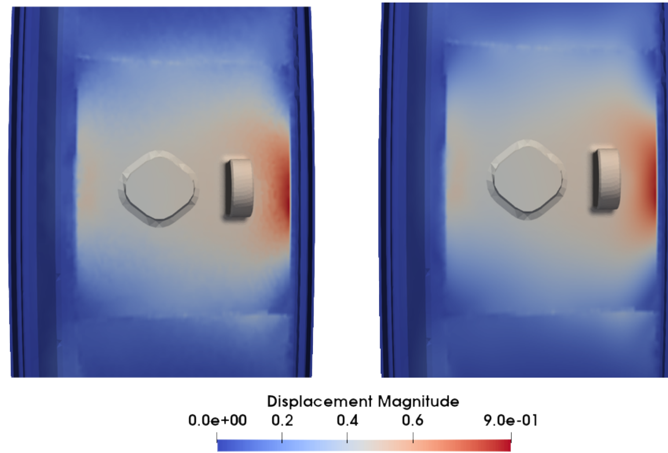


Fig. 3: Output of Nearest Point Method (left) and Element Shape Function (right), mapping displacement from a source mesh of 171,188 linear tetrahedron elements to a target mesh of 125,361 linear tetrahedron elements.

The difference in mapping output between the least accurate and most accurate mapping methods is perceptible from Figure 3.

In future work, the index can become more memory efficient by allocating child nodes in breadth-first order, whereby when a leaf node is split, all 8 children are allocated contiguously. This then allows the ability to only store a pointer to the first child to access all children, thereby saving 56 bytes assuming a 64-bit system. This allocation strategy does involve a trade-off as it means that nodes can no longer be allocated individually on-demand which is useful for sparse meshes. Another aspect to consider would be completely creating the index in parallel by using a linear Octree [5] with CPU [12] or GPU [6] parallelization. This could then be evaluated against larger source meshes that contain millions of elements.

Acknowledgments

This work was supported by the SURE Research Projects Fund of the University of Bradford.

References

1. Shukri Afazov, Eleonora Semerdzhieva, Daniele Scrimieri, Ahmad Serjouei, Bekmurat Kairoshev, and Fatos Derguti. An improved distortion compensation approach for additive manufacturing using optically scanned data. *Virtual and Physical Prototyping*, 16(1):1–13, January 2021.
2. Shukri M. Afazov, Adib A. Becker, and T. H. Hyde. Development of a Finite Element Data Exchange System for chain simulation of manufacturing processes. *Advances in Engineering Software*, 47(1):104–113, 2012.
3. Jens Behley, Volker Steinhage, and Armin B. Cremers. Efficient radius neighbor search in three-dimensional point clouds. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3625–3630, May 2015. ISSN: 1050-4729.
4. Jacobo Bielak, Omar Ghattas, and E.-J Kim. Parallel Octree-Based Finite Element Method for Large-Scale Earthquake Ground Motion Simulation. *CMES. Computer Modeling in Engineering & Sciences*, 10, November 2005.
5. Irene Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25:905–910, December 1982.
6. Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, 2012.
7. Donald Meagher. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. Technical report, Rensselaer Polytechnic Institute, October 1980.
8. G. M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Co., Ottawa, 1966. OCLC: 52181262.
9. Jagan Sankaranarayanan, Hanan Samet, and Amitabh Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31:157–174, April 2007.
10. Daniele Scrimieri, Shukri M. Afazov, Adib A. Becker, and Svetan M. Ratchev. Fast mapping of finite element field variables between meshes with different densities and element types. *Advances in Engineering Software*, 67:90–98, 2014.
11. Daniele Scrimieri, Shukri M. Afazov, and Svetan M. Ratchev. An in-core grid index for transferring finite element data across dissimilar meshes. *Advances in Engineering Software*, 88:53–62, October 2015.
12. Tiankai Tu, D.R. O'Hallaron, and O. Ghattas. Scalable Parallel Octree Meshing for TeraScale Applications. In *ACM/IEEE SC 2005 Conference (SC'05)*, pages 4–4, Seattle, WA, USA, 2005. IEEE.
13. Olek C. Zienkiewicz and Robert L. Taylor. *The Finite Element Method for Solid and Structural Mechanics*. Butterworth-Heinemann, 7th edition, November 2013.