Advances in Engineering Software 67 (2014) 90–98

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Fast mapping of finite element field variables between meshes with different densities and element types

Daniele Scrimieri *, Shukri M. Afazov, Adib A. Becker, Svetan M. Ratchev

Department of Mechanical, Materials and Manufacturing Engineering, University of Nottingham, Nottingham NG7 2RD, UK

ARTICLE INFO

Article history:
Received 5 January 2013
Received in revised form 16 August 2013
Accepted 19 August 2013
Available online 11 September 2013

Keywords:
Finite element analysis
FEA data transfer
Manufacturing chain simulation
Spatial indexing
Equal-sized cells
Meshing

ABSTRACT

In the simulation of a chain of manufacturing processes, several finite element packages can be employed and for each process or package a different mesh density or element type may be the most suitable. Therefore, there is a need for transferring finite element analysis (FEA) data among packages and mapping it between meshes. This paper presents efficient algorithms for mapping FEA data between meshes with different densities and element types. An in-core spatial index is created on the mesh from which FEA data is transferred. The index is represented by a dynamic grid partitioning the underlying space from which nodes and elements are drawn into equal-sized cells. Buckets containing references to the nodes indexed are associated with the cells in a many-to-one correspondence. Such an index makes nearest neighbour searches of nodes and elements much faster than sequential scans. An experimental evaluation of the mapping techniques using the index is conducted. The algorithms have been implemented in the open source finite element data exchange system FEDES.

1. Introduction

Finite element (FE) technology has become very well established as a main tool for engineering analysis. Since the early 1980s, engineering analysts have extensively used FE software for many engineering problems, ranging from a simple elastic analysis to non-linear deformations of biological structures and crash simulations. The affordability and versatility of FE software has helped to spread its popularity [1]. Specialised and general purpose FE software has been developed using formulations and algorithms for solid and structural mechanics, heat transfer, thermo-mechanics in solids, fluid dynamics and multi-physics for a variety of engineering applications [2–4]. In manufacturing, many processes exhibiting different physical phenomena can be involved in the production process. The simulation of each manufacturing process requires a careful selection of the most appropriate FE code, mesh and element type. This implies that different FE codes and meshes with different element types and densities can be utilised in the simulation of manufacturing processes chains.

The need for mapping FEA data across different meshes has arisen in metal forming and forging. The element shape function is used in [5,6], where the local coordinates are obtained by solving a system of non-liner equations using the Newton–Raphson method. In [7], the element shape functions are used for mapping FE field variables between different linear and quadratic, 2D and 3D element types in the open source finite element data exchange system (FEDES), which enables FEA data mapping and transferring across six commercial FE solvers [8]. Three additional mapping techniques are implemented in FEDES, namely a method using the nearest point, a method using fields of points and a method using elements. FEDES was successfully used for the simulation of manufacturing process chains of aerospace applications [7,9,10]. Fernandes et al. [11] develop an algorithm including extraction of the basic geometrical features, contraction of the core mesh, generation of a boundary mesh linking the core with the surface of the workpiece, applications of smoothing procedures to edges and transfer of field variables from the old to the new mesh. In [12], information between non-matching FE meshes is transferred. The study concerns the transfer of FE variables defined at the integration points of the meshes by using only local element-by-element matrices. A nearest-nodes finite element method is proposed in [13], where finite elements are used only for numerical integration while shape functions are constructed using a set of nodes that are the nearest to a quadrature point. A local multivariate Lagrange interpolation is used to construct the shape functions. With regard to interoperability of simulation software, Iványi [14] presents a method using regular expressions to convert different types of file formats for finite element meshes.

In the optimisation of manufacturing process chains, the mapping of FE field variables between different meshes needs to be

* Corresponding author. Tel.: +44 (0)1158468834.
E-mail addresses: Daniele.Scrimieri@nottingham.ac.uk (D. Scrimieri), Shukri.Afazov@nottingham.ac.uk (S.M. Afazov), A.A.Becker@nottingham.ac.uk (A.A. Becker), Svetan.Ratchev@nottingham.ac.uk (S.M. Ratchev).

0965-9978/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.advengsoft.2013.08.003

fast, as many data transfer operations are required at process, chain and assembly levels [15]. This requires the development of new efficient computational techniques to support the existing mapping methods. The objective of this paper is to present efficient algorithms for the mapping methods described in [7]. This is achieved by means of an in-core spatial index built on the mesh from which FEA data is transferred. The index allows to quickly locate the node or element searched for, in accordance with the mapping method. It has the form of a grid partitioning the underlying space of the mesh into equal-sized cells. Given a nodal or interpolation point of the mesh that is the destination of the mapping, the cell in which this point is projected can be accessed in constant time. The node or element searched for will be located either in this cell or in one of its neighbours. In our experiments, this technique has proven to be much faster than a sequential search. The algorithms have been implemented and tested in FEDES [8].

Multidimensional data structures have been extensively employed in many areas, such as computer graphics, geographic information systems and solid modelling [16]. One of the problems that they address is to quickly respond to nearest neighbour queries, the same problem dealt with in this paper. Two similar methods for decomposing a k-dimensional embedding space into k-dimensional rectangles are presented in [17,18]. The content of a rectangle is stored in a bucket on the disc. The use of a directory in the form of a k-dimensional array containing the addresses of the buckets allows to retrieve a record with only two disc accesses (one to the directory and one to the bucket). While in [17] cell division lines are equidistant, in [18] they are not. In the latter work, for each dimension there exists a linear scale specifying the intervals into which the dimension is partitioned. This paper follows the approach taken in [17], although nothing is stored on the disc. A family of spatial indices, called tetrahedral trees, is defined and analysed in [19]. These trees are octrees and kd-trees built on tetrahedral meshes. The aim is to perform spatial queries efficiently.

The paper is organised as follows: Section 2 describes the index structure and how to access it; Section 3 illustrates how to create an index; Sections 4 and 5 present the mapping algorithms; Section 6 contains an evaluation of the performance of the index creation and the mapping algorithms; Section 7 contains conclusions.

2. Access structure

The spatial index is built on the mesh containing the FEA data to transfer and has the form of a grid partitioning the underlying space into equal-sized cells whose borders are parallel to the coordinate axes. The grid is represented by a 3-dimensional array whose elements are the cells of the grid. The size of the grid is given by the number of cells along each dimension. Cells point to buckets that contain references to nodes. The elements of those nodes can also be referenced in the bucket. The correspondence between cells and buckets is many-to-one, i.e. several cells can point to the same bucket. The region spanned by a bucket is given by the union of the cells that share the bucket. Like cells, buckets are pairwise disjoint and the union of the regions that they span represents the entire underlying space.

Given a point in the underlying space, the access structure allows to quickly find the bucket associated with it. A scan of this bucket will then allow to determine whether or not a node with the same coordinates as the point has been indexed. Nearby nodes can be searched for in the same bucket and in nearby buckets. No order is imposed on the nodes in a bucket and any data structure can be used to represent buckets (e.g. arrays, linked lists).

Buckets contain references to nodes, not the actual nodes. A reference could be a pointer or an identifier, depending on the implementation. Handling references when copying, moving or comparing nodes is faster then operating directly on the referenced objects. When speaking about nodes we often mean references to them and not the nodes themselves. For example, if we say that a bucket contains a node or that a node is inserted into a bucket we intend a reference to the node. In FEDES, nodes are stored in an array separate from the spatial index and a node is referenced by its index into this array.

The mapping of a node into a cell is a function of its coordinates, which can be calculated in constant time. Suppose that a grid of size $D_1 \times D_2 \times D_3$ is constructed. Let $\min_i$ and $\max_i$ be, respectively, the minimum and maximum ith coordinate values over all nodes of the mesh indexed or being indexed. A node $n = (x_1, x_2, x_3)$ is associated to the cell $c = (c_1, c_2, c_3)$, where:

$$c_i = \begin{cases} \left\lfloor \frac{x_i - \min_i}{\max_i - \min_i} \cdot D_i \right\rfloor & \text{if } \min_i \leqslant x_i < \max_i \\ 0 & \text{if } x_i < \min_i \\ D_i - 1 & \text{if } x_i \geqslant \max_i \end{cases} \quad (1)$$

for $1 \leqslant i \leqslant 3$ ($\lfloor x \rfloor$ denotes the largest integer not greater than x). The cases $x_i < \min_i$ and $x_i > \max_i$ handle points out of the mesh indexed in the case that the mesh that is destination of the mapping does not cover exactly the same geometric region as the mesh indexed. A function *getBucket* (*idx*, *n*) is defined, which returns the bucket, in the index *idx*, pointed to by the cell associated with node *n* according to (1). This function is needed for both indexing a node (i.e. inserting it into the access structure) and performing nearest neighbour searches. Arrays are random access structures, i.e. access to an element takes constant time (it is independent of the number of elements). Therefore, insertion of a node in the grid can be performed in constant time when the associated bucket is not full and one keeps track of the next available slot in it.

The size of the grid is variable and grows as nodes are indexed and buckets get filled, as it will be described in Section 3.1. On the contrary, the size of the buckets is constant and when a bucket overflows (i.e. a node being indexed cannot be inserted in it

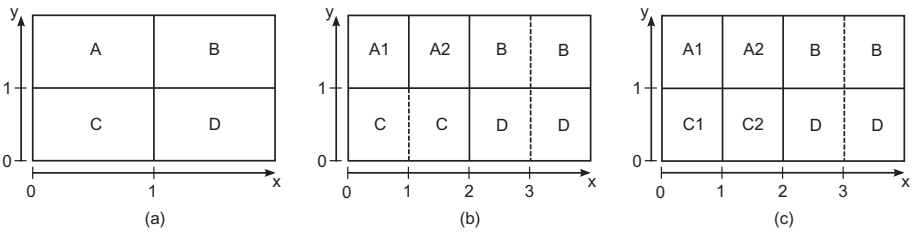

Fig. 1. Grid refinement and creation of new buckets. (a) Initial 2 × 2 grid. (b) Bucket A overflows: since it is not shared, the grid is refined along the *x*-axis, buckets B, C and D are shared (dashed lines indicate shared regions) and bucket A is split into A1 and A2. (c) Bucket C overflows: since it is shared, it is split into C1 and C2.

because it is full) a split occurs. This avoids having buckets growing excessively and slowing down searches. The initial size of the grid and the size of the buckets must be specified when creating the index.

3. Indexing

This section shows how to create indices of nodes and indices of elements, which are used, respectively, by the mapping techniques that will be presented in Sections 4 and 5.

3.1. Indexing of nodes

At the beginning, each bucket is associated with only one cell. Nodes are inserted into the index one by one, in the bucket indicated by the function *getBucket*. When a bucket overflows, the index needs to be expanded. This happens by splitting the overflowing bucket into two. The grid structure may also be refined by splitting cells and sharing buckets.

There are two types of split, depending on whether the overflowing bucket is shared or not. Let us first examine the case when the overflowing bucket is not shared, since this is the one that occurs first and causes buckets to be shared. In this case the grid is refined by splitting all cells along one dimension (keeping the division lines equidistant), thereby doubling the total number of cells. The interval which was covered by one cell along the refined dimension is now covered by two contiguous cells. The process is illustrated in a 2-dimensional case in Fig. 1 (a) and (b), where the split takes place along the x-axis. The direction to refine is chosen by cycling through all the three directions.

The number of buckets is not doubled. Instead, each bucket except the overflowing one is shared by two contiguous cells along the dimension that is being refined. The overflowing bucket is split into two buckets, one for each of the two cells resulting from the split of the cell originally pointing to the overflowing bucket. That is, if the cell c pointing to an overflowing bucket b is split into two cells c_1 and c_2, then b will be split into two buckets b_1 and b_2 with, let us say, c_1 pointing to b_1 and c_2 pointing to b_2. The split of b involves the redistribution of its contents between b_1 and b_2. The redistribution of a node of b is based on whether it is located in c_1 or c_2, according to (1). Afterwards, the bucket (either b_1 or b_2) in which to add the node to index is determined. If there is space in it, the node is inserted. Otherwise, if all the redistributed nodes have been indexed in this bucket (i.e. there is again an overflow), then another split with grid refinement must occur.

The case when the overflowing bucket is shared is simpler, as the size of the grid does not change. An overflowing bucket b is split into two buckets: one for the cell in which the node being indexed is located and one for the other cells sharing b. Fig. 1 (c) shows the split of an overflowing bucket shared by two cells. If after the split there is still no space available, a split with grid refinement is performed.

The first type of split is more expensive computationally, since it requires that new cells be allocated and pointers be updated. However, it occurs less often than the second type, because after a grid refinement all but two buckets are shared and it is likely that several shared buckets will be split before the next refinement.

Let us see in more detail the involved operations. The function *createNodeIndex* (Function 1) creates an index *nodeIdx* on a set of nodes *nodes*.

Function 1. createNodeIndex (nodeIdx, nodes, xs, ys, zs, bs)

```
Input: nodes, the nodes to index; xs, ys, zs, the size of the grid
  in each dimension; bs, the bucket size
Output: nodeIdx, the index built
 1:  initialiseIndex (nodeIdx, nodes, xs, ys, zs, bs)
 2:  for all n ∈ nodes do
 3:    bucket ← getBucket (nodeIdx, n)
 4:    if numOfNodes (bucket) < bucketSize (nodeIdx) then
 5:      insertNode (bucket, n)
 6:    else if numOfPointers (bucket) > 1 then
 7:      splitSharedBucket (nodeIdx, bucket, n)
 8:    else
 9:      recRefineGrid (nodeIdx, bucket, n)
10:    end if
11:  end for
```

The following functions are used. The index is initialised with the function *initialiseIndex (nodeIdx, nodes, xs, ys, zs, bs)*, which creates a grid of size $xs \times ys \times zs$ and buckets of size *bs*, and determines the minimum and maximum ith coordinate values over all nodes in *nodes*. The function *numOfNodes (bucket)* returns the number of nodes in *bucket*. The function *bucketSize (idx)* returns the size of the buckets in *idx*, in terms of number of nodes that they can contain. The function *insertNode (bucket, n)* inserts node n into *bucket*, assuming that *bucket* is not full. The function *numOfPointers (bucket)* returns the number of cells pointing to *bucket* (a pointer counter is stored in the bucket). If there is more than one pointer to *bucket*, it means that *bucket* is shared. The function *splitSharedBucket* (Function 2) splits an overflowing shared bucket, while *recRefineGrid* (Function 3) splits an overflowing non-shared bucket and refines the grid.

Function 2. splitSharedBucket (nodeIdx, fullBucket, n)

```
Input: nodeIdx, an index of nodes; fullBucket, an overflowing
  shared bucket; n, the node causing the overflow
Output: nodeIdx, with fullBucket split and n indexed
 1:  tempBucket ← copyBucket (fullBucket)
 2:  initialiseBucket (fullBucket, numOfPointers
     (tempBucket) - 1)
 3:  createBucket (nodeIdx, n)
 4:  newBucket ← getBucket (nodeIdx, n)
 5:  redistributeBucket (nodeIdx, tempBucket)
 6:  if numOfNodes (newBucket) < bucketSize (nodeIdx)
     then
 7:    insertNode (newBucket, n)
 8:  else
 9:    recRefineGrid (nodeIdx, newBucket, n)
10:  end if
```

In *splitSharedBucket*, a copy of the overflowing bucket *fullBucket* is created with *copyBucket*. Then *fullBucket* is emptied with the function *initialiseBucket (bucket, numOfPointers)*, where *numOfPointers* specifies the number of cells pointing to *bucket*, in this case the old value decremented by one. A new bucket is created with the function *createBucket (idx, n)*, which also redirects the pointer in the cell associated with n to the newly created bucket and sets the pointer counter to one. The other cells pointing to *fullBucket* are not changed. The nodes that were initially in *fullBucket* are redistributed in the grid using *redistributeBucket*. If there is space in the new bucket after redistributing, n is inserted in it. Otherwise the grid is refined with *recRefineGrid* (Function 3).

Function 3. recRefineGrid (nodeIdx, fullBucket, n)

```
Input: nodeIdx, an index of nodes; fullBucket, an overflowing
   non-shared bucket; n, the node causing the overflow
Output: nodeIdx, with a refined grid, fullBucket split and n
   indexed
1: bucket ← fullBucket
2: repeat
3:    refineGrid (nodeIdx, bucket)
4:    bucket ← getBucket (nodeIdx, n)
5: until numOfNodes (bucket) < bucketSize (nodeIdx)
6: insertNode (nodeIdx, n)
```

In *recRefineGrid*, the grid is recursively refined by invoking *refineGrid* (Function 4), until there is available space in the bucket associated with *n*. Node *n* will then be inserted in it. Usually the number of required iterations is very low; most of the times just one is sufficient as the nodes in *bucket* are redistributed.

Function 4. refineGrid (nodeIdx, fullBucket)

```
Input: nodeIdx, an index of nodes; fullBucket, an overflowing
   shared bucket
Output: nodeIdx, with a refined grid and fullBucket split up
1: copyIdx = copyIndex (nodeIdx)
2: resize (nodeIdx)
3: for i = 0 →xSize (nodeIdx) - 1 do
4:    for j = 0 →ySize (nodeIdx) - 1 do
5:       for k = 0 →zSize (nodeIdx) - 1 do
6:          if splitAxis (nodeIdx) = XAxis then
7:             i1 ← i/2, j1 ← j, k1 ← k
8:          else if splitAxis (nodeIdx) = YAxis then
9:             i1 ← i, j1 ← j/2, k1 ← k
10:         else {splitAxis (nodeIdx) = ZAxis}
11:            i1 ← i, j1 ← j, k1 ← k/2
12:         end if
13:         if getBucket (copyIdx, i1, j1, k1) = fullBucket then
14:            createBucket (nodeIdx, i, j, k)
15:         else
16:            setBucket (nodeIdx, i, j, k, getBucket (copyIdx, i1,
                  j1, k1))
17:         end if
18:      end for
19:   end for
20: end for
21: redistributeBucket (nodeIdx, fullBucket)
```

In *refineGrid* the grid is resized using *resize*. This operation doubles the number of cells in the grid, but does not update the pointers to the buckets. In order to do that, a copy *copyIdx* of the index is created with *copyIndex* before resizing, so that the pointers can be copied from it. The function *copyIndex* performs a shallow copy of the index, i.e. it creates a copy of the cells, including the pointers, but not of the pointed buckets. Each bucket pointer in *copyIdx*, except the one for *fullBucket*, is copied twice in the resized grid. Suppose, for instance, that the axis that is split is the *x*-axis and that *i* is an even number. For each *j* and *k*, the bucket pointer of the cell $(i/2, j, k)$ of *copyIdx* is copied into the cells (i, j, k) and $(i+1, j, k)$ of *nodeIdx*. The split axis is given by the function *splitAxis*. The setting of the bucket pointer in the cell (i, j, k) of an index *idx* to a bucket *b* is done by invoking *setBucket (idx, i, j, k, b)*. The function *getBucket (idx, i, j, k)* is similar to *getBucket (idx, n)*. Instead of using the coordinates of a node to identify a bucket, it uses directly the position (i, j, k) of a cell pointing to it. Analogously, the function *createBucket (idx, i, j, k)* allocates a new bucket and links the cell (i, j, k) to it. For the sake of simplicity, two new buckets are actually allocated in *refineGrid* and the content of *fullBucket* is redistributed between them. Both *copyIdx* and *fullBucket* can then be disposed of.

3.2. Indexing of elements

An index of elements is similar to an index of nodes and is created in a similar way. An element is indexed by indexing its nodes. The difference from an index of nodes is that, in this case, a bucket also contains, for each node, a reference to its element. If two or more elements have a node in common, that node will be processed multiple times, each time linking to a different element. A check can be done to see if a node has already been referenced in a bucket, so that there exists only one reference per node, linked to all the elements sharing it.

If one element's node is indexed in a cell, that element either is contained in or overlaps with the cell. On the contrary, if an element overlaps with a cell, the cell may not contain any of the element's nodes. Therefore, searching a cell for the elements referenced in it may not provide all the elements overlapping with the cell. An exhaustive search requires that also nearby cells be examined.

It is not sufficient to store in a bucket only elements; nodes must be stored as well. In case of split, elements are redistributed by redistributing their nodes as described above for indices of nodes. Elements must be represented explicitly by a list of their nodes because, given a node indexed in a bucket, it must be possible to easily find all other nodes of the same element. In FEDES, elements are stored in an array separate from the spatial index and the node array, and contain references to their nodes in the node array. An element is referenced by its index into such an array.

The function *createElementIndex* (Function 5) creates an index *elemIdx* on a set of elements *elems*. The function for inserting a node in a bucket, *insertNode (bucket, n, e)*, takes now one more argument, the element *e* of the node *n* to insert. Similarly, the functions *splitSharedBucket* and *recRefineGrid* take as additional argument the element of the node causing the overflow.

Function 5. createElementIndex (elemIdx, elems, xs, ys, zs, bs)

```
Input: elems, the elements to index; xs, ys, zs, the size of the
   grid in each dimension; bs, the bucket size
Output: elemIdx, the index built
1: initialiseIndex (elemIdx, elems, xs, ys, zs, bs)
2: for all e ∈ elems do
3:    for all n ∈ e do
4:       bucket ← getBucket (elemIdx, n)
5:       if numOfNodes (bucket) < bucketSize (elemIdx) then
6:          insertNode (bucket, n, e)
7:       else if numOfPointers (bucket) > 1 then
8:          splitSharedBucket (elemIdx, bucket, n, e)
9:       else
10:         recRefineGrid (elemIdx, bucket, n, e)
11:      end if
12:   end for
13: end for
```

4. Mapping nodes

Suppose that we want to transfer FEA data from a mesh *A* to a mesh *B*. The following two mapping methods use an index built on the nodes of *A* to quickly locate the nodes from which transferring.

4.1. Method using the nearest node

For each nodal or integration point n in B, this method searches A for the nearest node to n from which transferring FEA data. Without a spatial index, this method requires the scan of all nodes in A and the calculation of the distance between each of them and n. With a grid index built on A, it is sufficient to locate the grid cell associated with n using (1) and examine only its nodes and, if necessary, those of nearby cells.

The function *findNearestNode (nodeIdx, n)* (Function 6) finds the nearest node to n in the index *nodeIdx*.

Function 6. findNearestNode (nodeIdx, n)

```
Input: nodeIdx, an index of nodes; n, the node to map
Output: nearest, the nearest node to n
 1: minDist ← maxDist (nodeIdx)
 2: searchBucket (getBucket (nodeIdx, n), n, nearest,
    minDist)
 3: i ← getCellXPos (n), j ← getCellYPos (n), k ← getCellZPos
    (n)
 4: search ← true
 5: p ← 1
 6: while search = true do
 7:    search ← false
 8:    if i - p ⩾ 0 then
 9:       searchLowerXPlane (nodeIdx, i, j, k, p, n, nearest,
    minDist, search)
10:    end if
11:    if i + p < xSize (nodeIdx) then
12:       searchUpperXPlane (nodeIdx, i, j, k, p, n, nearest,
    minDist, search)
13:    end if
14:    if j - p ⩾ 0 then
15:       searchLowerYPlane (nodeIdx, i, j, k, p, n, nearest,
    minDist, search)
16:    end if
17:    if j + p < ySize (nodeIdx) then
18:       searchUpperYPlane (nodeIdx, i, j, k, p, n, nearest,
    minDist, search)
19:    end if
20:    if k - p ⩾ 0 then
21:       searchLowerZPlane (nodeIdx, i, j, k, p, n, nearest,
    minDist, search)
22:    end if
23:    if k + p < zSize (nodeIdx) then
24:       searchUpperZPlane (nodeIdx, i, j, k, p, n, nearest,
    minDist, search)
25:    end if
26:    p ← p + 1;
27: end while
28: return nearest
```

During the search, the variables *nearest* and *minDist* contain, respectively, the nearest node and its distance from n. Initially, *nearest* is undefined and *minDist* is set to the maximum distance between two points in the grid. First, the cell associated with n is examined. Then adjacent cells are examined because n may be closer to a node in one of them. Other cells further away may also be searched. The search is therefore an iterative process starting from the cell associated with n and going outwards until the nearest node is found. At each iteration, the offset p (i.e. the distance in terms of number of cells) from the cell associate to n is incremented by one.

The function *searchBucket (bucket, n, nearest, minDist)* searches *bucket* for the nearest node to n whose distance from n is less than *minDist*. If it is found, it updates both *minDist* and *nearest*. The functions *getCellXPos (n)*, *getCellYPos (n)* and *getCellZPos (n)* return the indices, in the grid, of the cell associated with n along, respectively, the x-, y- and z-axis. Given a cell with index (i, j, k), the nearby cells considered are (with $p = 1, 2, \ldots$):

- $(i - p,\ j',\ k')$, for all $j - p \leqslant j' \leqslant j + p$ and $k - p \leqslant k' \leqslant k + p$ (*searchLowerXPlane*)
- $(i + p,\ j',\ k')$, for all $j - p \leqslant j' \leqslant j + p$ and $k - p \leqslant k' \leqslant k + p$ (*searchUpperXPlane*)
- $(i',\ j - p,\ k')$, for all $i - p \leqslant i' \leqslant i + p$ and $k - p \leqslant k' \leqslant k + p$ (*searchLowerYPlane*)
- $(i',\ j + p,\ k')$, for all $i - p \leqslant i' \leqslant i + p$ and $k - p \leqslant k' \leqslant k + p$ (*searchUpperYPlane*)
- $(i',\ j',\ k - p)$, for all $i - p \leqslant i' \leqslant i + p$ and $j - p \leqslant j' \leqslant j + p$ (*searchLowerZPlane*)
- $(i',\ j',\ k + p)$, for all $i - p \leqslant i' \leqslant i + p$ and $j - p \leqslant j' \leqslant j + p$ (*searchUpperZPlane*)

Checks on the indices are done to make sure that they are not out of the grid bounds. The function *searchLowerXPlane* is shown in Function 7.

Function 7. searchLowerXPlane (nodeIdx, i, j, k, p, n, nearest, minDist, search)

```
Input: nodeIdx, an index of nodes; i, j, k, the indices of the cell
   associated with n; p, the offset from the cell (i, j, k) along the
   x-axis; n, the node to map; nearest, the nearest node to n
   found so far; minDist, the distance between n and nearest
Output: nearest, the nearest node to n found so far; minDist,
   the distance between n and nearest; search, indicates
   whether to continue searching or not
1: xLowerBound ← (i - p) × cellXSize (nodeIdx) + minX
   (nodeIdx)
2: if getXCoord (n) - xLowerBound < minDist then
3:    searchXPlane (nodeIdx, i - p), max (j - p, 0), min (j + p,
   ySize (nodeIdx)), max (k - p, 0), min (k + p, zSize (nodeIdx),
   n, nearest, minDist)
4:    search ← true
5: end if
```

The other search functions are defined similarly. A cell is examined only if its distance from n is less than *minDist*. In fact, if this condition is not true, the nearest node cannot be in it. For the sake of simplicity, in *searchLowerXPlane* only the distance between n and $(i - p, j, k)$ is calculated. If it is less than *minDist*, all cells $(i - p, j', k')$, with $j - p \leqslant j' \leqslant j + p$ and $k - p \leqslant k' \leqslant k + p$, are examined. This is accomplished with the function *searchXPlane (nodeIdx, xPlane, yStart, yEnd, zStart, zEnd, n, nearest, minDist)*, limiting the indices within the grid bounds.

The search stops when *search* is false, i.e. when there is no cell that has not been examined and whose distance from n is less than *minDist*. Usually the search stops after very few iterations, unless there is a considerable difference in density between the two meshes and the grid is very refined. In this case, locating the nearest node may involve searching nearby cells with higher values of p.

Since cells can share buckets, the same bucket can be scanned more then once while searching. To prevent this, scanned buckets can be flagged as such. A search counter keeps track of the number of nearest node searches made. A variable (*lastSearch*) is associated with each bucket, containing the value of the search counter when the bucket was last scanned. Thus, when a cell is examined, it is

possible to determine if the associated bucket has already been scanned in that nearest node search by comparing *lastSearch* to the search counter.

4.2. Method using fields of points

Each nodal or interpolation point n in B, projected in the underlying space of A, can be seen as the origin of a new coordinate system dividing the space into 8 regions whose borders are parallel to the coordinate axes. This method searches for the nearest node to n in each of these regions. The value of a FEA variable to map onto n is obtained by calculating the weighted mean of the values for the nearest nodes. In the special case when n coincides with a node m of A, only the data associated with m is copied. Some regions may be empty, for example when n is not located inside A. In this case the mean is calculated only on the neighbours found.

Once the nearest nodes $m_1, \ldots, m_k$ have been found ($k \leqslant 8$), a weight w_i proportional to the distance d_i between n and m_i is calculated, for $1 \leqslant i \leqslant k$:

$$w_i = \frac{\sum_{j=1}^{k} d_j}{d_i} \tag{2}$$

The value v_n to map onto node n is then given by:

$$v_n = \frac{\sum_{i=1}^{k} w_i v_{m_i}}{\sum_{i=1}^{k} w_i} \tag{3}$$

As with the nearest node method, by using a grid index on A it is sufficient to locate the grid cell associated with n using (1) and examine only the nodes in it and nearby cells. A limit on the offset from the cell associated with n must be set in order to avoid that empty regions are searched. The search algorithm is similar to the one for the nearest node method, with the difference that the nearest node is searched for in each region. For each cell c being examined, the region it is contained in or the regions it overlaps with are determined. For each of these regions, let us call it r, if the distance between c and n is less than the distance between n and the current nearest node in r, then c is searched. If c overlaps with more than one region, only the nodes falling in r are considered.

5. Mapping elements

Suppose that we want to transfer FEA data from a mesh A to a mesh B. The following two mapping methods use an index built on the elements of A to quickly locate the elements from which transferring.

5.1. Method using elements

For each nodal or integration point n in B, this method searches A for the element with the smallest average distance between its nodes and n, i.e. the element e that minimises $\frac{\sum_{i=1}^{k} d_i}{k}$, where d_i is the distance between the ith node of e and n. After locating such an element, the value of a FEA variable to map onto n is obtained by calculating the weighted mean of the values for the nodes of the element, by applying (3) and using (2) as weights. If n coincides with a node m of A, only the data associated with m is copied. If n is contained in an element, it is likely that this element will be selected. However, if n is contained in an element and is close to or on the border with a smaller element, the average distance from the latter may be smaller.

Using a grid index built on the elements of A, the grid cell c associated to n is located by applying (1) and its elements are examined. The elements of a cell are those of the nodes indexed in the cell. If there is no element in c, nearby cells are examined, with increasing offsets from c, until elements are found. Elements are normally found at very small offsets, unless there is a considerable difference in density between the two meshes and the grid is very refined. If there are elements in c, adjacent cells still have to be examined because n may be contained in an element which overlaps with c but none of its nodes are in c. In this case it is likely that such an element is the one searched for. The search stops at the smallest offset, greater than or equal to a fixed lower bound, where at least one element has been found.

Since the same element can be associated with more than one node in the same cell or in different ones, it may be examined more than once. This is also the case when several cells share the same bucket. To prevent this, elements examined can be flagged as such. The same technique described in Section 4.1 for flagging buckets can be applied.

5.2. Method using the element shape function

For each nodal or integration point n in B, this method searches A for the element into which n falls. The element shape function for the corresponding element type is then employed to map FEA data onto n. The search in the index is performed, as in the previous method, starting from the cell associated with n and going outwards until the element containing n is found.

Different element types described by shape functions have been developed in the finite element method (FEM). In FEM, shape functions are used to interpolate a variety of FEA state variables (e.g. displacements, velocities, accelerations, reaction forces, strains, stresses, temperatures) from nodal points into integration/Gauss points. Generally, they can be used for mapping from nodal points into any point inside the element. Given an element with k nodes $m_1, \ldots, m_k$, the value v_n of a FEA variable mapped onto a point n located inside the element is given by [2]:

$$v_n = \sum_{i=1}^{k} \eta_{m_i} v_{m_i} \tag{4}$$

where η_{m_i} is the element shape function for node m_i and v_{m_i} is the value of the variable for node m_i. The explicit formulations for 2D and 3D, linear and quadratic elements can be found in [20].

Generally, the local coordinates (with respect to the element) of the integration points are predefined in the finite element code. This allows the interpolation of FEA variables from nodal points into integration points by applying directly the element shape functions. With regard to mapping, while the coordinates of nodes and integration points are known, the local coordinates are not. Therefore, the local coordinates must be obtained in order to apply (4) for mapping FEA data. For triangular and tetrahedron elements, the local coordinates are obtained by solving a system of linear equations. For quadrilateral, wedge and hexahedron linear elements, the local coordinates are obtained by solving a system of non-linear equation using the Newton–Raphson method [7].

A special case occurs when n does not fall into any element of the mesh. This is the case when the finite element geometries of the two meshes are not exactly the same and nodes of B are located outside A. In this case no FEA data will be mapped using the exact ranges of the local coordinates. To deal with this scenario, if the element is not found within a certain offset, the local coordinates are updated by introducing incremental tolerances and the search restarts from the cell associated with n.

6. Experimental evaluation

Experiments were conducted to evaluate the performance of the algorithms for creating indices of nodes and elements, and for mapping using such indices. An analysis of the accuracy of the mapping methods is done in [7], and that is also valid for the implementations described in this paper. In this section only the computational efficiency is analysed. Displacements were mapped between meshes in two simulations of manufacturing chains, using all the mapping techniques illustrated:

- From a mesh with about 170,000 linear tetrahedron elements (mesh 1) to a mesh with about 120,000 linear tetrahedron elements (mesh 1′), shown in Fig. 2;
- From a mesh with about 60,000 linear hexahedron elements (mesh 2) to a mesh with about 80,000 linear tetrahedron elements (mesh 2′), shown in Fig. 3.

The first mesh represents an aero-engine vane component. FEA data transfer was performed from a heat treatment process to a macro-machining or bulk material removal process.

All experiments were run on a machine equipped with an Intel i7-Q740 CPU at 1.73 GHz and 8 GB of RAM. The operating system was GNU/Linux, kernel version 3.2.0. The algorithms were coded in Pascal, the language in which FEDES is written. The compiler used was the Free Pascal Compiler, version 2.6.0, with default optimisations. The times reported are wall-clock times. Each of them is the mean of three runs. They were calculated using the Pascal library function `Now()`, after killing all non-required processes.

Table 1 shows the parameters for the construction of the indices and some statistics. Bucket sizes and initial grid sizes were chosen based on the size of the meshes. As it can be observed, construction times were very short; only the construction of the element index for mesh 1 took more than 1 s. This suggests that there is no need to save on the disc an index of such dimensions for future mapping operations, as it can be quickly recreated when needed. As it can be seen from the difference between the initial and the final grid size, the node and element indices for mesh 1 were refined 5 and 7 times, respectively. While the node and element indices for mesh 2 were refined 2 and 3 times, respectively.

The average bucket occupancy was below 50% in all cases. This value was calculated with respect to the non-empty buckets. The memory for a bucket is allocated only when the first node is inserted into it. Therefore, for empty buckets it is not actually allocated any memory. For non-empty buckets the entire bucket size is allocated. A more elaborate strategy would allocate space stepwise until a maximum size is reached. This would require more time spent allocating memory but it would save space. Having a

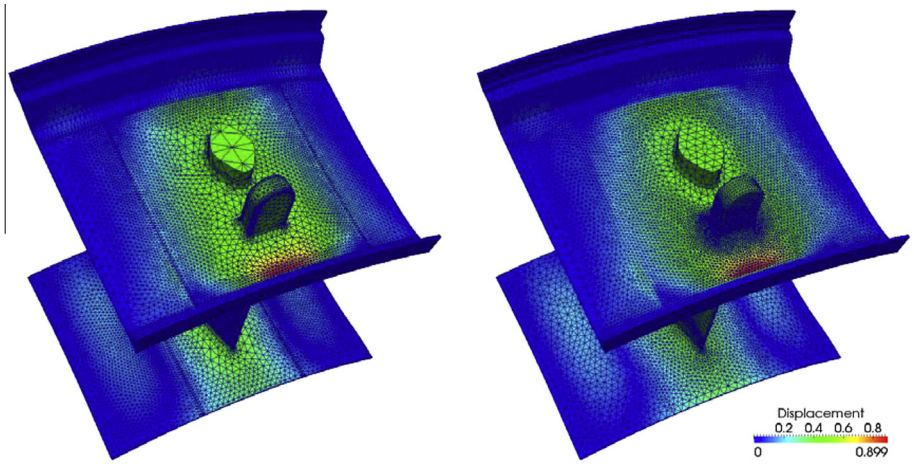

Fig. 2. Mapping of displacements from a mesh with about 170,000 linear tetrahedron elements (left) to a mesh with about 120,000 linear tetrahedron elements (right).

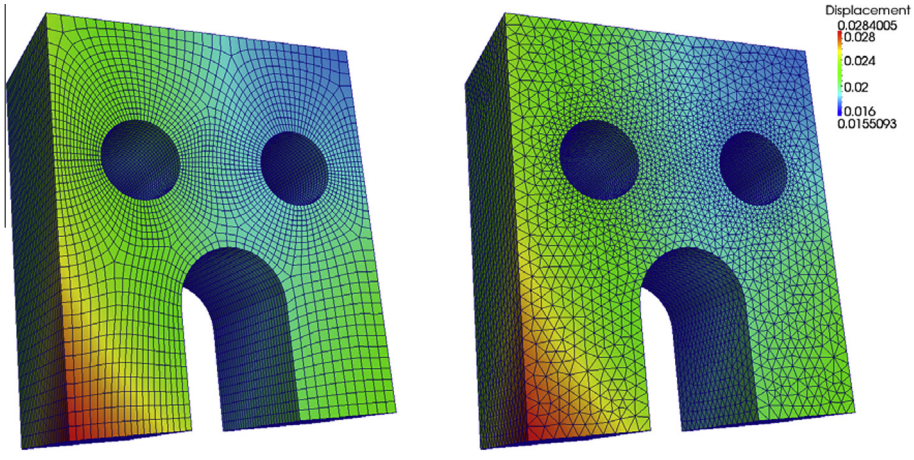

Fig. 3. Mapping of displacements from a mesh with about 60,000 linear hexahedron elements (left) to a mesh with about 80,000 linear tetrahedron elements (right).

Table 1
Parameters and statistics of the indices created.

Mesh	Index	Bucket size	Initial grid size	Final grid size	Time (s)	Buckets	Splits	Non-empty buckets (%)	Avg. bucket occupancy (%)
1	Nodes	40	$25 \times 25 \times 25$	$100 \times 100 \times 50$	0.07	16,751	1126	18.8	42.8
1	Elements	150	$50 \times 50 \times 50$	$400 \times 200 \times 200$	1.62	130,723	5723	10.9	31.9
2	Nodes	20	$20 \times 20 \times 20$	$40 \times 40 \times 20$	0.04	9275	1275	86.2	46.5
2	Elements	60	$30 \times 30 \times 30$	$60 \times 60 \times 60$	0.14	32,461	5461	67.7	40.5

smaller bucket size does not necessarily reduce the average bucket occupancy because this may result in more splits and a larger number of buckets.

Tables 2–5 contain the execution times for each mapping technique described, comparing a search in the index built on the mesh with a sequential search in the mesh. Each mapping technique and type of search was run with 1, 2, 4 and 8 threads. The used CPU has 4 cores and can therefore run up to 4 threads simultaneously. However, the CPU's hyper-threading technology provides 2 logical units per core so that the operating system can schedule 2 threads to each core. This means that if two threads are scheduled to one core and one is not running, that core can execute the other one. The difference in performance between running 4 and 8 threads depends on the application and is not always significant, as it can be noticed in this experimentation.

Table 2
Execution times (s) with the method using the nearest node.

Threads	Mesh 1 → Mesh 1′		Mesh 2 → Mesh 2′	
	Sequential	Index	Sequential	Index
1	29.59	1.08	21.80	0.43
2	17.30	0.73	12.59	0.30
4	12.25	0.55	8.85	0.23
8	12.16	0.51	8.83	0.22

Table 3
Execution times (s) with the method using fields of points.

Threads	Mesh 1 → Mesh 1′		Mesh 2 → Mesh 2′	
	Sequential	Index	Sequential	Index
1	94.46	15.89	69.32	2.98
2	56.75	10.29	41.42	2.22
4	40.84	7.64	29.69	1.51
8	39.38	6.90	29.20	1.37

Table 4
Execution times (s) with the method using elements.

Threads	Mesh 1 → Mesh 1′		Mesh 2 → Mesh 2′	
	Sequential	Index	Sequential	Index
1	1071.61	23.96	217.30	9.01
2	590.70	14.46	127.09	5.72
4	355.74	9.63	89.93	3.91
8	297.19	6.48	86.80	3.43

Table 5
Execution times (s) with the method using the element shape function.

Threads	Mesh 1 → Mesh 1′		Mesh 2 → Mesh 2′	
	Sequential	Index	Sequential	Index
1	477.26	45.71	116.29	10.31
2	285.33	26.39	70.90	9.41
4	199.95	24.67	50.93	8.07
8	158.83	24.03	50.80	6.96

The parallelism is at the data level. The threading code works by dividing in equal chunks among the running threads the nodes of the mesh which is the destination of the mapping. All threads execute in parallel the same code but on different chunks of data. They therefore access simultaneously the source mesh (in the case of sequential search) or the index built on it (in the case of search in an index). The access is read-only, so there is no synchronisation issue.

It can be observed that searches in an index were much faster than sequential searches, even including the index build time. The mapping times indicate that, for sequential searches, the techniques can be ordered from the fastest to the slowest as follows: method using the nearest node, method using fields of points, method using the element shape function, method using elements. For searches in an index, the order is different: method using the nearest node, method using fields of points, method using elements, method using the element shape function. (With the first mesh and 8 threads, the method using elements appeared to perform roughly as the method using fields of points.)

Generally, execution times with indices can be improved by increasing the initial grid dimension or the bucket size. This reduces the number of splits, thereby shortening the index creation time. In addition, it results in fewer shared buckets. As a consequence, searching in the index is faster, because the likelihood of scanning more than once the same bucket is lower.

7. Conclusions

An in-core spatial index has been proposed to speed up the mapping of FEA data between different meshes employed in the simulation of a chain of manufacturing processes. The index has the form of a grid partitioning the underlying space of the mesh on which it is built into equal-sized cells. For each nodal or interpolation point of the mesh onto which FEA data is being transferred, a search in the index is performed to find the node or element to use for interpolation.

Algorithms for the creation and use of such an index have been described. Also, they have been evaluated in two cases where displacements were mapped between meshes with number of elements in the range 80,000–170,000. In particular, the implementation of four mapping techniques have been presented: a method using the nearest node, a method using fields of points, a method using elements and a method using the element shape function. The first two techniques involve a mapping from nodes to nodes and require an index of nodes. The last two involve a mapping from elements to nodes and require an index of elements. For each of them, the performance of the mapping with the use of an index has proven to be much faster than a sequential search in all cases analysed, even including the index build time.

Search in an index can be performed in parallel by several threads. Experiments were run with 1, 2, 4 and 8 threads, with each thread operating on a separate chunk of data. The construction of the index took less than 2 s in all cases. This suggests that, for meshes of the dimensions analysed, there is no need to save on

the disc the created index when it is needed for future mappings. A new index can always be created "on the fly". Performance of index creation and mapping can still be improved by increasing the initial grid dimension and the bucket size. There is some memory overhead due to the index structure, which should not generally be an issue. Anyway, it can be reduced by a stepwise bucket allocation strategy.

It would be interesting to evaluate this indexing technique with larger meshes (with millions of elements) and on distributed computing systems, and to compare it with other indexing techniques.

References

[1] Becker AA. An introduction guide to finite element analysis. Professional Engineering Publishing Limited. UK: Suffolk; 2004.
[2] Zienkiewicz OC, Taylor RL. The finite element method for solids and structural mechanics. 6th ed. Oxford, UK: Butterworth Heinemann; 2005.
[3] Zienkiewicz OC, Taylor RL. The finite element method. Fluid dynamics, 5th ed., vol. 3. Oxford, UK: Butterworth Heinemann; 2000.
[4] Reddy JN, Gartling DK. The finite element method in heat transfer and fluid dynamics. Florida, USA: CRC Press LLC; 2001.
[5] Yoon JH, Yang DY. A three-dimensional rigid-plastic finite element analysis of bevel gear forging using a remeshing technique. Int J Mech Sci 1990;4:277–91.
[6] Martins P, Marmelo J, Rodrigues J, Marques MB. Plarmsh3 – a three-dimensional program for remeshing in metal forming. Comput Struct 1994;53:1153–66.
[7] Afazov S, Becker A, Hyde T. Development of a finite element data exchange system for chain simulation of manufacturing processes. Adv Eng Softw 2012;47(1):104–13.
[8] Fedes. <http://www.sourceforge.net/projects/fedes/>.
[9] Afazov S, Nikov S, Becker A, Hyde T. Manufacturing chain simulation of an aero-engine disc and sensitivity analyses of micro-scale residual stresses. Int J Adv Manuf Technol 2011;52:279–90.
[10] Tersing H, Lorentzon J, Francois A, Lundbäck A, Babu B, Barboza J, et al. Simulation of manufacturing chain of a titanium aerospace component with experimental validation. Finite Elem Anal Des 2012;51:10–21.
[11] Fernandes J, Martins P. All-hexahedral remeshing for the finite element analysis of metal forming processes. Finite Elem Anal Des 2007;43:666–79.
[12] Dureisseix D, Bavestrello H. Information transfer between incompatible finite element meshes: application to coupled thermo-viscoelasticity. Comput Methods Appl Mech Eng 2006;195:6523–41.
[13] Luo Y. A nearest-nodes finite element method with local multivariate lagrange interpolation. Finite Elem Anal Des 2008;44:797–803.
[14] Iványi P. Finite element mesh conversion based on regular expressions. Adv Eng Softw 2012;51:20–39.
[15] Afazov S. Modelling and simulation of manufacturing process chains. CIRP J Manuf Sci Technol 2013;6(1):70–7.
[16] Samet H. Foundations of multidimensional and metric data structures. Morgan Kaufman; 2006.
[17] Tamminen M, Sulonen R. The excell method for efficient geometric access to data. In: Proceedings of the nineteenth design automation conference; 1982. p. 345–51.
[18] Nievergelt J, Hinterberger H, Sevcik KC. The grid file: an adaptable, symmetric multikey file structure. ACM Trans Database Syst 1984;9(1):38–71.
[19] Floriani LD, Fellegara R, Magillo P. Spatial indexing on tetrahedral meshes. In: Proceedings of the 18th SIGSPATIAL international conference on advances in geographic information systems; 2010. p. 506–9.
[20] Afazov S. Simulation of manufacturing processes and manufacturing chains using finite element techniques. Ph.D. thesis, University of Nottingham, Nottingham, UK; 2009.