An in-core grid index for transferring finite element data across dissimilar meshes

Daniele Scrimieri[a,*], Shukri M. Afazov[b], Svetan M. Ratchev[a]

[a] *Department of Mechanical, Materials and Manufacturing Engineering, University of Nottingham, Nottingham NG7 2RD, UK*

[b] *The Manufacturing Technology Centre Limited, Pilot Way, Ansty Business Park, Coventry CV7 9JU, UK*

Abstract

The simulation of a manufacturing process chain with the finite element method requires the selection of an appropriate finite element solver, element type and mesh density for each process of the chain. When the simulation results of one step are needed in a subsequent one, they have to be interpolated and transferred to another model. This paper presents an in-core grid index that can be created on a mesh represented by a list of nodes/elements. Finite element data can thus be transferred across different models in a process chain by mapping nodes or elements in indexed meshes. For each nodal or integration point of the target mesh, the index on the source mesh is searched for a specific node or element satisfying certain conditions, based on the mapping method. The underlying space of an indexed mesh is decomposed into a grid of variable-sized cells. The index allows local searches to be performed in a small subset of the cells, instead of linear searches in the entire mesh which are computationally expensive. This work focuses on the implementation and computational efficiency of indexing, searching and mapping. An experimental evaluation on medium-sized meshes suggests that the combination of index creation and mapping using the index is much faster than mapping through sequential searches.

Keywords: Manufacturing chain simulation, Finite element data mapping, Grid indexing, Nearest neighbour search

1. Introduction

The manufacture of a product can involve the manufacture and assembly of several components, each requiring the application of a sequence of processes, integrated into a manufacturing chain. The finite element method (FEM) is commonly used to simulate every process of the chain, where a different modelling strategy can be adopted for each process. A strategy includes the selection of the most appropriate finite element solver, element type, element density, mesh refinement and material model.

FEM has been widely applied in industry to simulate static, dynamic, multi-physics and highly non-linear physical phenomena. In particular, FEM has been used to simulate manufacturing process chains, including forging, heat treatment and cutting of stainless steel SS316L [1]; multi-stage forging processes of carbon steel [2]; forming, material cutting and welding of frame structures in the vehicle industry [3]; extrusion and friction stir welding [4]; metal forming assembly [5].

To simulate a manufacturing process chain, finite element (FE) data may have to be transferred across different solvers and meshes. These issues are addressed by Afazov et al [6] in the development of FEDES, an FE data exchange system.[1] Afazov [7] reviews some manufacturing process chains of aero-engine components

*Corresponding author

Email addresses: Daniele.Scrimieri@nottingham.ac.uk (Daniele Scrimieri), Shukri.Afazov@the-mtc.org (Shukri M. Afazov), Svetan.Ratchev@nottingham.ac.uk (Svetan M. Ratchev)

[1] http://www.sourceforge.net/projects/fedes/

and their integration using FEDES. Tersing et al [8] simulates a manufacturing process chain of an aerospace component where FEDES is used for FE data transfer.

FEDES can transfer FE data between meshes with 2D and 3D solid linear and quadratic elements, including mixed element types. FEDES can read and write files compatible with six commercial packages, namely ABAQUS, ANSYS, DEFORM, Marc, Morfeo and Vulcan. It can also use a neutral XML file that can be visualised with the open source software ParaView. With relation to the interoperability of simulation software, a method using regular expressions to convert different types of file formats for finite element meshes is presented in [9]. Many mesh frameworks have been developed, including the Mesh-Oriented datABase (MOAB)[10], which can store and interpolate structured and unstructured mesh, and is optimised for efficiency in space and time. In MOAB, physical access to a mesh does not occur through individual entities, but in chunks. However, the MOAB interface is very flexible and supports individual entity access. GetFEM++[2] is a generic and efficient open source library for FEM elementary computations and offers interpolation methods and mesh operations.

In order to transfer an FE variable between a source and a target mesh, FEDES reads the elements and nodes of both of them and the values of the FE variable at the nodes of the source mesh. Solvers calculate FE variables at either nodal or integration points. If a variable (e.g. strain, stress) is calculated at integration points, the solver can usually obtain its values at nodal points by extrapolation. When transferring variables such as strain and stress, also the integration points of the target mesh have to be read or calculated. This is necessary because these variables are required by FE solvers to be associated with integration points. Four mapping methods are implemented in FEDES: a method using the nearest node, a method using fields of points, a method using elements and a method using the element shape function.

When transferring FE data across dissimilar meshes, a significant computational effort is spent in searching the source mesh for the nodes or elements specified by the mapping formulation. A sequential search requires a considerable amount of time, particularly for large meshes (i.e. more than 500,000 elements). The mapping time can be greatly reduced by creating a spatial index on the source mesh and then conducting a local search in the index instead of a sequential search in the entire mesh. Generally, a spatial index enables a rapid response to spatial queries, considering spatial relationships between objects such as points, lines, polygons. In the case of finite elements, the indexed objects are nodes and elements. Spatial data structures have been used extensively in computer graphics, databases, pattern recognition, solid modelling and other areas [11]. With relation to FE analysis, applications of spatial indexes include mesh generation [12], adaptive mesh refinement [13] and spatial contact search [14].

This paper presents a technique for performing searches in indexed meshes in order to map FE data between meshes within a manufacturing process chain. The meshes can have different densities or element types. An in-core grid index is created on the source mesh. For each nodal or integration point of the target mesh, the index on the source mesh is searched for a specific node or element, in accordance with the mapping method. The technique has been implemented and tested in FEDES.

The remainder of this paper is organised as follows: Section 2 presents the indexing technique, Section 3 describes the mapping methods employing the index structure, Section 4 contains an evaluation of the performance of index creation and mapping, Section 5 draws overall conclusions.

2. Indexing

The indexing technique that we propose partitions the 2- or 3-dimensional space underlying a mesh into a 2- or 3-dimensional orthogonal grid, respectively. The cells of the grid are not required to be equal-sized. This means that the splitting lines (for 2-dimensional meshes) or planes (for 3-dimensional meshes) do not have to be equidistant. Consequently, auxiliary structures, called *scales*, are necessary to specify the coordinates of the lines or planes subdividing each dimension.

The index access structure allows rapid location of the cell containing a given point and the points indexed in it. The spatial query we are interested in is the nearest neighbour query. That is, given a point

[2] `home.gna.org/getfem/`

in the underlying space, we want to find the nearest to it in the index. Of course, if the point itself has been indexed, it represents the answer. Some variants of the nearest neighbour query are employed in the mesh mapping methods described in this paper. Some mapping methods need to know which nodes are contained in a cell, while others need to know which elements are covered by a cell. Depending on the method, either a node index or an element index is used. A node index associates nodes with cells. An element index is a variation of a node index that contains also the elements of the indexed nodes.

Cells point to buckets where references to indexed nodes are stored. Each cell points to one bucket and several cells can point to the same bucket. Thus, the correspondence between cells and buckets is many-to-one. If some cells point to the same bucket we say that they share it and that such a bucket is *shared*. Similarly, a *non-shared* bucket is one that is pointed to by only one cell. All the buckets have the same fixed size in terms of number of references to nodes. In this work, all the data is kept in the main memory. Other solutions are also possible, where everything is stored on the disk or the mesh is on the disk and only the index structure is in-core.

The grid can be refined while indexing. An initial size is specified. While indexing nodes, new splitting lines or planes may be added, thereby creating new cells and increasing the grid size. A refinement occurs when a non-shared bucket *overflows*, i.e. a node being indexed cannot be inserted in it because it is full. A splitting line or plane passing through the cell pointing to the overflowing bucket is added. The overflowing bucket is also split into two. The following subsections describe in detail the indexing and splitting process.

The use of fixed-sized buckets and the splitting mechanism enable the grid to reflect the level of refinement of the mesh in every location. That is, the more refined the mesh is in a certain location, the more cells and buckets will be created in the corresponding location in the grid.

Our approach is similar to the grid file of Nievergelt et al [15], a database system where a directory associates each cell with a data bucket stored on a disk page. The grid file guarantees that a data item can be retrieved with only 2 disk accesses, one to the directory and one to the bucket. Another technique, called EXCELL, with the same objective of minimizing the number of disk accesses, is described in [16]. The difference from the grid file is that cells are equal-sized, so there is no need to maintain auxiliary structures to locate them. An index with equal-sized cells for FE data transfer is described in [17]. Although with this technique the identification of cells is faster (as it takes constant time), a refinement requires the splitting of all the cells to maintain the equal-size property. Therefore, at each refinement the number of cells doubles. With the method presented in this paper, a refinement involves only the introduction of a splitting line or plane and the update of the corresponding scale.

2.1. Grid representation

A 2- or 3-dimensional grid is represented by a 2- or 3-dimensional array, respectively. Each element of the array represents a cell of the grid, identified by the element's coordinates, and contains a pointer to the cell's bucket. When a 2-dimensional grid is refined, a new row or column is inserted in the array. When a 3-dimensional grid is refined, a new 2-dimensional array, a "slice", is inserted along one dimension. Insertion is performed by allocating a larger memory area and copying the cells' pointers. Since this operation is computationally expensive, the indexing mechanism tries to reduce the number of times it is required.

2.2. Scale representation

A grid spans a bounded interval along each dimension. A scale specifies the intervals into which a dimension of the grid is partitioned and the coordinate of each interval in the grid along the partitioned dimension. When a new splitting line or plane is added to the grid along one dimension, the corresponding scale is updated to reflect the change. Initially, all the splitting lines/planes are equidistant. However, this is not a requirement. In fact, the initial grid could be more refined where the mesh has more nodes. This would aim to reduce the number of subsequent refinements.

A scale contains the initial point of each interval of the partitioned dimension. To perform searches efficiently, values are sorted. Since their number is not known in advance, a static data structure such as an array is not suitable, as it would require reallocations and shifts whenever new elements are added. A tree structure (e.g. binary search tree, b-tree), instead, can grow dynamically and is therefore more appropriate.

Our implementation represents scales by binary search trees. If the tree is balanced, operations like insertion and searching take $O(\log_2(n))$ time in the worst case, since the number of items to check is halved at each level.[3] If the tree is a linear chain, the same operations run in $O(n)$ time in the worst case. Self-balancing binary search trees (e.g. red-black trees, AVL trees) and b-trees can adjust themselves so that they are always balanced (see, for example, [18]). However, in our application the relatively small size of the scales (i.e. less than 1000 values) and the relatively small number of grid refinements (i.e. less than 1000) did not justify a more complex implementation. In addition, the initial scales (i.e. the scales associated with the initial grid) are constructed in such a way that they are balanced. Thus, there should not be a significant degradation in performance, if any. An evaluation of the performance of four variants of binary search trees (unbalanced binary search trees, AVL trees, red-black trees and splay trees) in system software is contained in [19].

Let us briefly review the definition of a binary search tree. A binary tree is a tree whose nodes have at most two children. A binary search tree is a binary tree whose nodes store a key and, optionally, an additional value. Keys satisfy the following property:

Property 1. *(Binary-search-tree) For each node x, the following two conditions hold:*

1. *every key in the left subtree of x is less than x's key;*
2. *every key in the right subtree of x is greater than x's key.*

The binary-search-tree property allows a tree of depth h to be searched for a key in $O(h)$ time. From this property, it follows that keys are not duplicated. For a node x, let $key(x)$ be the key of x, $value(x)$ be the value of x, $left(x)$ be the left subtree of x, and $right(x)$ be the right subtree of x. If the left or right subtree is missing, then $left(x)$, respectively $right(x)$, is equal to *nil*.

A scale is represented by a binary search tree whose nodes represent the intervals of the partitioned dimension. The key of a node is the initial point of the node's interval, whereas the value of a node is the grid coordinate of the node's interval. A scale satisfies the following property:

Property 2. *(Contiguity) Let s be a scale partitioning a dimension in k intervals. An in-order tree walk[4] of s yields all nodes' values from 1 to k in ascending order.*

The contiguity property allows searches for grid coordinates to be performed efficiently. Similarly to keys, it guarantees that the search for a value in a tree of depth h takes $O(h)$ time.

Initial construction. The initial tree is constructed in such a way that it is balanced. Given a sorted sequence of points to insert, the algorithm consists of the following steps:

1. Select the midpoint of the sequence as root of the tree;
2. Apply recursively the algorithm to the left and right subtrees with all the points that are, respectively, less than and greater than the midpoint.

Note that if points were added in ascending or descending order, the tree would be a linear chain. Fig. 1 shows the initial scale for the intervals of length 0.5 between -2 and 2. Nodes are labelled by pairs of the form *key*/*value*.

Searching. The most frequent operation performed on a scale is searching. There are two types of searches:

1. Search for the grid coordinate of the interval containing a given point;
2. Search for the initial point of the interval having a given grid coordinate.

[3] A binary tree is balanced if, for each node x, the depths of the left and right subtrees of x differ by at most 1.

[4] i.e. by recursively visiting the tree in this order: left subtree, root, right subtree.

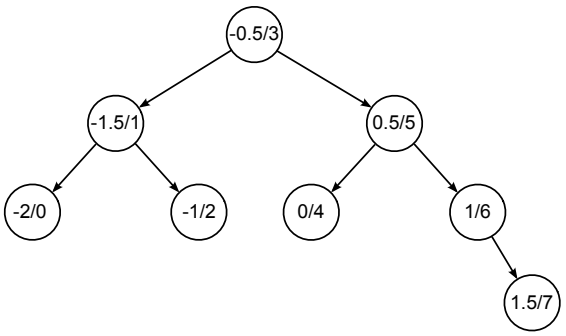

Figure 1: Initial scale for the intervals of length 0.5 between -2 and 2.

```
Function FindGridCoord (x, p)
if p = key(x) then
    return value(x)
else if p < key(x) then
    if left(x) ≠ nil then
        return FindGridCoord (left(x), p)
    else
        return value(x) − 1
else
    if right(x) ≠ nil then
        return FindGridCoord (right(x), p)
    else
        return value(x)
```

Figure 2: Search for the grid coordinate of an interval.

The first type of search works as follows. Let p be a point in the interval spanned by the grid along the partitioned dimension. The search starts from the root and, at each node x, p is compared to $key(x)$. Based on the result of the comparison, the search stops or continues in either the left or right subtree. The choice of which subtree to explore is made by applying the binary-search-tree property.

The algorithm is implemented in the function FindGridCoord shown in Fig. 2. A scale is identified with its root node. To understand the case when $p < key(x)$ and $left(x) = nil$ note the following. Let x' be the node such that $value(x') = value(x) - 1$ (which exists by definition of scale). By the contiguity property, there is no node y such that x' is in $left(y)$ and x is in $right(y)$. Thus, x must be in $right(x')$. This means that x' was examined before x during the search and, in particular, that $p > key(x')$. Hence, it is correct to return $value(x) - 1$. The case when $p > key(x)$ and $right(x) = nil$ is handled similarly.

The function LocateCell(n), which locates the cell containing a mesh node n, can be easily defined using FindGridCoord for each dimension.

The second type of search is simpler because the given grid coordinate is contained in one of the nodes. In this case the search is by the nodes' values and not the keys, and is made by applying the contiguity property. The function FindInitialPoint(x, i), which finds the initial point of the interval having grid coordinate i in a scale x, is assumed to be defined.

Splitting. Suppose that a split is to be made in an interval along the dimension partitioned by a scale. The midpoint of the interval is selected as the splitting point, as in general there is no knowledge of the distribution of points. Fig. 3 shows the result of splitting the interval with grid coordinate 1 in the scale of Fig. 1. The interval's midpoint -1.25 is inserted in the scale and the values of all the nodes whose key is greater than -1.25 are incremented by 1.

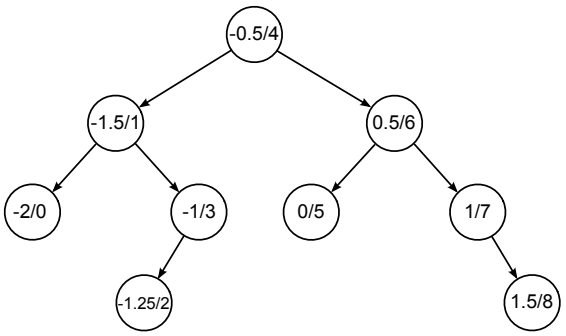

Figure 3: Splitting of interval 1: the interval's midpoint -1.25 is inserted in the scale and the nodes' values are updated.

To calculate the midpoint of an interval with grid coordinate i, the endpoints of the interval are retrieved from the scale by searching for the nodes with values i and $i+1$. The keys of these two nodes represent the endpoints. If i is equal to the size of the grid along the scale's dimension, then the high endpoint is that of the whole interval spanned by the scale.

Because of the update of grid coordinates, splitting in a scale of size n takes $O(n)$ time. An in-order traversal can be performed combining the search of the endpoints with the updating. When the traversal starts, the node with value i is searched for. When it is found, the traversal continues as follows. The next node that is encountered is the one with value $i+1$. The value of this last node, and of every node that is visited afterwards, is incremented by 1. The splitting point is then inserted, with associated value $i+1$.

Alternatively, given the node x representing the interval to split, the high endpoint can be found by accessing x's successor. To be implemented efficiently, the *successor* operator requires that the parent of a node be accessible [18]. The nodes to update can be accessed similarly by repeatedly applying the successor operator, starting from x.

2.3. Indexing of nodes

After the initial grid structure is created, nodes are indexed (i.e. added to the index) one by one. To index a node, the cell containing it is located using the scales. From an implementation point of view, only references to nodes are added to the index. Nodes are represented by their coordinates and stored in an array separate from the index. The position of a node in the array is used to identify the node. The same identification method is used to associate FE variables with nodes or interpolation points.

Figures 4-7 illustrate the indexing process. The axes indicate the intervals and their grid coordinates. An initial grid is constructed, with all the cells having the same size (Fig. 4). Associated with each cell there is a node bucket. The correspondence between cells and buckets is one-to-one at this point.

When a (non-shared) bucket overflows, the grid is refined along one dimension (Fig. 5). The dimension is chosen by cycling through all the dimensions. The interval, along the chosen dimension, containing the cell pointing to the overflowing bucket is split. Each cell in this interval is divided into two, both sharing the same bucket as the original cell, except for those resulting from the split of the cell c pointing to the overflowing bucket. The overflowing bucket is split into two, one for each of the two cells resulting from the split of c, and its content is redistributed between them, i. e. each of its nodes is inserted in the appropriate bucket based on the cell containing it. In the figures, a dashed line separating two cells indicates that they share the same bucket. Although buckets are depicted with different dimensions, they all have the same size in memory!

Grid refinement creates shared buckets. If a shared bucket overflows, the grid is usually not refined. Instead, the overflowing bucket is simply split into two: one for the cell containing the node being indexed and one for the other cells sharing it (Fig. 6). However, if after splitting a bucket (shared or not) and redistributing, the new bucket where the node being indexed is inserted is again full, then a (further) grid refinement is made (Fig. 7).

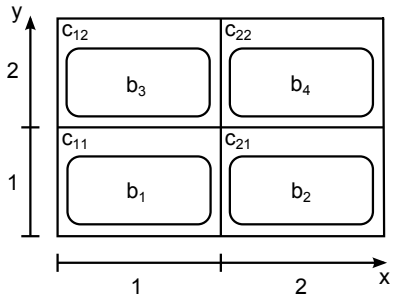

Figure 4: Initial grid of size 2×2; all cells have the same size.

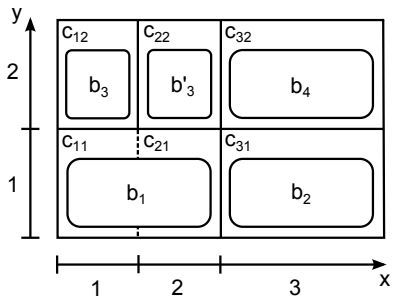

Figure 5: Bucket b_3 overflows: the grid is refined along the x-axis and b_3 is split into b_3 and b'_3.

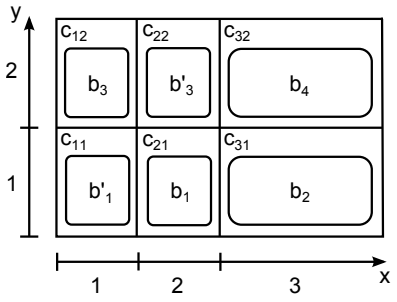

Figure 6: Bucket b_1 overflows: a split occurs and a new bucket, b'_1, is associated with C_{11}, the cell where the node being indexed is located.

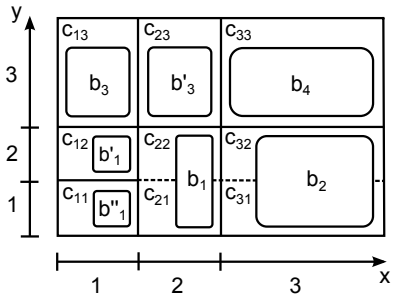

Figure 7: Bucket b'_1 is still full after redistribution: a further grid refinement is performed along the y-axis and b'_1 is split into b'_1 and b''_1.

```
Function RefineGrid (grid, axis, splitpos, cellb)
copy ← copyGrid(grid)
split grid at splitpos along axis
if axis = x then
    for i = 1 to splitpos do
        for j = 1 to y-size(grid) do
            SetCellBuf (grid[i, j], copy[i, j], cellb)
    for i = splitpos + 1 to x-size(grid) do
        for j = 1 to y-size(grid) do
            SetCellBuf (grid[i, j], copy[i − 1, j], cellb)
else
    for i = 1 to x-size(grid) do
        for j = 1 to splitpos do
            SetCellBuf (grid[i, j], copy[i, j], cellb)
    for i = 1 to x-size(grid) do
        for j = splitpos + 1 to y-size(grid) do
            SetCellBuf (grid[i, j], copy[i, j − 1], cellb)
foreach node n ∈ cellb do
    reinsert n in grid
```

Figure 8: Grid refinement.

Grid refinement is more computationally expensive than simply creating a new bucket and redistributing one bucket's nodes. Nonetheless, it occurs less and less often as nodes are indexed because the number of shared buckets increases while the number of non-shared buckets decreases.

Specifically, when indexing a node n in a cell c, three cases can be distinguished, depending on whether c's bucket, let us call it b, is full or shared:

1. *b is not full:* n is simply added to b.
2. *b is full and non-shared:* the grid is refined by splitting, along one dimension, the interval containing c; b is split and redistributed. If, after redistributing, the bucket of the cell containing n is not full then n is added to it. Otherwise, the grid is recursively refined until the bucket of the cell containing n is not full.
3. *b is full and shared:* a new bucket b' is created and associated with c. The content of b is redistributed between b and b'. The other cells sharing b are not changed. If, after redistributing, b' is not full then n is added to it. Otherwise, the grid is recursively refined until the bucket of the cell containing n is not full.

The function RefineGrid(*grid*, *axis*, *splitpos*, *cellb*) for performing one refinement on a 2-dimensional grid is shown in Fig. 8, where *grid* is split at *splitpos* along *axis* and *cellb* is an overflowing non-shared bucket pointed to by a cell having coordinate *splitpos* along *axis*. The procedure SetCellBuff(c_1, c_2, b) is defined as follows: if c_2's bucket is not b then SetCellBuff associates c_2's bucket with c_1, otherwise it associates a new bucket with c_1. For the sake of simplicity, RefineGrid creates two new buckets. However, it is sufficient to create only one and reuse the overflowing bucket.

The function RecRefineGrid(*grid*, n) for recursively refining *grid* and inserting n is shown in Fig. 9. In general, a grid refinement recursively splits a series of increasingly narrower intervals $i_1, i_2, \ldots, i_k$, where $i_{j+1} \subset i_j$. However, most of the times only one level of refinement is necessary.

2.4. Indexing of elements

Indexing of elements is carried out by indexing their nodes. Elements are represented by lists of nodes and are stored in a separate array. The indexing mechanism identifies elements by their position in the

```
Function RecRefineGrid (grid, n)
repeat
    choose axis
    get the coordinate x of n along axis
    splitpos = FindGridCoord(scale(axis), x)
    cellb = bucket(LocateCell(n))
    RefineGrid (grid, axis, splitpos, cellb)
until cellb is not full
```

Figure 9: Recursive grid refinement.

element array. The nodes of each element are indexed as described in Section 2.3. The bucket structure of an element index is extended to associate the nodes referenced in a bucket with their elements.

Since nodes are shared by several elements, they are processed multiple times while indexing. Before adding a reference to a node or element in a cell's bucket, a check can be made to see if it was already present. This allows duplicates to be avoided, thereby saving memory and simplifying redistribution and mapping.

Note that an element can overlap more than one cell. In such a case, each cell containing any of the element's nodes will reference the element. It can happen, however, that a cell is contained within an element or overlaps an element without containing any of its nodes. Such a cell will not reference the element. To deal with this case, the mapping methods presented in this paper examine neighbouring cells.

The nodes stored in an element index are not used when mapping, but only when indexing. In fact, they are only needed when redistributing elements between cells. When a cell is split, each element referenced in it is redistributed based on the redistribution of its nodes.

3. Mapping

Once an index has been built on the source mesh, the indexed nodes or elements can be mapped to nodal or integration points of the target mesh. The calculated FE variables can thus be transferred between the mapped entities. This section describes how to implement the mapping methods developed in FEDES using the proposed index. The method using the nearest node and the method using fields of points map nodes to nodes and thus require a node index, while the method using elements and the method using the element shape function map elements to nodes and thus require an element index.

Whether or not an index is used, mapping can be executed in parallel on a multiprocessing system by distributing the nodes or the elements of the target mesh across the running processes (or threads). Each process executes the same mapping code but on a different chunk of data. Processes can all access simultaneously the source mesh or the index without needing to synchronize, since the access is read-only.

3.1. Method using the nearest node

This method projects each node or integration point n of the target mesh into the space underlying the source mesh and finds the node which is nearest to n. The FE variables to transfer to n are taken from this node. The application of this method without a spatial index requires that each node of the source mesh be scanned, the distance between it and n be calculated, and the nearest node be identified. With the proposed index, the cell c containing n is located using the scales. Only the nodes indexed in c and, if required, in neighbouring cells are examined, thereby considerably reducing the search space.

The algorithm works as follows. First, cell c is located by finding its coordinates using FindGridCoord, its nodes are examined and the nearest is found. Then, if the distance between n and neighbouring cells is less than the current minimum distance (see Fig. 10) or no node has been found, the nodes of neighbouring cells are also examined. Neighbouring cells are located with increasing offsets from c, going outwards in all directions, i.e. all the adjacent cells along the 4 surrounding rows/columns in 2-dimensional grids (6 planes in 3-dimensional grids) are added to the search space. Usually neighbouring cells do not need to

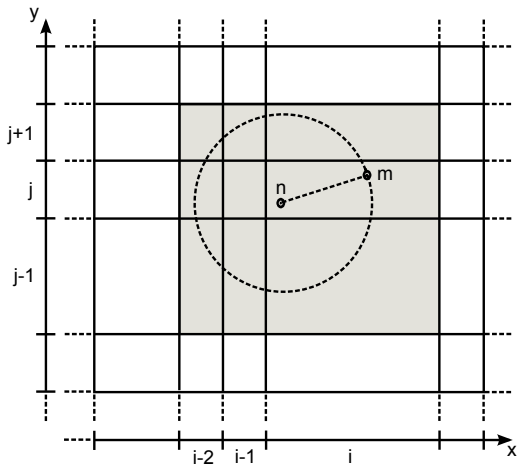

Figure 10: Nearest neighbour search in a grid. Node m is the nearest to n in the cell (i, j) containing n. The other grey cells with coordinates (x, y) such that $i - 2 \leq x \leq i$ and $j - 1 \leq y \leq j + 1$ must be examined because they may contain a nearer node.

be examined at all or the search stops at very small offsets. However, if the grid is very refined and the difference in density between source and target mesh is relatively big, then the search might stop at higher offsets.

The function FindNearestNode for searching a 2-dimensional grid index is shown in Fig. 11 (the function can be easily extended to search a 3-dimensional grid index by handling the additional dimension similarly). Instead of calculating the distance between n and each neighbouring cell, FindNearestNode calculates just one distance for each direction, that is the distances between n and the lower and upper bounds, on the x- and y-axis, of the rectangular area already examined. The search in direction i stops at the offset where the distance $d(n, b_i)$ between n and bound b_i is greater than or equal to the current minimum distance min, or there are no more cells to examine. The function EndSearch, which is used in the termination condition of the while loop, indicates when the search stops in every direction. EndSearch(k, n, min, b_1, b_2, b_3, b_4) is equal to *true* when, for $i = 1, \ldots, 4$, either $d(n, b_i) \geq min$ or there are no more cells to examine along direction i. The border of the examined area along each direction is determined using FindInitialPoint with the corresponding scale and the coordinate of the interval whose initial point delimits the area. This technique reduces the number of distances calculated at each iteration to the number of directions but, being a conservative approximation, may cause some cells to be unnecessarily scanned.

If some cells share a bucket, such a bucket will be scanned multiple times, once for each cell sharing it. Scanned buckets can be flagged as such to avoid being re-scanned. To indicate that a bucket has been scanned during the mapping of a node, it is flagged with the identifier of the node (i.e. its position in the node array). When analysing a cell, its bucket's flag is compared to the identifier of the node currently being mapped. If they are equal it means that that bucket has already been scanned.

3.2. Method using fields of points

This method projects each node or integration point of the target mesh into the space underlying the source mesh. The projection of a point n represents the origin a Cartesian coordinate system whose axes are parallel to the grid's axes. Such a coordinate system partitions a 2-dimensional grid in 4 regions and a 3-dimensional grid in 8 regions. In each region a nearest neighbour search is performed, similarly to the method using the nearest node. For each cell c being examined, the region containing it or the regions overlapping it are determined. For each of these regions, let it be r, the search proceeds as follows. 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 for the nearest node in r.

Function FindNearestNode ($grid$, n)
$i \leftarrow$ FindGridCoord (x-$scale$, n)
$j \leftarrow$ FindGridCoord (y-$scale$, n)
search cell (i, j)
x-$lower$-$bound$ = FindInitialPoint (x-$scale$, i)
x-$upper$-$bound$ = FindInitialPoint (x-$scale$, $i + 1$)
y-$lower$-$bound$ = FindInitialPoint (y-$scale$, j)
y-$upper$-$bound$ = FindInitialPoint (y-$scale$, $j + 1$)
$k = 1, min = \infty$
while EndSearch *(k, n, min, x-lower-bound, x-upper-bound, y-lower-bound, y-upper-bound) = false* **do**
  if $i + k < x\text{-}size(grid) \wedge d(n, x\text{-}upper\text{-}bound) < min$ **then**
    search all cells $(i + k, j')$ such that $\max(j - k, 0) \leq j' \leq \min(j + k, y\text{-}size(grid) - 1)$
  if $i - k \geq 0 \wedge d(n, x\text{-}lower\text{-}bound) < min$ **then**
    search all cells $(i - k, j')$ such that $\max(j - k, 0) \leq j' \leq \min(j + k, y\text{-}size(grid) - 1)$
  if $j + k < y\text{-}size(grid) \wedge d(n, y\text{-}upper\text{-}bound) < min$ **then**
    search all cells $(i', j + k)$ such that $\max(i - k, 0) \leq i' \leq \min(i + k, x\text{-}size(grid) - 1)$
  if $j - k \geq 0 \wedge d(n, y\text{-}lower\text{-}bound) < min$ **then**
    search all cells $(i', j - k)$ such that $\max(i - k, 0) \leq i' \leq \min(i + k, x\text{-}size(grid) - 1)$
  x-$lower$-$bound$ = FindInitialPoint (x-$scale$, $i - k$)
  x-$upper$-$bound$ = FindInitialPoint (x-$scale$, $i + 1 + k$)
  y-$lower$-$bound$ = FindInitialPoint (y-$scale$, $j - k$)
  y-$upper$-$bound$ = FindInitialPoint (y-$scale$, $j + 1 + k$)
  $k = k + 1$

Figure 11: Nearest node search.

The value of an FE variable to transfer to n is obtained by calculating a weighted mean of the values of the variable for the neighbours found. The weight associated with each neighbour is inversely proportional to the distance between it and n. If n coincides with a node of the source mesh, the value of this node is copied. If n does not fall in the source mesh or is close to its surface, there may be regions without any node. In this case the mean is calculated only on the neighbours found. To avoid empty regions being searched, a limit on the offset can be set. This way, the search in a region stops if no node has been found and the limit has been reached.

The weight w_i assigned to neighbour i ($1 \leq i \leq k$) is

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

where d_i is the distance between i and n. Let $x_i, \ldots, x_k$ be the values of an FE variable for the nearest neighbours. The value y to transfer to n is

$$y = \frac{\sum_{j=1}^{k} w_j x_j}{\sum_{j=1}^{k} w_j} \tag{2}$$

3.3. Method using elements

This method projects each node or integration point n of the target mesh into the space underlying the source mesh and finds the nearest element to n. The distance between an element and a point is defined as the average distance between the nodes of the element and the point.

The application of this method without a spatial index requires that each element of the source mesh be scanned, the distance between it and n be calculated, and the nearest element be identified. With the proposed index, the cell c containing n is located using the scales. Only the elements indexed in c and in

neighbouring cells are examined, thereby considerably reducing the search space. The elements indexed in a cell are those of the nodes indexed in the cell and are contained in the cell's bucket.

The search starts from c. If there are elements in c's bucket, the nearest is identified. Whether or not elements have been found, the search continues in neighbouring cells, with increasing offsets from c, as described for the method using the nearest node in Section 3.1. This is necessary because, if an element overlaps c without having nodes in it, it has to be found in neighbouring cells.

Usually elements are found at very small offsets, unless the grid is very refined and the difference in density between source and target mesh is relatively big. The search stops at the lowest offset, greater than a fixed lower bound, where at least one element has been found. To achieve the best accuracy with this method, the lower bound should be such that the corresponding area encloses all the elements overlapping c. This value depends on the level of refinement of the grid and the difference in density between source and target mesh. The more refined the grid is and the higher the difference in density is, the higher the lower bound should be.

An element can be processed multiple times during the same search if it has nodes in different cells. This can be avoided by flagging the examined elements, similarly to buckets (as described in Section 3.1).

The value of an FE variable to transfer to n is obtained by calculating a weighted mean of the values of the variable for the nodes of the nearest element. If n coincides with a node of the source mesh, the value of this node is copied. The weights are inversely proportional to the distances and are calculated using (1), with the nearest element's nodes as neighbours. The weighted mean is calculated using (2), with $x_1, \ldots, x_k$ being the values of the nearest element's nodes.

3.4. Method using the element shape function

This method projects each node or integration point n of the target mesh into the space underlying the source mesh and finds the element that contains n. The shape function of the element type is used to interpolate FE data at n. The search in the index is performed similarly to the method using elements. That is, the cell c containing n is located using the scales. Neighbouring cells are then examined going outwards from c.

Shape functions are used for interpolating from nodes to integration points or, in general, any point within an element. Let D_i be the value of an FE variable at node i and η_i be the shape function at node i, for $0 \leq i \leq k$. The interpolated value at n is:

$$D_n = \sum_{i=1}^{k} \eta_i D_i \tag{3}$$

For applying (3), the local coordinates of n with respect to the element containing it must be determined. The systems of equations to solve to find the local coordinates for linear triangular, linear quadrilateral, linear tetrahedron, linear hexahedron and linear wedge elements can be found in [6]. When searching, n may not be located in any element, as the geometries of the two meshes may be slightly different. In this case, the ranges of the local coordinates are updated by incrementing a tolerance parameter and the search restarts.

4. Performance evaluation

The performance of the proposed indexing technique was evaluated in the simulation of two manufacturing process chains. We analysed one mapping in each of them. The FE variables to transfer were displacement and stress. The first chain was relative to an aero-engine vane component. The mapping was performed from heat treatment to machining. Source and target meshes had, respectively, 83,316 and 154,407 tetrahedron elements. Fig. 12 shows the mapping of displacements. The second chain was relative to a piston. Source and target meshes had, respectively, 9,387 hexahedron elements and 28,635 tetrahedron elements. Fig. 13 shows the mapping of stresses.

Let A and A' be, respectively, the source and target meshes of the former mapping and let B and B' be, respectively, the source and target meshes of the latter mapping. Table 1 reports the times taken for

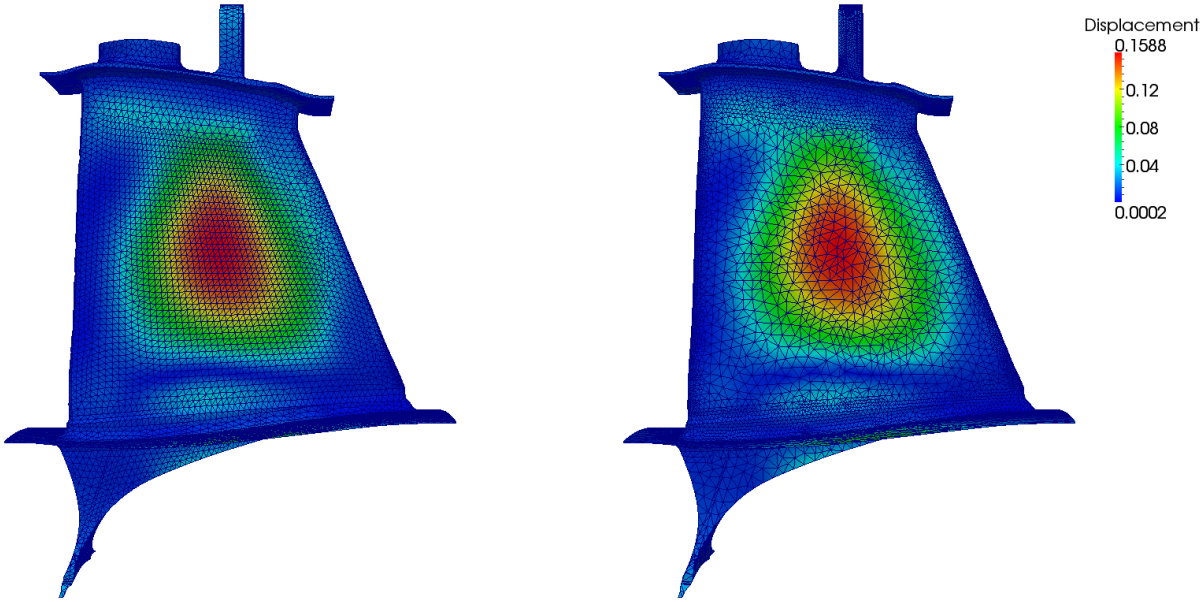

Figure 12: Mapping of displacements (using the element shape function). The source mesh (left) has 83,316 tetrahedron elements, while the target mesh (right) has 154,407 tetrahedron elements.

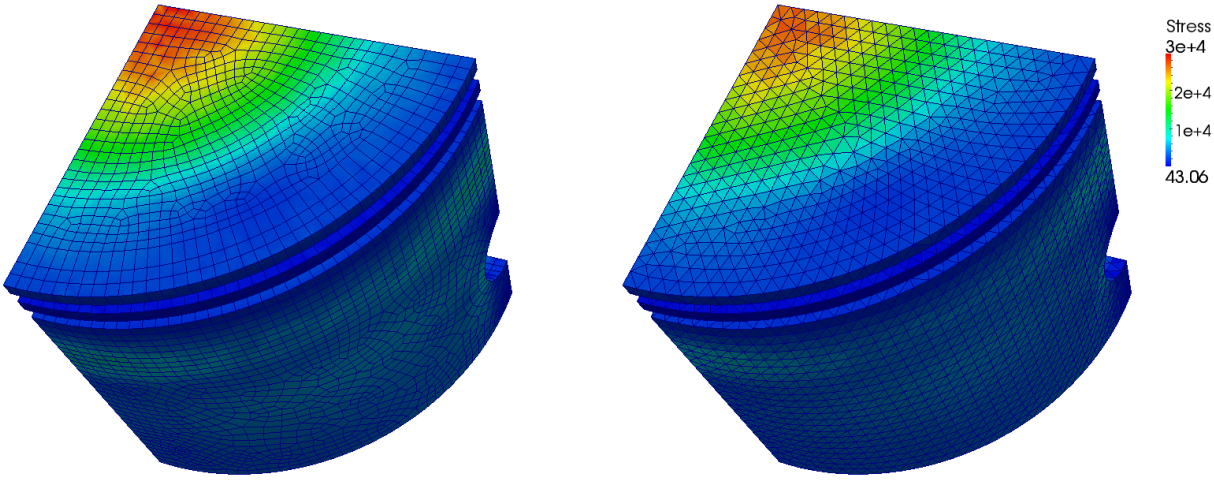

Figure 13: Mapping of stresses (using the element shape function). The source mesh (left) has 9,387 hexahedron elements, while the target mesh (right) has 28,635 tetrahedron elements.

Index	Mesh	Type	Initial grid size	Bucket size	Construction time (s)	Final grid size	Buckets	Splits
1	A	nodes	$25 \times 25 \times 25$	15	0.16	$50 \times 49 \times 49$	16,965	1,340
2	A	elements	$50 \times 50 \times 50$	60	2.951	$105 \times 105 \times 105$	127,454	2,454
3	B	nodes	$15 \times 15 \times 15$	10	0.018	$25 \times 25 \times 25$	4,176	801
4	B	elements	$25 \times 25 \times 25$	20	0.129	$46 \times 45 \times 45$	17,564	1,939

Index	Non-empty buckets (%)	Avg. bucket occupancy (%)
1	21	46
2	8.9	48.8
3	52.7	55
4	40.5	52.5

Table 1: Construction of the indexes.

	Mesh $A \rightarrow$ Mesh A'			Mesh $B \rightarrow$ Mesh B'		
Threads	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$
1	57.05	3.91	14.59	5.11	0.57	8.96
2	33.86	2.51	13.44	3.1	0.39	7.95
4	24.42	1.82	13.42	2.21	0.27	8.19
8	24.22	1.62	14.95	2.2	0.24	9.17

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

creating a node index and an element index on meshes A and B, and some statistics. Tables 2 to 5 report the time taken for mapping displacements and stresses from A to A' and from B to B' with each of the described mapping methods, comparing a search using the index with a sequential search. The mapping times do not include the construction times of the indexes. Each mapping was executed with 1, 2, 4 and 8 threads.

The experiments were conducted on a machine equipped with an Intel i7 CPU at 1.73 GHz and 8 GB of RAM, running GNU/Linux. The indexing and mapping code was written, as the rest of FEDES, in Pascal and compiled with the Free Pascal Compiler (version 2.6.0) with default optimizations. The reported times are wall-clock times, each calculated as the mean of three runs, measured using the Pascal library function `Now()`.

The initial grid size and the bucket size of the indexes were arbitrarily chosen based on the size of the mesh and are indicated in Table 1. The construction of the index took about 3 seconds for the largest (i.e. the element index of mesh A) and less than 0.2 seconds for the others. Such short times suggest that there is no need to save on the disk indexes of those dimensions if they are needed for future mappings, as they

	Mesh $A \rightarrow$ Mesh A'			Mesh $B \rightarrow$ Mesh B'		
Threads	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$
1	157.04	37.98	4.13	15.37	3.74	4.11
2	97.53	23.9	4.08	9.36	2.32	4.03
4	68.2	16.2	4.21	6.64	1.62	4.1
8	64.04	15.38	4.16	6.18	1.45	4.26

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

	Mesh $A \rightarrow$ Mesh A'			Mesh $B \rightarrow$ Mesh B'		
Threads	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$
1	1880.34	21.83	86.14	45.28	2.32	19.52
2	1057.16	14.57	72.56	26.94	1.53	17.61
4	677.83	9.14	74.16	19.01	1.06	17.93
8	601.35	8.09	74.33	18.65	0.92	20.27

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

	Mesh $A \rightarrow$ Mesh A'			Mesh $B \rightarrow$ Mesh B'		
Threads	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$	Sequential search	With index	$\frac{\text{Seq.}}{\text{Index}}$
1	1056.75	78.46	13.47	28.69	2.82	10.17
2	703.76	73.95	9.52	17.3	2.24	7.72
4	481.12	46.73	10.3	12.47	1.62	7.7
8	367.68	36.78	10	11.35	1.31	8.66

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

can be quickly recreated. By comparing the final grid size with the initial one, it can be observed that, for mesh A, the node index's grid and the element index's grid were refined, respectively, 73 and 165 times. While for mesh B, the node index's grid and the element index's grid were refined, respectively, 30 and 61 times. The number of splits in the table includes splits of both shared and non-shared buckets.

The average bucket occupancy of the indexes is roughly 50%. This is calculated on the non-empty buckets only, because for empty buckets no memory is actually allocated. In fact, a bucket is allocated when the first node is inserted in it. Note that reducing the bucket size does not necessarily increase bucket occupancy because more splits may occur, thereby resulting in a larger number of buckets. To save space, another approach to memory management could be followed. That is, one bucket's space would be allocated incrementally as nodes are indexed in it, up to a certain limit. When the limit is reached a split would take place.

It can be observed that mapping with an index was much faster than mapping with sequential searches in all the examined cases. In the worst case (i.e. with the method using fields of points), mapping with the index was still 4 times faster. Even if the index build time is included, mapping times with the proposed methods are still very short compared to the linear approach.

The difference in performance between running 4 and 8 threads is not always significant. This is due to the hardware architecture of the machine used for the experimentation, which has 4 cores but can handle up to 8 threads (2 per core). Although at most 4 threads can be run simultaneously, the switch between threads on the same core is fast.

5. Conclusions

An in-core grid index for transferring FE data between meshes in a manufacturing process chain has been presented. A grid decomposes the embedding space of a mesh into cells which can have different sizes. A scale for each dimension allows the cell containing a given node to be located. Scales are represented by binary search trees. The splitting mechanism enables the grid to reflect the level of refinement of the mesh. Four mapping methods employing the index have been described, namely, the method using the nearest node, the method using fields of points, the method using elements and the method using element shape functions.

An evaluation of the performance of index creation and mapping has been conducted. The results indicate that, in the examined cases, mapping with the proposed index is much faster than mapping with sequential searches for all the mapping methods, even including index build time. The creation of the largest index took about 3 seconds, whereas it took a fraction of second in the other cases. The use of the index structure involves a little memory overhead that, we believe, is not an issue in most applications. In addition, it can be reduced with an incremental allocation of a bucket's memory. Future work includes a comparison with other indexing techniques (based on, for instance, octrees or other grid structures) and an evaluation with meshes of larger sizes (of the order of millions of elements).

Acknowledgments

We would like to thank Jonathon Shaw for his comments on a draft of this paper.

[1] S. Hyun, L. Lindgren, Simulating a chain of manufacturing processes using a geometry-based finite element code with adaptive meshing, Finite Elements in Analysis and Design 40 (2004) 511–528.
[2] M. Pietrzyk, L. Madej, S. Weglarczyk, Tool for optimal design of manufacturing chain based on metal forming, CIRP Annals - Manufacturing Technology 57 (2008) 309–312.
[3] M. Zaeh, L. Papadakis, M. Langhorst, Simulation of the manufacturing process chain of welded frame structures, Production Engineering 2 (2008) 385–393.
[4] M. Zaeh, A. Tekkaya, M. Langhorst, M. Ruhstorfer, A. Schober, D. Pietzka, Experimental and numerical investigation of the process chain from composite extrusion to friction stir welding regarding the residual stresses in composite extruded profiles, Production Engineering 3 (2009) 353–360.
[5] A. Govik, L. Nilsson, R. Moshfegh, Finite element simulation of the manufacturing process chain of a sheet metal assembly, Journal of Materials Processing Technology 212 (7) (2012) 1453–1462.
[6] S. Afazov, A. Becker, T. Hyde, Development of a finite element data exchange system for chain simulation of manufacturing processes, Advances in Engineering Software 47 (1) (2012) 104–113.
[7] S. Afazov, Modelling and simulation of manufacturing process chains, CIRP Journal of Manufacturing Science and Technology 6 (1) (2013) 70–77.
[8] H. Tersing, J. Lorentzon, A. Francois, A. Lundbäck, B. Babu, J. Barboza, V. Bäcker, L.-E. Lindgren, Simulation of manufacturing chain of a titanium aerospace component with experimental validation, Finite Elements in Analysis and Design 51 (2012) 10–21.
[9] P. Iványi, Finite element mesh conversion based on regular expressions, Advances in Engineering Software 51 (2012) 20–39.
[10] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, C. Ernst, MOAB: a mesh-oriented database, Technical report SAND2004-1592, Sandia National Laboratories (2004).
[11] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, 2006.
[12] T. Tu, D. O'Hallaron, J. Loopez, Etree: a database-oriented method for generating large octree meshes, Engineering with Computers 20 (2004) 117–128.
[13] S. Popinet, Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries, Journal of Computational Physics 190 (2003) 572–600.
[14] P. Wriggers, G. Zavarise, Computational contact mechanics, in: Encyclopedia of Computational Mechanics, John Wiley & Sons, Ltd, 2004.
[15] J. Nievergelt, H. Hinterberger, K. C. Sevcik, The grid file: An adaptable, symmetric multikey file structure, ACM Transactions on Database Systems 9 (1) (1984) 38–71.
[16] M. Tamminen, R. Sulonen, The excell method for efficient geometric access to data, in: Proceedings of the Nineteenth Design Automation Conference, 1982, pp. 345–351.
[17] D. Scrimieri, S. Afazov, A. Becker, S. Ratchev, Fast mapping of finite element field variables between meshes with different densities and element types, Advances in Engineering Software 67 (2014) 90–98.
[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 2009.
[19] B. Pfaff, Performance analysis of BSTs in system software, SIGMETRICS Perform. Eval. Rev. 32 (1) (2004) 410–411.