Advances in Engineering Software 88 (2015) 53–62

Contents lists available at ScienceDirect

Advances in Engineering Software

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

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

ARTICLE INFO

Article history:
Received 20 January 2015
Received in revised form 2 May 2015
Accepted 2 June 2015
Available online 19 June 2015

Keywords:
Manufacturing chain
Simulation
Finite element data
Mesh mapping
Grid indexing
Nearest neighbour search

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.

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 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

* Corresponding author.
E-mail addresses: Daniele.Scrimieri@nottingham.ac.uk (D. Scrimieri), Shukri.Afazov@the-mtc.org (S.M. Afazov), Svetan.Ratchev@nottingham.ac.uk (S.M. Ratchev).

[1] http://www.sourceforge.net/projects/fedes/.
[2] home.gna.org/getfem/.

http://dx.doi.org/10.1016/j.advengsoft.2015.06.001
0965-9978/

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 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