Computer Assisted Methods in Engineering and Science, **23**: 3–17, 2016.

Design of interface modules for flexible coupling of finite element codes with solvers of linear equations

Krzysztof Banaś, Kazimierz Chłoń
AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Kraków, Poland
e-mail: pobanas@cyf-kr.edu.pl

This paper presents the design of flexible interfaces between finite element (FE) codes and solvers of linear equations. The main goal of the design is to allow for coupling FE codes that use different formulations (linear, non-linear, time dependent, stationary, scalar, vector) and different approximation techniques (different element types, different approximation spaces – linear, higher order, continuous, discontinuous, *h*- and *hp*-adaptive) with solvers of linear equations that use different storage formats for sparse system matrices and different solution strategies (such as, e.g., reordering of degrees of freedom (DOFs), multigrid solution or preconditioning for iterative solvers, frontal and multi-frontal strategies for direct solvers). Suitable data structures associated with the design are presented and examples of algorithms related to the interface between the FEM codes and linear solvers, together with their execution time and performance estimates, are described.

Keywords: finite element method, solvers of linear equations, *hp*-adaptivity, multigrid, multi-frontal strategies.

1. Introduction

The problem of optimal coupling of FE codes with solvers of linear equations can be analyzed from different perspectives. In the 1990s and 2000s this problem was investigated in the context of new programming languages and paradigms – object- and component-oriented (see, e.g., [1, 2, 9, 22, 23, 38]). Recently, the attention has been focused on the problem of efficiently utilizing the new computer architectures such as GPUs, massively multi-core processors and heterogeneous clusters [10, 15, 18, 24, 31, 34]. These new architectures favour data structures for which efficient vector accesses to data are enabled. For many codes such requirements induce the necessity to design new data structures or to design suitable interfaces between the currently utilized structures and the new ones.

Adaptive FE codes have to be equipped with sophisticated mesh data structures that allow for mesh adaptations and, possibly, dynamic load balancing in the message passing execution environments [7, 33]. Such data structures represent complex graphs that are difficult to transform to linear structures required by massively multi-core architectures. Fortunately, the execution times of operations related to mesh data are usually much smaller than the times associated with the creation and solution of systems of linear equations during the typical FE simulations [4]. Hence, for the efficient implementation of FE algorithms on modern hardware, only these two latter phases require the use of suitable linear structures. Such structures have to be designed together with the interface modules that translates the data from the FE mesh data structures.

The current paper is devoted to the description of one of such interface modules. First, the assumptions concerning the data structures associated with the FE meshes are presented, followed by the description of possible solver data structures. Then, a data structure is proposed for the interface coupling mesh and FEM solver. Finally, the use of the proposed data structure in several operations during the FEM solution procedure is presented. To show the advantages of the proposed design several computational experiments are performed for testing the performance estimates derived for the new data structures. We draw the summarizing conclusions at the end of this paper.

2. Fe mesh and solver data structures

FE meshes are understood in the current article as combinations of purely geometrical objects (termed "mesh entities") with the FEM solution objects, i.e., the FE DOFs (FE solutions are represented as the linear combinations of the FEM basis functions, and DOFs are the coefficients of these combinations) [30]. For the most popular linear FEM approximations DOFs are associated with the vertices of FEs, forming the FE nodes. For higher-order approximations, DOFs can be associated with mesh entities of any type. In such a case, they are often called generalized nodes [13]. In the current paper, we use the notion of a DOF entity that corresponds to any mesh entity with which a set of DOFs has been associated.

We concentrate on volumetric FEs and 3D problems (the application of the investigations in the paper to 2D or 1D problems is straightforward) and, hence, we consider four types of mesh entities: vertices, edges, faces (quadrilateral and triangular) and elements (tetrahedral, hexahedral, prismatic, pyramidal). These entities are interlinked by several relationships of different kinds: some entities *belong* to other entities, some entities are *composed* of other entities, some entities are *neighbours* of other entities, and some entities are *parents* of other entities (i.e., they have been divided into these entities, with the reverse relation of being a *child* entity).

To accommodate such diverse relations, complex data structures for mesh entities are designed [32]. We assume that these data structures are stored in standard CPU memories that are optimized for low-latency accesses.

Parallel to mesh data structures, the structures for storing the DOFs data are designed. They are usually simpler, and with less relationships, but they must follow the changes to mesh data structures induced by the FEM adaptivity processes. Hence, we also assume that they are stored in CPU main memories (at least during the mesh adaptation processes).

We want to design such a FEM code-linear solver interface that minimizes the number of memory accesses to the FE mesh and approximation data structures during the solution procedure. In order to do that we analyse the process of solving a single FEM problem. Such a problem may constitute a whole FEM simulation (but then, the problem of efficient execution may be of less importance) or may form one step in complex iterative strategies for non-linear and/or time dependent problems. We take into account the latter situation, since for such types of problems the efficiency of execution becomes even more important factor.

The process of solving a FE system of linear equations can be divided into three phases: the creating solver data structures, filling structures with particular problem data and solving the obtained system [3]. For systems solved for the same mesh (and approximation) many times during iteration processes, the same solver data structures can be reused, with only values of variables changed. Hence, the first phase of the process should be clearly separated in the design of solver interface modules. Similarly, the second and the third phases should be separated due to the fact that the second phase involves interactions between the FEM code and the solver, while the third phase is usually performed independently by the solver of linear equations.

Different solvers for the FE systems of linear equations, direct as well as iterative, can have different forms, especially when they are designed for complex problems and approximation types [8, 28, 29]. Nevertheless each such a solver has to use the entries obtained by integration (usually numerical) of the terms from the weak statement of the approximated problem. The integration is performed over the whole computational domain, divided into FEs, using the locally polynomial shape functions. The calculated entries contribute to the global system of linear equations, but in different solution strategies they may be used in several different ways. Contributions from an integration performed for a single element can be used separately or can be grouped into an element stiffness matrix and an element load vector. They can be used “on-the-fly”, just after calculating, in explicit or matrix-free methods [10, 19], they may be left in the form of element data structures for the use of frontal and multi-frontal solvers [27] or they may be assembled into a global, sparse system matrix and a vector of right-hand side.

In each case, the most important information associated with each individual entry is its position in the global system of linear equations. For each entry in the system matrix (and correspondingly in the right-hand side vector) its row and column positions must be specified. The form of this information is usually called “local to global” mapping, reflecting the essential meaning: at which position a given entry, obtained locally and hence having some local position, related to the local numbering of DOFs, should be assembled in the global system of equations. In traditional FE codes, a “local to global” mapping is related to the local and global numbering of FE nodes (usually vertices of FEs, see, e.g., Fig. 1). We adopt a more general perspective, where the global to local mapping explicitly reflects the relation between positions in local and global data structures.

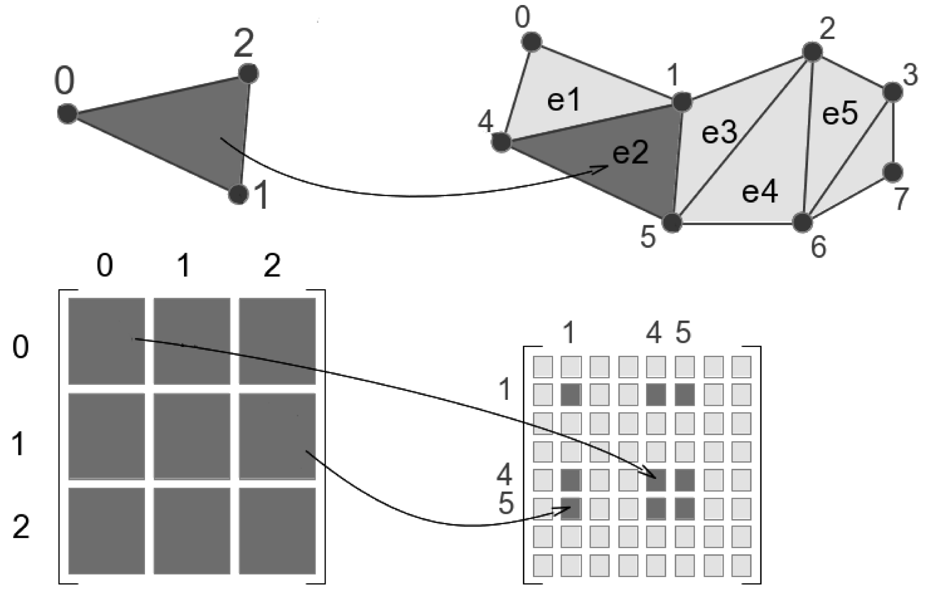

Fig. 1. Illustration of the FE assembly process.

We present a design of data structures and some related algorithms for storing the “local to global” mapping data. The proposed data structures can be used in different strategies of using integrated terms in the FE solution procedures. In the examples of practical applications, we concentrate on a situation, typical for FE codes, where the system matrix and right-hand side vector entries are obtained by the summation of contributions related to individual FEs. Small, dense element matrices and vectors are produced by the FE part of a code and should be submitted to the solver with the suitable information about the positions at which they should be assembled into the global structures.

3. Fe assembly process

3.1. Global matrix storage formats

The input to the FE assembly is formed by element system matrices and right-hand side (RHS) vectors. Their entries are added at proper places in the global system matrix and the right-hand side vector. The global vector of unknowns forms the third part of the system of linear equations. A single entry in the FE global vector of unknowns corresponds to a single DOF. The corresponding row of the global system matrix and the corresponding entry in the global right-hand side vector contain data related to this DOF. Similarly, due to properties of the FE approximation, each column of the system matrix also corresponds to a single DOF (see Fig. 1). As a result, each entry in the global system matrix corresponds to a pair of DOFs. The appearance of non-zero entries in the system matrix is related to properties of the FE approximation: DOFs, forming a pair for which a non-zero entry exists, are associated with the mesh entities that are close to each other in the mesh (including the case of DOFs associated with the same mesh entity). For such DOFs and DOF entities we say that there exist the relation of neighbourhood between them (hence, for each pair of neighbouring DOFs or DOF entities there exist one or several non-zero entries in the global system matrix). Since each mesh entity is close to only several other mesh entities, the system matrix is sparse and a single row in the matrix has usually at most several tens of non-zero entries (or several hundreds for vector problems and/or higher-order approximations). Since for large scale problems the total number of DOFs easily exceeds several millions, the number of non-zero entries may be smaller than 0.01%.

There are many storage formats for sparse matrices [6], row or column oriented, for single entries or blocks of entries, or even special hybrid formats developed recently for the GPU solvers [21]. All these formats are used in practice and their optimality depends on many factors. There is no single format considered optimal for all problems solved, approximations used and hardware employed. There are, however, several formats, such as COO, CRS or CCS [6], which are frequently used, due to their simplicity and flexibility. They are also often used as base formats for more sophisticated storage options [20], which are often designed with special functions to effectively transform sparse matrices from the base formats.

3.2. CRS storage format

We concentrate on one such format – the compressed row storage (CRS, sometimes also called compressed sparse row, CSR) [6]. In this format, all non-zero entries in a matrix are stored contiguously in one array, usually named `val`, row by row. The second array, named usually `row_ptr`, stores for each row the position of its first entry in the `val` array, while the third array, named `col_ind`, stores for each entry in `val` its column index. The format was designed with the purpose of efficiently performing the matrix-vector products for sparse matrices. The basic algorithm for the CRS (with several details omitted) looks as follows:

```
for(row=0; row<number_of_rows; row++) {
  for(i=row_ptr[row]; i<row_ptr[row+1]; i++) {
    y[row] += val[i] * x[ col_ind[i] ];
} }
```

One of the advantages of the CRS format is that the matrix-vector product presented above allows for easy and efficient parallelization, since the outer loop over rows has no dependencies. The CRS storage is also used for many direct solvers (such as, e.g., open source solvers PARDISO, SuperLU) as the base storage option. In the article we assume the CRS format as the base format for storing the FEM global system matrices and concentrate on assembly using the CRS, leaving the question of adapting to other formats to specialized solver procedures.

3.2.1. Iterative solvers and preconditioning with CRS

Despite enabling the efficient parallel execution of matrix-vector product, the CRS format allows also for efficient implementation of other important operations, e.g., operations related to preconditioning, which form a crucial ingredient of popular Krylov space iterative solvers, like preconditioned conjugate gradient or GMRES. As an example, we show the possibility of implementing the widely used Gauss-Seidel preconditioning that we employ in our computational experiments reported later in the paper.

The original Gauss-Seidel algorithm for solving systems of linear equations

$$\sum_{j=1}^{N} A_{ij} x_j = b_i, \qquad i = 1, ..., N$$

uses subsequent iterations, repeated until convergence, and a loop over subsequent rows of the system matrix for a single iteration. For k-th iteration, the operations for i-th row looks as follows:

$$x_i^{k+1} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{k+1} - \sum_{j=i+1}^{N} A_{ij} x_j^{k} \right).$$

Using the CRS format the algorithm (with several details omitted) can be implemented as

```
for(row=0; row<number_of_rows; row++) {
  for(i=row_ptr[row]; i<diag_ptr[row]; i++) {
    x[row] += val[i] * x[ col_ind[i] ];
  }
  for(i=diag_ptr[row]+1; i<row_ptr[row+1]; i++) {
    x[row] += val[i] * x[ col_ind[i] ];
  }
  x[row] = ( b[row] - x[row] ) * diag_precon[row]
}
```

The only additional data structures required are: `diag_ptr` array storing, for each row, the position of the diagonal entry in the `val` array and the `diag_precon` array storing inverted diagonal entries of the system matrix. We use this algorithm directly for preconditioning, without splitting into the matrix-vector product and triangular solution.

The algorithm above does not allow for direct parallelization, because of the real data dependency in each iteration over rows of the system matrix. There are several approximate parallelizations [35], in our computational experiments we employ the variant based on the domain decomposition paradigm. We parallelize the loop over rows in the straightforward manner (e.g., using "`parallel for`" OpenMP directive), but assume that each thread operates on its own copy of the vector `x`. The management of separate copies of `x` for different threads creates an overhead of parallel computations, while the convergence of the algorithm deteriorates, since each thread performs its operations independently (in the additive Schwarz or block-Jacobi manner).

For other forms of preconditioning and the use of the CRS storage format we refer to [6, 16, 35].

3.3. The "local to global" mapping

The input to the assembly procedure is formed by the element system matrices. Each entry in a local system matrix has its local indices. The question that we want to address is: how to find global indices for such an entry?

We start by simple observation that each such an entry corresponds to a pair of DOF entities. If these DOF entities have only a single DOF associated with them (as it is, e.g., in the case for scalar problems and linear approximation with element vertices, being DOF entities), the global row and column indices may be taken from the global identifiers of DOF entities (FE nodes). The situation becomes more complicated, when the DOF entities have more DOFs associated with them (forming a single vector, as we further assume), which is the case, for example, for higher-order approximations and/or vector problems. In such situations we adopt a simple rule stating that each single DOF has an identifier composed of a pair: the identifier of the corresponding DOF entity and the position in its vector of DOFs.

Another difficulty related to higher-order approximations is that DOFs are associated with different types of mesh entities. Usually each type (vertices, edges, faces, elements) has its own global numbering and, hence, a single global numbering of DOF entities has to be established by some convention. The convention that we adopt is again simple: an identifier for each DOF entity is composed of an identifier of the associated mesh entity and an identifier of the type of mesh entity.

The final problem that we address is related to the specific properties of adaptive codes. In many designs, the so-called constrained approximation [14] is applied, in which not all DOF entities that are used in the computations of local system matrices (in the procedure of FE numerical integration) are present in the final global solution procedure. This is the case, e.g., for, so-called, hanging nodes or other DOF entities, for which the values of DOFs are not computed as the global FEM solutions, but are interpolated using the values associated with the neighbouring DOF entities.

The information on constrained approximation can be applied either at the local level, when forming element matrices, or at the global level. The latter solution requires that the global system matrix is assembled for all DOF entities and then certain rows and columns are distributed among other rows and columns (the process corresponding to the interpolation of the values associated with the constrained DOF entities using the values of other DOF entities). This process means that the storage form of the global system matrix changes, the number of rows and columns is reduced, and a set of operations is performed on a compressed structure of the global system matrix.

However, we choose another strategy. Each local, element matrix in which the constrained DOF entities are present is rewritten prior to the assembly, in such a way that only unconstrained DOF entities are represented in local systems [14]. The advantage of this strategy is that the modifications are performed on small dense matrices and, hence, should be faster than modifications of large, sparse, compressed structures. In effect, all DOF entities that are associated with local systems, have their associated rows and columns in the global system of equations. Moreover, we assume that the information on local to global mapping (at that moment referring to the global numbering of DOF entities in the mesh) for rows and columns of the local matrix does not come from simple numbering of mesh entities composing the element, but has to take into account also the constraints. As a result, this information must be directly incorporated in the interface between the solver and the FEM approximation part, i.e., the routine that supplies the solver with element matrices, must also supply it with arrays storing the local to global information. This solution should have additional advantage of simplifying the assembly process.

The final strategy that we adopt in our investigations is the following. The FEM part of a code supplies the solver part with local element system matrices and right-hand side vectors. For each entry its local indices are induced from the position in the supplied data structures. Additionally, the FEM part supplies for each entry information that is further used for establishing the global numbering of DOFs. This information is composed of types and global identifiers in the FE mesh of the associated DOF entities. Based on this data, the module responsible for coupling the FEM part with a solver of linear equations produces a unique linear global numbering of DOFs for the solver.

The final numbering of DOFs may be optimized in different ways for different solvers, iterative as well as direct, with the numbering of DOFs for the latter having significant impact on their performance [36]. In our setting, the interface module between the FEM approximation code and the linear solver first creates a provisional numbering and then applies any form of renumbering to establish the final, optimized for the selected solver, global numbering that is also used in the assembly process.

4. Practical implementation

In this section, we present a simple implementation of the presented ideas for flexible coupling of the FEM approximation related modules with solvers of linear equations in FE codes. We concentrate on data structures, using C notation, but the presentation leaves the question of selecting programming language and environment as well as further modifications and enhancements, open.

We assume that the whole information provided by the FE part of the code is contained in arguments of the procedure returning element stiffness matrices and load vectors. The interface of such a procedure may look as follows:

```
return_local_stiff_mat( /* procedure returns stiffness matrix (SM)  */
                        /* and load vector (LV) for some mesh entity */
  int Mesh_entity_id, /* in: unique identifier of the mesh entity */
  int Control,        /* in: indicator for the scope of computations: */
                      /*   PDC_NO_COMP  - do not compute anything */
                      /*   PDC_COMP_SM - compute entries to stiff matrix only */
                      /*   PDC_COMP_RHS - compute entries to rhs vector only */
                      /*   PDC_COMP_BOTH - compute entries for sm and rhsv */
  int* Nr_dof_ent,    /* out: number of DOF entities associated with SM and LV */
  int* List_dof_ent_type,   /* out: list of types for DOF entities */
  int* List_dof_ent_id,     /* out: list of IDs for DOF entities */
  int* List_dof_ent_nrdofs, /* out: list of no of dofs for DOF entities */
  double* Stiff_mat,        /* out(optional): stiffness matrix */
  double* Rhs_vect,         /* out(optional): right hand side (load) vector */
);
```

The procedure allows the FEM part of the code to provide arbitrary numbering of DOF entities, separate for each type (vertex, edge, face, element interior) with the only assumption that each DOF entity is associated with a unique pair: type and ID. For higher-order approximations and/or vector problems the number of scalar DOFs associated with a single DOF entity is arbitrary and the solver usually treats them as a single block (vector). In each practical implementation some conventions must be applied for associating the order of DOFs on the returned lists with the order of stiffness matrix and load vector entries.

The presented design assumes that the procedure for returning stiffness matrices is used twice: first, it does not provide the actual stiffness matrix and load vector entries, but only lists of DOF entities. At this stage, it is used for creating the FEM code-linear solver interface data structure that is used for preparing the final numbering of DOFs. Then, in the stage when stiffness matrix and load vector entries are calculated the structure is used, either in the assembly process or in other forms of using computed entries.

The basic proposed data structure for the interface module is the following:

```
typedef struct{
  int dof_ent_type;  /* type of the associated FEM code (mesh) entity     */
  int dof_ent_id;    /* ID of the associated FEM code (mesh) entity       */
  int nrdofs;        /* number of single (scalar) DOFs                    */
  int pos_glob;      /* position in the global RHS vector, denoting       */
                     /* also the row of the global system matrix          */
  int nrneig;        /* number of neighbouring entities/blocks            */
  int* l_neig;       /* list of neighbouring DOF entities (structures)    */
  int* l_neig_bl;    /* list of their positions in the global RHS vector, */
                     /* denoting also the columns for non-zero entries    */
}  sit_DOF_struct;
```

Such a data structure (that we further call ”DOF structure”) is associated with each DOF entity. The number of DOFs associated with a DOF entity is stored in the *nrdofs* field. It is assumed that all such DOFs form a single small vector and that this vector is included in the global vector of unknowns. As a consequence, from the perspective of the proposed data structure, the global vectors of unknowns and right-hand side are composed of linear blocks (as a special case, these blocks can have one DOF), while the global system matrix is made of quadrilateral blocks. This storage is especially useful for higher order and *hp*-adaptive finite element approximations [30].

A DOF structure corresponding to a single DOF entity corresponds to a block of rows of the global system matrix. Non-zero entries in the block of rows form quadrilateral blocks with their column indices, derived from the data related to the neighbouring DOF entities, stored in *l_neig_bl* array. In that way, the interface is ready to present to a solver a graph like structure (using adjacency information) representing the non-zero pattern of the global stiffness matrix. In order to do so, first, a global numbering of all DOFs must be obtained. This is necessary for vector problems, higher-order approximations and non-consecutive numbering of DOF entities by the FEM part of the code. Then, having the provisional ordering, a renumbering algorithm can be applied, in order to produce optimal ordering from the solver point of view (for solvers that perform the final renumbering by themselves this step is skipped).

4.1. Assembly process for CRS format

The FEM code supplies the linear solver with the element system matrices and right-hand side vectors, e.g., by using the interface procedure presented in Sec. 4. The lists supplied by the FEM part of the code are used by the interface module to find the DOF structure (of type `sit_DOF_struct`). Then for each local DOF entity the position in the right-hand side vector, `pos_glob`, being also the row and the column in the global system matrix, is directly retrieved from the DOF structure.

Finding the final position for each local matrix entry in the global system matrix stored in the CRS format requires a linear search – knowing the global column index for an entry, the part of `col_ind` array has to be searched to find the position of the entry in the CRS `val` array. The pseudo-code for the main part of the assembly process (again with many details omitted) looks as follows:

```
for(local_row=0; local_row<size_of_local_SM; local_row++) {
  global_row = ... // find the global row based on global position pos_glob
  for(local_col=0; local_col<size_of_local_SM; local_col++) {
    global_col = ... // find the global col based on global position pos_glob
    for(i=row_ptr(global_row); i<row_ptr(global_row+1); i++) {
      if(col_ind[i]==global_col){
        val[i] += local_SM[local_row, local_col];
        break;
} } } }
```

Here, the lines where the use of `pos_glob` variable is indicated, require the application of the DOF structure, where global positions for each DOF in the final element mesh after renumbering have been stored.

It is possible to get rid of the search during the assembly process. Additional arrays can be created, prior to assembly, in the phase of creating the system of linear equations, which directly store the position in the `val` array for each entry in the local system matrix. During a creation of such arrays the search is performed in a similar way as in the assembly above:

```
for(local_row=0; local_row<size_of_local_SM; local_row++) {
  global_row = ... // find the global row based on global position pos_glob
  for(local_col=0; local_col<size_of_local_SM; local_col++) {
    global_col = ... // find the global col based on global position pos_glob
    for(i=row_ptr(global_row); i<row_ptr(global_row+1); i++) {
      if(col_ind[i]==global_col){
         int jaux = local_row+local_col*size_of_local_SM;
         Assembly_table[mesh_entity_id, jaux] = i;
         break;
} } } }
```

Thanks to this, the final assembly is simplified, becoming a straightforward matrix rewriting with indirect addressing in `val` for each entry:

```
for(local_row=0; local_row<size_of_local_SM; local_row++) {
  for(local_col=0; local_col<size_of_local_SM; local_col++) {
      int jaux = local_row+local_col*size_of_local_SM;
      int icrs = Assembly_table[mesh_entity_id, jaux];
      val[icrs] += local_SM[local_row, local_col];
} }
```

However, there is a price paid for this simplification in the form of the increased storage required for the arrays storing assembly data, as well as the additional time for creating the arrays. The size of arrays depends on the number of elements in the mesh and the number of DOFs associated with a single element. Hence, it scales linearly with the problem size.

During the assembly procedure the arrays `Assembly_table` and `local_SM` are accessed with the unit stride which is advantageous, especially when the hardware optimized for stream processing, such as GPUs, is used. The array `val` is accessed using the indirect addressing, but after suitable renumbering, the accesses for a single row of local stiffness matrix may lie closely to each other in the memory storing `val`.

4.2. Parallel assembly

Numerical integration in the FE codes is a perfectly scalable procedure for parallel execution. However, this is not the case for the assembly phase. Usually several entries from different elements contribute to the same entry in the global stiffness matrix. To avoid the race condition several strategies can be implemented [10]. One of the most popular, employed also in the computational experiments described later in the paper, uses coloring of elements. Elements are colored in such a way, that the elements having the same color do not contribute to the same entries in the global element stiffness matrix. Then, the loop over all elements in the integration and assembly processes is split into two: the outermost loop is performed over all colors in the mesh, while the innermost loop that is perfectly parallelized, is done over the elements of the same color.

5. Possible performance gains for FE assembly and solution of linear systems

As it can be seen from the original assembly algorithm, the main operations (from the performance point of view) are done in the innermost loop, indexed by `i`, over entries in one row of the global system matrix and include:

- retrieving the column index of an entry in the global system matrix from the `col_ind` table – with the accesses to consecutive memory locations,
- comparing the column index in the global system matrix with the column index of the considered entry provided by the interface module as input to the assembly procedure,
- when both indices are equal, storing the proper element matrix value in the CRS array `val`.

The last operation is performed once per each matrix entry and, hence, the computational cost of rewriting the entries is determined by the three accesses to memory, from which accesses to the element matrix should be mostly realized using L1 cache (the element matrix should fit into several cache lines and almost always fits into L1 cache). The accesses to the global system matrix may include more L1 and even L2 (or L3) misses, depending on the structure of the matrix that is always sparse, but with the sparsity pattern determined by the FEM mesh, approximation and even weak statement of the problem solved.

The cost above has to be paid by all assembly algorithms. When optimizations are applied, which involve special assembly tables aimed at removing the search over all entries of a given row of the global system matrix, the costs of the first two operations in the assembly algorithm can be eliminated. What can be gained in this case? The operations comprise of several accesses to memory and several comparisons per each assembled entry. Usually, memory accesses are much more expensive than comparisons, hence it may happen that removing comparisons will have no impact on performance. For memory accesses, one can expect that due to the fact of accessing subsequent memory locations, they can be again mostly realized using L1 cache.

When an assembly array is used, instead of several memory accesses and comparisons, one memory access for an index in the CRS matrix is done. The actual gains can be computed when data on the sizes of local matrices, the sparsity pattern of the global matrix as well as information on performed compiler optimizations together with details of the cache organization and performance are given. For typical sizes of local matrices with one row fitting in a single or at most several cache lines, the performance gains of only several tens of percents can appear, with even lower values possible for large FEM matrices (local and global), when more cache misses are induced during rewriting the matrix entries. However, the large number of factors influencing the final performance of the algorithm may result in a much larger performance increase.

The solver interface module allows also for performing arbitrary DOF renumbering. The performance gains associated with renumbering depend on the particular solver algorithms. For example, for many iterative solvers, the most time consuming operation is the matrix-vector product, presented for the CRS storage format in Subsec. 3.1.

In this case, the significant portion of execution time is spent on retrieving the values of the vector `x`, for which indirect addressing is used. Renumbering can change the order of DOFs in the vector of unknowns in order to reduce the bandwidth of the global system matrix, and by this reduce the number of cache misses when accessing the vector `x`. The exact estimates for performance gains once again depend on the actual structure of the global matrix, but one can expect reduction of execution time from several to several tens of percents.

6. Computational experiments

We present two computational problems, the first shows the performance of the proposed solutions and the second demonstrates its flexibility (the initial form of the interface was also used for *hp*-adaptive approximations described in [30]). To test the performance of the proposed FEM solver interface module, a simple Poisson problem in the unit cube was selected. The number of elements with linear approximation was equal to 782 336, with the number of faces with Neumann boundary data (for which the numerical integration and assembly was also performed) equal to 206 592. The number of unknowns (being also the number of rows of the global stiffness matrix) was equal to 446 265, with 8 608 425 non-zero entries in the system matrix. Hence, the average number of non-zero entries per single row of the global stiffness matrix was 19.29 (with the ratio of non-zero entries to the full size equal to 0.004%).

For this problem, several experiments have been conducted using a standard workstation with Intel Core i7 4790 processor, 4 DIMMs of 1600 MHz, 12800 MB/s, DDR3 RAM (in dual channel configuration) and Linux operating system (CentOS 7 with 3.10.0 kernel). Since the arithmetic performance of the processor is of less importance for the algorithms that we investigate (with the performance of execution bounded by the speed of memory accesses), we establish the performance of the processor in the STREAM benchmark [25] as the reference performance. The benchmark consists of several simple loops, e.g., for the COPY test case it is:

```
#pragma omp parallel for
  for (j=0; j<N; j++)  c[j] = a[j];
```

When compiled for large enough values of `N` (in our experiments we set `N = 20000000`), with proper parallelization and vectorization options, it is often used to indicate the memory bandwidth possible to be obtained for a given hardware in real-life applications (as opposed to theoretical limits).

The Intel Core i7 4790 processor has four cores with hyperthreading, thus the standard reference number of threads is equal to 8. The STREAM COPY benchmark (representative for testing assembly algorithm) gives the bandwidth around 23 GB/s, which translates into an average access time for a single double precision value close to 0.32 ns (the theoretical maximal bandwidth is 25.6 GB/s). Since the STREAM tests consist of operations on huge arrays, with the optimal use of all cache levels and prefetching, in the case of assembly, where there are thousands of small matrices rewritten to a single array, with indirect addressing and accesses not necessarily for subsequent locations in memory, the performance that is several times lower can be expected (for random accesses to memory the average access time can be several dozen times longer than for optimal accesses, such as in the STREAM tests).

For this problem and the hardware, the main phases of the FEM solution procedure were performed with their wall-clock execution times measured using system tools (`gettimeofday` Unix system call). To test the performance of assembling we used the interface coupled with a Gauss-Seidel preconditioned GMRES iterative solver. The execution times for the main phases of calculations are presented in Table 1 (for all the cases with multiple threads the GMRES solver converged in 77 iterations, while for the sequential run it required 76 iterations for convergence, i.e. in our case, the reduction of the initial residual by the factor 10^{12}). For parallel multithreaded runs, the integration and assembly phases were preceded by the coloring algorithm with the duration around 1.7 sec.

Table 1. Execution time (time) and the related parallel speed-up (pspd) for different solution phases and the Poisson test case (782 336 elements, 446 265 unknowns).

	Number of threads							
	1		2		4		8	
Phase of calculations	time	pspd	time	pspd	time	pspd	time	pspd
Integration	2.898	–	1.484	1.95	0.774	3.74	0.688	4.21
Assembly	0.232	–	0.121	1.92	0.065	3.57	0.044	5.27
Total matrix creation	3.203	–	1.657	1.93	0.882	3.63	0.809	3.96
Iterations	1.329	–	1.102	1.21	1.036	1.28	1.087	1.22
Total solver	4.622	–	2.839	1,63	2.008	2.30	1.986	2.33

For the assembly, two variants were tested – with and without assembly tables (the times for the former are included in Table 1). The total number of assembled entries in all element (and face) stiffness matrices was equal to 41 534 976. From the results in Table 1 it can be seen that, in a memory intensive assembly algorithm, the bandwidth of CPU-memory bus is almost fully exploited for 4 threads. Hence, we obtain a reasonable speed-up of 3.57 for 4 threads, while for 8 threads, the speed-up remains at 5.27, the value much lower than the perfect 8. The final average access time for a single scalar value (assuming four memory accesses per single entry assembled into the global system matrix) was equal to 0.28 ns, equivalent to the bandwidth close to 25 GB/s (since three memory accesses are done with double-precision values and one with integers). This result is very good, comparable to the best results from the STREAM benchmark and the theoretical bandwidth. The explanation for this fact may be related to the relatively large size of the processor cache memory (8 MB), with possibly some percentage of `val` entries retrieved from cache and not from the main memory during the calculations. Nevertheless, the algorithm shows good use of multithreading, with the assembly result for 8 threads still better than for 4 threads, while for the STREAM tests the results for 8 threads were the same as for 4 threads (and even for 2 threads).

For the case without assembly tables, the execution times for the assembly phase were respectively: 0.228 for 8 threads, 0.281 for 4 threads and 1.049 for 1 thread. Hence, despite the fact of relatively small changes in the source code with theoretically possible small impact on performance, the execution times were several times longer than when the assembly tables were used. This confirms the fact, well known in the HPC community, that even small changes in memory management can have a profound impact on the performance of codes.

In order to show the importance of reordering we present the results for the same problem, but on a smaller mesh with 97792 elements and the system of equations with 62845 unknowns. We use the presented interface coupled with the PARDISO direct solver [36], employed without internal reordering. The original numbering of DOF entities, provided by the FEM part of the code, resulted in the half-bandwidth of the system matrix equal to 62760. This strongly suboptimal ordering resulted in the solution time of 915 sec. After applying the reverse Cuthill-McKee (RCM) algorithm for bandwidth reduction [11], the half-bandwidth of the system reduced to 616 with the solver producing results just after 2.39 sec.

This example shows a situation, to certain extent, artificial since all popular direct solvers employ internally some form of renumbering. However, when the PARDISO solver is used with internal reordering for the same problem and the larger mesh with the problem size of 446 265, the reported execution time 4.45 sec includes 0.48 sec spent for reordering. This 10% of the time can be saved, when the internal solver reordering is switched off and the assembly is performed for already renumbered DOFs in the manner proposed in the paper.

The second example problem is the coupled simulation of incompressible fluid flow and heat transfer – the buoyancy driven flow in a cube cavity [12]. Similarly to the previous example, this test case is solved using the ModFEM code [26], in which the presented interface has been implemented. We present here neither the details of the FE formulation nor the results of calculations, but concentrate solely on the coupling between the FE part of the code and the solvers of linear equations.

The simulation consists of certain number of time steps (using the implicit Euler method), with several Picard's iterations to solve the non-linear problem at each time step. The iterations are performed alternately for each of the coupled problems with the proper information exchanged between the problems. In each iteration, a system of linear equations is solved. For the heat problem the problem is scalar, hence the number of DOFs for each DOF entity is 1, while for the vector Navier-Stokes problem the number is 4 (due to the used SUPG stabilization [5, 17]).

To test the flexibility of the approach two different solvers were used for each of the coupled problems, both employing the same interface presented in the paper. The heat problem was solved using the direct PARDISO solver (employing the CRS format), while the Navier-Stokes problem was solved using the iterative solver developed in-house for ModFEM. For the latter, the preconditioned GMRES method was applied with the ILU(1) preconditioning and the BCRS storage format.

The *h*-adaptivity was used for increasing the accuracy of the results, with derivative recovery error estimates for both problems [37]. The results are reported for the third consecutive adapted mesh with 670 738 elements and 363 784 nodes, which resulted in 363 784 unknowns for the heat problem and 1 358 768 unknowns for the incompressible fluid flow problem.

For each new mesh, the solver interface prepares the new data structures described in the paper (including the base data structure together with assembly tables and element coloring data structures). Then, for consecutive time steps and non-linear iterations, the same data structures are reused.

The reported execution times are presented in Table 2. The first observation is that the only algorithm that benefits from hyperthreading is assembly. For all the other, the execution times for 8 threads remain close to those for 4 threads, with even some results deteriorating (this concerns especially the ILU(1) decomposition). When up to 4 threads are used for the 4-core hardware, some algorithms exhibit good and some poor scaling. The coloring algorithm that scales poorly was always used in its sequential version (it took approximately 1.68 sec). The same concerned ILU(1) iterations (forward reduction and back substitution). The ILU(1) factorization scaled poorly and

Table 2. Execution time (time) and the related parallel speed-up (pspd) for different solution phases and the buoyancy driven cavity flow test case (670 738 elements, 363 784 nodes).

Phase of calculations	Number of threads							
	1		2		4		8	
Navier-Stokes problem	time	pspd	time	pspd	time	pspd	time	pspd
Integration	21.080	–	10.595	1.99	5.553	3.80	5.093	4.14
Assembly	1.007	–	0.505	1.99	0.266	3.79	0.210	4.80
Total matrix creation	22.176	–	11.197	1.98	5.916	3.75	6.245	3.55
ILU(1) factorization	56.729	–	39.684	1.43	42.510	1.34	75.990	0.75
Iterations	7.601	–	7.377	–	7.410	–	8.917	–
Total solver	86.646	–	58.395	1.48	55.954	1.55	91.302	0.95
heat transfer problem	time	pspd	time	pspd	time	pspd	time	pspd
Integration	4.127	–	2.107	1.96	1.098	3.76	0.997	4.14
Assembly	0.180	–	0.096	1.88	0.050	3.60	0.035	5.14
Total matrix creation	4.361	–	2.260	1.93	1.179	3.70	1.072	4.07
Direct solution	25.690	–	16.869	1.52	13.785	1.86	14.317	1.79
Total solver	30.180	—	19.260	1.57	15.074	2.00	16.138	1.87

only for 2 threads, the direct solver scaled slightly better, but still with the results for 4 threads being only slightly better than for 2 threads. As before, the integration and assembly procedures described in the paper scaled almost perfectly.

7. Conclusions

The main motivation for the introduced mechanisms in the interface modules between the FEM codes and solvers of linear equations was to increase the flexibility of the interface. The presented design can accommodate different FEM meshes and approximations (including *hp*-adaptive) as well as different storage formats for solvers. In addition, it does not require the FE code to produce optimal ordering of DOFs, can take any ordering as input to the interface, perform reordering in the interface module and create matrices for the solver already using the optimal numbering, so the time consuming procedure of rewriting a sparse matrix in compressed format is avoided. The adjacency graph of FE model is constructed in the interface module using the data associated with individual element matrices. In that way, the interface between the solver and the FE part of simulation codes can be reduced to a single procedure call (usually several additional procedures of less importance are also used). The information associated with the proposed interface can be used to create linear data structures that can increase the performance of the main steps of FE solution procedures. The paper presented the results, showing the increase in performance for FE assembly associated with the introduced data structures on an example of standard multi-core processor. We plan to present the effects of using the proposed design for execution on graphics processors and accelerators in forthcoming papers.

Acknowledgment

The work presented in this paper was supported by National Science Centre, Poland, grant no. DEC-2012/06/M/ST1/00363.

References

[1] Finite Element Interface to Linear Solvers (FEI). Project home page: http://trilinos.sandia.gov/packages/fei.

[2] E. Arge, A. Bruaset, H. Langtangen [Eds.]. *Modern Software Tools for Scientific Computing*. Birkhäuser, 1997.

[3] K. Banaś. On a modular architecture for finite element systems. I. Sequential codes. *Computing and Visualization in Science*, **8**: 35–47, 2005.

[4] K. Banaś, K. Michalik. Design and development of an adaptive mesh manipulation module for detailed FEM simulation of flows. [In:] P.M.A. Sloot, G.D. van Albada, J. Dongarra [Eds.], *Proceedings of the International Conference on Computational Science, ICCS 2010*, University of Amsterdam, The Netherlands, May 31 – June 2, 2010, Procedia Computer Science. *Elsevier*, **1**: 2043–2051, 2010.

[5] K. Banaś, K. Chłoń, P. Cybułka, K. Michalik, P. Płaszewski, A. Siwek. Adaptive finite element modelling of welding processes. [In:] M. Bubak, J. Kitowski, K. Wiatr [Eds.], eScience on Distributed Computing Infrastructure – Achievements of PLGrid Plus Domain-Specific Services and Tools, Lecture Notes in Computer Science. *Springer International Publishing*, **8500**: 391–406, 2014. http://dx.doi.org/10.1007/978-3-319-10894-0_28.

[6] B. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, PA, 1994.

[7] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klofkorn, R. Kornhuber, M. Ohlberger, O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. *Computing*, **82**(2): 121–138, 2008.

[8] P. Bastian, C. Engwer, D. Göddeke, O. Iliev, O. Ippisch, M. Ohlberger, S. Turek, J. Fahlke, S. Kaulmann, S. Müthing, D. Ribbrock. EXA-DUNE: Flexible PDE Solvers, Numerical Methods and Applications. [In:] L. Lopes, J. Zilinskas, A. Costan, R.G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S.L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, M. Alexander [Eds.], *Euro-Par 2014: Parallel Processing Workshops – Euro-Par 2014 International Workshops*, Porto, Portugal, August 25–26, 2014, Revised Selected Papers, Part II, Lecture Notes in Computer Science, **8806**: 530–541. Springer, 2014. http://dx.doi.org/10.1007/978-3-319-14313-2_45.

[9] D.E. Bernholdt, B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T.L. Dahlgren, K. Damevski, W.R. Elwasif, T.G.W. Epperly, M. Govindaraju, D.S. Katz, J.A. Kohl, M. Krishnan, G. Kumfert, J.W. Larson, S. Lefantzi, M.J. Lewis, A.D. Malony, L.C. McInnes, J. Nieplocha, B. Norris, S.G. Parker, J. Ray, S. Shende, T.L. Windus, S. Zhou. A component architecture for high-performance scientific computing. *International Journal of High Performance Computing Applications*, **20**: 163–202, 2006.

[10] C. Cecka, A.J. Lew, E. Darve. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, **85**(5): 640–669, 2011. http://dx.doi.org/10.1002/nme.2989.

[11] E. Cuthill, J. McKee. Reducing the bandwidth of sparse symmetric matrices. [In:] *ACM'69 Proceedings of the 1969 24th National Conference*, pp. 157–172, ACM New York, USA, 1969. http://dx.doi.org/10.1145/800195.805928.

[12] G. De Vahl Davis. Natural convection of air in a square cavity: A bench mark numerical solution. *International Journal for Numerical Methods in Fluids*, **3**(3): 249–264, 1983. http://dx.doi.org/10.1002/fld.1650030305.

[13] L. Demkowicz. *Computing with hp-Adaptive Finite Elements: Volume 1. One and Two Dimensional Elliptic and Maxwell Problems*. Taylor & Francis Group, 2006.

[14] L. Demkowicz, J. Oden, W. Rachowicz, O. Hardy. Toward a universal h-p adaptive finite element strategy, Part 1. Constrained approximation and data structure. *Computer Methods in Applied Mechanics and Engineering*, **77**: 79–112, 1989.

[15] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski. Generation of large finite-element matrices on multiple graphics processors. *International Journal for Numerical Methods in Engineering*, **94**(2): 204–220, 2013. http://dx.doi.org/10.1002/nme.4452.

[16] S. Fialko. Iterative methods for solving large-scale problems of structural mechanics using multicore computers. *Archives of Civil and Mechanical Engineering (ACME)*, **14**: 190–203, 2014.

[17] L. Franca, S. Frey. Stabilized finite element methods II: The incompressible Navier-Stokes equations. *Computer Methods in Applied Mechanics and Engineering*, **99**: 209–233, 1992.

[18] Z. Fu, T.J. Lewis, R.M. Kirby, R.T. Whitaker. Architecting the finite element method pipeline for the GPU. *Journal of Computational and Applied Mathematics*, **257**: 195–211, 2014. http://dx.doi.org/10.1016/j.cam.2013.09.001.

[19] S. Komatitsch, G. Erlebacher, D. Göddeke, D. Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, **229**(20): 7692–7714, 2010.

[20] Z. Koza, M. Matyka, S. Szkoda, Ł. Mirosław. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, **36**(2): C219–C239, 2014. http://dx.doi.org/10.1137/120900216.

[21] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A.R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM J. Scientific Computing*, **36**(5): C401–C423, 2014. http://dx.doi.org/10.1137/130930352.

[22] H. Langtangen, A. Bruaset, E. Quak [Eds.]. *Advances in Software Tools for Scientific Computing*. Springer, Berlin/Heidelberg, 2000.

[23] R. Mackie. Object oriented programming of the finite element method. *International Journal for Numerical Methods in Engineering*, **35**: 425–436, 1992.

[24] G.R. Markall, A. Slemmer, D.A. Ham, P.H.J. Kelly, C.D. Cantwell, S.J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, **71**(1): 80–97, 2013. http://dx.doi.org/10.1002/fld.3648.

[25] J.D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[26] K. Michalik, K. Banaś, P. Płaszewski, P. Cybułka. ModFEM – a computational framework for parallel adaptive finite element simulations. *Computer Methods in Materials Science*, **13**(1): 3–8, 2013.

[27] M. Paszynski, D. Pardo, V.M. Calo. Direct solvers performance on h-adapted grids. *Computers & Mathematics with Applications*, **70**(3): 282–295, 2015.

[28] M. Paszyński, D. Pardo, A. Paszyńska, L. Demkowicz. Out-of-core multi-frontal solver for multi-physics hp adaptive problems. *Procedia Computer Science*, **4**: 1788–1797, 2011.

[29] P. Płaszewski, K. Banaś. Performance analysis of iterative solvers of linear equations for hp-adaptive finite element method. [In:] V.N. Alexandrov, M. Lees, V.V. Krzhizhanovskaya, J. Dongarra, P.M.A. Sloot [Eds.], *Proceedings of the International Conference on Computational Science, ICCS 2013*, Barcelona, Spain, 5–7 June, 2013, *Procedia Computer Science*, **18**, 1584–1593. Elsevier, 2013.

[30] P. Płaszewski, M. Paszyński, K. Banaś. Architecture of iterative solvers for hp-adaptive finite element codes. *Computer Assisted Methods in Engineering and Science*, **20**(1): 43–54, 2013.

[31] I. Reguly, M. Giles. Finite element algorithms and data structures on graphical processing units. *International Journal of Parallel Programming*, **43**(2): 203–239, 2015. http://dx.doi.org/10.1007/s10766-013-0301-6.

[32] J.F. Remacle, B. Karamete, M. Shephard. *Algorithm Oriented Mesh Database*. Report 5, SCOREC, 2000.

[33] J.F. Remacle, O. Klaas, J. Flaherty, M. Shephard. A parallel algorithm oriented mesh database. *Engineering with Computers*, **18**: 274–284, 2002.

[34] K.A. Rojek, M. Ciznicki, B. Rosa, P. Kopta, M. Kulczewski, K. Kurowski, Z.P. Piotrowski, L. Szustak, D.K. Wojcik, R. Wyrzykowski. Adaptation of fluid model EULAG to graphics processing unit architecture. *Concurrency and Computation: Practice and Experience*, **27**(4): 937–957, 2015. http://dx.doi.org/10.1002/cpe.3417.

[35] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[36] O. Schenk, K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, **20**(3): 475–487, 2004.

[37] O.C. Zienkiewicz, J.Z. Zhu. The superconvergent patch recovery and a posteriori error estimates. Part 1: The recovery technique. *International Journal for Numerical Methods in Engineering*, **33**(7): 1331–1364, 1992. http://dx.doi.org/10.1002/nme.1620330702.

[38] T. Zimmermann, Y. Dubois-Pelerin, P. Bomme. Object-oriented finite element programming: I. Governing principles. *Computer Methods in Applied Mechanics and Engineering*, **98**: 291–303, 1992.