Computer Assisted Mechanics and Engineering Sciences, **16**: 117–131, 2009.

A block sparse shared-memory multifrontal finite element solver for problems of structural mechanics

S. Y. Fialko
Institute of Computer Modeling, Cracow University of Technology
ul. Warszawska 24, 31-155 Cracow, Poland
Software Company SCAD Soft, Ukraine

(Received in the final form July 17, 2009)

The presented method is used in finite-element analysis software developed for multicore and multiprocessor shared-memory computers, or it can be used on single-processor personal computers under the operating systems Windows 2000, Windows XP, or Windows Vista, widely popular in small or medium-sized design offices. The method has the following peculiar features: it works with any ordering; it uses an object-oriented approach on which a dynamic, highly memory-efficient algorithm is based; it performs a block factoring in the frontal matrix that entails a high-performance arithmetic on each processor and ensures a good scalability in shared-memory systems. Many years of experience with this solver in the SCAD software system have shown the method's high efficiency and reliability with various large-scale problems of structural mechanics (hundreds of thousands to millions of equations).

Keywords: finite element method, large-scale problems, multifrontal method, sparse matrices, ordering, multithreading.

1. Introduction

The dimensionality of present-day design models for multistorey buildings and other engineering structures may reach 1,000,000 to 2,000,000 equations and still grows up. Therefore, one of most important issues with modern CAD software is how fast and reliably their solver subsystems can solve large-scale problems. The present paper suggests a method for solving high-order systems of linear algebraic equations,

$$\mathbf{K}\,\mathbf{X} = \mathbf{B} \tag{1}$$

where $\mathbf{K}$ is a symmetric indefinite sparse matrix, $\mathbf{X}$, $\mathbf{B}$ are the respective sets of solution vectors and right-part vectors. This equation system has been derived by applying a finite element method (FEM) to problems that appear in mechanics of solids and structures.

Contemporary finite-element analysis (FEA) software tools use various methods to solve the equation system (1), direct methods for sparse matrices being most efficient among other direct methods [10]. As the dimensionality of many problems in structural mechanics exceeds the memory capacity of most computers available to average users, the present research is based on a method that involves external memory in large-scale cases.

This method extends an idea on which the frontal method is based [11, 13] onto the case of any arbitrary ordering. There is a number of major differences from the known multifrontal method [3, 4] which is an algebraic method and based on analysis of the sparse matrix structure.

The presented method is a substructure method, because it is based on step-by-step assembling of whole structure from separate finite elements and substructures, created on the previous assembling steps. When the completely assembled equations arise, they are immediately eliminated. The order

of assembling is strictly defined by the order of node elimination, that is, the elimination of group of equations associated with a given node.

These are the principles that the approach suggested here is based on:

- During the ordering, during the building of the front tree, and during the matrix decomposition analysis, the approach uses information about the finite-element model's nodes rather than equations. Each node of the FE model produces a block of equations associated with it.
- The source data for the ordering algorithm is an adjacency graph for the nodes rather than finite elements as in the frontal method [5].
- The source code is developed in C++, except for the front matrix factorization routine where the C language is used to make the optimizing compiler more efficient. The object-oriented principle helps develop a dynamic algorithm that controls the allocation and release of RAM flexibly.
- The whole matrix decomposition process is divided into steps, and the number of the steps is equal to that of the FE model's nodes. Each step eliminates one block of equations associated with the current node.
- The front is understood as a C++ class object that encapsulates No. of the node being eliminated at the current step (or the block of equations associated with the node); a list of finite elements added to the ensemble at the current step; a list of preceding fronts the matrices of which – incomplete fronts – contribute to the building of the dense matrix for the current front, which we call a frontal matrix as in the frontal method; a list of nodes the equations for which make up the frontal matrix; and other essential data.
- The whole process of decomposing the global sparse matrix $\mathbf{K}$ is reduced to factorization of a sequence of frontal matrices which consist of fully assembled equations and an incomplete front. The factorization comprises only complete (fully assembled) equations, but the incomplete front is modified, too. A block algorithm for factoring the frontal matrix has been developed; it ensures high performance and good scalability on multicore computers.

The earlier version of the method [7] used an LU decomposition in the frontal matrix, and only two ordering algorithms, ND and QMD [8]. The random-access memory for the frontal matrices was allocated in the page file which was a serious limitation of the dimensionality. In the later version [6], a more efficient scheme of a block $\mathbf{L} \cdot \mathbf{S} \cdot \mathbf{L}^{\mathrm{T}}$ decomposition of the frontal matrix was employed, the set of ordering algorithms was extended, the random-access memory was allocated in a special buffer so the special file was involved only when the buffer was full. The version of the method presented in this paper suggests a more effective scheme for the block $\mathbf{L} \cdot \mathbf{S} \cdot \mathbf{L}^{\mathrm{T}}$ decomposition of the frontal matrix, less expensive algorithms of memory buffer management for the frontal matrices including disk saving operations when the buffer is full, and a parallel approach based on multithreading.

2. Ordering

The method works with any ordering. The stiffness matrix's adjacency structure is represented by an adjacency graph for the nodes, and the result of the ordering is an array of permutations that defines the node elimination order. This makes the graph naturally much rougher than the adjacency graph for equations. Solution of a lot of practical problems has confirmed the suggestion made in [12] that the use of a rough adjacency graph instead of a detailed one will, as a rule, reduces the nonzero entries of the matrix more efficiently.

The following ordering algorithms are used: reverse Cuthill-McKee (RCM), Sloan's (Sloan) [14], factor trees (FT_MMD), parallel sections (PS_MMD), nested dissections (ND), minimum degree [9] (MMD), and a hybrid algorithm (ND_MMD) based on dividing the initial structure into substructures using the nested dissections method ND and ordering the internal nodes in each substructure

by the minimum degree method MMD [1]. The algorithms for the RCM and ND methods were taken from [8], while for the FT_MMD and PS_MMD methods their algorithms given in [8] were modified: internal nodes in every substructure were ordered by the minimum degree algorithm.

Each of the listed ordering algorithms is actually heuristic rather than strictly mathematical, therefore, first, whichever ordering is used will never be optimum but only an approximation to optimum; and second, it is never known beforehand for a given problem which of the algorithms will produce a result closest to an optimum ordering.

The solver presented here allows the user to choose an ordering algorithm or not to use any at all. There is also an AUTO option; it takes the MMD, ND_MMD, Sloan algorithms, the efficiency of which was found the best "in average", and performs a symbolic factoring [8] to choose the algorithm that produces the least nonzero entries.

3. Analysis

At the analysis step, the order of the front processing and the order in which the finite elements are fed to the assembling are established for a given node elimination sequence. A "Process Descriptor" data structure and a tree of fronts are created. Without limiting the generality of the method, we will illustrate these important steps by a simple example, a square plate with the mesh 2×2 (Fig. 1).

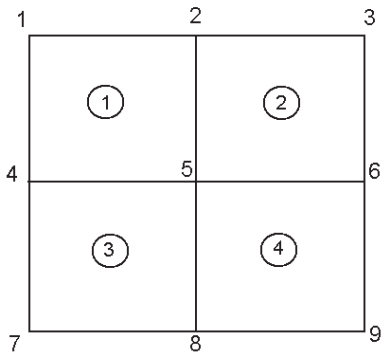

Fig. 1. A square plate with a 2×2 mesh. Nos. of finite elements are encircled

Suppose the ordering has produced the following node elimination sequence: 1, 3, 7, 9, 2, 6, 8, 4, 5. Now we need to determine in what sequence we should feed our finite elements to the assembling, to ensure the nodes are being eliminated in the given order.

The initial structure is divided into separate finite elements. The information about each particular element consists of a stiffness matrix for the element, $\mathbf{K}_e$, the number and list of its nodes, the number of degrees of freedom of the finite element, and a list of reference indexes – a list of Nos. of equations in the global matrix $\mathbf{K}$ to the coefficients of which this element makes a contribution.

We will call a node completely assembled if all the finite elements next to it have been feed to the assembling and duly assembled. Adding any finite element from ones that have left will not modify the coefficients of the equations associated with this node. Accordingly, completely assembled equations can be eliminated at once, as soon as they appear – this is the approach used in the classic frontal method. We will refer to this as the elimination of the respective node, meaning that we speak about the elimination of all equations associated with it.

Now we begin to assemble the structure by adding finite elements to the ensemble, one by one. To establish the order in which the finite elements will be fed to the assembling, we fill Table 1 by writing the given node elimination order in its first row and the list of finite elements adjacent to the current node in its second row. Each finite element can go through the assembling one time only. Therefore, if a finite element has been already fed to the assembling, we mark it as one used before and underline its No. in Table 1.

Table 1. Setting up the order in which to feed finite elements (FEs) to the assembling

No. of node	1	3	7	9	2	6	8	4	5
No. of FE	1	2	3	4	1, 2	2, 4	3, 4	1, 3	1, 2, 3, 4

Table 2. "Process Descriptor" data structures

No. of front, exclusion step	Node being excluded	List of nodes in the frontal matrix	List of preceding fronts	List of FEs fed to the assembling
1	1	1, 2, 4, 5	–	1
2	3	3, 2, 6, 5	–	2
3	7	7, 8, 4, 5	–	3
4	9	9, 6, 8, 5	–	4
5	2	2, 6, 4, 5	1, 2	–
6	6	6, 8, 4, 5	5, 4	–
7	8	8, 4, 5	6, 3	–
8	4	4, 5	7	–
9	5	5	8	–

Writing down Nos. of the finite elements not marked as already assembled in the order of node elimination (going through the table from left to right) will give us the sequence in which the elements will be fed to the assembling as defined by the node elimination order for the current FE model.

Table 2 demonstrates how the "Process Descriptor" data structure is being built. First column presents No. of the elimination step which is equal to the front's No. Second column contains No. of the eliminated node at the current step of exclusion. The last column contains results from Table 1. The other columns will be filled during the elimination.

To eliminate node 1 (Table 1), you need to feed element 1 to the assembling; the element contains nodes 1, 2, 4, 5. The respective frontal matrix is created by equations which are associated with nodes 1, 2, 4, 5 (Fig. 1), and it is

$$\begin{array}{cccc} 1 & 2 & 4 & 5 \\ \begin{pmatrix} x & & & \\ x & s & & \\ x & s & s & \\ x & s & s & s \end{pmatrix} \end{array} \tag{2}$$

where completely assembled equations are denoted as x and equations assembled partially as s. Here, every symbol of the matrix designates a block with its dimensionality equal to the number of degrees of freedom in the node. Nos. of the nodes are indicated on top of the matrix. As the matrix is symmetric, RAM contains only its lower triangular half. Node 1 is completely assembled, therefore all unknowns associated with it are subject to elimination. The factorization comprises only completely assembled equations; however, the part of the matrix that corresponds to partially assembled nodes 2, 4, 5, will undergo a transformation, too. The transformed columns of the frontal matrix, which correspond to the unknowns for node 1 being excluded, are a part of the factored global lower triangular matrix and are placed in a special buffer that is saved to disk as soon as it is full. The other part of the matrix, that corresponds to partially assembled nodes 2, 4, 5, makes up an incomplete front that will be used to assemble new frontal matrices in steps that follow.

Nodes 3, 7, 9 are eliminated similarly at the respective steps 2, 3, 4. Their respective frontal matrices look alike to 2, and Table 2 presents lists of nodes for them. To facilitate the orientation, the eliminated node is placed in column 3 of Table 2 at the beginning of the list of nodes of the

frontal matrix. After it has been excluded, No. of this node is used no longer. Remaining nodes are placed according to the nodal eliminating sequence, given by ordering algorithm.

Now let us consider the elimination step 5. No. of the node to be eliminated is 2. The coefficients of equations associated with node 2 can be assembled only from those of equations of incomplete fronts because the list of finite elements fed to the assembling has been exhausted. In more complex problems, the frontal matrix of the current front (or just the current front) can be assembled either from previous incomplete fronts (we will refer to them as preceding fronts) or from the matrices of finite element fed to the assembling at the current step. Node 2 is in the lists of the preceding fronts 1, 2, therefore the current front 5 is assembled from the preceding fronts 1, 2:

$$\left.\begin{array}{c} \begin{array}{ccc} 2 & 4 & 5 \end{array} \\ \begin{pmatrix} s_{11} & & \\ s_{21} & s_{22} & \\ s_{31} & s_{32} & s_{33} \end{pmatrix} \\ \begin{array}{ccc} 2 & 6 & 5 \end{array} \\ \begin{pmatrix} t_{11} & & \\ t_{21} & t_{22} & \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \end{array}\right\} + \rightarrow \quad \begin{array}{c} \begin{array}{cccc} 2 & 6 & 4 & 5 \end{array} \\ \begin{pmatrix} t_{11}+s_{11} & & & \\ t_{21} & t_{22} & & \\ s_{21} & 0 & s_{22} & \\ s_{31}+t_{31} & t_{32} & s_{32} & s_{33}+t_{33} \end{pmatrix} \end{array}. \tag{3}$$

The list of nodes of front 5 is a combination of the lists of nodes of incomplete fronts 1, 2, minus already eliminated nodes 1, 3. It is shown in third column of Table 2, and fronts 1, 2 are put on the list of preceding fronts for front 5. As soon the assembling of preceding front is finished as C++ object of this front is destroyed to free the core memory. It means that fronts 1, 2 do not exists after assembling of front 2 is finished.

The node 6 is eliminated on the elimination step 6. We search such of previous fronts, which contain the node 6, the list of finite elements fed to the assembling is exhausted. The node No. 6 is met in front 2, but front 2 has been destroyed on the previous elimination step. The node No. 6 is met in fronts 4 and 5, so, fronts 4, 5 are the preceding fronts for front 6, and are taken to be assembled.

And so on. We continue filling Table 2 until all elimination steps are completed.

The structure of levels [8] of the front tree (Fig. 2) is being created as Table 2 is passed from the bottom up. Front 9 is the last one – we put it on 1st level. The preceding front for it is front 8 – we put it on level 2. The preceding front for front 8 is front 7 – and it becomes 3rd level. The preceding fronts for front 7 are fronts 6 and 3 – we put it on 4th level. And so on.

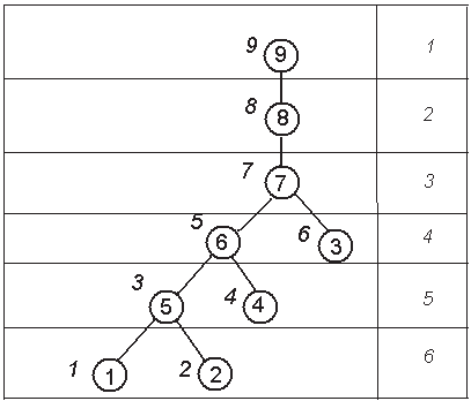

Fig. 2. Structure of levels of fronts included in a front tree

In the tree of fronts, we distinguish between nodal fronts, succeeding fronts, and initial fronts. The nodal fronts have more than one preceding front, the succeeding ones have exactly one preceding front, and the initial fronts have no preceding fronts at all.

The factorization of the matrix is a process of moving through the structure of levels from the bottom up. In order to reduce the memory volume required to store incomplete fronts, the fronts are re-numbered. Figure 2 shows a new order of front processing by numbers above circles. Now front 1 is first to be processed, front 2 is second, front 5 is third, front 4 is fourth, front 6 is fifth etc.

4. Factorization

In nodal and initial fronts (Fig. 3), a dynamic allocation of memory for the front matrix takes place. This process determines the dimensionality of the biggest front counting from the current nodal (initial) front to the closest nodal front higher in the structure of levels. For example, when allocating memory for the frontal matrix of front p, the required memory volume is calculated for the frontal matrices of fronts p, k and then the bigger one is used. Here we allow for the fact that the dimensionality of the matrix of the succeeding front k can be greater than that of the frontal matrix of the nodal front p because there are additional assembled matrices of the respective finite elements which are not present in the frontal tree but can be added during the assembling of the current front's frontal matrix.

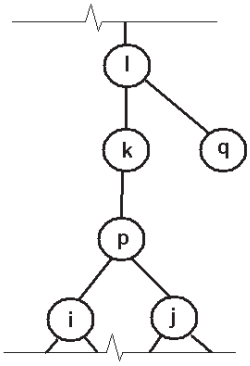

Fig. 3. Example: q is an initial front; i, j, p, l are nodal fronts, k is a succeeding front

In addition, memory is allocated for the list of nodes that make up the front and for the list of reference indexes. The list of nodes for the current nodal front is being built as we pass through the list of nodes of preceding fronts and the list of nodes of finite elements fed to the assembling at the current step. Next, using this list, we create a list of global Nos. of equations for the frontal matrix (a list of reference indexes). The described process of memory allocation is part of creating the "current front" object.

As soon as the preceding incomplete front is fully fed to the assembling of the new nodal front, the memory that stores the frontal matrix, the list of nodes of the preceding front, and the list of reference index will be released. In other words, the "preceding front" object will be destroyed.

No memory is allocated for the frontal matrices of succeeding fronts because a succeeding front inherits a memory block created for a nodal (initial) front which is closest from below in the structure of levels. In the example of Fig. 3, the succeeding front k receives a pointer to a memory block associated with front p when the latter was created. Thus we avoid having to copy data from one memory block to another and an expensive routine of dynamic memory allocation for front k. So it appears that the dynamic memory allocation and data transfers from memory areas occupied by one frontal matrix to another occur only when nodal or initial fronts are being created. Next,

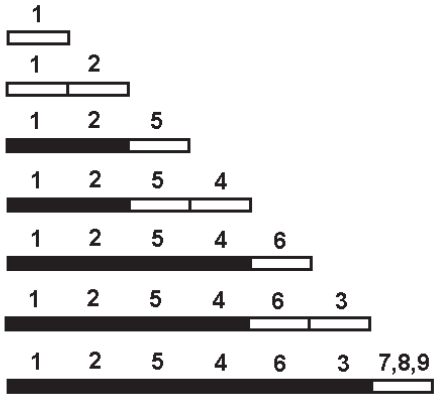

Fig. 4. Dynamic allocation and release of memory for frontal matrices of an ordered frontal tree presented in Fig. 3 Numbers show Nos. of fronts. Fronts for which the memory has been released are shown in black

an additional assembling is performed to allow for the contribution of the stiffness matrices of the respective finite elements if there are any at the current step.

This dynamic algorithm helps use the computer RAM resources efficiently. Figure 4 shows the process of the RAM dynamic allocation and release for the fronts of the model shown in Figs. 1 and 2. Although the total number of fronts is 9, no more than 3 fronts are stored in memory at a time, because the assembling of the frontal matrix of the current nodal front cannot be done without the frontal matrices of its preceding fronts. The preceding fronts will be destroyed only after the assembling is completed. This example illustrates the fact that the required memory amount proves to be relatively small even for large models.

Two allocation modes have been developed for the RAM management. The first mode allocates the memory dynamically in the virtual address space of the process. This scheme provides the fastest performance, but it is fitting only for medium-scale models: about 500,000 to 900,000 equations.

The second mode allocates a block of memory (BuffFront) before the matrix factorization; this block should be able to store at least the frontal matrix of the biggest dimensionality. If the dimensionality of the model and the RAM capacity allow this buffer to store all frontal matrices that need to be stored at a time, then all operations on incomplete fronts will take place solely in RAM. The BuffFront buffer shrinks as it is being filled because some of the preceding fronts have been used and are not needed anymore; if the shrinking does not release enough memory in the buffer for the current front's frontal matrix to fit in, the buffer will be saved to hard disk.

In order to reduce the data copying load, we keep the frontal matrices in a stack no longer because that would require allocating a separate memory block for the current frontal matrix and copying an incomplete front to its storage buffer. In our scheme, an incomplete front the data amount of which exceeds significantly that of the completely factored part of the frontal matrix moves nowhere at all (memory allocation mode 1); this improves the performance of the method. Memory allocation mode 2 moves the data only when the BuffFront buffer shrinks. If a model requires its incomplete fronts to be stored on hard disk, this deteriorates the solver's performance and scalability.

When incomplete fronts are demanded for assembling, a special algorithm determines whether the current front is stored in the memory (in the BuffFront buffer) or saved on hard disk. In the latter case the frontal matrix will be read from hard disk into a special buffer, IO_Buff, of a relatively small dimension (if the dimension of the frontal matrix exceeds that of the IO_Buff buffer, reading will be performed by parts) from where coefficients for its assembling will be taken.

To reduce the size of the file that stores incomplete fronts, next writing from the BuffFront is made to the end of the file only when it becomes impossible to write the buffer to free parts of the file that remain after reading the matrices of incomplete fronts fed to the assembling.

A block of memory of a comparatively small dimension is allocated to store the completely factorized part of the matrix, and when it is filled, its contents are saved to disk.

In both, the first and the second mode, the memory for the lists of frontal nodes and reference indexes is allocated dynamically as the front objects are created, and released when the objects are deleted.

Thus, the only essential limitation of the model's dimension is the capability of the computing system to allocate a buffer, BuffFront, big enough to store the frontal matrix of the maximum dimensionality.

5. Factorization of the frontal matrix

The factorization of the frontal matrix is based on a block $\mathbf{L}\cdot\mathbf{S}\cdot\mathbf{L}^{\mathrm{T}}$ decomposition where $\mathbf{S}$ is a sign diagonal. This is a scheme how the block factorization step is performed within completely assembled equations:

$$\begin{pmatrix} \mathbf{F} & \mathbf{W}_1^{\mathrm{T}} & \mathbf{W}_2^{\mathrm{T}} \\ \mathbf{W}_1 & \mathbf{A} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{W}_2 & \mathbf{B} & \mathbf{C} \end{pmatrix} = \begin{pmatrix} \mathbf{L} & \mathbf{0} & \mathbf{0} \\ \tilde{\mathbf{W}}_1 & \tilde{\mathbf{A}} & \tilde{\mathbf{B}}^{\mathrm{T}} \\ \tilde{\mathbf{W}}_2 & \tilde{\mathbf{B}} & \tilde{\mathbf{C}} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{S}_{\mathrm{L}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{L}^{\mathrm{T}} & \tilde{\mathbf{W}}_1^{\mathrm{T}} & \tilde{\mathbf{W}}_2^{\mathrm{T}} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{4}$$

where the first column $\begin{pmatrix} \mathbf{F} & \mathbf{W}_1^{\mathrm{T}} & \mathbf{W}_2^{\mathrm{T}} \end{pmatrix}^{\mathrm{T}}$ of the original matrix is a block of coefficients for completely assembled equations of dimensions $N \times l_{\mathrm{b}}$ where l_{b} is a block size. In a spatial design model, $l_{\mathrm{b}} = 6$ if the excluded node has any spatial frame or shell finite elements adjacent to it (which is the most frequent case in structural mechanics), and $l_{\mathrm{b}} = 3$ if it is adjacent to volume elements and spatial truss elements. If the node has any constraints imposed, l_{b} will be less than indicated. If all degrees of freedom are constrained in the node, then there are no completely assembled equations ($l_{\mathrm{b}} = 0$) and we go to the next elimination step.

The submatrix $\begin{pmatrix} \mathbf{A} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{B} & \mathbf{C} \end{pmatrix}$ corresponds to partially assembled equations, $\mathbf{S}_L$ is a sign diagonal, the dimensionality of which is equal to that of the diagonal block $\mathbf{F}$, and $\mathbf{I}$ is a unit matrix.

We switch 1st and 2nd columns, and then 1st and 2nd rows in each matrix:

$$\begin{pmatrix} \mathbf{A} & \mathbf{W}_1^{\mathrm{T}} & \mathbf{B}^{\mathrm{T}} \\ \mathbf{W}_1 & \mathbf{F} & \mathbf{W}_2^{\mathrm{T}} \\ \mathbf{B} & \mathbf{W}_2 & \mathbf{C} \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{A}} & \tilde{\mathbf{W}}_1^{\mathrm{T}} & \tilde{\mathbf{B}}^{\mathrm{T}} \\ \mathbf{0} & \mathbf{L} & \mathbf{0} \\ \tilde{\mathbf{B}} & \tilde{\mathbf{W}}_2 & \tilde{\mathbf{C}} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_{\mathrm{L}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \tilde{\mathbf{W}}_1^{\mathrm{T}} & \mathbf{L}^{\mathrm{T}} & \tilde{\mathbf{W}}_2^{\mathrm{T}} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix}. \tag{5}$$

This is the general structure of the frontal matrix, too (Fig. 5) – we cannot be sure the completely assembled equations will be located in its top part. As the structure of the frontal matrix proved to be too complicated, we failed to use factorization and matrix block multiplication routines from the BLAS, LAPACK libraries, and we had to develop our own implementations of the algorithms which are in accordance with the BLAS-III performance requirements [2].

First, we need to copy blocks $\mathbf{W}_1$, $\mathbf{F}$, $\mathbf{W}_2$ to the memory block for completely assembled equations (Fig. 5a,b) and then:

1. Factorize the block $\mathbf{F}$. It follows from Eq. (5) that: $\mathbf{F} = \mathbf{L}\cdot\mathbf{S}_{\mathrm{L}}\cdot\mathbf{L}^{\mathrm{T}}$. The sign diagonal $\mathbf{S}_{\mathrm{L}}$ permits to generalize the Cholesky method onto indefinite matrices. If the calculation of the diagonal element of matrix $\mathbf{L}$ produces a negative radicand, we change it into a positive one and put -1 in the respective position of $\mathbf{S}_{\mathrm{L}}$. If the radicand is positive, we do not alter the sign, just put $+1$ on the diagonal of $\mathbf{S}_{\mathrm{L}}$.

2. Modify the block $\mathbf{W}_1$. It follows from Eq. (5) that: $\mathbf{W}_1 = \mathbf{L}\cdot\mathbf{S}_{\mathrm{L}}\cdot\tilde{\mathbf{W}}_1^{\mathrm{T}} \rightarrow \mathbf{L}\cdot\mathbf{S}_{\mathrm{L}}\cdot\tilde{\mathbf{W}}_1^{\mathrm{T}} = \mathbf{W}_1 \rightarrow \tilde{\mathbf{W}}_1^{\mathrm{T}}$. In other words, we need to make a direct substitution with a set of right parts for the lower triangular matrix $\mathbf{L}$ and the diagonal sign matrix $\mathbf{S}_{\mathrm{L}}$ obtained during step 1.

3. Modify the block $\mathbf{W}_2$: $\mathbf{W}_2 = \tilde{\mathbf{B}}\cdot\mathbf{I}\cdot\mathbf{0} + \tilde{\mathbf{W}}_2\cdot\mathbf{S}_{\mathrm{L}}\cdot\tilde{\mathbf{L}}^{\mathrm{T}} + \tilde{\mathbf{C}}\cdot\mathbf{I}\cdot\mathbf{0} \rightarrow \mathbf{L}\cdot\mathbf{S}_{\mathrm{L}}\cdot\tilde{\mathbf{W}}_2^{\mathrm{T}} = \mathbf{W}_2^{\mathrm{T}} \rightarrow \tilde{\mathbf{W}}_2^{\mathrm{T}}$. Here we need to do the direct substitution with the set of right parts, too.

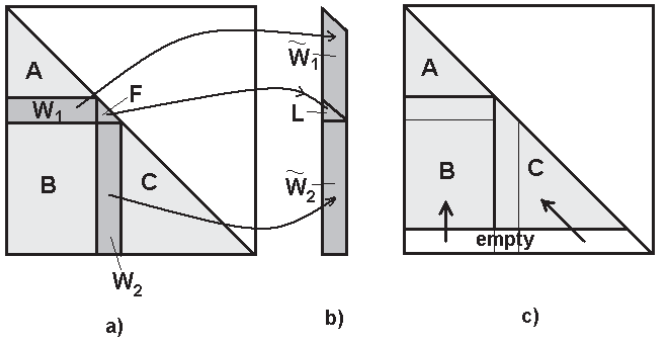

Fig. 5. (a) Structure of the frontal matrix before the block factorization step. (b) copying of completely assembled equations into a memory block for the completely factored matrix and factorization within those memory addresses. (c) Modification of an incomplete front, shifting of block B up by l_b and of block C up and to the left by l_b. "Empty" is an empty strip in the matrix, l_b wide

4. Modify the block **A**: $\mathbf{A} = \tilde{\mathbf{A}} \cdot \mathbf{I} \cdot \mathbf{I} + \tilde{\mathbf{W}}_1 \cdot \mathbf{S}_\mathrm{L} \cdot \tilde{\mathbf{W}}_1^\mathrm{T} + \tilde{\mathbf{B}}^\mathrm{T} \cdot \mathbf{I} \cdot \mathbf{0} \rightarrow \tilde{\mathbf{A}} = \mathbf{A} - \tilde{\mathbf{W}}_1 \cdot \mathbf{S}_\mathrm{L} \cdot \tilde{\mathbf{W}}_1^\mathrm{T}$.
5. Modify the block **B**: $\mathbf{B} = \tilde{\mathbf{B}} \cdot \mathbf{I} \cdot \mathbf{I} + \tilde{\mathbf{W}}_2 \cdot \mathbf{S}_\mathrm{L} \cdot \tilde{\mathbf{W}}_1^\mathrm{T} + \tilde{\mathbf{C}} \cdot \mathbf{I} \cdot \mathbf{0} \rightarrow \tilde{\mathbf{B}} = \mathbf{B} - \tilde{\mathbf{W}}_2 \cdot \mathbf{S}_\mathrm{L} \cdot \tilde{\mathbf{W}}_1^\mathrm{T}$ Move block $\tilde{\mathbf{C}}$ up by l_b positions (Fig. 5c).
6. Modify the block **C**: $\mathbf{C} = \tilde{\mathbf{B}} \cdot \mathbf{I} \cdot \mathbf{I} + \tilde{\mathbf{W}}_2 \cdot \mathbf{S}_\mathrm{L} \cdot \tilde{\mathbf{W}}_2^\mathrm{T} + \tilde{\mathbf{C}} \cdot \mathbf{I} \cdot \mathbf{I} \rightarrow \tilde{\mathbf{C}} = \mathbf{C} - \tilde{\mathbf{W}}_2 \cdot \mathbf{S}_\mathrm{L} \cdot \tilde{\mathbf{W}}_2^\mathrm{T}$ Move block $\tilde{\mathbf{C}}$ by l_b positions up and to the left (Fig. 5c).

Steps 1 through 3 are used to factorize completely assembled equations, and steps 4 through 6 modify the incomplete front.

The maximum size of the matrix **F** is 6×6. This small matrix is stored in the processor's cache, so its factorization is performed very fast at step 1.

Steps 2, 3 are used to do the direct substitution for the set of right parts. If the code is written well enough, it helps to keep the performance here, too.

Steps 4 through 6 are most important. Here we calculate the Schur complement by using the block multiplication of matrices which provides for the proper BLAS-III performance level [2].

Figure 5 illustrates what was said before and shows how data are stored in the frontal matrix. At the end of the block factorization step, a free strip l_b wide appears in the lower part of the matrix, and the dimension of the matrix becomes less by l_b equations than that of the original front matrix.

The efficiency of the block $\mathbf{L} \cdot \mathbf{S} \cdot \mathbf{L}^\mathrm{T}$ decomposition, comparing to the $\mathbf{L} \cdot \mathbf{U}$ decomposition (a non-block one) [7] is illustrated in [6]. In the presented example that contains 1,171,104 equations, the global matrix decomposition time reduced three times (the computer was Pentium III, 1024 MB RAM, 1.2 GHz CPU).

After the factorization is finished, the lower triangular matrix **L** is reproduced on the hard disk. To obtain the final solution, the direct and inverse substitution needs to be done.

6. A parallel block factorization algorithm for the frontal matrix in shared-memory computing systems

The bottleneck in multicore and multiprocessor shared-memory computing systems is an insufficient bandwidth of the system bus which holds back the computing performance when the number of processors grows. This is referred to as a poor scalability of a parallel computing system.

It turns out in problems of linear algebra where the dimensionality of arrays exceed the L2 cache size, that the increased number of processors hardly improves the computing performance for algorithms of the BLAS-I, BLAS-II levels (scalar multiplication of vectors, multiplication of a matrix by a vector); see Fig. 6a. The poor scalability affects also the data copying and the frontal matrix assembling algorithms.

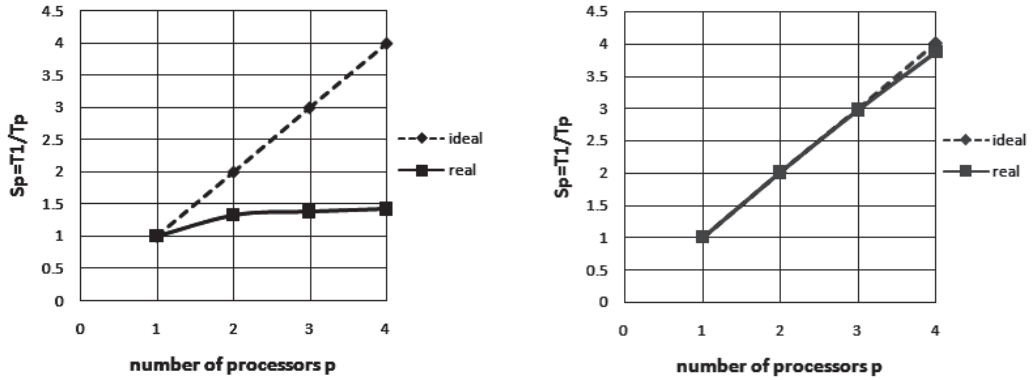

Fig. 6. (a) Scalability of algorithms of the BLAS-I, BLAS-II level. (b) Scalability of the algorithm for block multiplication of matrices of the BLAS-III level. T1 – the computing time with one processor, Tp – the computing time with p processors

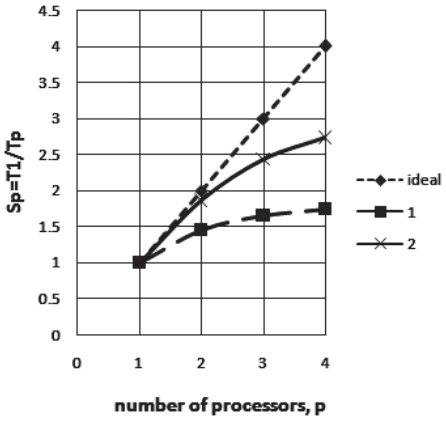

Fig. 7. Scalability of the factorization procedure in the frontal matrix. T1 – the computing time with one processor, Tp – the computing time with p processors; 1 – no front ordering, 2 – using front ordering

A totally different behavior is demonstrated by the matrix block multiplication algorithm of the BLAS-III level (Fig. 6b). Results shown here and below have been obtained with a computer with four-core processor Intel® Core^{TM}2 Quad CPU Q6600, 2.40 GHz, cache 4xL1: 32 KB (code), 4xL1: 32 KB (data), 2xL2: 4096 KB, random-access memory DDR2 800 MHz 4 GB, operating system: Windows XP. The dimensionality of any array involved is many times bigger than the size of the L2 cache.

Therefore, the computers of this class can give an actual acceleration of the computing by building up new processors only for the BLAS-III level algorithms. Such is exactly the frontal matrix factorization algorithm. The hardest things to accelerate are data transfer when some part of the matrix with completely assembled equations is put into a buffer (Fig. 5b) and another data transfer operation needed to shift an incomplete front (Fig. 5c). The latter can be avoided by renumbering the nodes in the front node list in the order opposite to the order of their exclusion, before doing the assembly of each nodal and initial front. For instance, in the example of Fig. 1, the order of node processing for front 5 must be 5, 4, 6, 2 rather than 2, 4, 5, 6. Then the completely assembled equations will be located in the lower part of the frontal matrix before the exclusion, and the incomplete front will be located in sector A only.

Figure 7 shows the scalability of the factorization procedure in the frontal matrix with and without the ordering of the front node list. While the former case shows the acceleration of computing with four processors 1.74 times, the latter gives 2.73 times.

Apparently, the global matrix decomposition procedure would show a poorer scalability because the assembling of the frontal matrix is based on data transfers and operations of lower BLAS levels, and thus it hardly accelerates when more processors are added. The scalability is even poorer when the incomplete fronts are stored on hard disk. The parallel factorization algorithm for the frontal matrix is implemented in OpenMP which is included in the C++ Visual Studio 2005 compiler.

7. Examples of computations

All computation examples have been taken from a collection of problems and models belonging to the software development company SCAD Soft (`www.scadsoft.com`). Its central development is a finite element analysis software, SCAD, used for design analysis of structures and buildings. The solver presented here has been implemented in this software. Seven years of operation have produced a vast collection of various design models developed by the users of the software. Examples 2 through 4 are borrowed from this collection. Example 1 is used because it is easily and credibly reproduced in any computing system or FE modeling software.

Example 1

The tested object is a square plate made of steel ($E = 2 \cdot 10^{11}$ Pa, $\nu = 0.3$), 1 m wide and 0.01 m thick, stiffly clamped in its two corners lying on one side and loaded by concentrated forces $F_x = F_y = F_z = 1000$ N applied in third corner. We use a 4-node shell finite element which has six degrees of freedom per node. The problem was solved on two grids: 400×400 and 800×800. The computing time with the BSMFM (block sparse multifrontal method) solver was compared to the computing time required by the direct sparse matrix solver ANSYS 11.0, where the Total Workspace parameter is set to 1200 MB, a maximum possible value on this computer. Results for the 400×400 mesh are presented in Table 3. The dimensionality of the problem is 964,794 equations. All incomplete fronts were stored in the random-access memory.

With one processor, the BSMFM solver demonstrates the same computation time as the multifrontal ANSYS 11.0 solver. However, with 4 processors the BSMFM solver works faster.

For the 800×800 mesh, the model's dimensionality is 3,849,594 equations. The size of the factored matrix is 9,723 MB, and the dimensionality of the biggest front is 8,400 equations. The direct sparse

Table 3. Time needed to decompose the global matrix, in seconds, for a 400×400 mesh. Dimension of the problem is 964,794 equations

Number of processors	ANSYS 11.0	BSMFM, mode 1 (incomplete fronts are stored in page file)	BSMFM, mode 2 (incomplete fronts are stored in a buffer)
1	221	220	226
2	176	146	152
4	159	112	121

Table 4. Time needed to decompose the global matrix, in sec, for a 800×800 mesh. Dimension of the problem is 3,849,594 equations

Number of processors	BSMFM, mode 2
1	2,091
2	1,450
4	1,080

matrix solver from ANSYS 11.0 failed to solve the problem on this computer because of lack of random-access memory. The BSMFM method succeeded. The results are given in Table 4. Hard disk space was used to store the frontal matrices of incomplete fronts during the solution.

Example 2

A model of a multistorey building contains 328,541 nodes, 351,421 finite elements, and 1,956,634 equations. Fragments of the design model are shown in Fig. 8. The size of the factored matrix is 6,761 MB, and the dimensionality of the biggest front is 8,760 equations. The factorization time on four processors was 893 s.

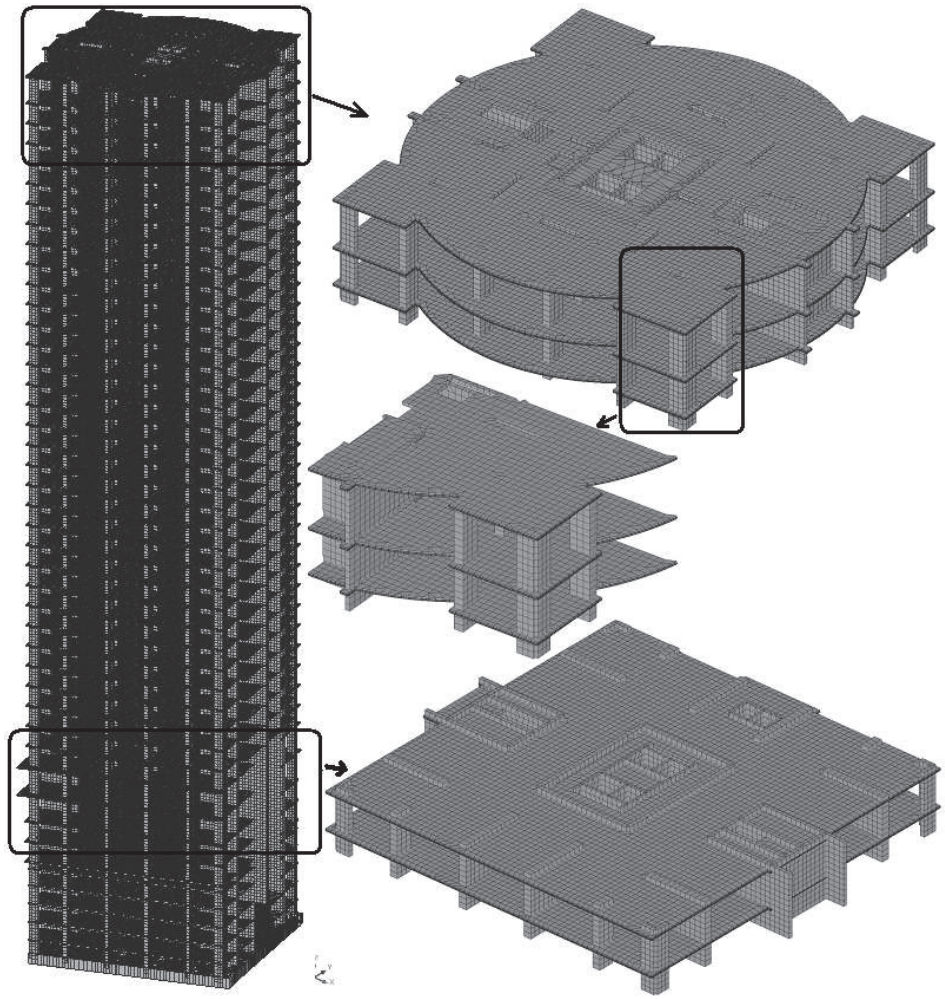

Fig. 8. A model of a multistorey building, 1,956,634 equations

Example 3

A model of an air terminal building in Vilnius, Lithuania contains 117,197 nodes, 127,735 finite elements, and 699,240 equations. Fragments of the design model are shown in Fig. 9. The size of

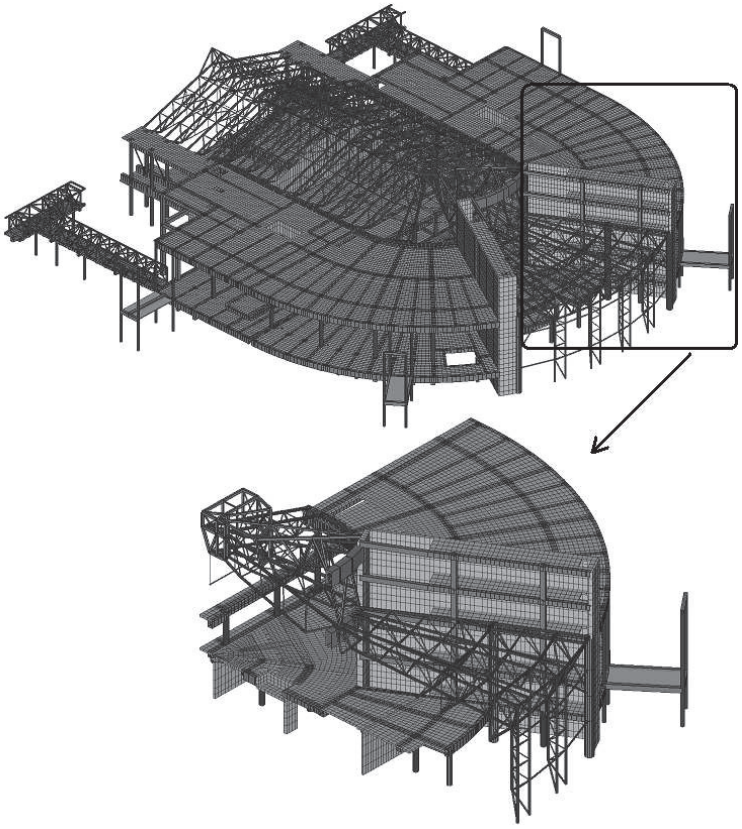

Fig. 9. A model of an air terminal building, 699,240 equations

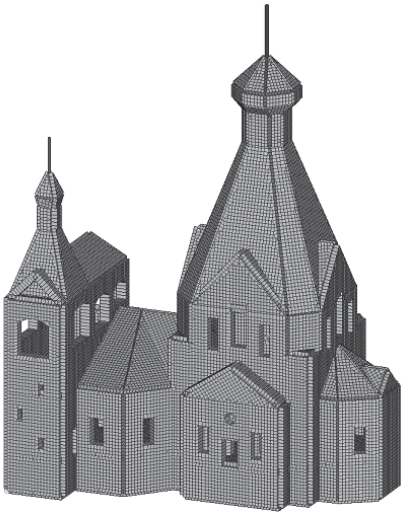

Fig. 10. A model of a church, 316,509 equations

the factored matrix is 1,044 MB, and the dimensionality of the biggest front is 1,614 equations. The factorization time on four processors was 45 s.

Example 4

A model of a church consists of 52,752 nodes, 55,062 finite elements, and 316,509 equations. Fragments of the design model are shown in Fig. 10. The size of the factored matrix is 622 MB, and the dimensionality of the biggest front is 3,042 equations. The factorization time on four processors was 30 s.

8. Conclusions

The first example illustrates that the performance of the proposed solver is close to the performance of the sparse direct solver from the famous FEM software ANSYS 11.0, but our solver is less consuming in terms of the core memory (Example 1 – a plate with the mesh 800×800).

The factorization time for a typical design model (200,000 to 900,000 equations) from structural mechanics (examples 3, 4) is about 1-2 min. on commonly used office computers. For respectively large problems comprising about 1,000,000 to 3,000,000 equations (Example 1 – a plate with the mesh 800×800, Example 2), the factorization time is about 10-40 min. It is a good results which makes it possible to analyze all design model within an acceptable computing time.

The dynamic management of the core memory (all fronts which have been used are immediately destroyed), the substructure-by-substructure assembling-eliminating procedure, virtualization of the decomposed matrix storage and the storage of incomplete fronts, and the use of a symmetrical storage scheme for frontal matrices entail relatively low requirements to the core memory. It is a very important factor for the implementation of the presented solver in FEM software intended for small and medium design offices.

Features of high performance – a multithreading parallelization and a block matrix multiplication algorithm developed specifically to take into account the particular structure of the frontal matrix and the symmetrical storage scheme – allow us to accelerate the computing significantly.

References

[1] C. Ashcraft, J.W.-H. Liu. *Robust Ordering of Sparse Matrices Using Multisection. Technical Report CS 96-01.* Department of Computer Science, York University, Ontario, Canada, 1996.

[2] J.W. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadellphia, 1997.

[3] F. Dobrian, A. Pothen. *Oblio: a sparse direct solver library for serial and parallel computations. Technical Report describing the OBLIO software library.* 2000.

[4] I.S. Duff, J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, **9**: 302–325, 1983.

[5] I.S. Duff, J.K. Reid, J.A. Scott. The use of profile reduction algorithms with a frontal code. *International Journal for Numerical Methods in Engineering*, **28**: 2555–2568, 1989.

[6] S.Yu. Fialko. A block sparse direct multifrontal solver in SCAD software. *Proceedings of the CMM-2005 – Computer Methods in Mechanics, June 21-24, 2005, Czestochowa, Poland*, pp.73–74, 2005.

[7] S.Yu. Fialko. Stress-Strain Analysis of Thin-Walled Shells with Massive Ribs. *Int. App. Mech.*, **40**(4): 432–439, 2004.

[8] A. George A., J.W.-H. Liu. *Computer solution of sparse positive definite systems.* Prentice-Hall, New Jersey, Inc. Englewood Cliffs, 1981.

[9] A. George A., J.W.-H. Liu. The Evolution of the Minimum Degree Ordering Algorithm. *SIAM Rev.*, **31**(March): 1–19, 1989.

[10] N.I.M. Gould, Y. Hu, J.A. Scott. *A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. Technical report RAL-TR-2005-005.* Rutherford Appleton Laboratory, 2005.

[11] B.M. Irons. A frontal solution program for finite-element analysis. *International Journal for Numerical Methods in Engineering*, **2**: 5–32, 1970.

[12] G. Karypis, V. Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report TR 95-035.* Department of Computer Science, University of Minnesota, Minneapolis, 1995.

[13] J.A. Scott. A frontal solver for the 21st century. *Communications in Numerical Methods in Engineering*, **22**: 1015–1029, 2006.

[14] S.W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, **23**: 1315–1324, 1986.