International Conference
ENGINEERING MECHANICS 2010
Svratka, Czech Republic, May 10 – 13, 2010

PERFORMANCE ANALYSIS OF A SPARSE DIRECT SOLVER IMPLEMENTATION

P. Pařík [1]

Summary: *The paper presents the results of a performance analysis of the new sparse direct solver and the existing frontal solver of the in-house finite element system PMD, which has been carried out on a set of finite element problems from real engineering applications. The sparse direct solver has been designed to efficiently solve large finite element problems (*10^6 *to* 10^7 *equations) to overcome limitations of the frontal solution method. Some modifications to existing numerical methods for storage, ordering and solution of linear equation systems have been implemented in the sparse direct solver to enhance its effectiveness, and are also briefly mentioned in the paper.*

1. Introduction

The solution of a system of linear equations lies at the basis of every computation using the finite element method. In the past decades, much attention has been directed towards an efficient implementation of solvers for large finite element problems. Direct methods, based on the Gaussian elimination, could not be overly used due to the limited capacity of available computers, which favored iterative methods that had lower computational demands but were much less robust. With the increasing performance of computers in late 1980s it became possible to implement skyline direct solvers, which exploited the fact that the skyline of the coefficient matrix was retained during the factorization. Today, skyline solvers are still mistakenly considered by some as the ultimate direct solvers for large finite element problems. However, the research conducted in the last decade showed that very large finite element problems could be solved more efficiently by a general sparse direct solver that would work only with nonzero entries of the coefficient matrix.

2. Implementation of the sparse direct solver

The basic methods involved in a sparse direct solution are the matrix storage method, the (pre)ordering method, and the solution (factorization) method, which are all interdependent to some degree. The storage of the whole coefficient matrix (or approximately one half in symmetric cases) is obviously never acceptable except the smallest problems that are however not practical. Matrix storage schemes thus try to exploit sparsity, symmetry and other properties of the coefficient matrix to store as few zero entries of the matrix as possible. Although it is

[1] Ing. Petr Pařík, Institute of Thermomechanics ASCR, Dolejškova 5, 182 00 Prague 8, tel. +420 2 6605 3441, e-mail: parik@it.cas.cz

perfectly reasonable not to store any nonzero entries, which is clearly the most efficient option and is indeed used for example in iterative methods, direct methods unfortunately spoil the sparsity structure by introducing new nonzero entries during the triangularization (factorization) of the coefficient matrix. This occurence of new nonzeros is called the *fill-in* and presents a major drawback and difficulty of direct methods. The initial nonzero structure of the matrix as well as the final nonzero structure (the amount of fill-in) can be substantially affected by using an ordering method. Storage of the coefficient matrix in the case of large problems requires a careful consideration, since it has a significant impact on the practical implementation of the solution method.

Ordering methods switch rows and columns of the matrix to obtain another, preferably more suitable, order of pivots on the main diagonal. An important consequence is that the resulting nonzero structure of the reordered matrix may allow an efficient storage and/or factorization using direct methods. Different ordering methods usually imply certain types of matrix storage schemes. For example, profile minimization algorithms move all nonzero entries close to the main diagonal, yielding a band or skyline matrix, therefore, a band or skyline storage format is ideal. The fill-in can occur only within the band or under the skyline of the coefficient matrix, thus it is also effectively reduced. Fill-in minimization algorithms are used specifically to reduce the fill-in, but they result in a more complicated nonzero structure requiring more complex storage schemes. Ordering methods usually work with graphs representing the nonzero structure of the matrix, and since operations on graphs are computationally expensive, the time complexity rises quickly in the case of large problems. However, without a suitable ordering, the direct solution of large problems is generally impossible due to uncontrollable fill-in.

The sparse direct solution is mostly performed using a variant of the Gaussian elimination. In order to be efficient, the method must exploit the sparsity of the coefficient matrix by avoiding unnecessary operations on zero entries. This largely depends on the employed matrix storage scheme and also on the used ordering method. Small and medium problems can be usually fully solved in memory (in core), but in the case of large problems, it may be necessary to store a part of the matrix data temporarily on the disk (out of core). Out-of-core solution of course involves much more complicated algorithms and since the disk storage is much slower than the memory storage, great care must be taken to implement the solution method efficiently.

2.1. Storage metod of the coefficient matrix

A simple and efficient matrix storage method for the out-of-core sparse direct solver is proposed in Parik (2010). It is partially based on the compressed row storage format and the dynamic block compressed row storage format described in Ueberhuber (1994). The proposed storage format has the following features:

- Only one-dimensional arrays are employed that are easy to understand and implement in FORTRAN 77 using the PMD memory allocation strategy.
- Integer and real data are stored in separate arrays and thus no index conversions are necessary.
- General finite element meshes with variable number of nodal degrees of freedom as well as the prescribed boundary conditions are taken into account, i.e., nodal submatrices can be of variable size.

- Nodal submatrices are numbered implicitly to allow an efficient out-of-core data manipulation using direct-access file.
- The size of the data structure is the same or less than in the K3/F77 storage format.

The proposed block sparse storage format is very efficient in both the storage overhead (resulting in low matrix storage size) and the algorithmic overhead (resulting in quick accessibility of matrix entries). It allows the sparse direct solver to perform all necessary operations on the coefficient matrix (assembly, factorization and substitution) either in-core or out-of-core, depending on the amount of available memory. Efficient out-of-core data manipulation is provided by using direct-access file.

2.2. Ordering method

The ordering method chosen for the sparse direct solver is the approximate minimum degree ordering algorithm, described for example in Amestoy, Davis and Duff (2004), to minimize the fill-in introduced in the sparse factorization, and consequently to reduce the storage requirements and the time needed to obtain the solution. Although established implementations of the approximate minimum degree ordering are available, their use in a commercial sparse direct solver is prohibited, and the possibility for modifications is limited. Therefore, an original version of the minimum degree algorithm is proposed in Parik (2010) for the use in the sparse direct solver. This version is capable of selective switching of various features (supervariables, approximate degrees, etc.) to allow the efficiency of the algorithm to be analyzed and tuned up thoroughly.

2.3. Solution method

The solution method chosen for the sparse direct solver is the block $\mathbf{LDL}^{\mathrm{T}}$ factorization, which is described (although without considering the block structure) for example in Ueberhuber (1994). The most efficient algorithm for an out-of-core $\mathbf{LDL}^{\mathrm{T}}$ factorization, proposed in Parik (2010), is the left-looking algorithm performed by rows, which is not commonly mentioned in literature. It allows efficient use of the available memory and also efficient out-of-core data manipulation.

3. Performance analysis

The analysis was performed on four different computer systems that are listed in Table 1. Systems `hastrman` and `rusalka` are UNIX-based mainframe servers[2] while `ds9` and `obkladac` are common Windows-based PCs. UNIX systems allow the 32-bit programs to utilize the full 2 GiB memory space for data, whereas Windows systems have somewhat lower limit, probably due to shared memory space between program data and code. On `rusalka` the limit 1 GiB is imposed by the administrator. On `obkladac` the physical memory is only 1 GiB but virtual memory is provided by the operating system, therefore the limit is the same as on `ds9`. Collectively, these various configurations simulate practical environments where the PMD[3] system can be used.

[2] The servers are located at the Institute of Thermomechanics ASCR.

[3] Details about the PMD system can be found at `http://www.vamet.cz/`.

On all four systems a 32-bit FORTRAN 77 compiler was used to compile the PMD code (including the frontal solver and the sparse direct solver) with full optimizations. The reasons for not testing also the 64-bit version (in UNIX) is that it is not up-to-date, because the PMD system is presently used mostly for finite element computations on 32-bit platforms in practice. Nevertheless, the sparse direct solver would benefit from the 64-bit memory space since it could perform all matrix operations in-core regardless of physical memory size. There are however no benefits from the 64-bit version of the frontal solver.

Table 1. Computer systems used for solver performance analysis

Name	Type	Operating system	Fortran compiler
`hastrman`	Server	HP Tru64 UNIX V5.1B	HP Fortran V5.5A-3548-48D88
`rusalka`	Server	HP Tru64 UNIX V5.1B	HP Fortran V5.5A-3548-48D88
`ds9`	PC	Windows XP V5.1.2600	Compaq Visual Fortran V6.6A
`obkladac`	PC	Windows XP V5.1.2600	Compaq Visual Fortran V6.6A

Name	Processor(s)	Frequency [GHz]	Memory [GiB]	Maximum core size `LI`
`hastrman`	4x DEC Alpha EV7	1.0	8	536,870,911
`rusalka`	4x DEC Alpha EV6	0.5	2	268,411,848
`ds9`	1x Intel Core 2 Duo T8100	2.1	4	498,000,000
`obkladac`	1x AMD Athlon 64 3000+	1.8	1	498,000,000

Storage requirements of the sparse direct solver and frontal solver are listed in Table 2. It can be seen that the memory requirements of the sparse direct solver are very high compared to the frontal solver, however, the sparse direct solver has to store the whole coefficient matrix in memory, unlike the frontal solver. Due to the optional out-of-core mode, the sparse direct solver is capable to process even problems that do not fit into the available memory, as is demonstrated for example by the problem BUBEN.

On the other hand, the disk storage requirements of the sparse direct solver are substantially lower than that of the frontal solver, up to 90% in some cases. It is due to the efficient block sparse storage and especially due to the use of the fill-in minimization ordering (minimum degree algorithm). The disk storage requirements of the sparse direct solver are slighly higher that its memory requirements because unlike the memory storage, the disk storage has to use fixed-size blocks.

Table 2. Solver storage sizes (in MB)

Problem name	Number of equations	Frontal solver Memory	Frontal solver Disk	Sparse direct solver Memory	Sparse direct solver Disk
P6_LIN	19,656	9	226	55	64
P6_QUAD	74,742	76	2,472	641	695
K1	136,354	14	1,676	300	493
ADS	359,504	14	2,974	362	607
K3	362,262	49	7,307	924	1,464
A98ABT001_1	365,043	76	8,763	758	869
DOCHL	387,829	43	4,820	542	837
BUBEN	1,739,211	127	46,851	10,369	10,924

Ordering times of the sparse direct solver are listed in Table 3 (frontal solver does not use any method of ordering). The ordering times depend on the system, but they are practically negligible compared to the solution times and therefore are not counted for the final comparison of the solvers.

Table 3. Solver ordering times

Problem name	Number of equations	Sparse direct solver `hastrman`	`rusalka`	`ds9`	`obkladac`
P6_LIN	19,656	00:00:00	00:00:01	00:00:00	00:00:00
P6_QUAD	74,742	00:00:04	00:00:09	00:00:01	00:00:02
K1	136,354	00:00:05	00:00:12	00:00:01	00:00:03
ADS	359,504	00:00:31	00:01:11	00:00:07	00:00:18
K3	362,262	00:00:32	00:01:13	00:00:07	00:00:19
A98ABT001_1	365,043	00:00:47	00:01:49	00:00:11	00:00:27
DOCHL	387,829	00:00:41	00:01:34	00:00:09	00:00:23
BUBEN	1,739,211	00:22:15	00:32:32	00:02:53	00:07:00

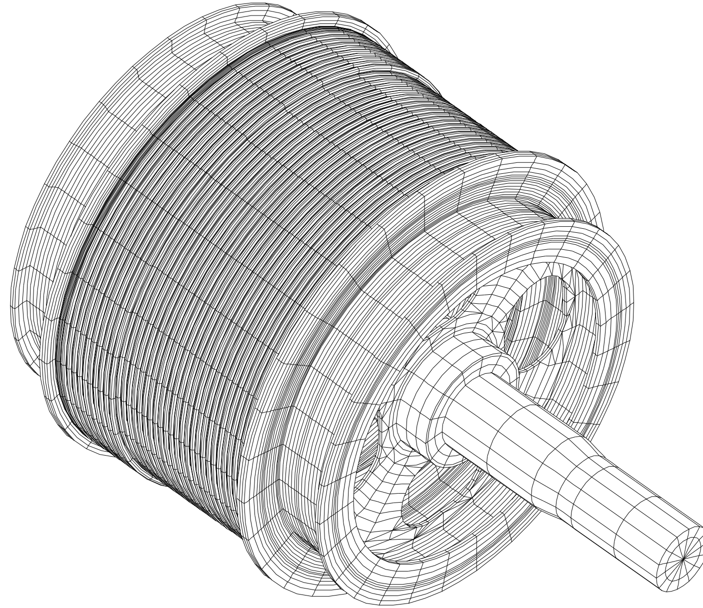
Finally, solution times of the sparse direct solver and the frontal solver are listed in Table 4. It can be seen that the sparse direct solver is indeed quite efficient, the block sparse solution can be up to 90% faster than the frontal solution in some cases. The largest computed problem, BUBEN (see Figure 1), achieved about 60% faster block sparse solution on most systems, except for `hastrman`, where the solution time was severely affected by the disk access speed.

Table 4. Solver solution times

Problem name	Number of equations	Frontal solver `hastrman`	`rusalka`	`ds9`	`obkladac`
P6_LIN	19,656	00:01:26	00:05:23	00:01:13	00:01:28
P6_QUAD	74,742	00:54:23	03:21:32	00:40:01	00:46:06
K1	136,354	00:11:36	00:47:37	00:09:49	00:11:40
ADS	359,504	00:09:59	00:27:19	00:05:59	00:10:07
K3	362,262	01:46:01	05:37:33	01:08:32	01:20:38
A98ABT001_1	365,043	02:12:30	09:20:33	01:52:17	02:10:02
DOCHL	387,829	00:42:46	02:33:13	00:32:21	00:39:50
BUBEN	1,739,211	12:30:47	×	10:07:12	11:46:44
Problem name	**Number of equations**	**Sparse direct solver `hastrman`**	**`rusalka`**	**`ds9`**	**`obkladac`**
P6_LIN	19,656	00:00:28	00:00:54	00:00:11	00:00:20
P6_QUAD	74,742	00:19:00	00:39:04	00:07:16	00:12:46
K1	136,354	00:03:54	00:07:47	00:01:31	00:02:43
ADS	359,504	00:03:43	00:08:26	00:01:48	00:03:06
K3	362,262	00:20:37	00:38:57	00:07:28	00:13:03
A98ABT001_1	365,043	00:13:18	00:31:32	00:07:18	00:11:53
DOCHL	387,829	00:06:38	00:13:56	00:02:58	00:04:59
BUBEN	1,739,211	12:21:06	×	05:29:27	07:12:17

× *not completed due to system limitations*

Figure 1. Mesh of the largest problem BUBEN (a winding engine drum)



4. Conclusions

The implemented sparse direct solver based on the block sparse matrix storage format, minimum degree algorithm and block sparse factorization has been confirmed to be an efficient and adequate complement to the existing frontal solver of the PMD system. It has high memory requirements, but in practice, this is not a difficulty for present computers and operating systems, and the savings in time and disk space required for the solution can be up to 90% or even more. The sparse direct solver is a suitable option especially in cases of very large problems where the frontal solver starts to have unreasonable requirements on disk storage. However, it should be noted that there are some cases where the frontal solution may be faster, especially when the frontwidth is small, because the minimum degree algorithm (being merely a heuristic) is not guaranteed to always give the best ordering.

5. Acknowledgment

The support of grant GA ČR 101/09/1630 is gratefully acknowledged.

6. References

Pařík, P. 2010: *An out-of-core sparse direct solver for very large finite element problems.* Doctoral thesis, Czech Technical University in Prague.

Duff, I. S. & Scott, J. A. 2005: *Towards an automatic ordering for a symmetric sparse direct solver.* Technical Report RAL-TR-2006-001, Rutherford Appleton Laboratory, Oxon.

Amestoy, P. R., Davis, T. A. & Duff, I. S. 2004: Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software* 30, 381–388.

Ueberhuber, C. W. 1994: *Numerical computation.* Springer, Berlin.