# A Fast Multifrontal Solver for Non-linear Multi-physics Problems*

Alberto Bertoldo, Mauro Bianco, and Geppino Pucci

Department of Information Engineering, University of Padova, Italy
{cyberto,bianco1,geppo}@dei.unipd.it

**Abstract.** The paper presents a highly optimized implementation of a multifrontal solver for linear systems arising in the FEM simulation of multi-physics problems related to the behaviour of porous media. The solver features a careful preprocessing phase that is crucial to considerably speed up both system assembly and Gaussian elimination. When run on a number of relevant test cases, the proposed solver compares very favourably with both its previous unifrontal counterpart and two general multifrontal solvers well known in the literature.

## 1 Introduction

The FEM simulation of non-linear fully coupled multi-physics problems is a challenging and, for many aspects, yet open problem. Due to their complexity, these problems require physical models based on very large, non-linear systems of PDEs. The specific multi-physics problem targeted in this paper is the simulation of porous media [4]. The computational kernel of the simulation is a solver of large unsymmetric linear systems, whose sparsity pattern does not change through iterations.

Due to their size, by virtue of their structure, and for stability reasons, the above mentioned linear systems are often solved using *frontal* approaches [6]. For the specific application under consideration, previous work [4] has studied the accuracy of standard pivoting strategies for the subfamily of *unifrontal* methods. Subsequently, Bianco [3] has proposed the novel *implicit minimum-degree* pivoting strategy, which affords an efficient implementation in terms of fast routines for dense matrices (e.g., the BLAS level 3 library.)

This paper extends previous work to the *multifrontal* approach, in which the assembly of the linear system proceeds in a tree-like fashion on the entire mesh. On four sizable test cases from our application domain, the resulting solver outperforms both the previous unifrontal solver of [3] tailored for the specific problem at hand, and other two prominent general multifrontal solvers: MUMPS [1] and SuperLU [5].

## 2 Logical Solution Algorithm

As stated above, the multifrontal approach entails a tree-like assembly of the linear system from repeated joins of subsystems corresponding to larger and larger regions of

* This work was supported, in part, by MIUR of Italy under COFINLAB and PRIN grants. Further support for the first and second author came from Consorzio "Roma Ricerche" and CISM of Italy, respectively.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 614–617, 2004.

the FE mesh. Given an *assembly tree* and one of its internal nodes $t$, let $\mathbf{A}^{[t]}\mathbf{u}^{[t]} = \mathbf{b}^{[t]}$ be its corresponding system after assembly of the associated region, and $\bar{\mathbf{A}}^{[t]}\bar{\mathbf{u}}^{[t]} = \bar{\mathbf{b}}^{[t]}$ be the *reduced system* obtained by eliminating *Fully-Summed* (FS) variables. Let also $\mathrm{lx}(t)$ and $\mathrm{rx}(t)$ denote, respectively, the left and the right children (subregions) of node (region) $t$. The logical sequence of steps that must be performed once two children regions are ready to be joined is the following:

1. **Assembly**: Merge the (reduced) matrices of the two (children) regions into a single (not reduced) matrix relative to the joined (parent) region. In the merged matrix $\mathbf{A}^{[t]}$, rows/columns relative to variables in the left child region are placed first, while the ones relative to the right child region are appended later.
2. **Swap**: Pack FS rows and columns into blocks (called *FS blocks*) towards the bottom and right sides of the matrix. (In order to make the computation efficient, and to ease subsequent assembly phases, we employ a minimal swap algorithm which does not move those rows/columns which are already in their destination blocks.)
3. **Copy**: Copy the FS blocks into temporary buffers which will be passed as input parameters to optimized linear algebra routines for fast UL factorization.
4. **Elimination**: Let

$$\mathbf{A}^{[t]} = \left[\begin{array}{c|c} \mathbf{N} & \mathbf{R} \\ \hline \mathbf{C} & \mathbf{S} \end{array}\right], \text{ where } \begin{cases} \mathbf{N} \in \mathbb{R}^{(f_t - n_t)\times(f_t - n_t)}, \mathbf{R} \in \mathbb{R}^{(f_t - n_t)\times n_t}, \\ \mathbf{C} \in \mathbb{R}^{n_t \times (f_t - n_t)}, \mathbf{S} \in \mathbb{R}^{n_t \times n_t} \end{cases} \quad (1)$$

   $\mathbf{C}$, $\mathbf{R}$ and $\mathbf{S}$ represent the three FS blocks ($f_t$ and $n_t$ are defined later). Obtain the factorization $\mathbf{S} = \mathbf{UL}$, compute $\mathbf{U}' = \mathbf{R}\mathbf{L}^{-1}$ and $\mathbf{L}' = \mathbf{U}^{-1}\mathbf{C}$, and finally compute the Schür complement with respect to $\mathbf{S}$ as $\bar{\mathbf{A}}^{[t]} = \mathbf{N} - \mathbf{U}'\mathbf{L}'$.
5. **Strip**: Strip blocks $\mathbf{U}'$ and $\mathbf{L}'$ and factors $\mathbf{U}$ and $\mathbf{L}$ and store them to finally solve the system with substitution algorithms. The region is now associated with the reduced matrix $\bar{\mathbf{A}}^{[t]}$.

In our application all the matrices involved in the above steps are dense [3], hence the partial factorization of $\mathbf{A}^{[t]}$ and matrix $\bar{\mathbf{A}}^{[t]}$ can be obtained efficiently in terms of BLAS level 3 routines.

We call the previous algorithm a *logical algorithm*, since it needs an actual implementation. To this aim, at the ouset we perform a preprocessing phase, called *symbolic analysis*, in order to gather relevant data of the solution process. In [3] it is shown how to merge the step 2 (swap) and the step 3 (copy) steps. Below, we describe a faster and more advanced merging of steps 1 (assembly), 2 (swap), and 3 (copy) into what we call a *super-assembly* phase.

## 3 Symbolic Analysis

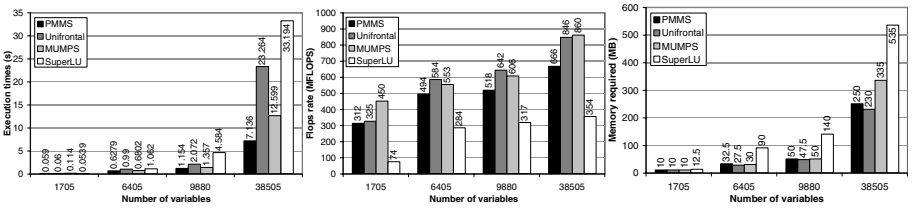
Our symbolic analysis takes advantage of the fact that the rows and columns that become fully-summed at each node of the assembly tree are the same *at every iteration* of the solver for a given mesh, so, for each region, the position of each variable in the blocks $\mathbf{N}$, $\mathbf{C}$, $\mathbf{R}$, and $\mathbf{S}$ can be computed beforehand. Moreover, we can keep the sub-matrices $\mathbf{N}$, $\mathbf{C}$, $\mathbf{R}$, and $\mathbf{S}$ of $\mathbf{A}^{[t]}$ within separate buffers $\mathbf{B_N}$, $\mathbf{B_C}$ $\mathbf{B_R}$ and $\mathbf{B_S}$. During symbolic analysis, the following information is computed for each node of the assembly tree $t$:

$f_t$ : the dimension of $A^{[t]}$, which is called *front size*;
$n_t$ : number of variables of $A^{[t]}$ that become fully-summed;
$\upsilon_t(\cdot)$ : for $i \in \{1, \ldots, f_t\}$, $\upsilon_t(i)$ is the global index of the $i$-th variable of $A^{[t]}$ soon after the swap step of the logical algorithm. Clearly, variables $\upsilon_t(i)$ for $i \in \{1, \ldots, f_t - n_t\}$ are not FS, while variables $\upsilon_t(j)$ for $j \in \{f_t - n_t + 1, \ldots, f_t\}$ are FS.

We assume to have available at a leaf $t$ the corresponding Schür complement $\bar{\mathbf{A}}^{[t]}$ obtained from static condensation of the associated element [3]. For what concerns composite regions, let $\bar{\mathbf{A}}^{[\mathrm{lx}(t)]}$ and $\bar{\mathbf{A}}^{[\mathrm{rx}(t)]}$ be the Schür complements of the left and right subregions of $t$, respectively. From the previous steps, we have available the correspondence between their row (resp., column) indexes and global variable indexes, respectively in $\upsilon_{\mathrm{lx}(t)}(\cdot)$ and $\upsilon_{\mathrm{rx}(t)}(\cdot)$. From these, we can easily compute two index maps $\gamma_l^{[t]}(k)$ and $\gamma_r^{[t]}(k)$ that, for the $k$-th variable in $A^{[t]}$, indicate the indexes associated with that variable in the matrices of the two children as follows:

$$\gamma_l^{[t]}(k) = \begin{cases} h \;\; \text{if } \exists h < \infty : \\ \quad \upsilon_t(k) = \upsilon_{\mathrm{lx}(t)}(h); \\ \infty \text{ otherwise.} \end{cases} \qquad \gamma_r^{[t]}(k) = \begin{cases} h \;\; \text{if } \exists h < \infty : \\ \quad \upsilon_t(k) = \upsilon_{\mathrm{rx}(t)}(h); \\ \infty \text{ otherwise.} \end{cases} \tag{2}$$

Using these definitions, we can think of $k$ as the destination row/column index of a variable in the assembled region and of $\gamma_l^{[t]}(k)$ and $\gamma_r^{[t]}(k)$ as the source indexes of that variable, respectively in the left component and in the right component during the assembly process (in case the $k$-th variable is not present in one of the two children, we map the variable to a "default" value of $\infty$, which is considered an index for a zero element) . These quantities can be computed for each node in the tree by simulating the assembly process of the logical algorithm without any floating point operation.




(a) Execution times (b) FLOP rate (c) Memory required

**Fig. 1.** Performance results for test cases using the four solvers.

The main purpose of the index maps defined in Eq. (2) is that of generating the submatrices $\mathbf{N}$, $\mathbf{C}$, $\mathbf{R}$ and $\mathbf{S}$ of $\mathbf{A}^{[t]}$ starting from $\bar{\mathbf{A}}^{[\mathrm{lx}(t)]}$ and $\bar{\mathbf{A}}^{[\mathrm{rx}(t)]}$, and placing them directly into buffers $\mathbf{B_N}$, $\mathbf{B_R}$, $\mathbf{B_C}$, and $\mathbf{B_S}$, thus merging together the first three steps of the logical algorithm. For efficiency reasons, we make sure that buffer $\mathbf{B_N}$ reuse as much memory space as possible from the one previously allocated to $\bar{\mathbf{A}}^{[\mathrm{lx}(t)]}$. In order to avoid overwriting entries of the latter matrix prior to their use, it is sufficient to compute $\mathbf{B_S}$, $\mathbf{B_R}$, and $\mathbf{B_C}$, before $\mathbf{B_N}$. In fact, observe that $\bar{\mathbf{A}}^{[\mathrm{lx}(t)]}(i,j)$ is used when updating $\mathbf{A}^{[t]}(m,n)$, with $\gamma_l^{[t]}(m) = i$ and $\gamma_l^{[t]}(n) = j$. By the properties of the precomputed maps, the only entries of $\bar{\mathbf{A}}^{[lx(t)]}$ in common with $\mathbf{B_N}$ that could be overwritten are

those entries $(i, j)$ such that both $\gamma_l^{[t]}(i)$ and $\gamma_l^{[t]}(j)$ are greater than $f_t - n_t + 1$, which have already been used, since they contribute to entries in $\mathbf{B_S}$, $\mathbf{B_R}$, and $\mathbf{B_C}$.

## 4 Performance Results

This final section compares some performance figures of the presented solver (dubbed PMMS, *Porous Media Multifrontal Solver*) with the ones of the previous unifrontal version [3] and those of two well-known general solvers: SuperLU version 3.0 [5] and MUMPS version 4.3 [1]. The test cases are four FE (square) meshes yielding linear systems with sizes varying from 1705 to 38505 variables. Our platform is an IBM Power3 processor at 375MHz and 4 Gbyte of memory.

The results of the tests are shown in Figs. 1.(a), 1.(b), and 1.(c) reporting, respectively the execution times, the FLOP rates, and the memory requirements for each iteration of the solver. On our grids, PMMS largely outperforms the unifrontal solver of [3] and is always faster than both MUMPS and SuperLU except for the smallest, and least significant test case (see Fig. 1.(a)). The worse performance exhibithed by SuperLU w.r.t. MUMPS had also been observed in [2]. For the sake of fairness, the running times measured for SuperLU do not include the assembly of the full matrix of the system.

Fig. 1.(b) shows that MUMPS exhibits a rather larger rate of execution of floating point operations than PMMS, which suggests that the former solver is better tuned than ours (which is still away from being optimized) although it resorts to an algorithm of higher complexity. From this point of view, matching the floating point performance of MUMPS appears as a challenging objective for the final release of our solver. For the memory usage, in Figure 1.(c) it can be noted how MUMPS has a slightly smaller space requirement than PMMS for the biggest test case, while SuperLU always has the largest. Finally, we want to remark that the numerical accuracy of the various solvers with respect to *modified component-wise backward error* metric [4] (figures not shown here due to space constraints) are all comparable with the roundoff unit of the 64-bit floating point precision used.

## References

1. P.R. Amestoy, I.S. Duff, J.Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Meth. Appl. Mech. Eng.* 184:501–520, 2000.
2. P.R. Amestoy, I.S. Duff, J.Y. L'Excellent, X.S. Li Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Trans. on Meth. Soft. (TOMS).* 24(4):388–421, 2001.
3. M. Bianco A high-performance UL factorization for the frontal method. In *Proc. Int. Conf. on Computational Science and its Applications (ICSSA 2003)*, pages 893–902, Montreal, CA, May 2003. Lecture Notes in Computer Science 2667, Springer-Verlag.
4. M. Bianco, G. Bilardi, F. Pesavento, G. Pucci, and B. A. Schrefler. A frontal solver tuned for fully-coupled non-linear hygro-thermo-mechanical problems. *Int. J. Num. Meth. Eng.*57(13):1801–1818, 2003.
5. J.W. Demmel and S.C. Eisenstat and J.R. Gilbert and X.S. Li and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM J. Mat. Anal. and Appl.* 20(3):720–755, 1999.
6. I. S. Duff and J. A. Scott. The design of a new frontal code for solving sparse, unsymmetric systems. *ACM Trans. Math. Soft.* (TOMS) 22(1):30–45, 1996.