A Static Parallel Multifrontal Solver for Finite Element Meshes*

Alberto Bertoldo, Mauro Bianco, and Geppino Pucci

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

Abstract. We present a static parallel implementation of the multifrontal method to solve unsymmetric sparse linear systems on distributed-memory architectures. We target Finite Element (FE) applications where numerical pivoting can be avoided, since an implicit minimum-degree ordering based on the FE mesh topology suffices to achieve numerical stability. Our strategy is static in the sense that work distribution and communication patterns are determined in a preprocessing phase preceding the actual numerical computation. To balance the load among the processors, we devise a simple model-driven partitioning strategy to precompute a high-quality balancing for a large family of structured meshes. The resulting approach is proved to be considerably more efficient than the strategies implemented by MUMPS and SuperLU_DIST, two state-of-the-art parallel multifrontal solvers.

1 Introduction

Finite Element (FE) applications typically rely on the numerical solution of systems of Partial Differential Equations (PDEs) modeling the behavior of some physical system of interest [16]. From a computational point of view, solving these PDEs involves the solution of large, sparse linear systems whose sparsity pattern depends on the topology of the FE mesh. Since the physical phenomena under simulation may be non-linear and evolving through time, a given FE mesh may require the solution of many linear systems with the same sparsity pattern but with different numerical values. In turn, each of these linear systems can be solved through iterative or direct methods [10]. The use of direct methods becomes particularly desirable if the FE application is such that numerical pivoting is unnecessary and the mesh topology remains unchanged over many iterations. Under this common scenario, successive solutions of (numerically) different linear systems can share the same computation schedule, hence heavy preprocessing, whose cost is amortized over the iterations, can be used for optimization purposes.

In what follows, we regard an FE mesh as a graph with N vertices, representing degrees of freedom (*unknowns*) of the physical system, and edges connecting

* Support for the authors was provided in part by MIUR of Italy under Project *ALGONEXT* and by the European Union under the FP6-IST/IP Project *AEOLUS*.

M. Guo et al. (Eds.): ISPA 2006, LNCS 4330, pp. 734–746, 2006.

any two unknowns interacting within some constraint. Under this view, an *element* becomes a fully-connected subgraph (*clique*) of M vertices, where M is a small constant (usually less than 100). Each clique is connected to other cliques by *boundary vertices* in a sparse pattern. The $N \times N$ linear system $\mathbf{Ax} = \mathbf{b}$ associated to the FE mesh at some iteration is *assembled* by computing $\mathbf{A} = \sum_{e=1}^{n_e} \mathbf{A}^e$, and $\mathbf{b} = \sum_{e=1}^{n_e} \mathbf{b}^e$, where n_e is the number of elements of the FE mesh, and the entries a_{ij}^e may be nonzero only when i and j are indices of vertices of the e-th element of the mesh.

Direct methods solve the assembled linear system by decomposing $\mathbf{A}$ into easy-to-solve factors, the most effective strategies for unsymmetric matrices being based on LU decomposition. The elimination order aims at reducing the emergence of *fill-ins* to preserve sparsity, and can be represented by an *Assembly Tree* (AT) whose nodes relate to the elimination of a set of unknowns with the same sparsity pattern. An effective implementation of this idea is the *multifrontal method* [11,15], which can be regarded as a post-order visit of the AT where eliminations at a node employ dense matrix kernels for improved performance [4]. If the linear system arises from an FE application, then the resulting AT can be directly related to the topology of the FE mesh, namely, the former represents a *hierarchical decomposition* of the latter into connected regions, with each AT node corresponding to one such region.

The multifrontal method interleaves phases of *assembly* of larger and larger portions of the FE mesh with *elimination* phases, where a partial LU decomposition is executed on *Fully-Summed* (FS) rows and columns, which are those that need never be updated by further phases. Visiting a leaf of the AT involves computing the matrix $\mathbf{A}^e$ and the right hand side vector $\mathbf{b}^e$ of the related mesh element. Visiting an internal node of the AT entails *merging* the two regions corresponding to its sons and then eliminating the resulting FS rows and columns[1].

Any efficient implementation of the multifrontal method maintains a compact representation of the matrices associated with the nodes of the AT into *dense* two-dimensional arrays storing the non-zero entries of the corresponding matrices. When two regions are merged together into a larger region associated with node n of the AT, during the assembly phase we build an $f \times f$ matrix $\mathbf{A}_n$. If s rows and columns of $\mathbf{A}_n$ become FS due to the assembly, the entries of $\mathbf{A}_n$ can be arranged as

$$\mathbf{A}_n = \begin{bmatrix} \mathbf{S}_n & \mathbf{R}_n \\ \mathbf{C}_n & \mathbf{N}_n \end{bmatrix}, \quad \text{where } \begin{cases} \mathbf{S}_n \in I\!R^{s \times s}, \mathbf{R}_n \in I\!R^{s \times (f-s)}, \\ \mathbf{C}_n \in I\!R^{(f-s) \times s}, \mathbf{N}_n \in I\!R^{(f-s) \times (f-s)} \end{cases}. \tag{1}$$

Submatrices $\mathbf{S}_n$, $\mathbf{R}_n$, and $\mathbf{C}_n$ are called the *FS blocks*. After each assembly, the elimination phase computes the following matrices from $\mathbf{A}_n$:

1. $\mathbf{L}_n \mathbf{U}_n \leftarrow \mathbf{S}_n$;
2. $\bar{\mathbf{U}}_n \leftarrow (\mathbf{L}_n)^{-1} \mathbf{R}_n$;
3. $\bar{\mathbf{L}}_n \leftarrow \mathbf{C}_n (\mathbf{U}_n)^{-1}$;
4. $\bar{\mathbf{A}}_n \leftarrow \mathbf{N}_n - \bar{\mathbf{L}}_n \bar{\mathbf{U}}_n$.

[1] To avoid ambiguity, we will use the term "vertices" exclusively for the vertices of the FE mesh, while the word "node" will be reserved for the vertices of the AT.

After the elimination phase, matrices $\mathbf{L}_n$, $\mathbf{U}_n$, $\bar{\mathbf{U}}_n$, and $\bar{\mathbf{L}}_n$ can be stored elsewhere in view of the final forward and backward substitution activities needed to obtain the final solution, whereas the Shür complement $\bar{\mathbf{A}}_n$ will contribute to the assembly phase associated with the parent of n. Steps 2, 3, and 4 can employ high-performance BLAS Level 3 routines [9], whereas any efficient LU decomposition algorithm can be used in Step 1. The pair of assembly and elimination phases for a region n will be referred to as the *processing* of that region.

This paper describes an efficient parallel implementation of the multifrontal method, whose main feature is a *static, model-driven* approach to work distribution and load balancing among the processing elements. By *static*, we mean that work distribution and communication patterns do not depend on the numerical characteristics of the solution process. This is possible whenever numerical pivoting can be avoided, since a simple *implicit minimum-degree* pivoting strategy ensures the same numerical stability [5]. The benefit of a static strategy is twofold. First, inter-processor communication patterns can be precomputed and optimized; second, heavy preprocessing can be performed to gather data needed to speed up the subsequent numerical computation with negligible memory overhead. Preprocessing time can be amortized over the repeated solutions of systems with the same sparsity structure.

The last few years have seen the emergence of a number of multifrontal solvers (see [2] and references therein). However, only two prominent solvers, MUMPS [1,3] and SuperLU_DIST [8,14], work in parallel on distributed-memory architectures and are capable of solving general unsymmetric linear systems. For this reason, we will compare the performance of our newly developed solver solely with MUMPS, and SuperLU_DIST[2]. MUMPS implements a multifrontal algorithm based on a parallel hierarchical LU decomposition approach similar to the one used in our solver, but it follows a *dynamic* approach to distribute the work between the computing processors. In contrast, SuperLU_DIST starts from the full matrix of the system and partitions it by means of a different technique (based on the identification of *supernodes*).

The results presented in this paper substantiate the claim that the fully static approach adopted by our solver may ensure considerable performance gains over the more complex dynamic solution, provided that simple and effective static load balancing techniques can be afforded by the application domain under consideration.

The rest of the paper is organized as follows. In Section 2 we describe the basic features of a general framework for a parallel multifrontal solver, and introduce the notation used throughout the paper. In Section 3 we provide the details of our parallel multifrontal algorithm. In Section 4 we focus on the model-driven partitioning algorithm devised and present a number of issues related to the AT topology. In Section 5 we compare the performance of our application with MUMPS and SuperLU_DIST on a number of benchmark FE meshes modeling the behavior of porous media. Finally, Section 6 reports some concluding remarks.

[2] In fact, a previous study [2] has proved the superiority of MUMPS over SuperLU_DIST. We decided to retain SuperLU_DIST in our comparison mainly for completeness.

2 A General Framework for a Parallel Multifrontal Solver

In a parallel environment, the work attached to each node of the AT can be speeded up by distributing this work among the available processors. Define a function Π that maps each AT node n into a subset $\Pi_n \stackrel{\text{def}}{=} \Pi(n)$ of available processors that will perform the work attached to that node. Similarly, let Π_n^l and Π_n^r denote the processor sets assigned, respectively, to the left and right son of an internal node n. We say that two arbitrary processors in the same set (either Π_n^l or Π_n^r) are on the *same side*, while a processor in Π_n^l is said to be on the *other side* from a processor in Π_n^r. The pair (Π, AT) defines a *static* processor allocation. In order to carry out the partial LU decomposition associated with node n, the processors in Π_n must first obtain and assemble the Schür complements $\bar{\mathbf{A}}_x$ and $\bar{\mathbf{A}}_y$ previously computed by the processors in Π_n^l and Π_n^r at nodes x and y, children of n. Within this framework, a parallel multifrontal algorithm defined over (Π, AT) must determine for each node n: 1) suitable communication patterns among the processors in the subsets Π_n, Π_n^l, and Π_n^r and 2) how the processors in Π_n cooperate to decompose the newly assembled matrix.

This general framework can be simplified by making some reasonable assumptions on the pair (Π, AT). First of all, it is natural to assume that the number of mesh elements n_e (hence, the number of leaves of the AT) is greater than the number of available computing processors n_p. Therefore, we can find a set of n_p disjoint subtrees that cover all the leaves of the AT and associate each of these subtrees with a single distinct processor. After an initial data distribution, computation on these subtrees can proceed in parallel without any communication involved. We call each of these subtrees a *Private Assembly Tree* (PAT). The computation on each PAT proceeds as in the sequential case amply described in [4]. The (uncovered) subtree of the AT that has the roots of the private subtrees at its leaves is called *Cooperative Assembly Tree* (CAT) since it involves explicit communication between processors. In order to maximize parallelism while limiting the communication volume and enhancing submachine locality, we will consider allocation strategies for which $\Pi_n = \Pi_n^l \cup \Pi_n^r$, for each node n of the CAT.

3 Distributed LU Decomposition Algorithm

Our parallel multifrontal strategy essentially computes the same matrices produced by the sequential algorithm. From here on, these matrices will be referred to as *virtual matrices*. In the parallel algorithm a virtual matrix associated with an internal node n of the CAT is distributed among the processors of Π_n, with each such processor working on a *partial (sub)matrix*. Consider the virtual matrix $\mathbf{A}_n$ of Eq. (1). We partition the rows of $\mathbf{C}_n$ and $\mathbf{N}_n$ into $|\Pi_n|$ subsets denoted as $\mathbf{C}_n^p$ and $\mathbf{N}_n^p$, where $p \in \Pi_n$. The task assigned to processor p is to decompose the partial matrix

$$\mathbf{A}_n^p = \begin{bmatrix} \mathbf{S}_n & \mathbf{R}_n \\ \mathbf{C}_n^p & \mathbf{N}_n^p \end{bmatrix} \tag{2}$$

by means of the four-step sequential algorithm described in Section 1.

Observe that *all* processors in Π_n need the *same* decomposition $\mathbf{S}_n \rightarrow \mathbf{L}_n\mathbf{U}_n$ and the solution to the same triangular system $\bar{\mathbf{U}}_n \leftarrow (\mathbf{L}_n)^{-1}\mathbf{R}_n$ in order to solve different triangular systems $\bar{\mathbf{L}}_n^p \leftarrow \mathbf{C}_n^p(\mathbf{U}_n)^{-1}$ and compute different partial Schür complements $\bar{\mathbf{A}}_n^p \leftarrow \mathbf{N}_n^p - \bar{\mathbf{L}}_n^p\bar{\mathbf{U}}_n$. These partial Schür complements form a row partition of the virtual Schür complement $\bar{\mathbf{A}}_n$ among the processors in Π_n, and will be used in the assembly phase of the father of n. In our solver, we choose to replicate the above common computation in each processor rather then having a single processor gather the relevant data, perform the computation, and then scatter the result to the other processors in Π_n. This master/slave scenario is instead the adopted solution in MUMPS[3].

The way we partition the rows of $\mathbf{C}_n$ and $\mathbf{N}_n$ affects the whole parallel algorithm. We now describe one possible solution that reduces the amount of data exchanged during the assembly phase. Let V_n and V_n^p be the sets of mesh vertices related, respectively, to the rows of the virtual matrix $\mathbf{A}_n$ and the rows of the partial matrix $\mathbf{A}_n^p$ assigned to $p \in \Pi_n$. Clearly, $\bigcup_{p\in\Pi_n} V_n^p = V_n$. Our partitioning makes sure that $V_n^p = V_n \cap V_\rho^p$, where ρ is the root of the PAT assigned to processor p, whence we call V_ρ^p the set of *initial vertices* of p. The main drawback of this simple strategy is that this subset keeps shrinking along the path toward the root of the AT, and will eventually become empty due to rows becoming FS, eventually leaving processor p *potentially idle*: we will address this cause of imbalance in Section 3.2. Note that the vertices related to the columns of partial matrices are the same of the corresponding virtual ones. In fact, the choice of partitioning with respect to the rows (rather than the columns) is totally arbitrary. A totally symmetric column-oriented algorithm can be easily obtained by switching the role of rows and columns.

3.1 The Assembly Phase

In order to build its partial matrix $\mathbf{A}_n^p$, p first has to upgrade the rows of the partial Shür complement computed by p itself in the previous elimination phase, to include the entries of the columns held by processors sharing a subset of its initial vertices. Observe that if more than one processor on the same side has an initial vertex v, then the corresponding rows of their partial Schür complements are the same. As a consequence, during the assembly of $\mathbf{A}_n^p$, processor p can exchange data with at most *one* other processor per vertex, and this processor is on the *other* side from p. No communication within the same side is required during this step. When this assembly step is finished, all processors having v as an initial vertex will have the (same) corresponding row in their partial matrix. This first step of the assembly phase is called the *merging* step.

[3] Note that both approaches afford employing optimized parallel routines for the LU decomposition of S_n at the root of the AT, without the large communication overheads which would be required at internal nodes.

After the merging step, in order to finish the assembly, p must still complete the blocks $\mathbf{S}_n$ and $\mathbf{R}_n$ with the FS rows relative to non-initial vertices. Observe that the missing FS rows can be found somewhere within the same side, since FS vertices are shared by both the left and the right components, and the merging step has already upgraded the corresponding rows. As a consequence, during this step each processor can exchange data with at most *one* processor per missing vertex, and one such processor can always be found on the *same* side. No communication with the other side is needed during this step. We call this second assembly step the *distribution* step, which completes the assembly phase for node n.

3.2 Communication Pattern and Load Balancing

In order to find the communication pattern, we define S_n as the set of *shared vertices* among the processors of the left and right son of node n. As we did for virtual matrices, for each processor $p \in \Pi_n$, we define $S_n^p = S_n \cap V_n^p$ to be the subset of vertices of p which are also shared. What processor p needs to receive during the merging step are the rows whose indices are in set S_n^p and arriving from processors within the other side. Analogously, for the distribution step, we define F_n as the set of vertices that become FS when processing node n of the CAT. Clearly, we have that $F_n \subseteq S_n$. For each $p \in \Pi_n$, we define $F_n^p = F_n \cap V_n^p$ to be the subset of vertices of p which become FS. What processor p needs to receive during the distribution step are the rows whose indices are in the set $F_n \setminus F_n^p$ and arriving from processors within the same side of processor p. Since each processor has multiple potential sources for gathering the data needed for the merging and distribution steps, deciding the optimum communication pattern and the amount of data to gather from each processor involves the solution of a computationally hard problem. To deal with such a problem efficiently, we make the reasonable assumption that the latency in setting up a communication is the real bottleneck. We are then left with minimizing the number of processors that each processor needs to contact, which is an instance of a *Minimum Set Cover* (MSC) problem. The well known greedy strategy for MSC [7] can then be employed to compute a communication pattern whose performance is not too distant from the optimal.

When all the vertices in V_ρ^p become FS, processor p becomes potentially idle. Depending on the shape of the region corresponding to the root of the PAT assigned to processor p, this may happen before the last elimination phase. To avoid a waste of computing power, an idle processor may be assigned a certain number of rows belonging to partial matrices of other processors that are still active. If we limit the possible "donors" of rows to Π_n, the resulting load balancing will feature a high degree of locality easing the parallel forward and backward substitution algorithm [12]. The communications required for balancing can be predetermined during the symbolic analysis phase described in the next subsection. Our approach to balancing is model-driven, in the sense that we use the same cost model described in Section 4 for partitioning the mesh to estimate the load of a processor, and adopt a threshold criterion to maintain a convenient computation/communication ratio.

3.3 Speeding Up the Assembly Phase

As mentioned in the introduction, there is no change in the sparsity pattern of the matrices involved in each of the numerous iterations of the multifrontal solver for a given FE problem. Moreover, our target application domains are such that numerical pivoting can be avoided. Therefore, as already done in the sequential application [4], we can spend time on a preprocessing activity that "simulates" the decomposition process without performing the actual numerical decomposition. During this *symbolic analysis* phase, crucial information regarding the decomposition process are gathered, which can then be used to speed up the subsequent numerical computation. Moreover, in the parallel case, our static work distribution and load balancing strategies make it possible to extend symbolic analysis to encompass the optimization of data exchange between processors during the merging and distribution steps. Specifically, a symbolic representation of a communication pattern is precomputed, so that the matrices can be assembled directly from the receive buffers into their final location so to avoid expensive intermediate buffering and reducing cache-inefficient indirect addressing and conditional branches.

The symbolic data used to speed up the numerical merging step are an extension of the γ-functions used to speed up the assembly phase of the sequential algorithm in [4]. These functions implement inverted references, in the sense that they map the indices of each entry of the *destination* buffers to the indices of the corresponding *source* buffers containing the values contributing to the assembly of that entry.

4 A Model-Driven Partitioning Algorithm

When processing an internal CAT node n, processors in Π_n synchronize their work in two different ways: *external* synchronization is required during the merging step, since processors on opposite sides exchange data, while *internal* synchronization takes place during the distribution step between processors exchanging FS rows within the same side. The load balancing strategy described in Section 3.2 aims at reducing internal synchronization, given that all processors in Π_n will be on the same side during the distribution step associated with the parent of node n. External synchronization time at a node depends on the discrepancy between the running times of the processors computing the left and the right subtree of that node. Balancing the *total* running time on these two subtrees is much trickier than balancing the work on a single node, since the total time depends on the overall topology of the AT and on the map between its leaves and the mesh elements. In order to produce an AT topology capable of yielding an adequate global balancing, we have developed a very simple recursive heuristic that simultaneously partitions the FE mesh into nested rectangular regions and determines the working processors for each region (see Fig. 1 for an example).

The mesh partitioning is driven by the cost function $f(n)$ that models the execution time of the block LU decomposition step performed at a node n of

the AT, depending on the block sizes of the matrices allocated to each processor in Π_n. On a fixed AT, based on $f(n)$ we are able to estimate the computing time $t(n)$ to process its subtree rooted at node n. We synthesize $f(n)$ by least square interpolation from the running times of a suite of sequential block LU benchmarks to be run during the installation of the solver library on a target architecture. When n is a node of a CAT, the function deals only with computation time and not with communication.

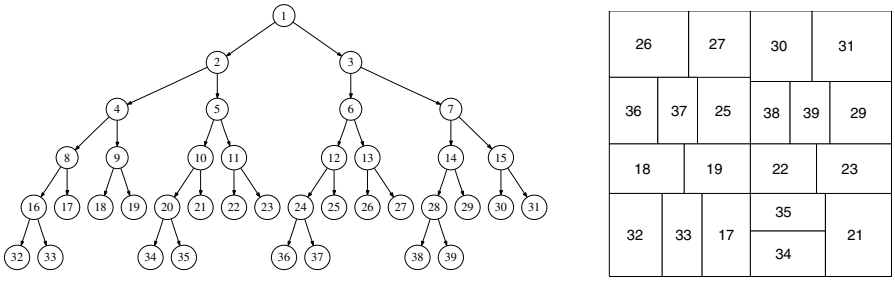

Fig. 1. CAT for 20 processors and the corresponding model-driven partition in 20 regions (one for each leaf) of a square FE mesh. Mesh regions are numbered with the corresponding leaf identifier.

Driven by function $t(n)$, we may determine the region corresponding to each node of the AT by visiting the fixed-shape CAT in a depth-first manner. Starting from the root, which corresponds to the whole mesh, we search for a bisection that minimizes the difference between $t(l)$ and $t(r)$, where $t(l)$ and $t(r)$ are recursively obtained in the same fashion. The resulting partitioning algorithm is exponential in the mesh size but fortunately we can employ simple heuristics to make it affordable. First, we seek nested partitions of rectangular shape, hence the number of possible bisections of a mesh region with m elements is $\Theta(\sqrt{m})$. Second, we limit the exhaustive search activity only at the levels of the CAT whereas we simply decompose each region within a PAT (which will be assigned to single processor) into roughly equally sized subregions.[4] As a result, very few evaluations of the cost function are generally required before finding such a minimum.

After partitioning, we are left with mapping the leaves of the CAT with the processors, trying to enforce as much sub-machine locality as possible. For example, when using a parallel machine made by SMPs with p processors each, every CAT subtree with p leaves should be processed by a single SMP node to avoid slower inter-node communications. To this purpose, our code features a very simple greedy strategy for confining communications between different SMP nodes as close to the root as possible.

[4] This latter simplification affects the quality of the resulting partition minimally, since most of the solver's work is done at the nodes of the CAT.

5 Results

We have compared our solver, dubbed *FEMS (Finite-Element Multifrontal Solver)* against MUMPS v. 4.6 and SuperLU_DIST v. 2.0. In order to quantify the contribution to performance of the model-driven partitioning strategy described in Section 4, we have also set up a modified version of FEMS (called FEMS-M in the following) which uses the METIS package [13] for partitioning. Our target machine is an IBM eServer 575 with 64 Power5 processors working at 1.5 GHz, each capable of a peak performance of 6 Gflop/s. Processors are grouped into 16-processor SMP nodes connected by an IBM High Performance Switch. We used MPI to perform communications among processes, and IBM libraries for sequential and parallel dense linear algebra routines.

Table 1. Test cases main characteristics

Mesh	N. of elements	Matrix order	N. of non-zeros
170×170	28900	436905	33955468
140×140	19600	296805	23028328
400×50	20000	304505	23497308
60×150	9000	137105	10573748

Our test suite comprises rectangular meshes of rectangular elements, modeling scenarios in porous media simulations [6], a computationally challenging application where each finite element has at least 40 degrees of freedom and the involved sparse linear systems are amenable to direct solution without numerical pivoting. For the sake of brevity, we report here the results obtained for four large FE meshes in the suite, whose features are summarized in Table 1. By "large meshes" we mean meshes for which the computation makes an intensive use of main memory. Indeed, as can be seen in Table 2, some execution times are missing due to memory limitations. FEMS, however, is usually able to solve each problem with the smallest number of processors w.r.t. its competitors, which is a clear indication of the fact that the overall memory requirements of FEMS never exceed those of MUMPS and SuperLU_DIST. In fact, the static approach of FEMS allows to compute the amount of memory required by the computation *exactly*, while dynamic approaches generally entail overestimation of memory requirements.

Table 2 summarizes the factorization time of one iteration for various processor configurations, where the lower-order terms due to symbolic analysis, data-distribution, and backward/forward substitution are not included. FEMS and FEMS-M employ implicit minimum-degree ordering, while we let MUMPS automatically choose the fastest pivoting strategy between minimum-degree and METIS ordering, and set up SuperLU_DIST to use minimum-degree ordering on $\mathbf{A}^T + \mathbf{A}$. In order to make a fair comparison, we disabled numerical pivoting in both competitor solvers. Results show that our solver outperforms MUMPS and SuperLU_DIST with both partitioning methods on all test cases and processors

Table 2. Running-time comparison of FEMS, FEMS-M, MUMPS and SuperLU_DIST. Times are in seconds (missing values are due to memory limitations arising when the large test cases are run on a few processors).

		Number of processors								
	Solver	**1**	**4**	**8**	**12**	**16**	**24**	**32**	**48**	**64**
170x170	FEMS	–	12.70	7.14	5.65	4.77	3.99	3.49	3.11	3.01
	FEMS-M	–	16.63	9.20	8.20	6.55	5.38	4.85	4.41	3.98
	MUMPS	–	–	25.21	18.69	18.64	14.79	11.75	8.64	6.68
	SuperLU_DIST	–	–	–	15.29	13.55	21.97	15.12	10.71	11.21
140x140	FEMS	–	7.22	4.15	3.34	2.77	2.38	2.06	1.89	1.80
	FEMS-M	–	9.32	6.89	4.74	4.69	3.82	3.32	3.04	2.84
	MUMPS	–	22.31	15.67	12.24	12.19	10.01	7.21	5.66	4.12
	SuperLU_DIST	–	17.80	11.31	9.57	8.54	8.26	7.60	7.04	7.51
400x50	FEMS	19.18	4.88	2.73	2.03	1.64	1.27	1.06	0.89	0.81
	FEMS-M	–	7.72	5.22	3.85	3.48	3.23	2.51	1.99	1.69
	MUMPS	–	11.82	7.81	5.90	6.86	5.91	4.47	3.84	3.35
	SuperLU_DIST	–	13.86	9.23	8.16	7.45	7.51	6.93	6.59	7.05
60x150	FEMS	8.06	2.08	1.15	0.97	0.77	0.66	0.60	0.53	0.52
	FEMS-M	9.91	3.14	1.71	1.55	1.28	1.16	1.05	0.79	0.78
	MUMPS	16.57	4.78	3.31	2.56	2.87	3.04	2.79	2.70	3.20
	SuperLU_DIST	16.08	6.06	4.01	3.57	3.41	3.26	3.13	2.81	3.18

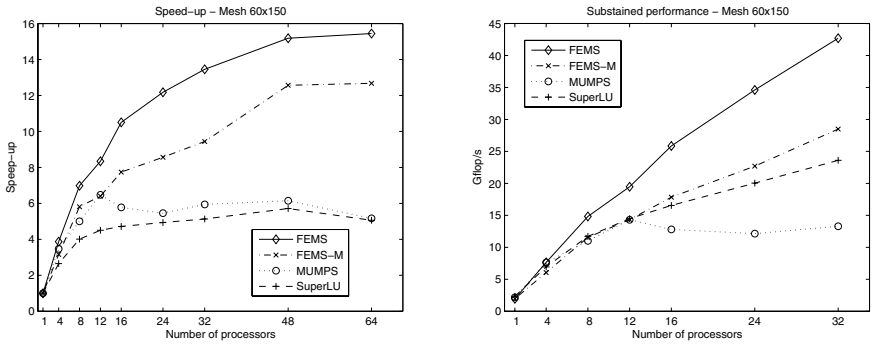

Fig. 2. Parallel performance of the solvers. The graph on the left shows the scalability, whereas that on the right the effective utilization of the computing power. The rate graph is limited to 32 processors due to flop-count limitations of our computing environment.

configurations. The better performance of our solver also in the sequential case shows the benefits of precomputing index functions to speed up the assembly phase. Moreover, it is clear that the model-driven partitioning algorithm substantially improves performance over the use of the general METIS partitioning routines.

Figure 2 gives a better insight into the parallel behavior of the solvers, where we chose to plot the speedup and performance rate graphs relative the 60×150 test case, since the corresponding matrix is small enough to be sequentially factorized by each solver within the available memory. Our solver exhibits considerably higher scalability than the others and makes a better use of the computing power, even if this test case is relatively small. The relatively worse performance of FEMS-M over FEMS mainly depends on a larger external synchronization time since the number of floating-point operations executed by the two versions of our solver is roughly the same.

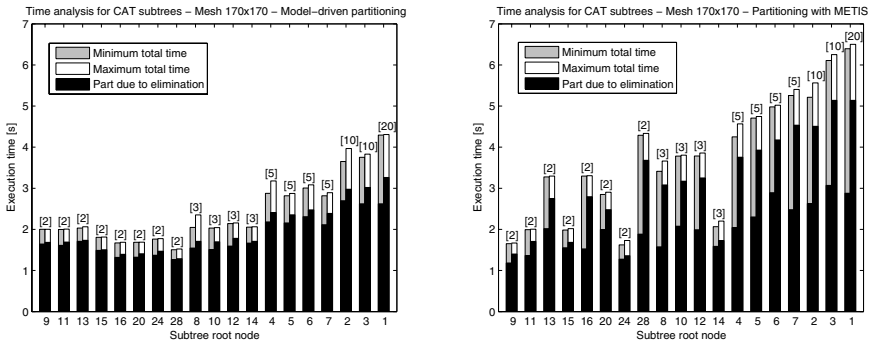

Fig. 3. Differential time analysis of FEMS and FEMS-M on the CAT of Figure 1

A better perspective on the effectiveness of our model-driven partitioning algorithm can be gained by looking at Figure 3 where we compare the running times achieved by FEMS and FEMS-M on each subtree of the CAT shown in Figure 1. Each group of bars represents the total time to compute the subtree rooted at node n (in abscissa), with $|\Pi_n|$ shown above the bars in square brackets; gray and white bars represent, respectively, the minimum and the maximum finishing time of processors in Π_n, with the black portion of each bar representing the fraction of the total time due to the elimination phase. The graph on the left proves that our cost model is very accurate, since the resulting balance of the elimination time is almost perfect. Furthermore, modeling only elimination time seems to be an effective choice also in guaranteeing a good balancing of the overall running time of sibling subtrees. In contrast, observe in the graph on the right that the balancing of the elimination time is much coarser when using METIS, which uniformly partitions the FE mesh into equally sized regions, without considering the distribution of the ensuing computation along the assembly tree.

6 Conclusions and Future Work

We presented a parallel multifrontal linear system solver especially tailored for FE applications, whose main features are a static allocation of work based on the

topology of the FE mesh and a model-driven load balancing technique. Future work will involve the release of a software library providing our FEMS solver. The library will adapt to the computing platform by running a carefully selected suite of microbenchmarks needed to determine the parameters of the cost model used to provide the partitioning. In order to enable to run FEMS also on heterogeneous machines, we are planning to provide the possibility of instantiating the cost model differently on different nodes of the parallel machine. Finally, further research effort will be devoted to the effective application of our model-driven partitioning and static work allocation strategy for unstructured and 3D meshes.

References

1. P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal of Matrix Analysis and Applications*, 23(1):15–41, 2001.
2. P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Trans. Math. Softw.*, 27(4):388–421, 2001.
3. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. Technical Report RR-5404, INRIA, 2004.
4. A. Bertoldo, M. Bianco, and G. Pucci. A fast multifrontal solver for non-linear multi-physics problems. *International Conference on Computational Science*, pages 614–617, 2004.
5. M. Bianco, G. Bilardi, F. Pesavento, G. Pucci, and B. A. Schrefler. An accurate and efficient frontal solver for fully-coupled hygro-thermo-mechanical problems. *International Conference on Computational Science*, 1:733–742, 2002.
6. 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. *International Journal for Numerical Methods in Engineering*, 57(13):1801–1818, 2003.
7. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
8. J.s W. D., S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.
9. J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.
10. J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.
11. I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.
12. A. Gupta and V. Kumar. Parallel algorithms for forward and back substitution in direct solution of sparse linear systems. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 74, 1995.
13. George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*, September 1998.

14. X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.
15. J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Rev.*, 34(1):82–109, 1992.
16. O. C. Zienkiewicz and R. L. Taylor. *The finite element method.* Butterworth-Heinemann, fifth edition, 2000.