FEMS: An Adaptive Finite Element Solver[*]

Alberto Bertoldo

University of Padova
Dept. of Information Engineering
Via Gradenigo 6/B, I-35131, Padova, Italy
cyberto@dei.unipd.it

Abstract

In this paper we investigate how to obtain high-level adaptivity on complex scientific applications such as Finite Element (FE) simulators by building an adaptive version of their computational kernel, which consists of a sparse linear system solver. We present the software architecture of FEMS*, a parallel multifrontal solver for FE applications whose main feature is an install-time training phase where adaptation to the computing platform takes place.* FEMS *relies on a simple model-driven mesh partitioning strategy, which makes it possible to perform efficient static load balancing on both homogeneous and heterogeneous machines.*

1 Introduction

Most of the time and expertise spent in developing today's scientific applications is devoted to address optimization and portability issues. Nevertheless, most of the applications are able to exploit only a small fraction of the available computing power, since a deeper knowledge of the target architecture would be required to get higher performance. The concept of *adaptive software* has become more and more popular as the complexity of computing architectures increased, making hand-tuned codes not a viable solution any more. Moreover, the great development of grid infrastructures in the last few years has been increasing the demand of performance portability and parallel efficiency, which in turn are the main goals of adaptivity. Various adaptive techniques have been recently proposed for the computation of the Discrete Fourier Transform [12, 17] and the solution of basic linear algebra problems [18], which also include some performance modeling [19].

Finite Element (FE) applications are typically used to solve systems of partial differential equations (PDEs) that stem from a mathematical model of some physical phenomena of interest [20]. From a computational point of view, solving an FE problem involves the solution of large sparse linear systems whose sparsity pattern depends on the underlying mesh topology. Since the physical phenomena to be simulated may be non-linear and evolving through time, a given problem may require the solution of many linear systems with the same sparsity pattern but different numerical values. At each iteration, the sparse linear system representing the computational problem can be solved through iterative or direct methods. Direct methods generally produce more accurate solutions than iterative ones, and are generally more stable [11]. On the other hand, they are difficult to parallelize because of the strong data dependencies, and are often memory intensive. In direct methods based upon LU decomposition, the coefficient matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is first decomposed as $\mathbf{A} = \mathbf{LU}$ in $\mathcal{O}(N^3)$ work, where $\mathbf{L}$ is a lower-triangular matrix with unitary diagonal and $\mathbf{U}$ is an upper-triangular matrix. After such decomposition, solving the initial system reduces to the easier problem of solving two triangular systems $\mathbf{Ly} = \mathbf{b}$ and $\mathbf{Ux} = \mathbf{y}$, each taking $\mathcal{O}(N^2)$ work.

The solution strategy studied in this paper is based upon a direct technique called *multifrontal method* [16] and applies to unsymmetric FE problems where *static pivoting* is enough to guarantee numerical stability. This latter constraint is motivated by the fact that numerical pivoting can dramatically reduce the effectiveness of extracting and using structural information to optimize the numerical computation. The multifrontal method makes it possible to turn the problem of an LU decomposition of a huge sparse matrix $\mathbf{A}$ into a sequence of partial LU decompositions of smaller

[*]This work was partially carried out under the HPC-EUROPA project (RII3-CT-2003-506079), with the support of the European Community - Research Infrastructure Action under the FP6 "Structuring the European Research Area" Programme. Further support came from the European Union under the FP6-IST/IP Project *AEOLUS*.

dense matrices. The 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 a subset of rows and columns that will never be further updated by future assembly phases.

The assembly/elimination process can be represented by an *Assembly Tree* (AT) that is a full, but generally not complete, binary tree, whose leaves correspond to elements, internal nodes to mesh regions, and the root to the whole mesh. Therefore, the AT represents a hierarchical partition of the FE mesh into nested regions. In our parallel implementation of the multifrontal method, the computation related to the AT is distributed among the computing processors. We can find a set of disjoint subtrees that covers all the leaves of the AT and associate each of these subtrees to 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 uncovered part of the AT, whose leaves have PAT roots as their children, is called *Cooperative Assembly Tree* (CAT) since it involves explicit communication between processors.

The global matrix $\mathbf{A}_i$ to be partially decomposed at the assembly/elimination step i and the local (sub)matrix $\mathbf{A}_i^p$ assigned to a processor p can be arranged into the following block form:

$$\mathbf{A}_i = \begin{bmatrix} \mathbf{S}_i & \mathbf{R}_i \\ \mathbf{C}_i & \mathbf{N}_i \end{bmatrix} \qquad \mathbf{A}_i^p = \begin{bmatrix} \mathbf{S}_i^p & \mathbf{R}_i^p \\ \mathbf{C}_i^p & \mathbf{N}_i^p \end{bmatrix} \tag{1}$$

where $\mathbf{S}_i^p$ and $\mathbf{R}_i^p$ are obtained as a row partition of $\mathbf{S}_i$ and $\mathbf{R}_i$. Similarly, $\mathbf{C}_i^p$ and $\mathbf{N}_i^p$ are obtained as a row partition of $\mathbf{C}_i$ and $\mathbf{N}_i$. Then, the elimination algorithm consists of computing the following four steps:

1. $\mathbf{L}_i\mathbf{U}_i \leftarrow \mathbf{S}_i$ [complete LU decomposition of $\mathbf{S}_i$]
2. $\bar{\mathbf{U}}_i \leftarrow (\mathbf{L}_i)^{-1}\mathbf{R}_i$ [lower-triangular system solution with multiple right-hand sides]
3. $\bar{\mathbf{L}}_i^p \leftarrow \mathbf{C}_i^p(\mathbf{U}_i)^{-1}$ [upper-triangular system solution with multiple right-hand sides]
4. $\bar{\mathbf{A}}_i^p \leftarrow \mathbf{N}_i^p - \bar{\mathbf{L}}_i^p\bar{\mathbf{U}}_i$ [Schür complement w.r.t. $\mathbf{S}_i$]

where the partial factors $\mathbf{L}_i$, $\bar{\mathbf{L}}_i^p$, $\mathbf{U}_i$, and $\bar{\mathbf{U}}_i^p$ can be stored elsewhere in view of the final forward and backward substitution activities needed to obtain the final solution. The local Schür complements $\bar{\mathbf{A}}_i^p$ will contribute to the multifrontal step related to the parent node. In the step related to the root, only the block $\mathbf{S}_i$ is non-zero and the elimination algorithm only consists of Step 1. In order to respect data dependencies, specific parallel algorithms for the first two steps are required, which can consider different communication patterns and data replication schemes. A detailed description of our approach to the parallel multifrontal method can be found in [4, 6].

Table 1. Features of computing platforms used for testing. *MareNostrum* **(MN) and** *Kadesh* **are sited at BSC (Spain),** *CLX* **at Cineca (Italy), and** *Opter1* **at the University of Padova (Italy).**

	Computing platforms				
Features	**MN**	**Kadesh**		**CLX**	**Opter1**
Vendor	IBM	IBM	IBM	Intel	AMD
Proc.	PPC970	Power3	Power4	Xeon P4	Opteron
Clock	2.2 GHz	375 MHz	1.0 GHz	3.06 GHz	2.2 GHz
L1 size	32 KB	64 KB	32 KB	8 KB	64 KB
L2 size	1 MB	8 MB	1.5 MB	512 KB	1 MB
L3 size	–	–	32 MB	–	–
Mem.	4 GB	4 GB	4 GB	2 GB	8 GB
SMP size	2	16	4	2	2
N. proc.	10240	128	36	768	2
Network	Myrinet	IBM SP Switch2		Myrinet	–
BLAS	ESSL	ESSL		MKL	ATLAS

In this paper we present the software architecture of an adaptive tool called `FEMS` (Finite Element Multifrontal Solver) to solve sparse linear systems coming from the solution of FE problems on both homogeneous and heterogeneous clusters of SMPs, which cover almost 80% of the current parallel architectures in the Top500 list [9]. Some of the sequential and parallel machines used for testing the package are summarized in Table 1. Our main goal is to provide FE application developers with a parallel library that automatically adapts to the target computing architecture and to the input data. The first type of adaptation is performed during the installation of the library by means of a training phase, whereas the second type of adaptation is performed at execution time during a *symbolic analysis* phase aimed at speeding up the subsequent numerical computation. While the first adaptation is done only once for a given computing architecture, the second one must be done whenever the user specifies a new FE problem to be solved. Nevertheless, the computing cost associated to such adaptation is usually affordable, since FE applications generally need to obtain the numerical solution of many linear systems sharing the same structure.

The rest of the paper is organized as follows. In Section 2 we propose an overview of the software architecture of `FEMS` and a brief description of its main components. Section 3 presents a more detailed description of the mesh partitioner, which is the main component of the static load-balancing strategy we devised. Finally, in Section 4 we draw some conclusions on the work done and offer an overview of future research in this area.

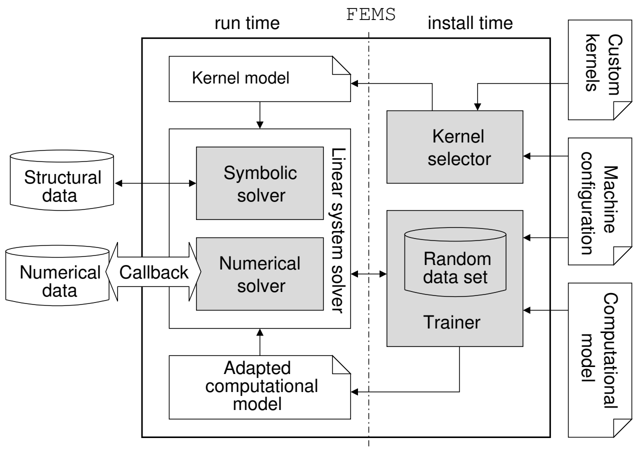

Figure 1. `FEMS` **software architecture.**

2 Software architecture

An overview of the software architecture of `FEMS` is given in Figure 1. At installation time, users can specify some information describing the computing architecture, which includes a hierarchical description of the parallel architecture where the FE application will run, called the *Machine Tree* (MT). Each node of the MT corresponds to a submachine whose processing elements are the leaves of the subtree rooted at that node, and each edge corresponds to a communication subnetwork[1]. Nodes and edges of the MT come along with a set of parameters describing the capabilities of the corresponding submachine and subnetwork. In the current version of the library, users can describe the memory hierarchy of each processing element by specifying the number of memory levels and their sizes. Note that this description is powerful enough to describe many parallel computing architectures, including clusters of Chip Multi Processors (CMPs) that are about to dominate both the high-end and the low-end computing markets. Moreover, the descriptions of the computing capabilities can be easily enriched as soon as the employed performance models become more complex than the simple ones already implemented. For example, *Kadesh* is a heterogeneous cluster of SMPs which can be represented by the MT in Figure 2. The machine has 8 nodes (S_0 to S_7) with 16 IBM Power3 processors each, sharing the same network N with another set of 9 nodes (S_8 to S_{16}) each featuring 4 IBM Power4 processors.

After providing such a description of the computing architecture, users can execute the *kernel selector*, whose task is to benchmark various built-in implementations of sequential and parallel algorithms for the LU decomposition of dense matrices, as well as user-defined routines, to pick up the best performing one. This tool also finds which are the implementation parameters, such as block sizes, that better adapt the computational kernels to cache and network features. This testing process is performed once for each target computing machine at installation time, and it usually takes up to few minutes, depending on the hardware speed and memory size. The benchmark results form a *kernel model* used by the solver to select the best performing kernel for each elimination step, in a way similar to what `ATLAS` and `FFTW` do [18, 12]. The purpose of the *trainer* is to monitor the execution of the solver running on suitable training data sets. The collected data is used to adapt both sequential and parallel computational models that describe the execution time of the solver running on the target machines, and that will be used at run time to perform static load balancing. The trainer is usually executed once at installation time, but it can also used at run time to perform a finer tuning for sufficiently long running FE applications.

At run time, the *symbolic solver* uses the structural data defining the given FE problem, coupled with the architecture-dependent information obtained at installation time, to compute an efficient mesh partitioning for load balancing purposes, and to optimize communication patterns. The numerical computation is then simulated on symbolic data to extract information used to speed up the next numerical iterations, as described in [4, 5]. This is done *once* for each FE problem and its execution time is generally less than the time required for one numerical iteration. On the other hand, the *numerical solver* is called as many times as required by the FE application. At each iteration of the solution process, the FE application provides the solver with the required input data through a user-defined callback function which returns a partial linear system for each required mesh element. Since each processor gets only the numerical data it needs during the solution process, the input does not need to be precomputed, thus reducing the total memory requirements, and avoiding costly data distribution and computational bottlenecks. The main components of this software architecture are described with greater detail in the following sections. For a deeper insight into the devised algorithms and experimental results, please refer to [4].

2.1 Kernel selector

An LU decomposition [14] can be visualized as a three nested-loop algorithm on i, j, and k whose inner basic operation is the following update:

$$a_{ij} \leftarrow a_{ij} - \frac{a_{ik} \cdot a_{kj}}{a_{kk}}$$

By changing the loop order, six different algorithms may be obtained, which still maintain the same computational com-

[1]We assume that the edges connecting siblings to their common father are of the same type, i.e. the components of a submachine share the same communication network.

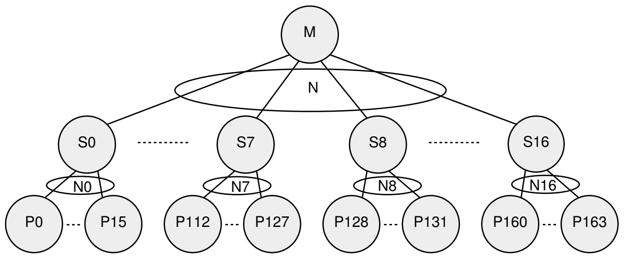

Figure 2. A simple Machine Tree.

plexity, but change the memory access pattern. When executed on computing architectures with a deep hierarchical memory, these implementations may have different performance [8]. Out of the six versions obtained by permuting the loop order, we focus on those that are column-oriented, since our solver is written in Fortran. We identify these versions through the respective loop order: `kji`, `jki`, and `jik`.

We implemented various versions of these algorithms that perform LU decomposition of dense square matrices without numerical pivoting. The first implementation does not make use of any optimized library routine such as the `BLAS` [10]. The computation performed by these scalar versions of the LU decomposition algorithm can be made into vector computations, thus making it possible to employ vector routines such as `BLAS` Level 2. The next step is to group together the vector computations into blocks to exploit `BLAS` Level 3 routines. Doing so, the outer loop becomes a block loop for a given block size. At each iteration of such loop, we have to solve the (constant size) LU decomposition of the diagonal block. All the previous scalar and vector algorithms can be used to solve each subproblem, and also one of the blocked algorithms with a smaller block size can be employed in a recursive fashion, thus obtaining a sort of *cache oblivious* algorithm [13]. An approach similar to the one adopted in `FFTW` [12] could be also devised, where the leaves of the recursion tree (corresponding to the *codelets* in the `FFTW` lingo) are small pieces of automatically generated straight-line code. Moreover, users can define custom kernels for the LU decomposition of dense matrices without numerical pivoting, by providing the source code at installation time. These kernels will be compiled and then included in the benchmark process as well as the built-in kernels provided by the `FEMS` package. This makes it possible to include routines from other libraries, such as those provided by `LAPACK`, whenever they become available on the computing platform, by just writing the required wrappers[2].

We have devised specific parallel benchmarks to investigate the performance of such implementations. For each memory level of the current processor, the kernel selector benchmarks all registered kernels solving randomly generated problems that fit in that level. If the best performing kernel is a block algorithm, it also searches for the best block size. The performance is evaluated by measuring the running time by means of high-precision cycle counters, like the ones used in `FFTW`. If such counters are not accessible, standard timing routines are used, which however guarantee a much lower accuracy. At the end of this testing process, the identifiers of the best performing kernels (along with their implementation parameters) for each memory level of each processing element constitute the kernel model. The data computed by the kernel selector running on some of the tested computing platforms is illustrated in Table 2. Block algorithms call the best kernel for the L1 level, whereas recursive algorithms call the best kernel for the previous smaller level in a recursive fashion. We also find the best base kernel (among scalar and vector kernels) which is used in the block kernels for L1. Note that vector kernels using `BLAS` Level 2 perform better than the corresponding scalar ones, probably due to a better usage of the processor registers, whereas block and recursive kernels generally perform better in the highest levels of the memory hierarchy [4]. As expected, the chosen block size is related to the memory level size.

[2]Note that `LAPACK` only features routines for LU decomposition with numerical pivoting, which is not supported by `FEMS`. `LAPACK` LU without pivoting essentially corresponds to our block `kji` implementation.

Table 2. Kernel model for some tested architectures. The tested kernels are the three loop variants for the (s)calar, (v)ector, (b)lock, and (r)ecursive algorithms. Values in brackets are block sizes.

Level	Computing platforms			
	Mare Nostrum	**Kadesh** (Power4)	**CLX**	**Opter1**
Base	`jki-v`	`jki-v`	`kji-v`	`jik-v`
L1	`jki-v`	`kji-b` (20)	`kji-v`	`kji-b` (10)
L2	`jki-v`	`kji-b` (60)	`kji-b` (30)	`kji-b` (50)
L3	–	`kji-r` (290)	–	–
$>$ **L3**	`jki-b` (62)	`kji-b` (60)	`kji-b` (30)	`kji-r` (300)

2.2 Trainer

The trainer first monitors the solver to collect some parameters describing each step of the solution process and the corresponding computing time. Then it uses this information to adapt a given computational model to the target architecture. In turn, the model is at the core of the load-balancing strategy. Users can choose to train the solver on randomly-generated FE problems or to provide their own data sets. In the first case, users can specify the problem size and the trainer automatically generates a suitable FE mesh and random partial linear systems corresponding to its elements. Then, it simulates a parallel FE application that calls `FEMS` to solve the related computational problem. In the second case, the solver can be trained through a real FE application, thus increasing the model accuracy by taking into account also the computing time of the callback function, which is heavily application-dependent. The training process can be executed in parallel on the whole computing machine or on a smaller submachine: the training on PAT nodes adapts a sequential computational model for the related processor type, whereas the training on CAT nodes adapts a parallel model. This also makes it possible to use `FEMS` in heterogeneous parallel architectures.

Let m be the number of assembly/elimination steps involved in the solution of the training FE problem, which in turn depends on the number of mesh elements. Let $\mathbf{p}(i)$ be the vector of $s \geq 1$ parameters that characterize step i, which have been fixed since they depend on the parallel multifrontal algorithm. The values of such parameters can be set up in the symbolic solver since they depend only on structural data. During the numerical solution of the training problem, the solver measures the execution time $t(i)$ of each assembly/elimination step i by means of the same cycle counters used by the kernel selector. The collected data is then given back to the trainer to be processed. Finally, let $\mathbf{f}(\mathbf{p}) \in I\!R^n$, $n \geq 1$ be the functions that make up the computational model of the solver. Such set of functions is provided by `FEMS`, but can be easily modified by the user though a registration mechanism similar to the one adopted to manage the LU kernels.

We want to compute a set of architecture-dependent model parameters $\mathbf{m} \in I\!R^n$ such that $\sum_{j=1}^{n} m_j f_j(\mathbf{p}(i))$ gives a good estimation of $t(i)$. Since $m \gg n$ this is a least-square problem. The trainer builds the matrix $\mathbf{M} = [f_j(\mathbf{p}(i))]_{ij} \in I\!R^{m\times n}$ whose rows correspond to decomposition steps and whose columns are obtained evaluating the model functions w.r.t. the step parameters. Let $\mathbf{t} \in I\!R^m$ be the vector of the computing times of the m steps. The model parameters are found by solving the least-square problem

$$\min_{\mathbf{m}\in I\!R^n} \|\mathbf{Mm} - \mathbf{t}\| \tag{2}$$

by means of SVD or equivalent methods [14].

In the current version of `FEMS`, where the first two steps of the elimination algorithm at node i are replicated on the whole blocks $\mathbf{S}_i$ and $\mathbf{R}_i$ [6], we included a very simple unified computational model given by the following functions:

$$\begin{aligned}
f_1(\mathbf{p}) &= p_1^3 \quad \text{[Elimination Step 1]}\\
f_2(\mathbf{p}) &= p_1^2 p_2 + p_1^2 p_3 \quad \text{[Elimination Step 2 and 3]}\\
f_3(\mathbf{p}) &= p_1 p_2 p_3 \quad \text{[Elimination Step 4]}\\
f_4(\mathbf{p}) &= (p_1 + p_2)(p_1 + p_3) \quad \text{[Assembly]}\\
f_5(\mathbf{p}) &= p_4 \quad \text{[Overhead]}
\end{aligned}$$

Table 3. Sequential model parameters computed by the trainer for *MareNostrum*.

Reg. type	**Elem.**	**Comp.**	
Mem. lev.	L1	L1	L2
m_1	0.24E-08	0.46E-10	0.47E-09
m_2	0.75E-08	0.14E-09	0.14E-09
m_3	0.41E-08	0.63E-09	0.43E-09
m_4	0.39E-09	0.43E-07	0.85E-07
m_5	0.64E-04	0.24E-04	-0.81E-03

where the step parameters are related to the size of the matrix blocks shown in (1), i.e. $p_1(i) = \mathrm{rows}(\mathbf{S}_i)$, $p_2(i) = \mathrm{rows}(\mathbf{C}_i^p)$, $p_3(i) = \mathrm{columns}(\mathbf{R}_i)$, and $p_4(i)$ is the number of processors computing node i. During the training process, the symbolic solver collects the values of such parameters, which depend only on how the mesh is recursively partitioned, and the numerical solver collects the running time $t(i)$. Note that these functions also give a sequential model for the assembly/elimination steps, since on PAT nodes $p_2(i) = p_3(i)$ and $p_4(i) = 1$.

The trainer evaluates such functions on the values of the parameters $\mathbf{p}$ collected by the solver and builds the matrix $\mathbf{M}$. Data points, corresponding to the rows of $\mathbf{M}$, are divided in three groups: one for the assembly/elimination steps of the mesh elements (PAT leaves), one for composite regions at PAT internal nodes, and one for composite regions at CAT nodes. Moreover, in order to reduce the fitting error, data points are filtered depending on the levels of the memory hierarchy specified by the user: for each memory level, we solve a different least-square problem like (2) obtained by considering only those decomposition steps whose matrix blocks fit in that level, but not in the previous smaller one (if any). The solution of each least-square problem gives the values of the model parameters $\mathbf{m}$, which will be used to estimate the computing time as described in Section 3. Table 3 shows the model parameters found by the trainer for one of the tested machines when using a random data set.

In order to appreciate the effectiveness of the adapted computational models, we report in Table 4 the average fitting and validation errors for some of the tested computing platforms, computed as:

$$\text{Error} = \frac{1}{m}\sum_{i=1}^{m} \frac{|\sum_{j=1}^{n} m_j f_j(\mathbf{p}(i)) - t(i)|}{t(i)} \tag{3}$$

We have validated the model using a different data set with respect to the one used to fit the training data. Note that both types of error are below 10% for most of the adapted models, and are generally higher on machines running Linux due to the noise introduced by the operating system. The fitting error can be used to decide whether repeating the training phase or adjusting each memory level threshold.

Table 4. Fitting and validation errors for some adapted computational models.

Platform	Region Type	Memory level	Fitting error	Validation error
MareNostrum	**Elem.**	L1	16%	25%
	Comp.	L1	4.0%	5.3%
Linux		L2	1.6%	4.3%
Kadesh	**Elem.**	L1	2.7%	2.9%
(Power3)	**Comp.**	L1	3.1%	4.2%
AIX		L2	2.1%	1.0%
CLX	**Elem.**	L1	2.6%	3.8%
	Comp.	L1	3.9%	7.6%
Linux		L2	1.7%	3.9%
Opter1	**Elem.**	L1	4.3%	11.0%
	Comp.	L1	7.7%	7.7%
Linux		L2	0.9%	1.9%

2.3 Symbolic solver

The symbolic solver executes the symbolic analysis of the FE problem, whose task is to prepare optimized data structures supporting numerical system solution [5, 6]. Out of the many parts composing the symbolic solver, we describe here only those which follow an adaptive approach. The mesh partitioner is also executed in the symbolic analysis phase, but it will be described in grater detail in Section 3.

MT-to-AT mapping. Recall from Section 1 that the CAT is a binary tree with $\frac{N_P}{2}$ leaves, where each node is associated with a subset of the N_P computing processors. The topology of the CAT determines how the computing processors interact to carry out the assembly/elimination steps related to its nodes, so it is crucial to exploit submachine locality, in order to reduce communication overheads. This goal can be achieved by embedding the CAT into the MT describing the parallel architecture.

The FE application may be executed on a submachine of the target architecture. On clusters of SMPs, users generally choose the number of processors they want to employ for the execution. Then the solver must first determine the run-time Sub-Machine Tree (SMT) as the minimal connected subgraph that includes such processing elements. Let i be an internal SMT node and C_i be the set of its children, which are interconnected by the network N_i. Since the SMT is generally not binary, the subgraph induced by such nodes has to be mapped into a binary tree whose root corresponds to i and leaves to the nodes in C_i. This binary tree becomes part of the final CAT, whose nodes will be computed by processing elements that communicate through the network N_i. By applying this substitution process on all the internal SMT nodes, we completely define the topology of the CAT. There are many ways of turning SMT nodes into CAT subgraphs. One simple way is to create a balanced binary tree by equally dividing the submachines between the children, whereas more complex solutions should also try to balance the number of SMT leaves.

AT-to-KERNEL mapping. Once the AT and the related mesh partitioning have been computed (see Section 3), the symbolic solver associates each AT node to the best LU decomposition kernel for that elimination step, by using the kernel model found by the kernel selector (see Section 2.1).

Each computing processor loads the kernel model corresponding to its type, which gives the best kernel implementation for each of its memory levels. Note that the size of the matrix blocks involved in the elimination phase (see Section 1) depends only on how the mesh has been recursively partitioned. Therefore, for each node of its PAT and for the CAT nodes it contributes to process, each processor easily estimates the smallest memory level that contains the matrix blocks involved in that elimination step by using a simple memory model based on a direct mapping mechanism, which in turn gives the best kernel along with the implementation parameters. At run time, the numerical solver automatically calls such routines on the required numerical data with negligible overhead.

3 Parallel model-driven mesh partitioner

The computational model adapted by the trainer is used to obtain an efficient static load balancing strategy. By static we mean that work distribution and communication patterns do not depend on numerical data and can be determined in the symbolic analysis phase based only on structural information. Indeed, using *implicit minimum degree* as pivoting method [7] implies that the step parameters $\mathbf{p}$ depend only on how the FE mesh is recursively partitioned into nested regions. Therefore, we look for the partition that produces the best load balancing among the processors, thus reducing the time wasted on synchronization.

We developed a simple partitioner that works on rectangular meshes made of rectangular elements and recursively partitions it in rectangular regions[3]. Each region is divided into two parts along the longest dimension to maximize the aspect ratio in terms of number of elements, since the amount of work to perform the elimination phase is related to the size of the region boundary. For the sake of simplicity, regions in charge of a single processor are split in the middle of the longest dimension and the model is used only to estimate the related computing time. For regions in charge of more than one processor, the bisection point guaranteeing the best balancing is obtained via a local search

[3]A similar model-driven approach could be adopted to partition general FE meshes and could be extended to three-dimensional meshes.

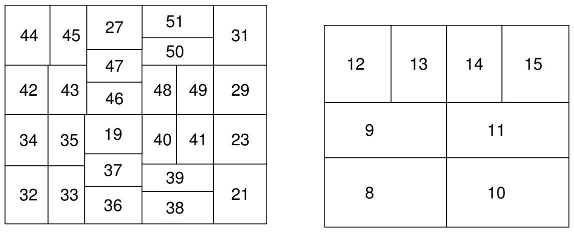

Figure 3. Model-driven partitions of a rectangular mesh obtained in homogeneous (left) and heterogeneous (right) environments.

around the point in the middle of the longest dimension. The objective is to minimize the difference between the estimated computing time related to the two subregions that 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 [6].

During the symbolic analysis phase, each processor loads the computational models previously adapted to its type by the trainer, and uses them to estimate the computing time of each multifrontal step it will have to compute. Let us first consider the partitioning step related to an internal PAT node i. The values of the step parameters $p_1(i) = \mathrm{order}(\mathbf{S}_i)$ and $p_2(i) = p_3(i) = \mathrm{order}(\mathbf{N}_i)$ depend only on the shape of the related mesh region and on how it is partitioned into two subregions. Given a bisection of the region, we estimate the computing time as

$$\hat{t}(i) = \sum_{j=1}^{5} m_j f_j(\mathbf{p}(i)) \tag{4}$$

where the right adapted model (determined by $\mathbf{m}$) is chosen by finding the smallest memory level that contains the required matrix blocks. The estimated computing time of the subtree rooted at i is obtained by adding $\hat{t}(i)$ to the estimated time needed to compute the left and the right subtrees, which are obtained in the same way in a recursive fashion. Let us now consider the partitioning step related to a CAT node i. The partitioner searches for a partition of the related mesh region that minimizes the estimated computing time of the subtree rooted at i, which is obtained adding $\hat{t}(i)$ to the *maximum* between the estimated time needed to compute the left and the right subtrees.

Starting from the AT root, each processor cooperates with the other processors to find the best partitions on CAT nodes. Note that the CAT topology is determined by the MT-to-AT mapping before partitioning the mesh, whereas each processor determines the topology of its own PAT while partitioning the mesh. This parallel approach to mesh partitioning makes it possible to embrace the case of hetero-

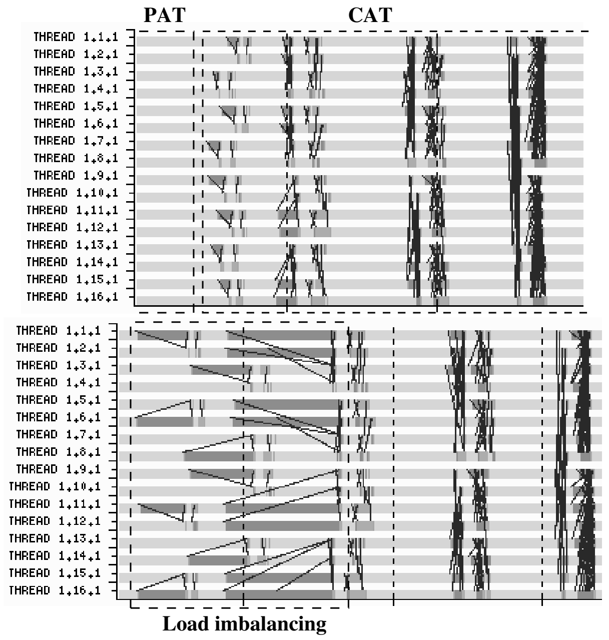

Figure 4. Trace of one iteration using both the adapted (top) and non-adapted (bottom) computational models to partition the mesh.

geneous architectures and considerably reduces the search time to find an optimal partition with respect to balancing.

The results of such model-driven mesh partitioning can be appreciated by looking at Figure 3, which shows the mesh regions associated to the PAT root of each processor when solving a real FE problem. The partition on the left refers to 26 (homogeneous) processors in *MareNostrum*. Note that when trying to balance the work done by each processor, the regions close to the mesh border are generally wider because they involve less computation, even if in this case the adapted computational model is the same for every processor. The difference can be more relevant when increasing the mesh size and the number of processors. The partition on the right has been obtained by using 4 Power3 processors of node S_0 and 4 Power4 processors of node S_8 in *Kadesh*. Regions from 12 to 15 will be computed by slower processors, so that they are quite smaller than regions from 8 to 11. Moreover, the same effect on the border regions can be observed within homogeneous groups.

Figure 4 shows a portion of parallel traces obtained with PARAVER [1] on 16 Power3 processors in *Kadesh*. They represent one iteration of the numerical decomposition process related to random FE problems, using the adapted and the non-adapted computational models. The execution time is in abscissa with the same scale for both traces, whereas the computing processes are in ordinate. Light gray regions correspond to computation, dark gray regions to waiting time due to synchronization, and the black lines are the

MPI communications. The two dashed rectangles in the top trace show part of the computation in each PAT and in the CAT, with the model-driven partitioner using the adapted computational model found by the trainer. Note that the computation proceeds synchronously and the waiting time is very low without involving any additional data redistribution. On the other hand, when the model-driven partitioner does not use the model parameters to estimate the computing time, load imbalance causes longer waiting times, as shown in the bottom trace; in this case, the last two steps are still balanced due to mesh symmetries. Note that, when using a general graph-partitioning tool like METIS [15], the balancing worsens even more, as already proved in [6]. The use of adapted computational models becomes crucial when increasing the number of processors and in heterogeneous environments.

4 Conclusions and future work

We investigated how to provide developers of FE applications with an efficient parallel direct solver which can be automatically tuned to the target computing architecture and to the input problems. We specifically address those FE applications that do not require numerical pivoting to get the desired numerical stability and accuracy and that need to perform many iterations of the solution process for each given FE problem. Our target architectures are high-performance clusters of homogeneous or heterogeneous SMP nodes. We developed a new software architecture called `FEMS` where adaptation is performed both at installation and run time. Using a simple description of the computing platform provided by the user, we automatically select the best elimination kernels and adapt a model of the computation by means of a training phase, both performed at installation time. Such model is used at run time in a preprocessing phase to achieve good static load balancing that reduces synchronization and communication volume involved in the numerical solution process.

The work presented in this paper lays the foundations of future research. Even if the computational model is very accurate in estimating sequential computing time, it actually does not take into account the communication cost in the cooperative phase of the algorithm. A better model should embody network bandwidth/latency parameters that reflect how the processors are grouped together into SMP nodes to improve load balancing, especially when employing parallel kernels described in [4]. Finally, an extensive testing activity is required to compare our adaptive approach with other solvers based on dynamic scheduling [2, 3], which generally require more communication. Scalability and memory requirements need also be explored when using FE applications on problems featuring linear systems with a number of unknowns ranging from 10^5 to 10^7.

References

[1] Details at `http://www.cepba.upc.es/paraver`.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. of Matrix Anal. and Appl.*, 23(1):15–41, 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. *Adaptive Finite Element Applications*. PhD thesis, University of Padova, Italy, 2006.

[5] A. Bertoldo, M. Bianco, and G. Pucci. A fast multifrontal solver for non-linear multi-physics problems. In *Proc. of ICCS*, pages 614–617, 2004.

[6] A. Bertoldo, M. Bianco, and G. Pucci. A static parallel multifrontal solver for finite element meshes. In *Proc. of ISPA*, pages 734–746, 2006.

[7] 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. In *Proc. of ICCS*, pages 733–742, 2002.

[8] J. Dongarra, F. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, 1984.

[9] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-performance computing: Clusters, constellations, mpps, and future directions. *Computing in Science and Engineering.*, 7(2):51–59, 2005.

[10] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Trans. on Math. Soft.*, 16(1):1–17, Mar. 1990.

[11] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, Philadelphia, PA, USA, 1998.

[12] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[14] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[15] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proc. of Supercomputing*, pages 1–13. IEEE Computer Society, 1998.

[16] J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Rev.*, 34(1):82–109, 1992.

[17] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. of the IEEE*, 93(2):232–275, 2005.

[18] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *9th SIAM Conf. on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[19] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS*, pages 141–150, 2005.

[20] O. C. Zienkiewicz and R. L. Taylor. *The finite element method*. Butterworth-Heinemann, fifth edition, 2000.