Interfacing a commercial Finite Element Code to HP's MLIB running on a V-Class System

A. J. Svobodnik, E. Schnabler

Numerical Analysis and Design GmbH & Co KG, Landskrongasse 5, A-1010 Wien, Austria
email: {as,es}@NADwork.at
phone: +43.1.533.53.06-0

Abstract

Today in many engineering applications one of the key issues is to solve large systems of linear equations using low memory and time resources. Typical industrial applications need to solve up to 1 or 2 million of equations. One of the most common tools are finite element codes as widely used in structural mechanics. This paper will show the SW-strategy to interface an existing commercial finite element code to HP's MLIB (especially VECLIB) to speed up the process of solving the equations. Furthermore, the incredible increase in performance will be shown for various typical applications compared to traditional (hand-coded) solving strategies. A comparison for different numbers of processors will be presented to show the effect of parallelization.

1 Introduction

There is a long tradition in structural mechanics to solve practical problems via the use of finite element codes. Primary this method was developed to solve for deformations and stresses of structures due to mechanical loads as forces or thermal stresses within a linear theory. Today there exist many more fields of applications which go far beyond the original research in former times. In the last two decades, there has been much research and progress to solve for nonlinear problems like forming or crash simulations, dynamic problems in the frequency and time domain, field problems like thermal distribution or electromagnetic waves, just to name a few applications. Even new fields, traditionally not mentioned within the context of structural mechanics, have been developed in the past, like the simulation of acoustic sound propagation, which is called Computational Acoustics. Beside the classical finite element method there is also a wide use of boundary element methods, especially in acoustics for external problems, and other approximation methods.

Since 1990 we are working on the (further) development of a general purpose analysis code for structural mechanics entitled *NAD*work, including comprehensive pre- and postprocessing facilities and various interfaces to CAD- and CAE-systems. The primary goal of this code was the solution of linear and nonlinear, static and dynamic problems using the finite element method, especially in the area of mechanical engineering. Today, there exist beside the standard version of this code two special versions. One is intended for use in civil engineering applications for reinforced concrete structures and steel structures (*NAD*bau), and the other for computational acoustics (*NAD*work/Acoustics), where we use finite element as well as boundary element methods to solve for the distribution of sound fields.

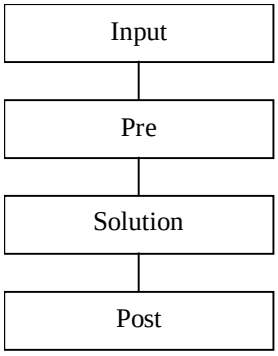
In the beginning of 1997 we started a research project entitled *NAD*work/HPC with the primary goal to adapt our existing software products for the use in a high performance computing environment, especially for multiprocessor computers. This effort was mainly driven by two fields of applications of our software. One is the simulation of forming processes and the other is the calculation of sound radiation of vibrating structures. In the first one, the long computation times are due to the highly nonlinearities like contact and plasticity at finite strains. In the latter one, one has to solve for a coupled fluid-structure problem in the frequency domain for large problem sizes. In a first step of this class of application the structural eigenfrequencies are calculated in the frequency domain of interest. In typical applications of vehicle industry there arise systems with several hundreds of thousands of unknowns and several hundreds (up to 2,000) eigenvalues. In further steps the motion of the body due to some excitement is calculated, and finally the propagation of sound waves in the surrounding fluid due to the motion of the structure are calculated. For the fluid (sound propagation) we get systems of symmetric, complex and fully populated (BEM) or sparse matrices (FEM). In the case of BEM we typically get up to 20,000 of unknowns, and in the case of FEM we get up to 1,000,000 of unknowns. Thus the term large scale analysis seems to be valid.

In a first phase of the research project, our focus was on the solution phase of linear systems. Our target was to implement a software interface for our existing code to various solver libraries. On the one hand the effort was to make an interface to proprietary solvers from hardware vendors, like HP's MLIB, which usually use direct solvers, whereas on the other hand we started the developing of iterative solver methods. This paper will show the strategy to implement such an interface in our existing code, and to show the increase in performance (for the solution phase of a linear system of equations) for some typical applications.

All work was done on a HP V2200, running under HP-UX 11.0, equipped with 8 processors, 2GB of memory and several GB's of harddisk space on an AutoRAID system. The language used in our software development is pure ANSI C.

2 The SW-Strategy

Typically, a structural mechanics code based on FEM or BEM consists of the following program phases:



During the *Input* phase the data describing the problem (nodes, elements, materials, forces, etc.) are being read. In the *Pre* phase the data structure of the global system matrix is being evaluated, which defines the storage scheme of the system of equations. Furthermore, the local stiffness matrices of the elements are calculated and assembled into the global system matrix.. In the *Solution* phase the system of equations is being solved for one or more right-hand-side vectors. In this phase typically deformations (displacements and rotations) are being calculated. And finally in the *Post* phase the ultimate results like strains or stresses are being carried out. In this paper we concentrate on the *Solution* phase, and will show how the other phases are influenced by the new interface.

The main goal in the development of the solver interface was strict data encapsulation. This means, that functions of a specific solver library are never called directly by the analysis code itself. We designed an intermediate software layer, which connects the analysis code to the different solver libraries and the functions within these libraries to access the data. Thus the application programmer needs not to know specifics of the solver library to target, for instance when implementing a new element, and at runtime it is easy to switch between the various solvers available.

This intermediate software layer is called the SlvLIB. This library primary consists of the following functions (the functions according to MLIB which are being accessed are given too):

`SlvInitializeSolver`	This function initializes the solver context structure during the *Input* phase. The following MLIB functions are used: `dslein`
`SlvSetMatrixFormat`	This function is used to set up the structure of the global system matrix during the *Pre* phase. Here the storage scheme (*Column Pointer, Row Index Sparse Matrix Representation*) is defined without doing any actual allocation of memory. No MLIB functions are used.

`SlvCreateMatrix`	This function is used to create the global system matrix (allocate the actual storage used to hold the nonzeros of the global system matrix) during the *Pre* phase. Furthermore, reordering and symbolic factorization is done. The following MLIB functions are used:
	`dsleim`
	`dsleor`

`SlvAddElement`	This function adds the local element matrix to the global system matrix during the *Pre* phase. The following MLIB functions are used: `dslevc`

`SlvPreSolveLinEquSys`	This function defines a pre phase before actually solving the equation system during the *Solution* phase. Here the numeric factorization is done. The following MLIB functions are used: `dsleco`

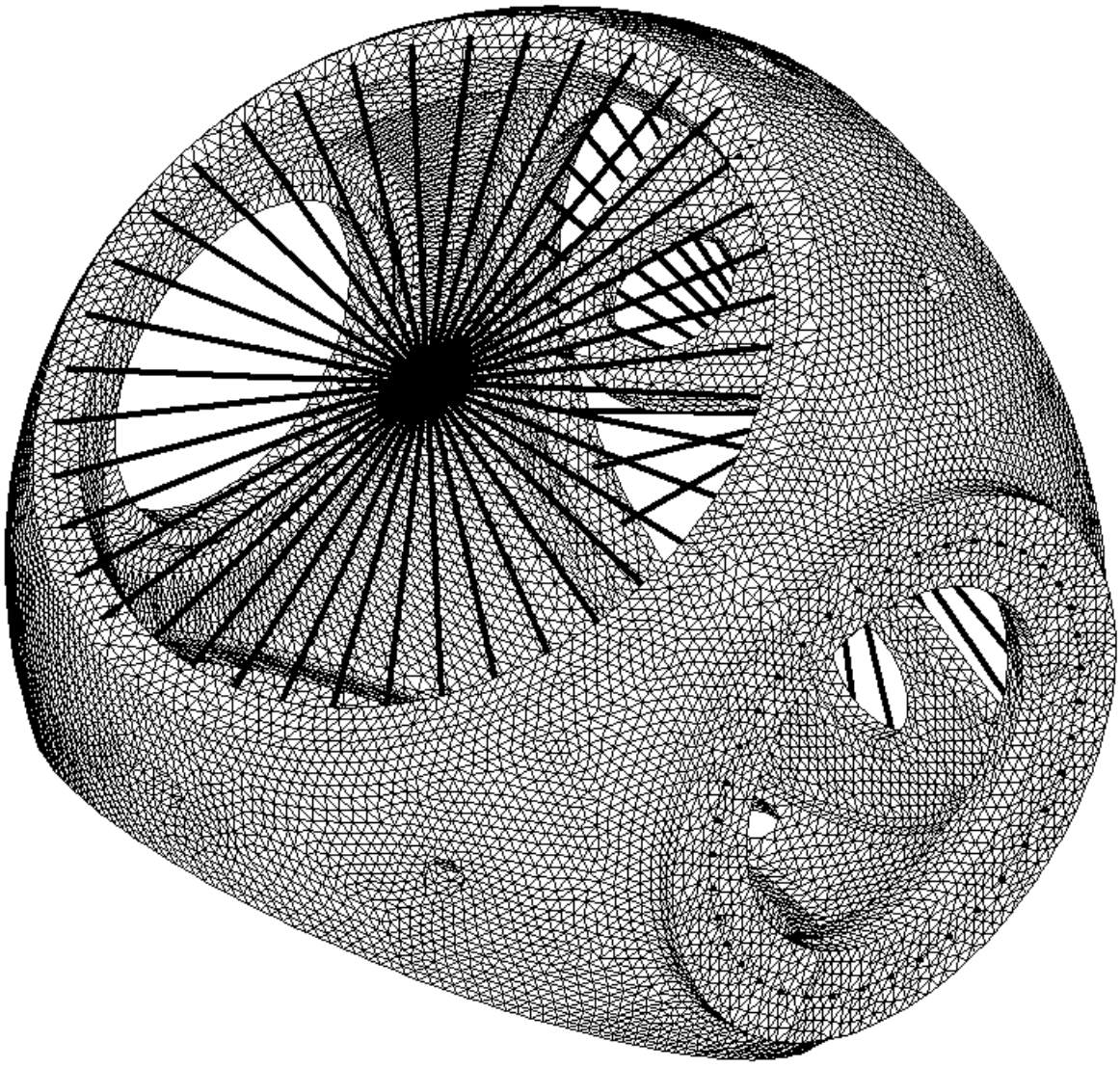
`SlvSolveLinEquSys`	This function evaluates an actual solution for a given right-hand-side vector during the *Solution* phase. The following MLIB functions are used:
	`dslesl`
	`dsleps`

`SlvFinalizeSolver`	This function destroys the solver context structure at the end of program execution. The following MLIB functions are used: `dsleda`

Only these seven functions are called directly by the analysis code itself. Thus a strict concept of data encapsulation has been realized.

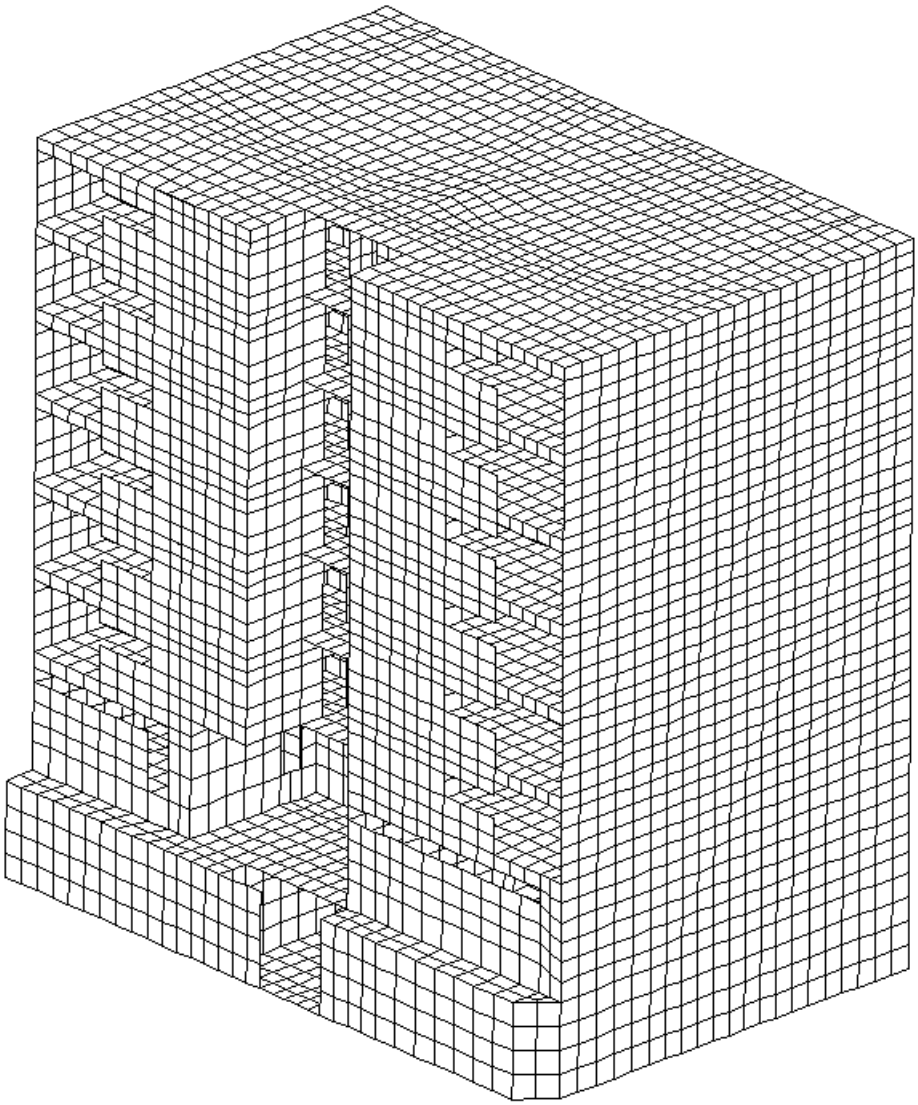
3 Benchmarks

In the following we will show some results we evaluated using the above described SW-strategy. We took two real-life examples for a linear stress analysis. The first benchmark is a typical mechanical engineering construction called *nabe*, which describes a hub made of cast-iron of a wind energy construction. The finite element model is shown in the following picture:



The model consists of 54,588 nodes and 255,025 elements (254,909 4-noded solids with three unknowns per node and 516 2-noded beams with six unknowns per node) resulting in an equation system of 165,321 of unknowns. The global system matrix itself contains 3,347,640 nonzeros (only one half of the matrix due to symmetry, including diagonal elements).

The second benchmark is a civil engineering construction entitled *dfh*, describing an apartment house made of reinforced concrete. The finite element model is shown in the following picture:



The model consists of 12,926 nodes and 32,101 elements (4-noded shells with six unknowns per node) resulting in an equation system of 72,426 of unknowns. The global system matrix itself contains 2,036,841 nonzeros.

The following table shows the evaluated results (best values are printed in boldface):

Benchmark	Solver	Memory [MB]	Time1 [s]	Time2 [s]
nabe	colsol	5,887	-	-
	lu	3,468	-	-
	pcg	**905**	408	803
	mlib	1,478	**167**	**562**
dfh	colsol	884	3825	3,894
	lu	645	2202	2,271
	pcg	**123**	493	562
	mlib	276	**32**	**101**

All results shown where carried out using only one processor. A comparison between various solvers is shown. Here *colsol* is the classical *Active Column Solver* with a *Skyline Storage Scheme* and a *Reverse Cuthill-McKee* algorithm for reordering. *lu* is an *LU sparse factorization* using a *Quotient Minimum Degree* algorithm for reordering. *pcg* is a *Preconditioned Conjugate Gradient* algorithm, where we use a *Jacobi* algorithm for preconditioning. And finally *mlib* represents HP's MLIB. The amount of *Memory* given shows the total maximum memory for the process of the analysis code during runtime. *Time1* and *Time2* show CPU timing measurements. *Time1* measures the time for the *Solution* phase (the time needed to extract a solution due to a given right-hand-side vector including factorization or iteration) whereas *Time2* additionally includes the time needed for evaluating stresses and strains.

As we can see, *mlib* gives always the best timing results. However, due to the additional fill-in required for direct sparse methods, it can be seen that *pcg* needs the least memory consumption. Iterative methods typically need only some extra vectors to do the preconditioning and the iteration itself, so the memory consumption is prior compared to direct methods. But the solution time actually needed heavily depends on the type of problem to solve. For a well-conditioned problem iterative methods need less iterations to solve the problem than for an ill-conditioned problem. This can be seen in the benchmarks shown here. Benchmark *nabe* has a much better condition number than benchmark *dfh*. So the speed-up of *mlib* is much better for benchmark *dfh* than for *nabe* if compared to *pcg*, which is due to the drawback of the many iterations needed.

When using iterative methods the biggest drawback arises when solving for more then one right-hand-side vector. In direct methods it is very cheap to carry out solutions for different right-hand-side vectors due to the factorization performed. Iterative methods never do a factorization, they try to evaluate a solution by using some special operators within an iterative context. Thus the whole process of iteration has to be done for every right-hand-side vector. And so for typical applications in linear statics where it is common to have up to 30 or more right-hand-side vectors, which are often to referred as load-cases, it seems to be better to use direct methods than iterative methods.

Next we will show the speed-up for HP's MLIB when using more than one processor. The following table shows the speed-up for benchmark *dfh*:

Nr. of proc.	Time [s]	Speed-up	Efficiency [%]
1	33.85	-	-
2	18.27	1.85	92.50
4	10.91	3.10	77.50
6	8.18	4.14	69.00
8	7.06	4.79	59.88

The column *Time* shows here the *Numerical factorization wall time* due to timing statistics within MLIB. We used this data as a reference for measurement, as the only function from MLIB which exists in a parallel version we are using is `dsleco`. However, it turned out that this function is the most time intensive section of the solution process. Column *Speed-up* shows the relative enhancements in performance compared to one processor, and column *Efficiency* is simply the relation of the speed-up to the number of processors used. As can be seen the efficiency decreases while using more processors, as to be expected. Thus it seems to be senseless to use more than four processors when doing analysis in a real-life engineering environment.

Currently we are working on the parallelization of iterative methods. There seems to be a big potential in speed-up when using more processors. In iterative methods there are primary matrix/vector operations which nearly scale linear with the number of processors used. So this could be a big advantage over direct methods.

4 Conclusion

We have shown the SW-strategy to implement bindings for various solvers in an engineering analysis code. For typical engineering applications some benchmarks were presented. We could show the excellent performance behavior of HP's MLIB for solving problems in linear statics. It turned out that there is a tremendous speed-up in the uniprocessor case if compared to classical hand-coded direct solver algorithms. Furthermore the advantage of using MLIB compared to iterative methods could be presented too. However, it turned out that iterative methods need much less memory consumption than direct methods due to the required fill-in for direct methods.

Also the speed-up in performance could be shown when using MLIB with more than one processor. It turned out that in an industrial environment an economic raise in efficiency using up to four processors is guaranteed.