Finite Element Code Design with Sandia’s Library Framework Trilinos:
Efficient data structures, solver interfaces and parallelization

Dominik Schillinger, Ralf-Peter Mundani
Lehrstuhl für Computation in Engineering, Technische Universität München
schillinger@bv.tum.de

Abstract: The library framework Trilinos can considerably facilitate and speed up the implementation of finite element software. The present contribution gives an overview of basic Trilinos capabilities related to typical FE requirements, points out their potential to reduce coding efforts, and exemplifies their simple integration into object-oriented C++ programs by code fragments. Additionally, the built-in MPI options of Trilinos are addressed, which minimize efforts to parallelize code for the application on distributed memory architectures. The paper concludes with results of benchmark tests with a Trilinos based FE code that demonstrate its computational efficiency with respect to the stand-alone code AdhoC and satisfactory scalability on the supercomputer HLRB2.

1 Introduction

Implementing a finite element program by coding every routine on one’s own involves an enormous amount of time and effort. Therefore, the interest in suitable libraries such as Dune (www.dune-project.org), PetSc (www.mcs.anl.gov/petsc) or Trilinos (http://trilinos.sandia.gov), which provide finite element related functionalities, has considerably grown within the last years. The present paper demonstrates how the design and implementation of an object-oriented FEM code in C++ can be considerably facilitated and accelerated by using some of the very basic capabilities of Trilinos.

Trilinos developed and maintained by Sandia National Laboratories is an open-source object-oriented software framework for the solution of large-scale, complex multiphysics engineering and scientific problems (Sala, Heroux et al. 2010). The key philosophy of Trilinos is its focus on packages that provide a variety of capabilities, ranging from development-level functionalities, such as data structures or interfaces, up to complete user-level applications, such as finite element programs. Trilinos offers a robust framework for the integration and interaction of different packages, which facilitates fast and easy combination of complex functionalities. Currently, Trilinos comprises around 30 packages, the most important of which provide linear algebra (Epetra), solvers and solver interfaces (Amesos, AztecOO, Nox, Loca), multi-grid methods (ML), Eigensolvers (Anasazi), mortar methods (Moertel), mesh generation / adaptivity (Mesquite), utility functionalities (TriUtils, Teuchos), interoperability tools (Intrepid), load balancing (Isorropia, Zoltan), finite element simulation (Sundance).

The paper is organized as follows: Section 2 outlines the selective installation process of Trilinos packages and third-party libraries. Section 3 discusses the implementation of efficient dense and sparse matrix and vector containers by the package Epetra, their direct linkage to Blas/Lapack routines, and the easy assembly of local data into a global matrix. In section 4, packages Amesos and AztecOO are introduced, which
provide interfaces to a range of direct and iterative solvers. Section 5 shows how the built-in MPI parallelization can be exploited for portability to distributed memory architectures. Finally, section 6 presents the results of two benchmark tests carried out with our Trilinos based in-house code FIDOS, which demonstrate its computational competitiveness with respect to AdhoC \(^1\) and satisfactory scalability on the HLRB2 supercomputer of the Leibniz-Rechenzentrum, Garching.

2 Selective installation of Trilinos packages

The possibility of selectively combining packages according to the specific tasks to be solved and the level of sophistication required is the major strength of the Trilinos framework. Compilation of selected Trilinos packages, adjustment of special options and integration of third-party libraries can be conveniently handled via ready-made CMake scripts in the style of the following excerpt.

```cmake
 cmake \
 -D MPI_C_COMPILER:STRING="mpicc" \ 
 -D MPI_CXX_COMPILER:STRING="mpiCC" \ 
 -D TPL_ENABLE_MPI:BOOL=ON \ 
 -D Trilinos_ENABLE_Epetra:BOOL=ON \ 
 -D Trilinos_ENABLE_Amesos:BOOL=ON \ 
 -D Amesos_ENABLE_SuperLUDist:BOOL=ON\ 
 -D SuperLUDist_INCLUDE_DIRS:FILEPATH=~/dominik/SuperLU_DIST_2.4/SRC \ 
 -D SuperLUDist_LIBRARY_DIRS:FILEPATH=~/dominik/SuperLU_DIST_2.4/lib \ 
 -D SuperLUDist_LIBRARY_NAMES:STRING="libsuperlu_dist.a"\ 
 -D BLAS_LIBRARY_DIRS:FILEPATH="/lrz/sys/intel/mkl91_21/lib/64" \ 
 -D BLAS_LIBRARY_NAMES:STRING="mkl; mkl_lapack" \ 
 -D LAPACK_LIBRARY_DIRS:FILEPATH="/lrz/sys/intel/mkl91_21/lib/64" \ 
 -D LAPACK_LIBRARY_NAMES:STRING="mkl; mkl_lapack"
```

3 Organizing data structures with Epetra

The package Epetra defines the basic classes for the construction of serial and parallel data structures. Each Trilinos package accepts Epetra objects as input, which is the basis for the combination of Trilinos functionalities from different packages.

3.1 Serial dense vectors and matrices

The classes `Epetra_SerialDenseVector` and `Epetra_IntSerialDenseVector` provide dense vectors of doubles or integers, respectively. In a finite element framework, double precision serial vectors are for example used to store the local force vector of each element, whereas integer vectors usually contain sets of element indices or global IDs of local degrees of freedom. Epetra serial vectors provide convenient vector notation by the ( ) operator, which automatically enforces bound checking, and can be constructed and reshaped by the Size( ) and the Resize( ) functions, respectively.

```cpp
#include "Epetra_SerialDenseVector.h"
Epetra_SerialDenseVector b;
int Length = 50;

b.Size(Length);
b.Resize(2*Length);
b(0) = 1;
```

\(^1\) Standard in-house FE code developed at the Chair for Computation in Engineering, TU München
The classes `Epetra_SerialDenseMatrix` and `Epetra_IntSerialDenseMatrix` provide dense double and integer matrices, which provide analogous functionalities. In a finite element framework, serial matrices are needed to store the multitude of matrices that occur at each Gauss point (for example the B-Matrix or the constitutive matrix) or in each local element (for example the element stiffness matrix).

```cpp
#include "Epetra_SerialDenseMatrix.h"

Epetra_SerialDenseMatrix K;
int RowLength = 50;
int ColumnLength = 100;
K.Shape(ColumnLength, RowLength);
K.Reshape(2*ColumnLength, 2*RowLength);
K(0,0) = 1;
cout << K;
```

Epetra overloads the C++ object `cout` to output vectors and matrices in a clearly arranged layout to console or file. In contrast to more common C++ allocations such as `new`, all Epetra serial containers guarantee that allocated memory is automatically deleted upon destruction of the container, so that memory leaks are avoided. This is achieved by a “runtime garbage collector”, as already well established in Java.

### 3.2 Connecting to Blas / Lapack routines

Epetra provides a range of convenient wrapper functions that connect serial containers to basic linear algebra operations, carried out by the highly efficient Fortran routines of the Blas/Lapack libraries. For the application in a finite element context, linear algebra functionalities such as matrix-vector and matrix-matrix multiplication are required during numerical integration of the stiffness matrix. Corresponding Blas/Lapack operations can be invoked by one line of code with the function `Multiply()`.

```cpp
#include "Epetra_SerialDenseMatrix.h"
#include "Epetra_SerialDenseVector.h"

Epetra_SerialDenseMatrix B(3,100);
Epetra_SerialDenseMatrix C(3,3);
Epetra_SerialDenseVector b(100);
Epetra_SerialDenseMatrix K(B.ColDim(),C.RowDim());
Epetra_SerialDenseVector k(C.ColDim());

// Set elements of matrices B, C and vector b here
// Matrix-matrix multiplication K = C*B and matrix-vector multiplication k = C*b
K.Multiply('N','N',0.0,C,B,1.0);
C.Multiply('N',b,k);
```

### 3.3 Sparse vectors and matrices

Sparse vectors and matrices belong to the class of distributed objects, which are partitioned across several processes to exploit parallelism on distributed memory machines. For the time being, we confine ourselves to the sequential case, where all data is kept on one process, and will come back to the distributed case in section 5. Epetra offers a range of sparse container classes for different purposes, such as `Epetra_CrsMatrix` for point matrices, `Epetra_VbrMatrix` for block matrices or `Epetra_FECrsMatrix` and `Epetra_FEVector` for finite element discretizations. Efficient memory allocation and
data organization of these containers are accomplished independently by Epetra without user interaction. Epetra provides a range of useful functions for sparse vectors and matrices, such as matrix-matrix and matrix-vector multiplication with Multiply( ), computation of dot products and norms with Dot( ), Norm2( ), or extraction of local elements with ExtractGlobalRowCopy( ), ExtractDiagonalCopy( ), and many more.

```cpp
#include "Epetra_SerialComm.h"
#include "Epetra_Map.h"
#include "Epetra_FECrsMatrix.h"
#include "Epetra_FEVector.h"

// Estimate total number of nonzero entries NumGlobNz of K
// Estimate number of nonzero entries NumNz[i] in each row i of K
Epetra_SerialComm Comm;
Epetra_Map Map(NumGlobEntries,0,Comm);

// Initialize global stiffness matrix K and force vector F
Epetra_Vector F(Map);
Epetra_FECrsMatrix K(Copy,Map,NumNz);
```

### 3.4 Assembly of non-local elements

In a finite element context, sparse containers are needed to store global force vector and stiffness matrix, which are typically assembled from corresponding local element data. Given a location vector, which relates local to global IDs, `Epetra_FECrsMatrix` and `Epetra_FEVector` containers can be automatically assembled from corresponding local matrices and vectors with SumIntoGlobalValues( ). The final call to GlobalAssemble( ) optimizes the internal data structures in terms of minimal memory consumption and efficient data access. In the parallel case, it also distributes matrix and vector entries over available processes and establishes a communication pattern. Continuing the previous code fragment, the assembly looks as follows

```cpp
// LocVect denotes an STL vector of Epetra_IntSerialDenseVectors, containing the // location vector for each element

Epetra_SerialDenseMatrix k;
Epetra_SerialDenseVector f;

// Loop over elements and Gauss points
for(int i=0; i<numEle; i++){
  for(int j=0; j<numGP; j++){
    // Assemble local k and f computed at each Gauss point j into global K and F
    K.SumIntoGlobalValues(LocVect[i],k);
    F.SumIntoGlobalValues(LocVect[i],f);
  }
}

// Optimize sparse data structure and distribute over processes
K.GlobalAssemble();
F.GlobalAssemble();
```

### 4 Interfacing solvers with Amesos and AztecOO

The Trilinos framework offers a variety of different solvers for different tasks and at different levels of sophistication. The basic advantage is that users do not need to create a multitude of solver interfaces on their own, but can change the solver by just changing the specification strings.
4.1 Direct solvers

Many powerful direct solvers for the solution of a linear system of equations have been proposed over the last years outside the Trilinos framework. The Amesos package provides a unified interface to the following third-party libraries: the serial solvers Klu, Lapack, Pardiso, SuperLU, Umfpack and the parallel solvers Mumps, Scalapack and SuperLU_DIST. With Epetra_FEChsMatrix K, Epetra_FEVector F known from the assembly example, the Amesos interface can be used as follows

```cpp
#include "Amesos.h"
#include "AmesosClassType.h"

// Initialize the solution vector
Epetra_Vector x;

// Initialize Amesos
Epetra_LinearProblem Problem(&K, &x, &F);
Amesos_BaseSolver* Solver;
Amesos_Factory;

// Specify the solver: Use Klu here
char* SolverType = "Amesos_Klu";

// Solve
Solver = Factory.Create(SolverType, Problem);
A_Base->SymbolicFactorization();
A_Base->NumericFactorization();
A_Base->Solve();

// Show the solution vector
cout << x;
```

4.2 Iterative solvers and pre-conditioners

The AztecOO package extends the older Aztec library and provides a unified interface to a multitude of standard iterative solvers of Krylov type, such as conjugate gradient, GMRES, Bi-CGSTAB or TFQMR, as well as corresponding pre-conditioners. With the Epetra_FEChsMatrix K and the Epetra_FEVector F known from the assembly example, the AztecOO interface can be used similar to Amesos.

```cpp
#include "AztecOO.h"

// Initialize the solution vector
Epetra_Vector x;
Epetra_LinearProblem Problem(&A, &x, &b);
AztecOO Solver(Problem);

// Choose a simple point Jacobi pre-conditioner
Solver.SetAztecOption(AZ_precond, AZ_Jacobi);

// Default solver is GMRES, use max. 100 iterations and residual threshold of 10^-9
Solver.Iterate(100, 1.0E-9);

// Show solution vector
cout << x;
```
5 Exploiting the built-in MPI parallelization

If a finite element code is intended for high-performance computing, for example to tackle expensive problems in nonlinear structural mechanics, parallelization with MPI is required due to the distributed memory architecture of common clusters and supercomputers. However, explicit parallelization of the code requires firm knowledge of MPI commands and substantial experience with parallelization strategies. However, most Trilinos packages offer a built-in MPI parallelization, which can be efficiently used with limited experience. In the scope of a finite element code, the parallel execution of the solver is of special interest, since the ratio of solving to total computation time grows substantially with increasing problem size.

In order to be able to solve in parallel, the global stiffness matrix, the force vector and the vector of unknown coefficients need to be distributed across the available processes. For Epetra’s sparse distributed vectors and matrices, this is accomplished by the Epetra_MpiComm class, which encapsulates the general information and services usually assigned to the MPI communicator, and the Epetra_Map class, which handles the actual distribution process. To create a distributed framework for parallel execution of solvers with MPI, the following code fragment is sufficient

```c++
#include "mpi.h"
#include "Epetra_MpiComm.h"

// Start main program
int main( int argv, char *argv[]) {
  MPI_Init(&argc, &argv);
  Epetra_MpiComm Comm(MPI_COMM_WORLD);

  // Estimate total number of nonzero entries NumGlobNz in K
  Epetra_Map Map(NumGlobNz,0,Comm);

  // Here follows the rest of the FE code
  MPI_Finalize();
}
```

Note that the only MPI calls users have to explicitly introduce in their codes are MPI_Init() and MPI_Finalize(). The solver itself is automatically parallelized, if MPI has been activated during compilation of the Trilinos package (AztecOO) or third-party library (Amesos). For more advanced users that would like to parallelize further regions of their code on their own, the Epetra_MpiComm class provides access to most of the common MPI functionality, for example through functions Barrier(), MyPID(), NumProc(), GatherAll(), and many more.
6 Benchmark tests

The following two benchmark tests carried out with our in-house Trilinos based code FIDOS give a tentative idea of efficiency and scalability of Trilinos based software.

6.1 Computational efficiency of data structures

The first benchmark consists of a linear plane stress problem discretized by 64 high-order finite elements as shown in Fig. 1(a) and demonstrates the efficiency of the data structures and solvers of Epetra and AztecOO. It is solved sequentially by both FIDOS and our standard in-house code AdhoC for polynomial degrees $p = \{2, 4, 6, 8\}$, using the same integrated Legendre basis and iterative CG solvers. Although FIDOS and AdhoC are run under different circumstances (see Tab. 1), the measured computation times shown in Fig. 1(b) indicate that Trilinos based FIDOS is nearly competitive with the stand-alone code AdhoC, whose performance has been optimized over the last years.

<table>
<thead>
<tr>
<th>$p$</th>
<th>FIDOS (Trilinos based)</th>
<th>AdhoC (stand-alone)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Intel’s icc, icpc, ifort</td>
<td>GNU gcc, gfortran</td>
</tr>
<tr>
<td>4</td>
<td>-fast (Trilinos), -O3 (FIDOS)</td>
<td>-O5</td>
</tr>
<tr>
<td>6</td>
<td>Intel(R) Core2 Duo T5500 @ 1.83GHz</td>
<td>Intel(R) Core2 Duo E4500 @ 2.20GHz</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 1.** FIDOS and AdhoC have been compiled on different machines with different compilers and optimization options. Note that FIDOS is better optimized than AdhoC, but run on the slower processor.
6.2 Scalability on distributed memory architectures

The second benchmark, which consists of a compression test simulation of a nonlinear metal foam structure as shown in Fig. 2(a), demonstrates the efficiency of FIDOS on large-scale distributed memory architectures, based on the built-in MPI parallelization. The key aspect to be tested is the scalability of the third-party solver SuperLU_DIST, fed with distributed Epetra objects via the interface Amesos. The FIDOS simulation is run with $n = \{1,2,4,8,16,32\}$ processes on the SGI Altix supercomputer HLRB2\(^2\), installed at the Leibniz-Rechenzentrum, Garching. The corresponding strong speed up of the solver $S = T_1/T_n$, which denotes the ratio of the solver time $T_1$ in the single process case with respect to the solver time $T_n$ required if the simulation is run with $n$ processes, is plotted in Fig 2(b). The example illustrates that the built-in MPI parallelization permits satisfactory scalability of the solver and thus allows for a simple portability of finite element codes on large-scale distributed memory machines.

Acknowledgements

The first author gratefully acknowledges support of the Munich Centre for Advanced Computing (MAC) and the International Graduate School of Science and Engineering (IGSSE) at Technische Universität München.

References


\(^2\) 9728 processor cores; 39 TB RAM; peak performance of 62.30 TFlops