ulm university universität
uulm

Vorlesung 'High Performance Computing'

Summer Term 2013

Mazen Ali
Christopher Davis

Dr. Andreas Borchert
Prof. Dr. Stefan Funken
Prof. Dr. Karsten Urban

Institute for Numerical Mathematics

Title.....

Mazen Ali, Christopher Davis

July 20, 2013

Contents

1 Introduction

The basis of this lecture course has been the parallelisation of various numerical methods, with a particular focus on the Finite Element Method. For this project we have been supplied with a software package (which we shall henceforth call *the FEM package*) that computes the solution to the Poisson partial differential equation using the Finite Element Method. This package is self contained - it includes its own custom matrix/vector types, and its own implementations of linear algebra operations on these types. Our goal is to make improvements to the package with the help of the FLENS.

FLENS (*Flexible Library for Efficient Numerical Solutions*) is a C++ library written by Dr. Michael Lehn, which offers a comprehensive collection of matrix and vector types. Included is a C++ -based BLAS (*Basic Linear Algebra Subfunctions* implementation, which provides linear algebra operations, such as matrix-vector multiplication, on these types.

The advantages of using the FLENS in the FEM package are numerous. Firstly, use of an external library for matrix/vector types adds some standardisation to the package, aiding, for example, a user who is new to the FEM package, but has experience of the FLENS from other projects. Secondly, the library can be linked, with almost trivial effort, to any optimised BLAS or LAPACK libraries available, such as *ATLAS* or *Goto BLAS*, for instant speed improvements of BLAS operations. Thirdly, the FLENS offers overloaded operators for the BLAS operations. We recognise that users from different backgrounds may have a preference regarding the noation used for linear algebra operations, be it the tradition BLAS notation:

```
1    blas::mv(NoTrans, One, A, p, Zero, Ap);
```

or a notation more akin to that of MATLAB:

```
1    Ap = A*p;
```

(for matrix A, vector p). With FLENS, we have the choice.

We therefore summarise the aims of this project as follows:

1. Replace all data storage objects with FLENS-based objects.
2. Where possible, replace linear algebra operations with their BLAS equivalents, for speed-up from optimised BLAS libraries.
3. Offer two versions of solvers, one with BLAS notation, one with overloaded operators.

Thus it is worth noting that we are primarily improving the existing implementation from a software design point of view, with possible performance improvements, rather than adding new methods.

2 Matrix and Vector Types

A major part of converting the FEM package to the FLENS framework is the transition from the package's custom storage types to FLENS-based types. Of course, some types, such as the package's *Vector* class, have exact FLENS equivalents. Others, however, contain bespoke objects and methods for the MPI communications. Thus we must create our own custom storage types in these cases.

2.1 Equivalent Types and Index Base

We adopt the following direct conversions from the FEM package to FLENS framework:

$$\begin{aligned} \texttt{Vector} &\rightarrow \texttt{DenseVector}\langle\texttt{Array}\langle\texttt{double}\rangle\ \rangle \\ \texttt{IndexVector} &\rightarrow \texttt{DenseVector}\langle\texttt{Array}\langle\texttt{int}\rangle\ \rangle \end{aligned}$$

However, we must note that the default index base in FLENS, which we are using here, is 1, as opposed to that of the package, which is 0. We make this change, despite the awkwardness it adds to the transition, because this is the natural index base regarding the mesh geometry. Numbering of the mesh nodes starts at 1, and assembly of the system of linear equations frequently accesses vector elements corresponding to node identities. For example in the FEM package:

```
1 || someVec_FEMpackage(nodeID - 1) = someValue;
```

as opposed to using FLENS:

```
1 || someVec_FLENS(nodeID) = someValue;
```

Whilst this is purely cosmetic, it may help to avoid bugs caused by forgetting to subtract 1 from node identities. For consistency, we implemented all FLENS matrices/vectors with index base 1.

FLENS includes a storage scheme for sparse matrices of CRS (Column-Row-Storage) type, offering an alternative to the FEM package's CRSMatrix type. However, these must be initialised from a FLENS sparse matrix with a coordinate storage scheme, effecting a change in the implementation of the type, but not requiring the creation of a custom type.

??? Mazen's CRS sorting?

2.2 TypeI and TypeII

The FEM package uses the nomenclature defined in the lecture course, of *TypeI* and *TypeII* to distinguish between vectors that contain values corresponding to the problem

posed on a compute node's local mesh, or on the global mesh:

- **TypeI**: global values.
- **TypeII**: local values.

We adopt this definition in this paper and in our code, and refer to these types as *MPI vector types*.

2.3 FLENSDataVector

(Christopher Davis)

The FEM package's *DataVector* class is the package's primary custom vector type. Included members are:

- Vector object: stores the values of the DataVector.
- Coupling object: contains mesh geometry information required for MPI communications.
- A vectorType enumerated type: determines the MPI vector type of the vector (see Section 2.2).

We propose a FLENS-based replacement for this class called FLENSDataVector, which incorporates a few small yet profound modifications to the structure.

Firstly, we make the obvious choice of using a FLENS `DenseVector⟨Array⟨double⟩ ⟩` to store our vector values. However, we choose to *derive* our class from this FLENS type, rather than specifying a `DenseVector` as a member object. I.e. we use a 'is-a' `DenseVector` approach, rather than a 'has-a' approach. The advantage here is that our class inherits all methods and overloaded operator from the `DenseVector` class, and can be passed directly to the FLENS BLAS functions. This lends itself to a more parsimonious implementation - under the 'has-a' approach we would need to overload every BLAS function, such as:

```
1  double
2  dot(FLENSDataVector x, FLENSDataVector y) {
3       blas::dot(x.vec, y.vec);
4  }
```

and clutter our FLENSDataVector class with trivial operators, such as:

```
1    double &
2    operator()(int index) {
3         return vec(index);
4    }
```

neither of which are desirable.

The Coupling object is stored as a constant reference - the same way as in the FEM package.

Next we consider the enumerated type that specifies the MPI vector type. Here we wish to change the structure somewhat - whilst setting the MPI vector type in this fashion is easy and flexible, allowing the type to be changed as and when required, we find this to be *too* flexible, and not ideal from a software design perspective. To help explain the situation, we consider a blind man and his socks. The man in question loves to wear socks, and therefore does so at all times. He commands an extensive collection, consisting of many different colours. Each morning he changes his socks, taking care to pair the dirty socks together so that they remain paired after washing. His system appears to work well - he always wears matching socks, and as a result leads a successful life. But what about if one morning whilst in the middle of changing his socks he is distracted by the telephone ringing, and as such forgets to change one of his socks - now his socks don't match! Whilst we would like to think that some kind person may notify him of his mistake, the world can be a cruel place. Odd socks are rarely tolerated by modern societies, so we can imaging that his life's achievements would probably crumble around him. Returning to the world of the FEM package, we hope that the difficulties that could arise from the enumerated type are clear - there is no way of determining whether the values contained in a TypeI DataVector are actually global values. There is always the chance that some rogue function changed the type without modifying the values. Such a problem would not make for an enjoyable debugging task.

Thus our FLENSDataVector is implemented as a template class, requiring the MPI vector type to be defined (permanently) at instantiation. The following classes are permitted as specialisation types:

- `class FLvNonMPI`
- `class FLvTypeI`
- `class FLvTypeII`

Clever implementation of the FLENSDataVector constructors limits specialisation of the class to these types *only*, as well as ensuring the specification of a Coupling object for MPI types (and not for the non-MPI). We use specialisations of the constructor function for these types, for example:

```
template <>
FLENSDataVector<FLvNonMPI>::FLENSDataVector(int n)
        :       DenseVector<Array<double> >(n),
                coupling(Coupling())
{
        //Permits instatiation of FlNonMPI specialisation.
}
```

and we add a line to the unspecialised constructor that will cause an error at *compile time* if scope ever reaches there (which would be due to a wrong type specialisation):

```
template <typename VTYPE>
FLENSDataVector<VTYPE>::FLENSDataVector(int n)
        :       coupling(Coupling())
{
        VTYPE::CHK;                     //<-- If scope ever reaches here,
                                        //compilation will fail.
                        //e.g. if FLENSDataVector<double> instantiated.
}
```

Thus the following instantiation would cause a compiler error:

```
FLENSDataVector<double>  oops(5);
```

As such, we have limited the potential for undefined behaviour.

All communication-related member methods of the FEM package's DataVector are added to the FLENSDataVector with no significant changes. For conversion methods, we require the object to be of the destination type. For example:

```
FLENSDataVector<FLvTypeI> myVec(5, Coupling());

////////////////////////////////////
// *** fill with local values *** //
////////////////////////////////////

myVec.typeII_2_I();             <-- perfect
//myVec.typeI_2_II();           <-- would cause compiler error
```

2.4 BLAS Overloading

Most BLAS functions can be used in the FEM package in their usual form. The copy and dot product functions, however, require attention.

Copying between two FLENSDataVectors of the same MPI vector type can use the BLAS copy function without further assistance. The types match, and we can't do anything about the constant reference to the Coupling object (this must be left to the user to ensure).

When copying vectors of differeing MPI vector types, we overload the BLAS copy function. Within this overloaded function, we call the BLAS copy function to copy the vector values by *upcasting* the FLENSDataVectors to their parent class `DenseVector⟨Array⟨double⟩ ⟩`, then perform the appropriate conversion, for example:

```
void
copy(FLENSDataVector<FLvTypeII> &orig, FLENSDataVector<FLvTypeI> &dest)
{

        //Copy data as usual (masquerading as a DenseVector :) ):
        blas::copy(*static_cast<DenseVector<Array<double> > *>(&orig),
                   *static_cast<DenseVector<Array<double> > *>(&dest));

        //Perform vector type conversion:
        dest.typeII_2_I();
}
```

We use a similar technique for the dot product - the dot product of the two supplied vectors is calculated using BLAS via upcasting, and then the appropriate MPI communication is performed.

By ensuring that our overloaded functions still use the FLENS BLAS implementation, we maintain the possibility for objective (2) in the Introduction.

3 Part I (Christopher Davis)

3.1 The CG Solver

This class stores its vector values in

Furthermore, now that we have a 'proper' object type to define the MPI vector types, we can remove many type asserts that are present in the FEM package, such as:

```
double dot(DataVector &u, DataVector &v)
{
        if(u.type==nonMPI && v.type==nonMPI)
          return u.values.dot(v.values);

        // we only multiply typeI and typeII vectors
        assert(u.type != v.type);
```

We can simply replace all instances of *Vectors* with FLENS

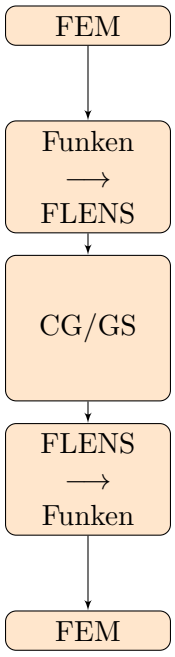
4 Part II (Mazen Ali)

This section continues describing the performed changes to the FEM package in order to replace the provided LinearAlgebra library with FLENS. Note that the Makefiles have to be slightly adjusted as well, please refer to the provided package. Further adjustments, not described here, were performed for the main functions (main-serial and main-parallel).

4.1 GS Solver

One of the main components of the FEM package is the solver that provides the end-result. Implemented in the FEM package were the CG (*Conjugate Gradient*) and the GS (*Gauss-Seidel*) solvers, parallel and serial versions. The adapting of the CG-Solver to FLENS was described in the previous section.

Analogue to the CG solver, the GS solver was initially integrated into FEM via a wrapper. Schematically the wrapper does the following:



Function wrapper

The wrappers transform the data from the original format to FLENS and from FLENS back to the original format upon completion of the solving procedure. This functionality

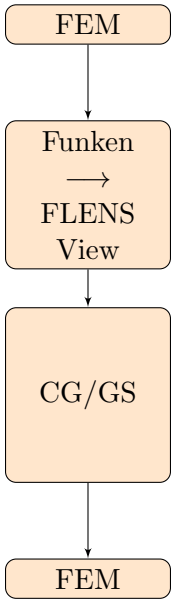
is provided in Flens_supl by the headers funk2flens.h and flens2funk.h. For the solvers the transformation is required for a matrix and a vector type. Although for the Poisson problem and the chosen basis functions it suffices to consider transformations for sparse matrices, Flens_supl includes both transformations for sparse and dense matrices, as well as a GS-solver for dense matrix classes (possible future utilization as pre-conditioner for e.g. multigrid in problems with resulting non-sparse systems).
The implemented transformation in Flens_supl copies the data manually from the original storing format into FLENS and back. However, FLENS also provides other alternatives for transforming data to FLENS using *Matrix/Vector-Views*. The concept is illustrated below, where data from the original storing format is transformed to FLENS:

```
int
main()
{
        // Typedefs
        typedef flens::DenseVector<flens::Array<double> >
                                                        Fl_Vec;
        typedef flens::DenseVector<flens::Array<double>::View>
                                                        VecView;
        typedef flens::Array<double>::View
                                                        View;

        // Initilaize with 0s
        Vector  v_funk(5); // Funken
        VecView v_flens_view = View(5, v_funk.data()); // Flens View
        Fl_Vec  v_flens = v_flens_view; // Flens

        // Change values
        v_funk(0) = 1;
        v_flens_view(5) = 1;

        // Print
        cout << "funk=" << endl;
        print(v_funk, 0, 4);
        cout << endl;
        cout << "flens_view=" << endl;
        print(v_flens_view, 1, 5);
        cout << endl;
        cout << "flens=" << endl;
        print(v_flens, 1, 5);
        return 0;
};
```

which would produce the output:

```
# funk=
1 0 0 0 1
flens_view=
1 0 0 0 1
flens=
0 0 0 0 0
```

FLENS views can be thus used in two ways: reference data from any other linear algebra package, as long as it provides access to the underlying C-Array, maintaining all the functionality of FLENS; or initialize FLENS non-view data structures via the internal FLENS copy constructor, thus avoiding copying the values manually.
Note that this offers another form of "wrapping" for the solvers which does not involve data copying. The original data can be referenced via FLENS Matrix/Vector-Views using non-const view for the solutions vector and const views for the rest. The idea is schematically illustrated below:



Function wrapper w. Views

After completing the modification process for all data structures in the FEM package the wrappers for the solvers are turned off, since not required anymore.

The implementation of the solvers themselves remained mostly unchanged. The required changes included:

- Index base: by default the index base in FLENS is 1 (FORTRAN style), as opposed to the original data structures where the index base is 0 (C style). FLENS provides the possibility to set the index base (and range), as well as operators to access first and last indexes of its data structures. Thus, it is possible to utilize these operators in order to provide maximum flexibility w.r.t. choice of index base. However, we decided not to follow this approach, since this would make the code longer and less readable and the mentioned flexibility does not provide any foreseeable advantages. Thus, the index base was assumed to be 1 in all FEM data structures, as by default in FLENS, and the code was modified accordingly.

- CRS matrices access: FLENS CRS matrices can be accessed much in the same way as the original CRS matrices:

```
const auto &_rows = A.engine().rows();
const auto &_cols = A.engine().cols();
const auto &_vals = A.engine().values();
```

The keyword *auto* is recognized by the C++11 standard, i.e. this necessitates the use of a C++11 conform compiler, such as g++ version 4.7 or higher[1]. Compile via:

```
# g++ hello.cpp -std=c++11 <options>
```

The keyword *auto* allows the compiler to deduce the type of the declared variable, applying rules similar to template argument deduction during function calls. Hence, this saves the programmer a lot of typing and makes the code more readable for the user.
engine() is the storage mechanism for FLENS and for CRS matrices it contains the getter methods rows(), cols() and values(), that return the corresponding vectors of type *DenseVector*. In order to preserve the functionality of FLENS data structures, it is recommended to use references instead of pointers to the data (which would also be possible). Usage of the attained vectors is analogue to the standard usage of CRS data structures, e.g. as implemented in the original FEM package.

Note, however, that FLENS sparse matrices do not allow to access elements directly (which can otherwise result in unintentional densifying), i.e.:

```
A(2,2)          += 3; // works fine for CoordStorage
double _tmp     = A(2,2); // compile error
```

The second access would produce a compile error for all sparse FLENS classes. Thus, for retrieving values of the matrix a search loop is required.

- Underscore operator _ : FLENS provides the utility of an underscore operator that can be used to set or copy portions of matrices and vectors, similar as in e.g. MATLAB:

```
const Underscore<IndexType> _
// Set MyVector to 3rd column of MyMatrix (must be same length)
MyVector = MyMatrix(_,3);
// Set Myvector's elements 1 to 3 to first elements of first row
MyVector(_(1,3)) = MyMatrix(1,_(1,3));
```

The underscore operator substituted the original set() and memcpy() calls of the LinearAlgebra data structures.

- Type I,II vectors: the original *DataVector*s were replaced by templated *FLENS-DataVector*s that were described in Part I. As already mentioned, this preserves type safety. Usage is in many ways identical to the original DataVectors, main difference being that type conversions are performed during copy operations or during explicit method calls, such as typeII_2_I or vise versa.

[1]Note that this is required for the entire FLENS library.

- Templates: Both solvers were templated, as well as most of Flens_supl headers. The only exceptions are some of the headers that were used as temporary "adapters" to introduce FLENS into the original FEM package. Apart from some minor flexibility, this would allow to utilize Flens_supl via headers only. Note that the FLENS library is headers only as well.

The dense GS solver required the same modifications, CRS matrix access excluded. The GS solver was tested and debugged.

4.2 Data Structures

In this section the adjustments for the Mesh and Coupling data structures are briefly described.

4.2.1 Mesh

Apart from the self explanatory data type changes, the required adjustments included:

- Index Base: as described in previous section.
- Read and write methods: Flens_supl includes the templated read and write methods, implemented similar to the original read and write methods of the FEM package.
- Const-Correctness: note that another advantage of the FLENS library is that it ensures const-correctness. Therefore the constructor calls had to be modified accordingly. E.g. in the original Mesh constructor the input vector *_dirichlet* is declared as *const* although write access is required, which would cause a compile error in FLENS.
- Underscore operator _: as described in the previous section.
- CRS_sort: for the method provideGeometricData() the original FEM package utilized the constructor for the CRSMatrix class to sort the values of a vector. The constructor accepted 3 vectors of same size, the first contained the row index, the second the column index and the third the corresponding value in the matrix. This functionality is provided by the templated header function CRS_sort.h, implemented analogue to the constructor call of the original FEM package. Alternatively, this can be done entirely without intermediary headers, utilizing the FLENS CoordStorage format and converting to GeCRSMatrix:

```
1 // Initialize (_numRows, _numCols assumed to be already computed in
      a preceding for-loop)
```

```
flens::GeCRSMatrix<flens::CRS<ElementType> >
                                    vals_crs;
flens::GeCoordMatrix<flens::CoordStorage<ElementType> >
                                    vals_coord(_numRows, _numCols);
// Fill
for (IndexType i=1; i<=rowi.length(); ++i)
{
        if (vals(i)!=0)
        {
                vals_coord(rowi(i), coli(i)) += vals(i);
        }
}
// Convert
vals_crs = vals_coord;
// Access
const auto &_vals = vals_crs.engine().values();
```

The latter way requires less writing but internally does more than actually necessary. However, even for large vectors this does not have a substantial influence on run-time, so that the end-choice is a matter of taste.

After adjusting the Mesh class, the dependencies in Fem.cpp have to be adjusted accordingly. The modified data structures were tested and debugged.

4.2.2 Coupling

Other than the self explanatory data type changes, no further modifications for the Coupling class were required. The dependencies have to be adjusted accordingly in Mesh, Fem, FLENSDataVector and the GS solver. The modifications were tested and debugged.

4.3 Debugging notes

The modifications described above required extensive debugging for both parts, as well as the merging process. Since this amounted to the biggest portion of the entire work time, this section will illustrate on a simple example of a common segmentation fault how to debug using core dumps. This will hopefully save future users a lot of time when modifying the FEM package.

The FLENS library utilizes *assert()* to ensure the index is not out of bounds. However, since these assertions are "hidden" deep in the templates and there is no exception handling implemented, the resulting error message will not provide any useful information about the origin of the faulty access. The system will attempt to produce a core dump, which is however turned off by default on most modern OSs. For example, the following code will produce a segmentation fault:

```
1 int
2 main()
3 {
4         flens::DenseVector <flens::Array<double> > V(5);
5         V(6) = 3;
6         return 0;
7 };
```

Compile and run:

```
# g++ seg_fault.cpp -o seg_fault -g
# ./seg_fault
seg_fault:  /usr/local/FLENS/flens/storage/array/array.tcc:108:
flens::Array<T, I, A>::ElementType& flens::Array<T, I, A>::operator()
(flens::Array<T, I, A>::IndexType) [with T = double; I = flens::IndexOptions<>;
A = std::allocator<double>; flens::Array<T, I, A>::ElementType = double;
flens::Array<T, I, A>::IndexType = int]:  Assertion 'index<=lastIndex()' failed.
Aborted (core dumped)
```

As can be seen at the end of the error message, it claims that a core was dumped, which does not have to be true. To indeed enable core dumping type in the command line:

```
# ulimit -c unlimited
```

Instead of *unlimited* you can specify any other data size (in number of 512-byte blocks). After running the executable again, the crash now generates a core dump, by default in the same directory. Note that for debugging it is advisable to compile with the -g option, since the resulting machine code then contains more information. To analyze the produced core one can e.g. use the standard gdb debugger and proceed by typing:

```
# gdb -c core seg_fault
```

where the executable has to be specified as well, otherwise the machine code messages will not be "linked" with the source code. The last statements will usually be some calls to the kernel, e.g. a request to terminate the program. Working up the core file, one then can get to the observed error message:

```
# Program terminated with signal 6, Aborted.
#0 0x00007f3fa01f1425 in raise () from /lib/x86_64-linux-gnu/libc.so.6
(gdb) up
#1 0x00007f3fa01f4b8b in abort () from /lib/x86_64-linux-gnu/libc.so.6
(gdb)
#2 0x00007f3fa01ea0ee in ??  () from /lib/x86_64-linux-gnu/libc.so.6
(gdb)
#3 0x00007f3fa01ea192 in __assert_fail () from /lib/x86_64-linux-gnu/libc.so.6
(gdb)
#4 0x0000000000400835 in operator() (index=6, this=0x7fff85fc2108)
at /usr/local/FLENS/flens/storage/array/array.tcc:108
108 ASSERT(index<=lastIndex());
```

Working up further will lead to preceding calls and then finally to the origin of the faulty access:

```
# 4 0x0000000000400835 in operator() (index=6, this=0x7fff85fc2108)
at /usr/local/FLENS/flens/storage/array/array.tcc:108
108 ASSERT(index<=lastIndex());
(gdb)
#5 operator() (index=6, this=0x7fff85fc2100)
at /usr/local/FLENS/flens/vectortypes/impl/densevector.tcc:201
201 return _array(index);
(gdb)
#6 main () at seg_fault.cpp:22
22 V(6) = 3;
```

Obviously the benefits of analyzing core dumps extend to any sort of program crashes and it is generally a good way to make debugging much more efficient.
Unfortunately, the approach does not apply in general to MPI. Though a core dump can still be generated, the content of the core is usually useless for debugging purposes. This, together with the fact that standard debuggers are not suited for parallel programs, often forces the programmer to debug "by hand", i.e. via cout and Barrier().

5 Flens_supl (Mazen+Chris)

This section summarizes the headers in Flens_supl.

- CRS_sort.h: sorting function required in Mesh; accepts a vector of values with the corresponding row and column coordinates, returns a sorted vector values; sorted according to the order in a CRS matrix; templated;
- FLENSDataVector.h: Type I and II (2 separate data types) data vectors derived from FLENS DenseVector; required for parallel computations; methods implemented as in the original FEM package; templated;
- FLENS_read_write.h: read method for FLENS and write methods for FLENS dense vectors and matrices; implemented as in the original FEM package; templated;
- cg_mpi_blas.h: CG sparse solver for MPI; implemented as in the original FEM package; templated;
- cg_nompi_blas.h: CG sparse solver; standard implementation; templated;
- gs_mpi_blas.h: GS sparse and dense solvers for MPI; implemented as in the original FEM package; templated;
- gs_nompi_blas.h: GS sparse and dense solvers; standard implementation; templated;

- flens2c.h: converts FLENS DenseVector to C array; initially required for wrapping the original package with FLENS data structures; not required in the final version; non-templated;
- flens2funk.h: converts data from FLENS data structures to the original FEM package; initially required for wrapping; not required in the final version; templated;
- funk2flens.h: converts data from the original FEM package to FLENS; initially required for wrapping; not required in the final version; templated;
- wrappers.h: wraps the CG and GS solvers for use with the data structures from the original FEM package; not required in the final version; non-templated;