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 19, 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.
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

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.

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 far from ideal from a software design perspective. To better explain the situation, we consider hair colour as an analogy.

This class stores its vector values in

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