Challenge 2. A sparse matrix

Luca Formaggia, Paolo Joseph Baioni, Beatrice Crippa

AA 24-25

1 Introduction

In large scale computations, it is often necessary to exploit sparsity to be able to operate with large matrices. Indeed, the space used up by full storage of a matrix grows as N^2 and can rapidly fill up all your memory, even when most elements are zero. Moreover, if the matrix is sparse, operations like matrix-vector product can be made more efficient if we avoid the useless multiplications by the zero elements.
First of all, let's define what a sparse matrix is. We can give different (equivalent) definitions, for instance,

- It is a matrix whose number of non-zero elements grows as N, being N the linear size of the matrix. For non square matrices N may be the square root of the product of the number of rows and columns.
- It is a matrix where the average number of non-zero elements per row is bounded above by a constant independent of N.
- The number of zero elements increases as N^2.

The third definition implies that a large sparse matrix has normally "plenty of zeros", so the advantage of a sparse representation can be substantial in large scale problems.
Clearly, there are drawbacks. While the full storage of a matrix requires just a mapping $(i, j) \to k$, where k is the index in a sequential linear container (e.g. a standard vector or array), the way to store a sparse matrix is not unique and more complex. We can distinguish between two families of storage techniques:

- **Dynamic (or uncompressed) storage techniques**. They allow to add (and sometimes also eliminate) new non-zero elements easily, normally at the price of higher memory requirements;
- **Compressed storage techniques**. They are the most efficient in terms of memory and of the computational efficiency of basic operations like matrix-vector product. But they do not allow changing the pattern of sparsity.

Among the *dynamic* storage formats, we mention the following ones:

COO In a basic form it is made by a vector containing the couples i, j and a corresponding vector with the value A_{ij} for all non-zero elements. You can have a tuple to keep the information for each element in the same place.

COOmap It is just a different version of the COO format, where the map $(i, j) \to A_{ij}$ is performed with a std::map (or a std::unordered_map) where the couple i, j acts as the key. Different orderings may be used so that the elements are logically ordered column-wise or row-wise. In other words, with the appropriate ordering traversing the map provides the matrix elements "by row" or "by column".

Among the *compressed* techniques, we mention

CSR Compressed Sparse Row. We store two vectors of indexes. The first (the inner indexes), of length the number of rows plus one, contains the starting index for the elements of each row. The last element is the index of "one off" the end of the matrix (in the same spirit of the end() iterator). The second vector of indexes (the outer indexes), of length the number of non-zeroes, contains the corresponding column index. Finally, we have the vector of values, again of length the number of non-zeroes. To be more specific, the elements of row i are stored in the elements of the vector of values in the interval $\text{inner}(i) \leq k < \text{inner}(i + 1)$ (the interval is open on the right) and the corresponding column index is $\text{outer}(k)$. Using this scheme, we have a row-wise storage, since traversing the vector of values provides the non-zero elements "by row".

CSC Compressed Sparse Column. As before, but with the role of row and column exchanged. The matrix is thus stored column-wise.

MSR (Modified Sparse Row) and the companion MSC (Modified Sparse Column). They are a modification of CSR and CSC, respectively, that provides an extra gain in memory, and also allows extracting the principal diagonal of the matrix easily. But it can be applied only to square matrices.

For an illustration of CSR and CSC formats, you may have a look at Wikipedia or the book of Y. Saad [1], available freely at this site.
We have also some variants. For instance, in the CSR (or CSC) we may impose that for each row (column) the elements are stored with increasing columns (row) index. This requirement is not present in the original version of the storage technique, but it can help improve cache performance. Another variant is to have instead for each row (or column) the diagonal elements first. This way, it is easy to extract the diagonal part (but it makes sense only for square matrices, and when having the diagonal handy is important).
Often a sparse matrix is built so that one may switch between a dynamic format (useful during matrix construction) to a compressed format (more efficient for arithmetic operations ... if you do things correctly of course). The `Eigen` matrices have this capability.

2 The matrix class

Build a dynamic matrix class template called Matrix<T,StorageOrder>, in the namespace `algebra`, that takes as template parameter the type of the elements and an enumerator the indicates the storage ordering (row-wise or column-wise). The matrix should have these characteristics:

- Be capable of dynamic construction. If the matrix is in an uncompressed state, you should have a method that takes two integers and inserts or changes the corresponding value, and handles the case where the element is not already present. The map is ordered

so that you can easily extract a whole row (a whole column in case of column ordered matrices). Remember that a standard array has a defaulted comparison operator that operates lexicographically using less-than. So if you use a std::map<std::array<int,2>,double> for storing in a COO-type format, you have by default a row-major ordering for the matrix when traversing the map (why?). If you want column-major ordering it is sufficient to change the comparison operator.

- Have a compress() method that converts the uncompressed storage to a compresses sparse row (or columns, depending on the matrix ordering). Think on how to perform the conversion nicely, maybe you want to do it in parallel using parallel algorithms. When the matrix is compressed, the storage used for the dynamic format should be cleared to avoid waste of memory.
- It is nice also to have an uncompress() method that brings back the matrix to the uncompressed state, while emptying the vectors used for the compressed format.
- It must have a method bool is_compressed() that does what it says.
- The call operator that returns the elements of the matrix should behave as expected, for both the const and non-const version.
- The class has a constructor that may take the size of the matrix, and a method to resize a given matrix (leaving it in an uncompressed state).
- And now the nice part: you add a friend operator for the multiplication of the matrix with a std::vector<T>.
- (extra) If you are brave you can implement matrix times matrix. Is a bit tricky if you want to do it efficiently for all cases. There is an example in the pacs-examples repository (I let you to find which) where I experiment all possible ways. You may take inspiration.

Let's describe an important linear algebra operation: matrix-vector product, and its efficient implementation.
The efficient implementation of the matrix-vector product depends on the storage type. What we want is a "cache friendly" computation where we traverse the matrix elements in the order they are stored in memory, which is of course different in the two cases.
Let $A \in \mathbb{R}^{m\times n}$ be our sparse matrix, and let's indicate with $\mathbf{r}_i \in \mathbb{R}^n$ and $\mathbf{c}_j \in \mathbb{R}^m$ the rows and columns of A, i.e.

$$A = \begin{bmatrix} \mathbf{c}_1 & \dots & \mathbf{c}_n \end{bmatrix} = \begin{bmatrix} \mathbf{r}_1^T \\ \vdots \\ \mathbf{r}_m^T \end{bmatrix}$$

Let $\mathbf{v} \in \mathbb{R}^n$ be a vector.

Row-wise multiplication If the matrix is stored row-wise, the most efficient way (and in fact the only reasonable way) to perfom $A\mathbf{v}$ is with the classical algorithm: row-times-vector:

$$[A\mathbf{v}]_i = \mathbf{r}_i^T \mathbf{v}, \quad 0 < i < m-1.$$

With a CSR storage, it can be implemented with a double loop, the inner loop being the one over the non-zero elements of row i. The matrix elements are traversed in their natural order (think about it!).

Column-wise multiplication If we have column-wise storage the classical way is extremely inefficient. But it is sufficient to recall that matrix-vector multiplication is equivalent to perform a linear combination of the columns (indeed this is the profound meaning of the operation in linear algebra). We have,

$$A\mathbf{v} = \sum_{j=0}^{n-1} v_j \mathbf{c}_j^T .$$

Consequently, if I have a CSC format I can implement the matrix-vector multiplication as a linear combination of the columns of the matrix, and again I will traverse the elements of the matrix in the natural order (think about it!).
An important note: If the matrix is still in the uncompressed state, the simplest way to perform matrix-vector product is to create a zero vector of the right size, traverse the map and add the result of the multiplication in the right place (I do not give more details since it is up to you to think about it). No major difference of the algorithm for the two storage orderings. We take into account the storage ordering in the std::map just to make the "compression" operation more efficient (think about it). You may also decide that matrix-vector multiplication is possible only if the matrix is in a compressed state.

2.1 The challenge

1. Code the matrix as described;

2. Implement $A\mathbf{v}$, where $\mathbf{v}$ is a std::vector<T>. The operator should be able to perform the computation if the matrix is in a compressed and uncompressed state, and use the correct algorithm according to the storage ordering, as explained above. Make sure that the operator returns the correct type whether the matrix and/or the vector store arithmetic types or std::complex<T> types.

3. Implement (as a method or as a friend function) a reader of matrices written in *matrix market format*, see math.nist.gov/MatrixMarket/ (you may consider just the case of general real matrices).

4. Test your code:
   read the matrix in math.nist.gov/MatrixMarket/data/Harwell-Boeing/lns/lnsp_131.html.
   Generate a vector of the right dimension and try to time the matrix-vector for the compressed and uncompressed case, row-major and column-major storage (you may use the Chrono.hpp utility, if you use directly the one of the Examples remember to link with the libpacs.so library). Remember to activate optimization when compiling. What you should find is that the use of compressed format speeds up computation.

5. Overload the matrix vector operator to accept also as vector a Matrix with just one column.

6. Implement a template method norm() that takes as template value an enumerator that may be One, Infinity or Frobenius. According to the value of the a template parameter value, it computes the corresponding norm:

$$\|A\|_1 = \max_j \sum_i |A_{ij}|, \quad \|A\|_\infty = \max_i \sum_j |A_{ij}|, \quad \|A\|_F = \sqrt{\sum_i \sum_j |A_{ij}|^2}$$

   Find the best way of doing it. And make sure it may work even if A has complex coefficients.

7. (Extra) Write working views for diagonal and transpose matrix.

General rules

- We will use GitHub classroom as in the previous challenge;
- The files you put in the git repo should include header and source files, a Makefile, a README file with a description. But **NOT** executable, binary libraries or object files. Those should be regenerated by the Makefile.
- If you refer to stuff in the `Examples/` directory of the course, use the environmental variable `PACS_ROOT` to indicate it in the Makefile, to simplify things. Or, copy the needed files from the Examples.
- You may look at the Examples of the course for inspiration, but please follow your ideas.

Evaluation criteria

The evaluation will be based on the following requirements:

- README file, comments in the code, documentation;
- Quality of the code;
- Matrix implementation;
- Implementation of the matrix-vector product;
- Test of the performance of the implementation, e.g. Chrono utility;
- Possible extras:
  1. Extension of the matrix-vector product to Matrix with just one column;
  2. Matrix-matrix multiplication;
  3. Implementation of the method norm<>();
  4. Parallelization using standard algorithms or use of the blas library.

References

[1] Yousef Saad. *Iterative methods for sparse linear systems.* SIAM, 2003.