CMOR 420/520, Homework #3: LaTeX Submission

bi3

November 18, 2023

Discussion

1

SparseMatrixCSR represents a sparse matrix by only storing nonzero values in its structure. In addition to these values, two arrays store the indices of the values, one for the row indices and the other for the column indices. The column indices are listed in the same order as the values appear in the matrix from left to right, top to bottom. The row indices are stored differently to conserve memory. For an mxn matrix, the array R containing the indices has $m+1$ entries, where the ith row of the matrix has R[$i+1$] - R[i] entries. As a result of this ordering, R[0] = 0 and R[$m+1$] will equal the number of elements in the matrix.

2

I think it depends on the size of the matrix. If the matrix has less columns than rows, then using the CSC structure would lead to a storage of column values that would be smaller in size compared to using the CSR structure in this case. Given that, I would expect CSC to be faster. Conversely, if the matrix has less rows than columns, I expect CSR to be faster for similar reasons.

3

Value array: [8, 3, 7, 1, 5, 8, 6, 1, 9]
Column array: [0, 1, 0, 2, 3, 0, 2, 2, 3]
Row array: [0, 2, 5, 7, 7, 9]

Compilation

```
make
./main
```

Implementation

For the allocating version, the following operators were implemented to calculate Vectors:

```
Vector & operator=(const Vector & copy_from);

Vector & operator+=(Vector x);

Vector operator*(AbstractMatrix & A, Vector & x);

Vector operator-(const Vector & x, const Vector & y);
```

For the non-allocating version, the following operators were used. The below operators were implemented in the Vector class, avoiding the creation of new Vectors.

```
Vector & operator=(const Vector & copy_from);

Vector & operator*=(double s);

Vector & operator+=(Vector x);

Vector & operator-=(Vector x);
```

Verification and Timing

Below is the output produced by main.cpp. The code was implemented in allocating and non-allocating versions.

```
********Allocating version********
DENSE MATRIX
Iteration 0, norm(r) = 7.14143
Iteration 18341, norm(r) = 0.000999652
Elapsed time: 3.392 seconds
Average time per iteration: 0.000184941 seconds

SPARSE MATRIX
```

```
Iteration 0, norm(r) = 7.14143
Iteration 18341 , norm(r) = 0.000999652
Elapsed time: 0.251 seconds
Average time per iteration: 1.36852e-05 seconds


******Non-allocating version******
DENSE MATRIX
Iteration 0, norm(r) = 7.14143
Iteration 18341, norm(r) = 0.000999652
Elapsed time: 2.746 seconds
Average time per iteration: 0.000149719 seconds

SPARSE MATRIX
Iteration 0, norm(r) = 7.14143
Iteration 18341, norm(r) = 0.000999652
Elapsed time: 0.247 seconds
Average time per iteration: 1.34671e-05 seconds
```

From the above output, we can see that there is a notable difference in computation between the allocating and non-allocating versions of the calculations for dense matrices. However, the difference is much smaller for sparse matrices, and given the fluctuations between these two, it is hard to say whether the allocating or non-allocating version is better for performing operations with sparse matrices. In either case, using sparse matrices is much more preferable compared to dense matrices when possible, given the above computation times.