# Creation of Sparse Matrices in MATLAB using GNU's stdlibc++

Viktor Wase

March 2016

## 1 Introduction

The sparse matrices in MATLAB are simplest to create using MATLAB's own *sparse* function. Unfortunately this function fails to take make use of the fact that many computers have multiple cores nowadays. A parallel approach would be preferable.

The main focus of this short paper is to investigate the efficiency and speed up that comes from using the parallel sorting algorithms found in GNU Standard C++ Library, or libstdc++ for short.

All code is written in C++ and built using MATLAB's mex compiler. The parallelism is provided by the OpenMP library.

## 2 Compressed Column Storage

One way of storing the sparse matrix is in the Compressed Column Storage (CCS) format **?**. Then three vectors are required: the value vector, the row index vector and the column pointer vector.

The value vector and row index vector contains $N$ elements, where $N$ is the number of non-zeros elements in the matrix to be stored. Each of the elements in both of these vectors gives the information of the row and actual value of an element. The vectors are assumed to be sorted primarily according to the column of each element and secondarily according to the row.
The column pointer vector is of length $M$, where $M$ is the number of columns in the matrix. It contains the index of the first element of each column.

## 3 Problem Description

The problem at hand is to turn the given input data into the three vectors that define CCS. The input data is also given on the form of three vectors: the row and column vectors and the value vector. The *i*:th element in these vectors define the an element in the matrix using its row number, column number and the actual value of the element.

The problem is then one of sorting these elements. However there might be several elements in the vectors that point to be same combination of row and column. In that case, the value of these elements are to be added.

## 4 Algorithm

The algorithm will now be described:

Step 1: Sort according to column using the multiway mergesort from GNU's parallel library. This algorithm is an extension of the commonly used mergesort, which allows for better parallel performance than the original.

Step 2: Iterate through the matrix to find where the each column begins. This information is stored in the set up the column vector. This is done in serial.

Step 3: Each column is sorted according to the row number of each element. Since the sorting of the columns is independent of each other this can be done in parallel using OpenMP's *parallel for(dynamic)* command. The ordinary quicksort method is used for the sorting.

Step 4: Since the vectors are now sorted (according to their column primarily and row secondarily), all elements which are to be summed together stand next to each other. It is therefore easy to go through the columns in parallel and count the number of these multiple occurrences.

Step 5: Create a cumulative sum of the number of multiple occurrences in each column and subtract this vector from the column vector.

Step 6: Prepare the value vector and the row index vector for the CCS matrix. In parallel, go through each column and sum the multiple occurrences and copy the result for each column to the new vectors.

Step 7: Combine the value vector, column vector and row index vector to create the sparse matrix by using the mex functions *mxCreateSparse*, *mxSetJc*, *mxSetPr* and *mxSetIr*.

## 5 Experiments

In order to measure the speedup of this algorithm and compare it to MATLAB's own sparse method 3 experiments were run. In each of the experiments random matrices were created. These matrices were defined by the parameters *siz*, *nnzrow* and *nrep*, where *siz* is the number of rows and columns in the matrix, *nnzrow* defines how many non-zeros there are in each row, and each nonzero entry results from about *nrep* collisions.

The row, column and value vector for each of these matrices were given as input to the new algorithm and MATLAB's *sparse* respectively. The execution time for both of these were measured, and each experiment were repeated 40 times to avoid variance.

The parameter settings for the three experiments were:

Experiment 1: *siz*=10000 *nnzrow*=50 and *nrep*=50

Experiment 2: *siz*=50000 *nnzrow*=50 and *nrep*=10

Experiment 3: *siz*=50000 *nnzrow*=10 and *nrep*=50

## 6 Results

## 7 Conclusions