CITS3402 Assignment 1 2019: Sparse Matrices

Nicholas Pritchard

August 2019

1 Details

- Due Date: 25th September 2019
- Worth: 25%

This assignment is designed to provide students with experience implementing some fundamental mathematical codes. You are encouraged to read further than the provided notes on sparse matrices in order to complete the assignment. Please set aside enough time to complete the assignment as all functionality will require time to test, analyse and report upon.

Your ability to investigate code-performance is of more interest to us than your code's performance. (We care more about what you do, not your machine).

2 Description

Matrix algebra underpins many compute intense applications in many fields (most fields of Engineering, Machine Learning, Scientific Modelling etc.); computers were originally used to crunch numbers so numbers we shall crunch.

Your goal, broadly speaking, is to:

- Create a piece of software implementing a variety of sparse matrix operations
- Ensure this software conforms to our input/output specification
- Performance test your software and comment on any speedup you observe (or lack thereof) in a brief report.

3 Sparse Matrix Introduction

3.1 What is a Matrix

A mathematical matrix is a rectangular array of numbers arranged in rows and columns. For simplicity we use the convention of (rows, columns) when giving dimensions. An example of a 3×2 matrix (read 'three by two') is

$$A = \begin{bmatrix} 2 & 6 \\ 2 & -3 \\ 0 & 1 \end{bmatrix} \tag{1}$$

In many practical cases involving matrix algebra, many elements will contain zero as their value. Storing all elements at all times wastes a potentially vast amount of memory in these cases. For a given machine this heavily restricts the size of problem able to be computed and so sparse matrices were developed to save space at the cost of more time. Loosely speaking, a matrix is considered 'sparse' if it contains $\approx 10\%$ non-zero elements.

The general idea is to save space by storing fewer zero elements (since they generally do not matter in any calculation). There are three broadly accepted sparse matrix representations used:

3.2 Coordinate Format (COO)

The most intuitive sparse matrix format specifies each non-zero element as a triple

$$A_{i,j} = (i, j, \texttt{val}) \tag{2}$$

For example, the matrix,

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \tag{3}$$

becomes

$$X = [(0, 2, 1), (1, 0, 3), (1, 2, 2)] \tag{4}$$

This representation while simple can be cumbersome when wanting to look for all values in a particular row or column.

3.3 Compressed Sparse Row Format (CSR)

The compressed sparse row format stores a matrix with three arrays:

- NNZ: The non-zero values stored in row-major order (left to right, top to bottom)
- IA: The number of elements in each row. An extra element $IA[0] = 0$ is used by convention. This array can be used to index into the NNZ array for each i-th row

- JA: Stores the column index of each non-zero element.

For example, the matrix,

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \tag{5}$$

becomes

$$NNZ = [1, 3, 2] \tag{6}$$
$$IA = [0, 1, 3, 3] \tag{7}$$
$$JA = [2, 0, 2] \tag{8}$$

3.4 Compressed Sparse Column Format (CSC)

The compressed sparse column format is very similar to CSR format except the non-zero values are stored in a column-wise ordering, the IA array corresponds to the extent of each column and the JA array provides row indices. For example, the matrix,

$$X = \begin{bmatrix} 0 & 0 & 1 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \tag{9}$$

becomes

$$NNZ = [3, 1, 2] \tag{10}$$
$$IA = [0, 1, 1, 3] \tag{11}$$
$$JA = [1, 0, 1] \tag{12}$$

4 Tasks

Your code will be a simple command-line application that will:

- Read in up to two dense matrix files
- Convert this matrix (these matrices) to a suitable sparse format.
- Perform a single matrix algebra routine specified by command-line
- Time the whole process and log the results to a file
- Cleanup all memory usage and terminate

We elaborate on the tasks required of you (and your code) below:

4.1 Reading in Files

Depending on the algebra operation requested one or two matrices will be required. These should be provided by command-line as `.in` files (see Section B).

4.2 Sparse Matrix Representation

You have a choice of using any of the three sparse matrix representations presented in Section 3. There will be pros and cons to each representation depending on what operation is requested; you will need to comment on this in your report.

4.3 Matrix Algebra Routines

We request five different operations to be implemented. They range in difficulty (and are presented in roughly increasing difficulty). Marks are distribution accordingly.

4.3.1 Scalar Multiplication

Type: Single matrix
Command-line argument: -sm %f——d
For a matrix A and scalar (float) α compute:

$$Y = \alpha \cdot A \tag{13}$$

i.e. multiply each element of A by α.
The command line argument expects the scalar to be supplied immediately after (either an int or float). You only need to consider a floating point scalar when using floating-point matrices.

4.3.2 Trace

Type: Single matrix
Command-line argument: -tr
The trace(t) of a matrix A is given as:

$$t = \sum_{i=1,n} (A_{i,i}) \tag{14}$$

i.e. the sum of each diagonal element. Note: the Trace is only well defined for square matrices.

4.3.3 Matrix Addition

Type: Double matrix
Command-line argument: -ad
For two matrices A and B (of identical dimensions (n, m) their pair-wise sum is:

$$Y_{i,j} = A_{i,j} + B_{i,j} \forall i, j | i \leq n, j \leq m \tag{15}$$

i.e. add each element of both matrices together.

4.3.4 Transpose

Type: Single matrix
Command-line argument: -ts
The transpose of matrix A (denoted A') is given by:

$$A'_{j,i} = A_{i,j} \forall i \leq n, j \leq m \tag{16}$$

i.e. The rows of A become the columns of A' and vice-versa for the columns.

4.3.5 Matrix Multiplication

Type: Double matrix
Command-line argument: -mm

Note: Sparse matrix multiplication is a notorious bottle-neck in many codes; achieving speedup will be difficult and that is okay. Focus on implementing a simple, correct solution and working from there.

4.4 Timing and Logging

Your code is required to time the following events (in seconds)

- The time to load in and convert any matrix files.
- Time to execute the requested operation

This information must be logged to a `.out` file with the a header containing the following information:

- Operation requested
- File1
- File2 (if needed)
- Number of threads used

Followed by the result of your computation (single value or dense-matrix depending on the operation).

4.5 Report

Your report should be fairly brief but covers the following:

- The sparse matrix representation(s) used
- For each matrix operation implemented:
  - Description of the parallelism implemented
  - Some informal reasoning about expected run-time and scalability
  - Testing results
  - Comment on the performance observed

We expect the report to be around six pages (including figures).

4.6 Restrictions

We place a number of restrictions on the type of matrix your solution will encounter and the design of your solution. They are briefly summarised below as a reminder

- Input File format - See App. B for a detailed description of the matrix file-type we use.
- Log-File format - See App. D for an example. Some further clarifications:
  - Operation requested - Provide the command line argument fed to the program as the label
  - Input Filenames - Provide the command line argument fed to the program as the label
  - Number of threads - Given as an integer
  - Output filename - Provide the date and time of execution plus your student number(s) `.out` as the filename

  - Computation result - Finally, output the result of the requested operation on the provided matrix (matrices). *Floating point results will be tested up to* $1e-6$ *difference*
  - We provide example output files for all types of operations for reference

- Matrix Datatype
  - Integers
  - Floats
  - You will never be required to perform operations involving matrices of different data-types

- Command-Line Arguments - See App. C for a detailed description. We guarantee that we will test your solution with arguments presented in the order described and will only provide valid command line arguments.

- Input matrices - We provide a number of guarantees in our test matrices
  - All command line arguments provided will be valid
  - All input files will be well-formed, valid and of a single data-type
  - The dimensions of tested matrices will range in powers of two in the range $[2, 16, 384]$
  - Input matrices are not guaranteed to be square
  - Input matrices are not guaranteed to be the correct dimensions for the requested operation (e.g. we may ask for the Trace of a non-square matrix or addition of two matrices differing in dimension(s))

Marking Rubric

	Critera	Highly Satisfactory	Satisfactory	Unsatisfactory
File Reading (5)	Successfully reads in matrix files in the format specified. Implements arguments as specified.	Reads in matrix files correctly.	Attempts to read matrix files and attempts to implement command line arguments.	Fails to read matrix files and does not implement correct command line arguments.
Sparse matrix representation (10)	Implements suitable sparse matrix formats and coverts from dense to sparse formats correctly.	Uses appropriate sparse matrix formats and converts from dense to sparse formats correctly.	Attempts to convert from dense to sparse formats but is not always correct.	Does not convert from dense to sparse matrix formats correctly.
Scalar multiplication (5)	Able to multiply both integer and floating point sparse matrices by a floating point scalar.	Is able to multiply both integer and floating point maitrices by a floating point scalar correctly.	Is able to multiply one type of sparse matrix by one type of scalar correctly.	Is unable to multipy a sparse matrix by a scalar.
Trace (5)	Computes the trace of both integer and floating point sparse matrices.	Can compute the trace of integer and floating point matrices correctly.	Computes the trace of one type of sparse matrix correctly.	Is unable to compute the trace of a sparse matrix.
Matrix addition (10)	Correctly adds two sparse matrices together. Exits if not possible.	Adds two integer or two floating point matrices together correctly. Exits gracefully if addition is not possible.	Correctly adds sparse matrices together. Does not exit gracefully if not possible.	Is unable to add two sparse matrices together correctly.
Transpose (10)	Computes the transpose of integer and floating point sparse matrices	Computes the transpose of integer and floating point matrices correctly.	Computes the transpose of one type of sparse matrix correctly.	In unable to compute the transpose of a sparse matrix correcly.
Matrix multiplication (15)	Correctly multiplies two matrices together. Exits gracefully if not possible	Correctly multiplies two integer or two floating point matrices. Exits gracefully if multiplication is not possible.	Attempts to multiply two sparse matrices together. Does not exit gracefully if not possible.	Is unable to multiply two sparse matrices together.

Log file format (10)	Output files conform to the assignment specification	Output files contain the correct header and content in all cases.	Produces output files containing most information. Some minor formatting issues present.	Does not write correct content to output files.
Report (30)	Addresses all required points and is presented clearly. Testing is thorough and clear.	Report addresses all points effectively and is presented clearly. Performance testing is thorough and meaningful.	Report does not address all points required or some minor readiblity issues.	Report addresses few points or is very difficult to read.

A Some words of advice

Writing, testing and reporting on a piece of threaded software will feel slightly different to your previous projects. Here are some humbly offered tips to get started:

- Start early - Said with every project but is especially true here. You will need time to test your final solution
- Practice good code management - Write your code in multiple files, comment as you go, use a good text editor and a debugger etc. Make life easy for yourself
- On sparse matrices - Start by thinking carefully about how you will manage your data separate to the matrix functions
- On matrix functions - Focus on correctness to begin with. A 'sub-optimal' solution that exists will be more useful to you than a fantastic solution that you have not finished. Get something simple working, then make it fast.
- On command-line-arguments and log files - Read the specification carefully and clear up any possible confusions early.
- On the report - We are most concerned with your testing method and results. Again, start early while writing your code.

Finally, feel free to ask the lab demonstrators and lecturer for advice if you get stuck.

B Dense Matrix File Format

There exist many sophisticated methods to store matrices in industry. For the sake of simplicity however we will use a plain-text representation of the data. Our format is quite simple

- Datatype: "int" or "float"
- Number of rows: An integer $n > 0$
- Number of columns: An integer $m > 0$
- $n \times m$ space-separated integers/floats representing each value

For example `int1.in`

```
int
4
4
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
```

This file represents the identity matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

For an example of a float matrix `float1.in`

```
float
4
4
1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.5 0.0 0.0 0.0 0.0 0.75
```

This file represents the following matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1.0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.75 \end{bmatrix} \tag{18}$$

C Command Line Arguments Glossary

To give direction in how to design your solution we require that your solution implements specific command line arguments (CLAs). You are allowed to implement more than these but these are a required minimum.

- Several operators, determines what matrix operation will be performed (have been discussed previously)
  - `-sc` - Scalar multiplication
  - `-tr` - Trace
  - `-ad` - Addition
  - `-ts` - Transpose
  - `-mm` - Matrix multiplication
- `-f %s (%s)` depending on the operation requested, one or more matrix files will need to be passed
  - Example: `./mysolution.exe -sc -f matrix1.in`
  - Example: `./mysolution.exe -mm -f matrix1.in matrix2.in`
- `-l` Log. If present, results will be logged to file

Again, you are allowed to add extra command line arguments but for full marks, our specified options *must* be implemented.

D Log-file Format

In addition to the specified command line arguments we also have specific requirements for the `.out` files your solution must produce. The following file would be the result of calling

```
./mysolution.exe -tr -f int1.in -t 4
```

at 11 : 59pm on the 22nd of August.
Filename: `21726929_22082019_2359_tr.out`

```
tr
int1.in
4
4
```

The following file would be the result of calling

```
./mysolution.exe -mm -f float1.in float2.in -t 4
```

at 12 : 00am on the 23rd of August.
Filename: `21726929_23082019_0000_mm.out`

```
mm
float1.in
float2.in
4
float
4
4
0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.25 0.0 0.0 0.0 0.0 0.375
```

We have included a number of other example output files for your interest.