*Nicholas Cannon 22241579*

*University of Western Australia*

*CITS3402: Project 1*

Sparse Matrices

Table of Contents

[Machine & Operating System Specifications 1](#_Toc20330723)

[Hardware specifications 1](#_Toc20330724)

[Software specifications 2](#_Toc20330725)

[Timing 2](#_Toc20330726)

[Sparse Matrix Representations 2](#_Toc20330727)

[Parsing Coordinate Format (COO) 2](#_Toc20330728)

[Parsing Compressed Row Format (CSR) 3](#_Toc20330729)

[Comparing COO and CSR 4](#_Toc20330730)

[Matrix Algebra Routines 5](#_Toc20330731)

[Scalar Multiplication 5](#_Toc20330732)

[Parallelism 5](#_Toc20330733)

[Expected Runtime and Scalability 5](#_Toc20330734)

[Testing 5](#_Toc20330735)

[Trace 6](#_Toc20330736)

[Parallelism 6](#_Toc20330737)

[Expected Runtime and Scalability 6](#_Toc20330738)

[Testing 6](#_Toc20330739)

[Addition 7](#_Toc20330740)

[Parallelism 7](#_Toc20330741)

[Expected Runtime and Scalability 7](#_Toc20330742)

[Testing 7](#_Toc20330743)

[Transpose 8](#_Toc20330744)

[Parallelism 8](#_Toc20330745)

[Expected Runtime and Scalability 8](#_Toc20330746)

[Testing 8](#_Toc20330747)

[Matrix Multiplication 9](#_Toc20330748)

[Parallelism 9](#_Toc20330749)

[Expected runtime and Scalability 9](#_Toc20330750)

[Testing 9](#_Toc20330751)

# Machine & Operating System Specifications

## Hardware specifications

The machine used to conduct these tests was a 2019 15-inch MacBook pro. The exact hardware specifications are:

* 2.6 GHz Intel Core i7 processor
  + 1 processor
  + 6 cores
* 16 GB 2400 MHz DDR4 RAM
* Hyper-threading enabled
* Cache
  + 256 KB L2 cache (one per core)
  + 12 MB L3 cache

## Software specifications

The application was written with the C programming language and compile with the GCC version 9.2.0 compiler (threading model = POSIX, target = x86\_64-apple-darwin18). All threading was implemented using the OpenMP library. My MacBook Pro is running OSX Mojave (10.14.5).

## Timing

All timing operations are represented as the total wall time of the operation, that is we start the timer before the function is called and stop it after it returns and calculate the elapsed time in seconds.

# Sparse Matrix Representations

The ‘parseMatrix.c’ file is responsible for parsing all sparse matrices and converting them to an appropriate compressed format for computation. Currently the application supports two compressed formats; the coordinate frame and compressed sparse row formats. These are implemented as two separate structs in the ‘parseMatrix.h’ header file. The use of structs allowed me to organize all relevant information required for each matrix in a single “object” like fashion. Each compressed format struct has a corresponding entry struct that represents a single non-zero entry in the sparse matrix. The compressed sparse struct holds an array of memory addresses to a generic entry struct called an entry base. When loading in a sparse matrix, the type of the matrix is read from the ‘.in’ file and stored in the struct, this then allows the application to dynamically allocate memory for either a float or integer entry struct and cast between a generic entry struct and the corresponding type struct.

## Parsing Coordinate Format (COO)

Parsing the sparse matrix into the coordinate format (COO) is simple, first the relevant header information is read from the sparse matrix file and stored in the struct for later use in relevant calculations (basically meta information on the sparse matrix). The application then processes the sparse matrix data line and tokenizes it processing each number individually. Depending on the type read from the previous lines the number is converted to it’s appropriate numerical format and checked if it’s non-zero. Upon finding a non-zero entry, an entry struct is dynamically created and added to the compressed struct. Below is a chart comparing the load times between float and integer matrices using the same COO matrix parser (Note 1=4x4, 2=64x64, 3=128x128, 5=256x256, 5=1024x1024).

## Parsing Compressed Row Format (CSR)

The CSR parser works in a very similar way to the COO parser. It first extracts the header information form the matrix file and sets up the compressed struct. The parser then processes the matrix data line the same was as COO by tokenizing it and converting the tokenized value to the appropriate format. If the value is a non-zero it dynamically creates a CS entry struct and stores the required information for that entry (the JA value). The application tracks the number of non-zero elements it’s seen for that row and when the column counter wraps around it updates the IA array for that row. Below is the exact same graph as above showing average load times for the CSR parser (Note 1=4x4, 2=64x64, 3=128x128, 5=256x256, 5=1024x1024).

From both of the charts above we can see that parsing floating point matrices is slightly costlier than parsing integer matrices (regardless of parser) which makes reasonable sense as we are dealing with finer precision at the floating point level and dealing with more characters from the matrix file. This data was collected by taking the average of 20 samples of the full wall time of the loading alone (that is excluding an algebra operations it was going to do).

## Comparing COO and CSR

When comparing COO and CSR we can see that the CSR format holds 3 separate arrays (NNZ, IA and JA) in which two (NNZ, JA) are dynamically sized. They are dynamically sized because the length of each is dependent on the number of non-zero elements in the matrix. Comparing this to COO we only have one dynamically sized array NZ. This means that for the CSR parser we have twice as much memory management as we do with COO parser. The graphs below compare the average load time for the 1024x1024 integer and float matrices.

From the graphs above we can see that managing fewer dynamic arrays with less memory management is what gives COO better performance against the CSR format. In terms of space complexity COO’s memory footprint is lower than CSR’s. COO’s space complexity can be defined as O(3n) where n is the number of non-zero elements in the matrix. CSR on the other hand has a space complexity of O(2n + m + 1) where n is the number of non-zero elements in the array and m is the number of rows in the sparse matrix. The IA array is the problem in this case as it holds redundant information across rows with no non-zeroes.

# Matrix Algebra Routines

## Scalar Multiplication

### Parallelism

The parallelism implemented in the scalar multiplication is quite simple. The approach parallelised the for loop that is responsible for looping through the non-zero elements and multiplying them by the scalar. This parallelisation is possible because of the lack of dependencies between each of the non-zero elements allowing multiple threads to work on a section of contiguous data without issues.

### Expected Runtime and Scalability

The non-parallel method requires each non-zero entry of the matrix to be multiplied by the scalar and stored in an identical answer matrix giving a time complexity of O(n) where n is the number of non-zero entries in the matrix (this is possible because 0\*x = 0 where x is any scalar). Parallelising this approach we can divide the expected complexity by the number of threads, for example if we have 4 threads then out expected complexity would be O(n/4). In general we can see that scalar multiplication has a linear time complexity and only scales when the number of non-zeroes increases.

### Testing

From graph above we can see that there’s not really a large difference between floating point matrices and integer matrices when performing scalar multiplication as they are both processing the same number of non-zero elements. The minor discrepancy would be because of the extra precision floating point numbers are calculating to. I think the overall performance is reasonably good, there’s no way to process n elements without a linear time complexity. Implementing the multi-threading approach slowed it down by a few microseconds which I think is due to the added logic and complexity of a multi-threaded program (e.g. barrier implementation).

## Trace

### Parallelism

The trace operation works by summing together all of the diagonal elements in the matrix. The parallel approach I took here was to simple reduce the value of each diagonal element of the matrix. The function loops through all the non-zero elements, checks if it is a diagonal element (an element where the row and column index are equal) and then adds that elements value to a sum variable. Each thread works on its own contiguous block of non-zero elements and summing only the diagonal elements values.

### Expected Runtime and Scalability

The expected runtime of this function would be linear and can be described as O(n) where n is the number of non-zero elements. This is faster than going through a sparse matrix of size n x n and summing the diagonals. Converting to the compressed format is what really gives us our speed and the multi-threading gives us an optimization on top of that.

### Testing

From the graph above we can see pretty much identical performance between matrix data types, which is to be expected as we are operating on the same number of non-zero elements. With regards to performance I think it works quite well. It checks all the non-zero entries and is able to skip some operations due to non-zeroes not being diagonals. The worst case performance would be when we trace an identity matrix giving us a time complexity of O(m) where m is the number of rows (or columns as it’s square). In the average case we would get a complexity of O(n) where n is the number of diagonal non-zero entries.

## Addition

### Parallelism

The addition function has the least parallelism used out of all the functions and in some cases none at all. The function is designed similar to the merge function in the merge sort algorithm. The reason the main loop is not parallelised is because of its undetermined loop count, it’s implemented as a while loop that breaks when one of the matrices has ran out of non-zero elements (that is all its non-zero elements have been put into the answer matrix by either direct insertion or addition with a corresponding non-zero element in the other matrix). It’s undetermined how many non-zero elements in both matrices have matching positions and hence can be added together. The function starts at the first non-zero element of both matrices and compares the position of both, if they are the same then add them together and insert into the new matrix and increment to the next non-zero of each matrix, if however they are not the same then we insert the non-zero element with the lower row order position than the other and increment the counter and so on. The parallel aspect comes in when we break out of the loop, we must add in the remainder of the elements (that is if one matrix run out of non-zeroes before the other). Now we know how much we must loop and can therefore allocate enough memory before we start looping and then split this work amongst several threads. Each thread can safely index into the correct memory address as all memory has been allocated unlike the while loop when we allocate memory as we need it.

### Expected Runtime and Scalability

In the worst case this function will run in O(nm) time complexity where n is the number of non-zero elements in matrix 1 and m is the number of non-zero elements in matrix 2 and none of them overlap. For example say matrix 1 has non-zero elements in every odd index and matrix 2 has non-zero elements in every even index the while loop will add them into the answer matrix one after the other, thankfully we are dealing with sparse matrices so this worst case should never happen. Best case is we only have a few overlap that require addition and then we insert until one maxes out and we then parallelise the rest of the insertions.

### Testing

An interesting point to take away from this graph is how well it performs with smaller matrices, the operation time for a 256x256 matrix is very similar to the operation time for the trace function. Given that this function has much less parallel code, there is less logic (no barriers etc) and thread management needed. This clearly shows how the added complexity of parallel code can sometimes be a downside when working on smaller tasks, that is the simplicity of single threaded code is more performant than the added complexity that comes with managing multiple threads. It’s a different story when we operate on a 1024x1024 matrix as this is our costliest operation yet!

I’d like to see how this implementation goes against a more parallelised version. I’m interested to see if parallelising this process more has better performance. When writing this I was very much locked into a sequential mindset (as I was just trying to get a working solution). Compared to a non-compressed format addition there is an obvious performance boost with this implementation as we don’t have to add redundant zeroes together (or to another non-zero entry).

## Transpose

### Parallelism

The transposition function exploits the simple idea that we can flip the row and column indexes of each non-zero element to transpose the entire matrix. This is as simple as looping through all the non-zero elements and exchanging the row and col variables in the COO entry struct. This can then be simple parallelised with OpenMP to share contiguous non-zero elements with each thread.

### Expected Runtime and Scalability

This has the same expected runtime as the scalar multiplication as we are manipulating each non-zero element therefore it has a linear time complexity O(n) where n is the number of non-zero elements. Again this can be divided depending on the number of threads we have working (excluding any extra time needed to manage / barrier threads).

### Testing

Comparing transposition operation time to other operations we can see that it is one of the cheapest operation so far. Comparing the average time with the scalar multiplication we can see that they have very similar average times which is to be expected as they both have very similar time complexities but the transposition function has 3 operations per loop and scalar multiplication has only 1.

I’m very happy with the performance of this function, it’s more than reasonable when compared to something like matrix addition. It’s quite easy to parallelise as we know exactly how many loop iterations we must do in order to complete the tasks and each iteration has zero dependencies on any other iterations making for a very clean parallel process.

## Matrix Multiplication

### Parallelism

Unfortunately I was unable to parallelise my implementation of the matrix multiplication operation, I tried several strategies though but was unable to remove the loop dependencies. I was able to successfully parallelise a certain type of matrix multiplication because I was able to figure out the number of non-zero elements in the resultant matrix before actually doing the multiplication. For matrix multiplication where the resultant matrix was the same dimensions as the first matrix I could see that the number of non-zero elements between the first matrix and the answer matrix was the same (even the IA array was the same!). This made the process of parallelising the matrix multiplication easy as we could pre allocate all the memory needed to store the non-zero elements for the resultant array (instead of allocating on the fly), this removed a counter variable loop dependency. Know that we know how many non-zeroes are present and we know the IA of both matrices is the same we can then independently calculate the non-zero value and insert it into the NNZ array in the correct position. Unfortunately this only works for a specific case of matrix multiplication and therefore my implementation does not know much information about the resultant matrix before the actual multiplication and hence not able to implement a multithreaded solution with the compressed matrix formats.

### Expected runtime and Scalability

The current single threaded solution loops through all rows (that is the number of actual rows in the matrix), checks if there are any non-zeroes then loops through all columns of that row checking if that column has any non-zeroes and finally computing the dot product of that row and column. It takes advantage of the fact that if there exists no non-zeroes in either the row or column then the dot product is 0 and we can skip the calculation. Inside both for loops we have a while loop that calculates the dot product of the row and column given that the non-zeroes can be in any position in the vectors (possible resulting in a dot product of 0!). The time complexity of this function is complex but in the case of an average sparse matrix multiplication we can express the complexity as O(nm\*(a + b)) where n is the number of rows with non-zero elements in the first matrix, m is the number of columns with non-zero elements in the second matrix and a and b would be the average number of non-zeroes per row in the first matrix and the average number of non-zeroes per column in the second matrix respectively.

### Testing

As we can see in the graph below this is definitely the most expensive operation to compute (not surprisingly). Looking at the time between the 256x256 matrix multiplication and the 1024x1024 we can see a very large difference (larger than our other operations) so I decided to analyse the average times of all floating point matrices. From this analysis I calculated a Pearson’s correlation of 0.987 which tells us that we have a strong positive linear relationship between the size of the matrix multiplication and the matrix multiplication compute time. I also calculated the slope of this data which equalled 0.0021 seconds per increase in matrix size (this can explain why our 1024x1024 calculation time is much larger than our 256x256 calculation time). Keep in mind that we are working with limited samples of matrix sizes, there is quite a large gap between 256x256 and 1024x1024 but we can still confidently say there exists a linear relationship. See graphs below for further details.