Sparse Matrices Report

Bruce How 22242664

September 2019

1 Abstract

The motivation behind this project is to understand the fundamentals behind the time and space complexity of algebraic routines on sparse matrices, given different conditions. A major part of this project is understanding the importance of preconditioning, where the use of different preconditioners, which are the different sparse matrix representations; as well as multi-threading and parallelism, can lead to a significant reduction routine computation times.

2 Sparse Matrix Representations

When working with sparse matrices, substantial storage requirement reductions can be realised by storing fewer zero elements as these elements are typically a non-factor when working with most algebraic routines. The three broadly accepted sparse matrix representations outlined in the project brief have all been implemented in this project. These include the Coordinate List (2.1), Compressed Sparse Row (2.2) and Compressed Sparse Column (2.3) representations.

Each of these representations have proved to demonstrate different performance results when used with various matrix algebraic routines and have all been used in this project.

2.1 Coordinate List (COO)

The idea behind the coordinate list (COO) format is that each non-zero element is stored alongside a pair of coordinates as it's name suggests. These coordinates refer to the position in the matrix where the element is located.

COO is represented in this project using a C *struct* which contains an array of element *struct* each with an x, y and *value* variable representing the (x, y) coordinates for the non-zero value respectively. The Union datatype has been used to allow for both int and float data type matrices to be stored under the same structure.

As the input data is given in a row-across format, data converted into the COO format is automatically sorted by the x and then y values respectively.

This makes the COO format a straight forward implementation that allows for easy outputting. The space complexity required for the COO is $O(n)$ where n is the number of non-zero elements. This linear space complexity beats the other 2 representations making COO an ideal representation for simple routines.

2.2 Compressed Sparse Row (CSR)

The Compressed Sparse Row format is similar to COO, but contains compressed row indexes rather than every (x,y) coordinate to allow for faster row access.

The Compressed Sparse Row (CSR) format is represented in this project in a similar manner as the COO, through the use of a C *struct*. This structure contains two integer pointers *IA* and *JA* representing the number of total elements seen so far, and the column index of each non-zero element respectively; as well as an *NNZ* array representing the non-zero elements in the matrix. A union data structure is used for the same reason as mentioned previously.

Processing data for the CSR format was similar similar to COO. The only exception being that extra variables such as the number of total elements seen for each row (IA) had to be recorded. The space complexity required for the CSR is comprised of three parts. $O(n)$ for the NNZ array, $O(n+1)$ for the IA array and $O(n)$ for the JA array and hence a total space complexity of $O(3n+1)$, which for very large matrices, is simply just $O(n)$.

2.3 Compressed Sparse Column (CSC)

The Compressed Sparse Column (CSC) format is very similar to the CSR format, but contains compressed column indexes instead which allows for faster column access.

CSC is represented in a similar manner to the CSR format through the use of a C *struct*. This structure contains the same variables as the CSR format with slight changes to variables *NNZ* and *JA* which represents an array of non-zero element in a column traversal and the row index of each non-zero element respectively.

The processing of data for the CSC format was very different to the other two because the supplied data was given in a row-across column-down format. This means that in order to traverse through the elements in a column-down order, the program would need to skip $n-1$ elements, where n is the number of columns. This proved to be quite inefficient because the same elements had to be traversed through multiple times when processing each column. An alternative efficient solution that was implemented involved storing all elements into a 0 initialised 1D-matrix, where it is then possible to directly access each value in column order. The overall complexity was $O(2nm)$ where n and m are the number of columns and rows respectively. It is easy to see that the constant value 2 becomes a non-factor when dealing with large matrices.

The space complexity required for the CSC representation is the same as the CSR format, $O(n)$, which can be understood intuitively.

3 Algebraic Routines

3.1 Scalar Multiplication

For a matrix A and a scalar (float) α compute:

$$Y = \alpha \cdot A \tag{1}$$

i.e. each element of A is multiplied by α.

3.1.1 Performance

When performing scalar multiplication, each non-zero element must be accessed in order to compute its new value. We can see intuitively that any sparse matrix representation which can allow us to directly access every non-zero element (to perform the calculation), will be the most suitable representation for this routine.

The COO representation allows us to access each non-zero element in the sparse matrix through the *struct element* array pointer. Although COO does not have the ability to directly access non-zero elements in specific locations in constant time (unlike CSR and CSC), the need for such functionality is unnecessary in this routine as we need to traverse through all elements regardless. Hence, if there are n non-zero elements in a sparse matrix, it would take *O(n)* time to traverse through them all.

Similar time performance can be achieved when using CSR and CSC representations. As the order in which each non-zero element is multiplied by a scalar value α, has no effect on the outcome, the *NNZ* variable can be easily traversed to compute the routine. Both CSR and CSC are suitable representations for performing scalar multiplication as they would also take *O(n)* time for traversal.

Table 1 displays the execution time required to perform scalar multiplication using each of the three implemented sparse matrix representation. These values are the average values of 20 trials on the same testing environment (Section 4.0) and are displayed in microseconds (μs).

Format	int64	float64	int1024	float1024
COO	211μs	186μs	402μs	323μs
CSR	201μs	189μs	426μs	387μs
CSC	241μs	181μs	413μs	391μs

Table 1: Scalar multiplication execution times

Due to the similar traversal speeds with all three representation, other factors such as space requirements and time taken to process raw data into each representation were taken into consideration. The COO representation requires less space than the other two representation as it does not keep any row or

column counters.

Format	int64	float64	int1024	float1024
COO	0.914ms	1.44ms	137ms	272ms
CSR	0.773ms	1.43ms	139ms	268ms
CSC	0.807ms	1.68ms	143ms	291ms

Table 2: Data process and conversion times

Table 2 depicts the execution time required to process the raw data and create the respective sparse matrix representation. In each entry, the average value of 20 trials are used. These values unlike the ones used in table 1 are displayed in milliseconds (ms) and are recorded without parallelism.

As the raw data is given to us in a row by row format, the program was required to skip *(row - 1)* elements at each iteration in order to traverse column by column. This was required to produce a CSC representation which proved to have a slightly slower processing speed than the remaining two representations. While both CSR and COO representations demonstrated similar performance in terms of routine execution time (Table 1) and data processing time (Table 2), the simplicity of the COO representation makes it very straight forward to output results. Hence, the choice to use the COO representation over CSR and CSC, for scalar multiplication.

3.1.2 Parallelism

When executing the routine, a loop traverses through each non-zero element and computes its scalar product. Each loop iteration is independent meaning that they can safely execute in any order without any loop-carried dependencies. The following computation is carried out:

$$elements[i] = \alpha \cdot elements[i] \tag{2}$$

The lack of loop-carried dependencies allows us to utilise the

#pragma omp parallel for shared(matrix,scalar) num_threads(param.threads)

directive without any repercussions. This implementation of parallelism results in multiple threads performing scalar multiplication on each portion of the non-zero elements. The use of $shared(matrix, scalar)$ is to ensure that there is proper cache coherence.

The below Table 4 displays the scalar multiplication routine computation time when using different number of threads. The overall computation speed in

this case is faster with lesser threads, and is the fastest when run under single-threading. This pattern may be a result of dominating thread spawning times or even simply due to the simplicity of the computation (scalar multiplication).

We can see that for a small data set, using multiple threads may end up slowing the entire process because the time required to spawn these threads may be longer than just running the program under single thread straight away. However, as we approach a larger data set, we see similar run times (E.g. Single threading and using 4 threads for float1024).

Threads	int64	float64	int1024	float1024
1	55μs	57μs	302μs	323μs
2	168μs	144μs	374μs	364μs
4	211μs	186μs	402μs	323μs
8	293μs	267μs	393μs	468μs

Table 3: Scalar multiplication execution times using multi-threading

To test this, larger data sets were generated with the use of a simple Python3 script. Table 4 displays the scalar multiplication execution time for four larger data sets $int4096$, $float4096$, $int16384$ and $float16384$. The graph below provides a more visual depiction of the speedups acquired from multi-threading.

Threads	int4096	float4096	int16384	float16384
1	6.998ms	4.832ms	96.836ms	137.188ms
2	3.412ms	2.710ms	58.882ms	129.498ms
4	2.394ms	2.643ms	57.213ms	116.104ms
8	2.945ms	2.697ms	61.603ms	75.179ms

Table 4: Scalar multiplication execution times on a large data set

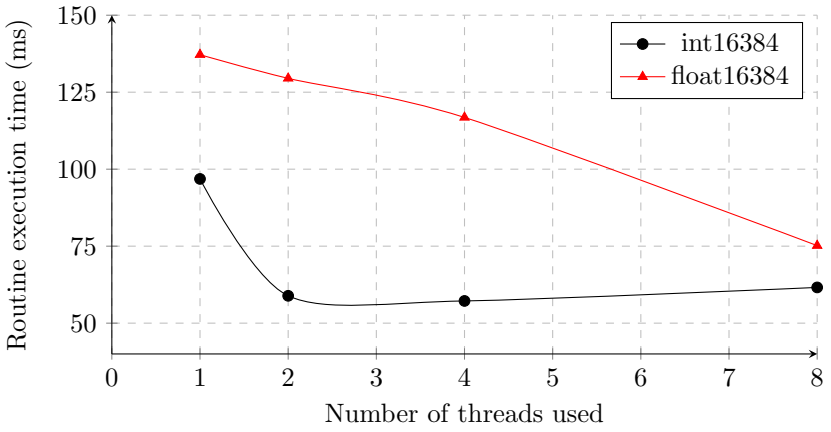

Figure 1: Scalar multiplication multi-threading execution times

There is a clear speed up when performing the scalar multiplication routine with multi-threading over running it in a single thread. From the table above, we can see that the routine is most effective when running with 4 threads for $int16384$ and 8 threads with $float16384$. Unlike the int data set, the float data set does have a peak speed at 4 threads.

3.1.3 Run-time & Scalability

The current scalar multiplication implementation using a COO format requires *O(n)* time where n is the number of non-zero elements as each non-zero element is traversed in order to perform the routine computation. This describes a linear time complexity where the routine's complexity will grow in direct proportion to the number of non-zero elements in a given sparse matrix. This number of non-zero elements in a sparse matrix is approximately equivalent to at least 10% of the total number of elements in the matrix, however this ratio becomes insignificant when n is very large.

Linear time complexity is the theoretically the best time complexity because all n elements must be processed (in this case, every non-zero element). As linear time complexity scales extremely well, the routine is expected to perform consistently even with very large matrices.

3.2 Trace

For a matrix A, the trace t is is given as:

$$t = \sum_{i=1,n} (A_{i,i}) \tag{3}$$

i.e. the sum of each diagonal element, assuming that A is a square matrix

3.2.1 Performance

Unlike scalar multiplication, when calculating the trace of a matrix not every non-zero element must be accessed. Knowing this, a sparse matrix representation that can allow us to quickly identify all diagonal non-zero elements (i.e where $x = y$) while minimising the number of traversals, would be the most ideal representation for this routine.

The COO representation requires us to traverse through every non-zero element due to its lack of index records. This means that given n non-zero elements, we would have to spend *O(n)* time to calculate its trace. To fully understand its shortfalls with access, lets consider an example.

In algorithm 1, the variable p is used as an iterator and must traverse through the entire list of non-zero elements. Even after identifying a diagonal element on row i (line 8), there is no way to identify the location of the next non-zero elements on row *(i+1)* without traversing through all remaining elements on row i (lines 5-6).

Algorithm 1 COO Trace Calculation

```
1:  procedure CalculateTrace(matrix)
2:      p ← 0
3:      trace ← 0
4:      for i ← 1 to rows do
5:          while matrix[p].x < i do
6:              p ← p + 1.
7:          while matrix[p].x = i do
8:              if matrix[p].y = i then
9:                  trace ← trace + matrix[p].val
10:                 p ← p + 1
11:                 break
12:     return trace
```

If we consider the CSR representation, we can use the *IA* and *JA* variables together in order to access the non-zero elements for any row i, without having to traverse through all the non-zero elements. There are two clear benefits for this; firstly, we now know how many elements are there in each row without having to traverse through them using the following equation:

$$elements(i) = matrix.ia[i] - matrix.ia[i-1] \tag{4}$$

Secondly, we can easily identify the starting index within our non-zero element array *NNZ*, for each row i using the following:

$$pos = matrix.ia[i-1] \tag{5}$$

Both equations demonstrate the convenience of the conventional 0 value in the first location of IA as it allows us to generalise the starting index with a single equation. By using these two equations, we can discard the need to traverse through every non-zero element. These two simple heuristics allow us to do the following: If there are no elements or remaining elements on row i we can skip row i (4); and once we have identified whether a non-zero diagonal value in row i exists, we can simply skip the remaining non-zero elements in that row (if any) and proceed to the first element in the next row (5). Hence, the algorithm will be spending at most *O(n/2)* time where n is the number of non-zero elements. In other words, the algorithm beats linear time.

3.2.2 Parallelism

The CSR approach also removes the loop-carried dependency value p, which was required when using a COO representation (Algorithm 1). This ensures that each loop iteration is independent and can be safely execute in any order making parallelism simple to implement. Parallelism is implemented using the following directive

#pragma omp parallel for reduction(+:trace) shared(matrix)

The *num_threads*(*param.threads*) clause has been omitted in the report for the sake of visibility. The reduction clause allows each thread to attain its own local copy of the reduction variable, *trace*. Each thread modifies only its local copy ensuring that there is no data race. The global *trace* variable is then computed by combining each local copy of *trace* from each thread.

Threads	int64	float64	int1024	float1024
1	111μs	61μs	269μs	259μs
2	209μs	163μs	377μs	406μs
4	198μs	239μs	372μs	343μs
8	260μs	288μs	411μs	442μs

Table 5: Trace execution times using multi-threading

Based on Table 5, there is no visible speed up running the trace routine with multi-threading compared to single threading. The results were also highly inconsistent even when taking an average of 20 test cases which may be due to different thread spawning times. As done previously with scalar multiplication, several additional tests were carried out on larger data set as shown in the table and graphs below. 8192x8192 matrices have been used instead of 16384x16384 to speed up the testing process.

Threads	int4096	float4096	int8192	float8192
1	5.414ms	5.166ms	17.989ms	19.002ms
2	3.032ms	3.117ms	11.516ms	12.583ms
4	2.592ms	2.632ms	8.705ms	9.627ms
8	3.283ms	2.957ms	10.797ms	9.726ms

Table 6: Trace execution times on a large data set

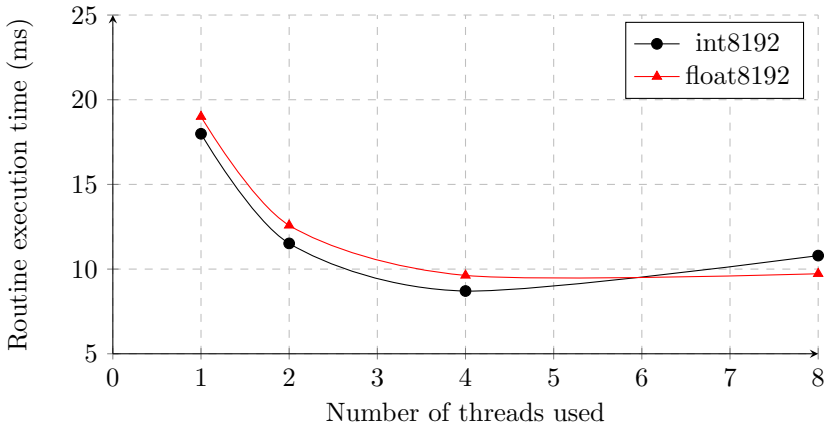

Figure 2: Trace execution times with multi-threading

Speed-ups can be seen in Table 6 and Figure 2 when running with multiple threads over a single thread. Both $int8192$ and $float8192$ execute with the least computation time with 4 threads and start to plateau from this point.

3.2.3 Run-time & Scalability

The trace routine implementation uses a CSR format which requires O(n/2) time where n is the number of non-zero elements. For each row, only a maximum of $c/2$ non-zero elements are traversed where c is the number of columns in the matrix, in order to identify the middle element. This describes a time complexity that beats linear time, $O(n/2)$, where the routine's complexity will grow in direct proportionate (essentially for large enough matrices) to the number of non-zero elements in a given sparse matrix.

This complexity was achieved by using the CSR format and two simple heuristics, as mentioned above. A complexity that beats linear time is extremely fast and scales very well for large values of n. Hence the routine is expected to perform consistently even with very large matrices.

3.3 Matrix Addition

For two matrices A and B (of identical dimensions (n,m)) their pair-wise sum is:

$$Y_{i,j} = A_{i,j} + B_{i,j} \forall i, j | i \leq n, j \leq m \tag{6}$$

i.e. add each element of both matrices together

3.3.1 Performance

As addressed in Section 3.2.1, we can see that two simple heuristics used in conjunction with CSR and CSC can lead to a speed up. We know how many elements there are on each row, and have a constant time access $O(1)$, to the first element of each row. Due to the nature of CSC (column access) and inefficiently of COO (lack of access control), CSR has been used to perform the matrix addition routine. Although the CSC format produces a similar routine performance to the CSR, its processing time is the slowest amongst all three representations (Table 2).

The algorithm to perform this routine is simple. For each row, if there are any matching JA values from both $matrix1$ and $matrix2$, then compute their sum. Otherwise, the resulting value is simply just the value of either $matrix1$ or $matrix2$ (whichever is non-zero).

In the worst case, if every non-zero element in $matrix1$ corresponds to a zero value in $matrix2$ and vice versa, the routine will be executed in $O(n+m)$ time at worst where n and m refers to the number of NNZ elements in $matrix1$ and $matrix2$ respectively.

3.3.2 Parallelism

Modifications to the routine code was required to ensure that parallelism could be implemented. Prior to the modification, the routine would execute a for loop where each iteration would computes each row of the resultant matrix based on the NNZ values from both $matrix1$ and $matrix2$. The issue was that each iteration utilised a counter $result.count$, to figure out the index where the next element should be placed in the $result$ matrix.

The modification to the code involved creating an array where result values could be temporarily stored. This removes the loop carried dependency from $result.count$. The modification allows each iteration to have access to its own counter, $result\_local[i].count$, allowing each iteration to figure out the index to place its next result. At the end of the routine, these local results are merged back into the $result$ matrix and freed from memory. The $result$ matrix is finally returned.

The routine return type had to be changed from CSR to COO due to this. Merging local copies of CSR results proved to be quite inefficient as the IA variable for a row i, depends on its previous row $i-1$. Producing an output in the COO format was simple due to the lack of dependencies for each element. Each

element only required an x and y coordinate which were both easily obtained without any dependencies. Merging local results of type COO was not a difficult task as they were already naturally sorted. The following directives were used in order to produce parallelism in the routine.

#pragma omp parallel for reduction(+:totalcount) shared(matrix,matrix2)

The above specifies that multiple threads should be used and declares both *matrix* and *matrix*2 (the two input matrices) as a shared variable to allow for better cache utilisation. As stated previously, the *num_threads*(*param.threads*) clause has been omitted for the sake of visibility.

The reduction clause allows each thread to attain its own local copy of the reduction variable, *totalcount*. Each thread modifies only its local copy ensuring that there is no data race. The global *totalcount* variable is then computed by combining each local copy of *totalcount* from each thread. This variable is used to figure out the total number of elements that will need to be stored in the *result* COO structure and is used for accurate memory allocation.

Threads	int64	float64	int1024	float1024
1	218μs	229μs	20301μs	19265μs
2	254μs	264μs	22511μs	17342μs
4	368μs	349μs	18285μs	18737μs
8	401μs	378μs	18188μs	19094μs

Table 7: Matrix addition execution times using multi-threading

From Table 11, we can see that there is no clear speedups for any given number of threads. To test this thoroughly, the routine was ran against two large matrices, 8192x8192 and 16384x16384. Table 8 and Figure 3 depicts these results.

Threads	int8192	float8192	int16384	float16384
1	363.352ms	337.125ms	3.176s	4.192s
2	252.512ms	282.802ms	1.521s	2.851s
4	234.191ms	247.974ms	1.274s	2.115s
8	222.103ms	236.086ms	1.179s	1.917s
16	235.771ms	222.840ms	0.755s	1.784s

Table 8: Matrix addition execution times on a large data set

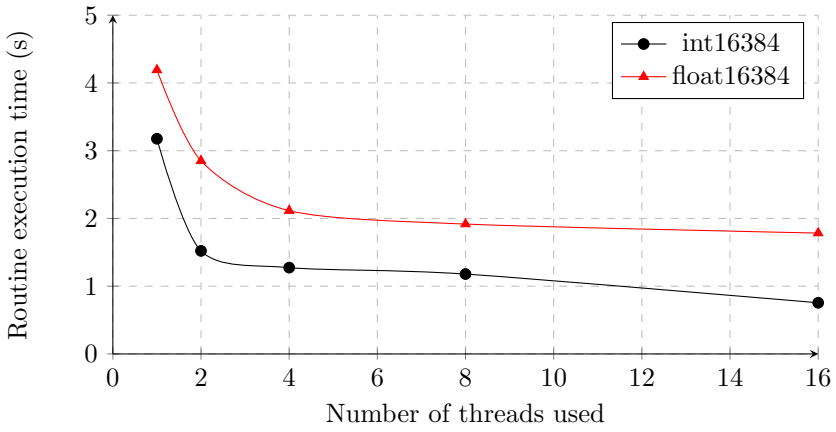

Figure 3: Matrix addition execution times with multi-threading

The matrix addition routine has the lowest execution time when ran with 16 threads (8 threads for int8192). Similar to the other routines when ran under parallelism, the speed-up plateaus after 4 threads. The speed-up is most noticeable when going from single threading to 4 threads. However, in this instance, we can see that the graph plateaus downwards as opposed to upwards after 4 threads.

3.3.3 Run-time & Scalability

The run-time complexity of this routine is made from two parts. The first is to calculate the matrix sum for each row. This result is stored into a *result_local* COO structure and the total time complexity required for this is $O(n + m)$ where n and m are the number of NNZ elements in $matrix1$ and $matrix2$ respectively.

The second part involves putting these local copies together into the final *result* COO structure. The implementation this utilises the C function *memcpy* over the standard assignment operators, which is the most efficient way of copying blocks of memory from one location to another. Each instance of *result_local* from each thread is copied onto the *result* COO structure which takes $O(n+m)$ time in the worst case.

Hence it's run-time complexity is $O(2(n + m))$ and while is not as good as the previous 2 routines, is still linear time. While the complexity is on paper linear time, it is important to note that both n and m scale as a polynomial function to the matrices' dimensions. In order to parallelise the routine, an extra $O(n + m)$ time of pre-computation was required and although multi-threading reduces complexity, running the routine strictly in single threading without the need of additional pre-computation was generally faster for smaller sets of data.

Although run-time may be faster for a smaller data sets when running strictly with a single thread (without the need of pre-computation), for very large ma-

trices the routine will scale very well (in polynomial time) when parallelism is implement (Table 8).

3.4 Transpose

The transpose of matrix A (denoted A') is given by:

$$A'_{j,i} = A_{i,j} \forall i \leq n, j \leq m \tag{7}$$

3.4.1 Performance

By definition, the transpose of matrix $A_{(n,m)}$ (with n rows and m columns) which we will call A', is simply itself with its rows and columns switched. In other words $A' = A_{(m,n)}$. Consider the definition of the CSC and CSR representation, where non-zero values are stored column down and row across (JA) respectively. If we were to transpose any of these representations, then the CSR JA variable would then represent the CSC JA variable and vice versa. In other words, the CSR representation of a matrix A is equivalent to the CSC representation of its transposed matrix A'. Equation 9 demonstrates this.

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} A' = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 0 \\ 1 & 2 & 0 \end{bmatrix} \tag{8}$$

$$JA_{CSR(A)} = JA_{CSC(A')} = [2, 0, 2] \tag{9}$$

Therefore if we were to use the CSC or CSR representation, we can easily compute its transpose by converting its representation accordingly. This would also require $O(n)$ time where n is the number of non-zero elements, to copy the data from one representation to another.

We can see similar performance results on paper with the COO representation where we would traverse through all non-zero elements $O(n)$, and at each iteration, flip their x and y coordinates. This algorithm although seems simple at first, results in an unsorted list of non-zero elements making it difficult to output without proper sorting methods.

3.4.2 Parallelism

This approach of converting a CSR format to CSC naturally has no loop-carried dependencies. This allows us to utilise many of the OpenMP directives without issues. The following directive was used to implement parallelism

#pragma omp parallel for shared(result,matrix)

As mentioned earlier, the $num_{(}param.threads$) clause has been omitted in the report for the sake of visibility. The below Table 9 displays the trace routine computation time when using different number of threads.

Threads	int64	float64	int1024	float1024
1	63μs	56μs	618μs	812μs
2	197μs	133μs	583μs	673μs
4	167μs	245μs	437μs	594μs
8	285μs	291μs	505μs	575μs

Table 9: Transpose execution times using multi-threading

Slight speed-ups can be seen with multiple threads as the data size increases. The results were highly inconsistent at when testing with a the smaller data sets $int64$ and $float64$ which may be due to inconsistent thread spawning times. Several additional tests were carried out on larger data sets as done previously to visualise the effects of multi-threading as shown below (Table 10).

Threads	int4096	float4096	int8192	float8192
1	16.322ms	16.981ms	41.561ms	63.840ms
2	9.884ms	11.551ms	28.322ms	38.077ms
4	7.238ms	8.685ms	19.425ms	29.768ms
8	6.522ms	7.958ms	18.677ms	26.325ms

Table 10: Transpose execution times on a large data set

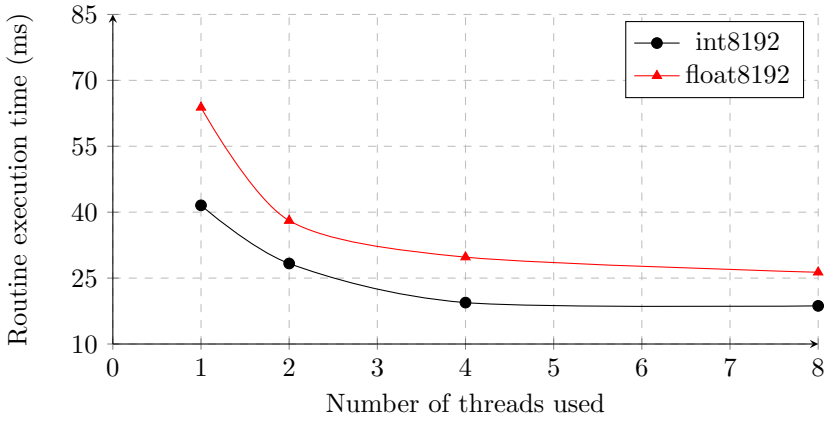

Figure 4: Transpose execution times with multi-threading

There is a significant speed-up with both data sets $int8192$ and $float8192$. Evident from the Figure 4, we can see that the transpose routine starts executing quicker as the number of threads is increased. There is plateau from 4 to 8+ threads where the the speed-up is not as noticeable compared to running with a single thread to 4 threads. This graph carries a similar shape an inverse Natural Logarithm function.

3.4.3 Run-time & Scalability

Conversion from the CSR to CSC format requires $O(n)$ time, where n is the number of non-zero elements, to copy the data from one representation (NNZ_{csc}) to another (NNZ_{csr}). The values of IA and JA from one representation to another are the same (given the transpose routine) as explained above. Hence, the routine will take linear time for execution that will grow in direct proportion to the number of non-zero elements in a given sparse matrix.

As mentioned earlier, linear time complexity is scales extremely well and is expected to perform consistently even with very large matrices.

3.5 Matrix Multiplication

For two matrices $A_{(n,m)}$ and $B_{(m,p)}$ (where $A_{(n,m)}$ denotes n rows and m columns), their matrix product is given by:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j} \forall i \leq n, j \leq p \tag{10}$$

3.5.1 Performance

If matrix multiplication were to be performed on two matrices, A and B, then we can see intuitively that a representation that can allow easy access to each row elements from the matrix A, and each column elements from matrix B, would be and ideal way to implement this routine. The CSR and CSC formats both naturally index elements in a row and column fashion respectively, and when used in conjunction with each other is desirable for matrix multiplication.

Consider yet again two matrices, A and B as shown below. The former is represented is a CSR format while the latter is represented in a CSC format.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \end{bmatrix} B = \begin{bmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{bmatrix} AB = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \end{bmatrix} \tag{11}$$

In equation 11, the result of c_1 would be the dot product $[a_1, a_2, a_3]$ and $[b_1, b_2, b_3]$. As A is represented in a CSR format, the index of the values in the first row of A can be easily accessed with A_{IA}. Similarly, the index of the values in the first column of B is easily accessed with B_{IA}. In the worst case, a dot

product is performed on every element of A against every element of B resulting a time complexity of $O(nmn)$ where n is the number of rows in A and m is the number of columns in B. The definition of matrix multiplication requires that $n == m$, hence a complexity of $O(n^3)$.

3.5.2 Parallelism

Modifications to the routine code was also required to ensure that parallelism could be implemented. The modifications are similar to the matrix addition routine (3.3.2). Prior to the modification, the routine would execute two for loops to perform the required dot product on each row on $matrix1$ and column on $matrix2$ (given that they are non-zero elements). A similar issue to the matrix addition implementation was encountered where the use of a $result.count$ counter created loop dependencies.

The modification to the code involved creating an array where result values could be temporarily stored ($result\_local$). This was created for the same reason as mentioned in 3.3.2.

The following directives were used in order to produce parallelism in the routine.

#pragma omp parallel for reduction(+:totalcount) shared(matrix,matrix2)

Note that the $num\_threads(param.threads)$ clause has been omitted. The above specifies that multiple threads should be used and declares both $matrix$ and $matrix2$ as a shared variable to allow for better cache utilisation.

The reduction clause is used to figure out the total number of elements that will need to be stored in the *result* COO structure and is used for accurate memory allocation.

Threads	int64	float64	int1024	float1024
1	721μs	757μs	2.096s	2.213s
2	669μs	703μs	1.043s	1.327s
4	631μs	635μs	0.788s	0.795s
8	634μs	683μs	0.801s	0.810s

Table 11: Matrix multiplication execution times using multi-threading

From the above data, we can see a clear speedup when running the routine under multi-threading as opposed to single threaded. To examine this behaviour in more depth, the routine was ran against three large matrices, 4096x4096x, 8192x8192 and 16384x16384. The below Table 12 and Figures 5 and 6 depicts these results. Note that due to extremely long execution times with very large data sets, some tests were not completed to full and others have been omitted due to time constraints.

Threads	int4096	int8192	int16384
1	140.584s	1049.960s	>9000s
2	73.334s	609.206s	5481.132
4	58.387s	478.831s	3215.671s
8	54.603s	448.330s	2931.481s
16	56.795s	415.071s	3751.326s

Table 12: Matrix multiplication execution times on a large dataset

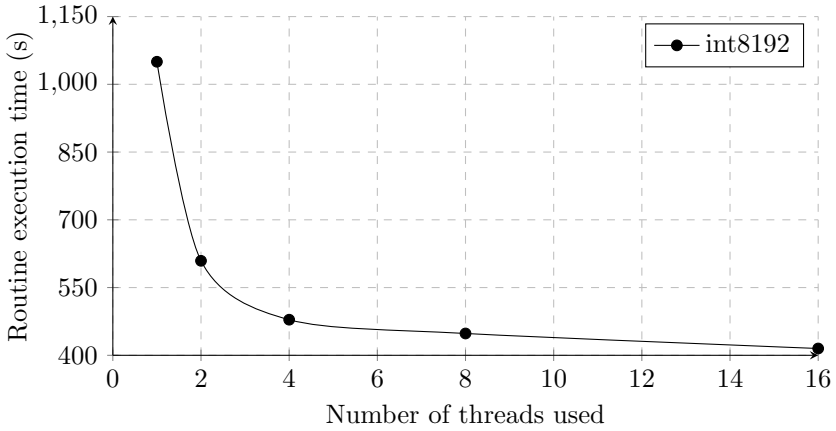

Figure 5: int8192 matrix multiplication execution times with multi-threading

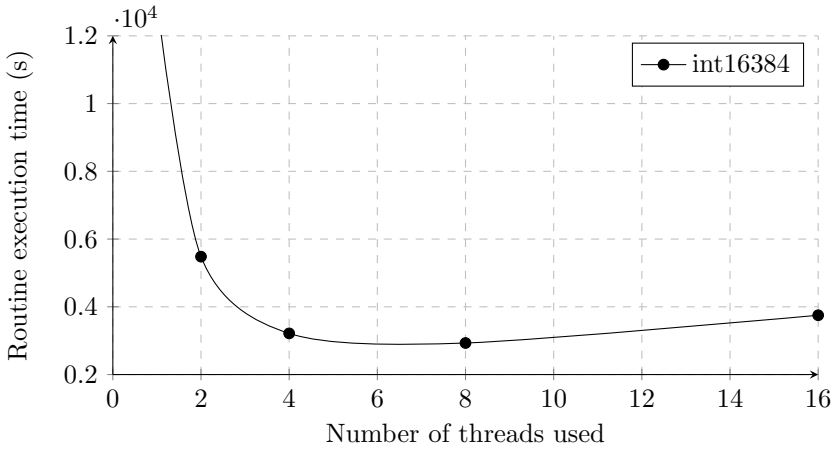

Figure 6: int16384 matrix multiplication execution times with multi-threading

From Figures 5 and 6, it is evident that there is a significant speed-up in the routine execution when running the routine under multi-threading. Similar to the other 3 routines, there is plateau from 4 to 16+ threads where the the speed-up is not as noticeable compared to running with a single thread vs 2 or 4 threads. The routine is at its fastest when run with 16 threads, which is similar to the matrix addition routine.

The difference between running the routine with a single thread as opposed to multiple threads is day and night.

3.5.3 Run-time & Scalability

As mentioned in section 3.5.1, in the worst case, a dot product is performed on every element of A against every element of B resulting a time complexity of $O(nmn)$ where n is the number of rows in A and m is the number of columns in B. The definition of matrix multiplication requires that $n == m$, hence a complexity of $O(n^3)$. The addition of the post-processing code required to enable parallelism requires at worst $O(n^2)$ time in order to copy the local results together. Hence, a total worst case complexity of $O(n^3 + n^2)$. If we consider it's run-time complexity, and the general case that we are only dealing with sparse matrices, then we have a run-time complexity of $\theta(P(sparse)n^3)$ where $P(sparse)$ is the probability of an element being non-zero.

While the time complexity for post-processing required for parallelism may seem huge, $O(n^2)$, the overall routine complexity is dominated by its polynomial term n^3. In other words, the post-processing for parallelism becomes a non-factor when dealing with scaled up against large values of n, but can provide a significant reduction in routine execution times as outlined above (3.5.2). The scalability of this routine comes down to how quickly the program needs to run. In table 12, we can see an increase of (on average) 7 times the execution speed when doubling the matrix dimensions (4096 to 8192 to 16384).

Although the scalability of this naive implementation is not bad, it can be improved by implementing algorithms such as the Strassen and Coppersmith–Winograd algorithms which produce a worst case time complexity of $O(n^{2.807})$ and $O(n^{2.3737})$ respectively.

4 Project Environment

All program results have been run on the same Operating System for consistently. Table 13 outlines some of the hardware specifications used during the testing phase.

Category	Details
Operating System	macOS Mojave 10.14.5
Memory	8GB
Processor Name	Intel Core i7
Processor Speed	2.4 GHz
Num of cores	2

Table 13: Project test environment specs

To aid with the testing process, an additional CLA parameter $-s$ was added. This parameter stands for silent and when present, will output everything except the resulting matrix. This is extremely useful when testing very large matrices where the time taken to print the result matrix is significantly long.

The project output file format is based on the sample output provided with the exception of the different double decimal places.

5 Conclusion

There was much success in reducing sparse matrices routine execution times through the use of proper preconditioners and multi-threading. We saw that different sparse matrix representations and heuristics can lead to minor run-time speed-ups as well as memory reduction. By generalising the results acquired in this report, we can see that there is always a noticeable speed-up with all execution given that the matrix size is large enough.

Much of the Project's research went into the investigation of the use of the OpenMP directive *simd*. This directive when applied to a loop allows multiple iterations of the loop to be executed concurrently using the $SIMD$ instructions. The nature of the matrix multiplication routine makes this directive extremely useful by reducing the number of re-computations performed. I was unfortunately, unsuccessful with implementing this directive.