CITS3402 Assignment 1

Sparse Matrix Operations (SMOPS)

By Alex Brown (21955725)

0 Specifications
0.1.a Computer Specifications for Testing

```
Linux version 4.15.0-64-generic (buildd@lgw01-amd64-038) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)) #73-Ubuntu

SMP Thu Sep 12 13:16:13 UTC 2019
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  2
Core(s) per socket:  2
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               78
Model name:          Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
Stepping:            3
CPU MHz:             500.009
CPU max MHz:         3100.0000
CPU min MHz:         400.0000
BogoMIPS:            5184.00
Virtualisation:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            4096K
NUMA node0 CPU(s):   0-3
```

0.1.b Specifications of the Log File

The log file is saved in the following format in the same directory as where the SMOPS executable is used in terminal.

```
[command tag for operation used]
[input filename a]
[optional input filename b]
[data type of result]
[(not for trace) the number of rows for the result]
[(not for trace) the number of cols for the result]
[the result (either 1D dense matrix or for trace an int or float)]
[wall time for loading in input files]
[wall time for performing the operation (excludes time saving as result)]
```

The reason the wall time was chosen to measure performance is because the sequential implementation of SMOPS uses the same algorithms as the parallel implementation and hence the CPU time would be similar. User time would be the preferred time metric of performance, but due to simplicity and averaging over 10 trials for each input file, wall time was selected.

0.2 Sparse Matrix Representations Used In SMOPS

The following sparse matrix representations were used for the following operations. Further discussion on why each representation was used for each operation will be discussed in later sections.

1. Coordinate Format (COO)
   Where the non-zero elements of a matrix are stored in struct that holds the row, column and value of each non-zero element.

   Operations that use the COO format for Sparse Matrices:
   - Scalar Multiplication (see Section 1)
   - Trace (see Section 2)
   - Transpose (see Section 4)

2. Compressed Sparse Row Format (CSR)
   The non-zero element values are stored in row-major order in the nnz array and the respective column is stored at the same index in the ja array. The ia array is such that to iterate through the elements i^{th} row of the matrix an iteration of the indexes [ia[i], ia[i+1]) would suffice.

   Operations that use the CSR format for Sparse Matrices:
   - Addition (see Section 3)
   - Matrix Multiplication (see Section 5)

3. Compressed Sparse Column Format (CSC)
   Similar to the CSR format but the nnz is order by column-major order, the ja array stores each respective row index and the ia is used for iterating through an entire column instead.

   Operations that use the CSC format for Sparse Matrices:
   - Matrix Multiplication (see Section 5)

0.4 Notes on Report Structure and Source Code

- The report is structured by theorising, testing and analysising the un-omptimised version of SMOPS stored in `smops_unomptimised/src`. This is done to identify issues related to the performance of the operations and attempt to fix them for the final version submitted in the `src/` directory.

- Seeing the load times of the matrices in `smops_unomptimised/test_performance/results/speed_up.txt` (see appendix 1) and the graphs stored in `smops_unomptimised/test_performance/results/graphs/fileload`, the obvious enormous bottleneck of the program is loading in the matrices and formating them to sparse matrix representations. However, taking the strict definition of the marking rubric, the performance of the file reading and converting to sparse matrix formats are not assessed. Therefore, optimising the file input and conversion is one of the lowest priorities for this assignment, even though it is an issue that needs to be fixed.

- One of the major contributors to the file load issue is the use of `quicksort()` to sort the COO format into row-major and column-major order before converting to CSR and CSC format respectively. Quicksort especially reduces the performance of converting format since how the COO format is formed from file it is already in row-major order, meaning that quicksort would then sort in the worst-case scenario of O(n^2) time. Instead the parallel bucket sort algorithim (or another parrallelised sorting algorithim) should be used to optimise the conversion.

1 Scalar Multiplication

1.1 Parallelism Method

The algorithim used for Scalar Multiplication was iterating through the elements stored in the COO format and multiplying by the specified constant. This was parallelised by using the `#pragma omp for` directive. Assuming that no cache misses occur, andconsidering that f is the fraction of common non-parallel sections and using N threads the speed up ratio (S) is:

$$S = \frac{T_1}{T_N} \leq \frac{T_1}{\left(f + \frac{1-f}{N}\right) T_1} = \frac{N}{(N-1)f - 1}$$

A rough estimation for f by analysing the number of source code lines from starting the timing is $f = \frac{5}{6}$. Inputting this back into equation for S will get.

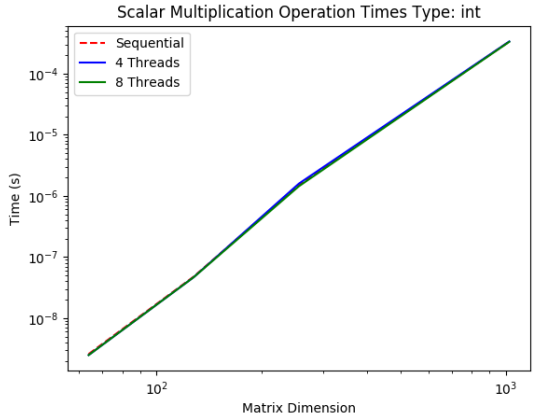
$$S \leq \frac{6N}{5N - 11}$$

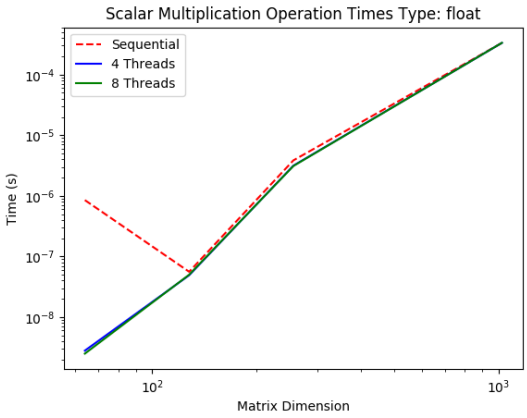
Therefore for 4 and 8 threads, we should expect a maximum speed up ratio of only 2.67 and 1.66 respectively.

However, the implementation of the scalar multiplication does not directly access the memory of the data arrays in the COO format, and may cause a significant number of cache misses reducing the practical speed up ratio.

1.2 Performance Test Results

Below are the matrix dimension compared to time graphs for scalar multiplication using both integer and float data types.





Graph 1 and 2: Minimal to no speed up of parallelising scalar multiplication due to cache misses previously stated.

As shown in Graphs 1 and 2, the speed up ratio for matrices of dimension 1024 are shown in the table below (see Appendix 1 for raw results).

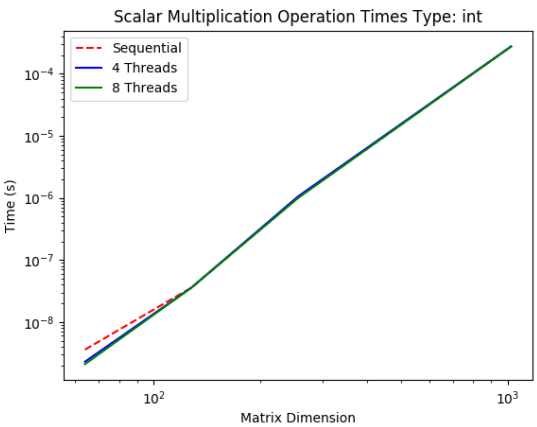
Data Type:	Integer	Float
4 Threads	0.99	1.00
8 Threads	1.01	1.00

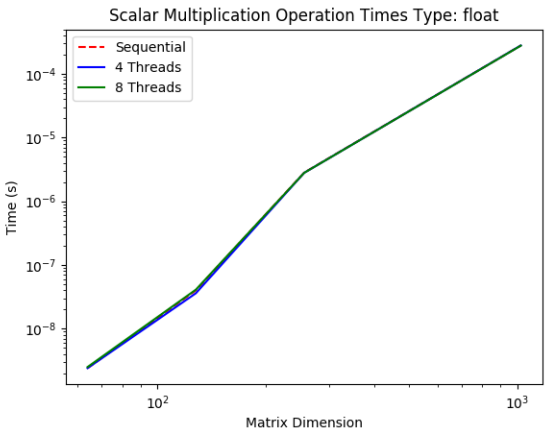
1.3 Result Analysis

The results support the theory that by not having the parallel region having direct access to the data arrays it causes significant cache misses. The obvious improvement is to have the parallel regions to have direct access to these data arrays and minimise the number of cache misses. An alternative to the current implementation would be to directly modify the value of the input matrix instead of copying the data to a result matrix as implemented.

1.4 Improving Results and Conclusions

The unomptimised version for Scalar Multiplication was modified so that the parallel regions have more direct access to the data arrays. The graphs and table below show the new matrix dimension and time graphs and speed up ratios (see appendix 2 for raw data on speed up ratios).





Graph 3 and 4: No speed up ratio just like the results from section 1.2.

Data Type	Integer	Improvement: Integer	Float	Improvement: Float
4 Threads	1.00	1.01	1.00	1.00
8 Threads	1.00	0.99	1.00	1.00

This shows that the analysis in sections 1.1 and 1.3 are not the cause of the no improvement due to parallelising. This then implies that copying the coordinates and results to a seperate matrix may be culprit for causing cache misses and slowing down the program. For a future implementation, it should be tested if directly modifying the input matrix would result in a significant speed up.

2 Trace

2.1 Parallelism Method

For calculating the trace of a matrix, the COO format was used and it would iterate using the `#pragma omp for` directive through the COO representation adding any values to the trace sum if both the row and column are the same.

Using the same formula for the speed up ratio used in section 1.1 and estimating that $f = \frac{4}{5}$ we should expect a maximum speed up of roughly:

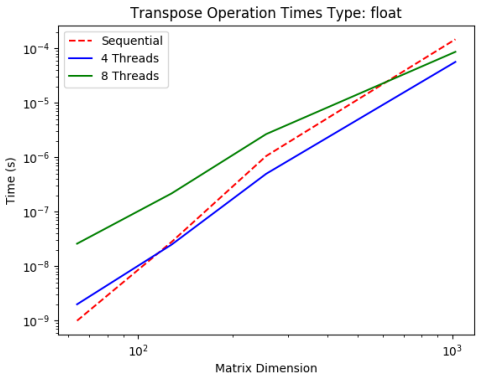
$$S \leq \frac{5N}{4N-9}$$

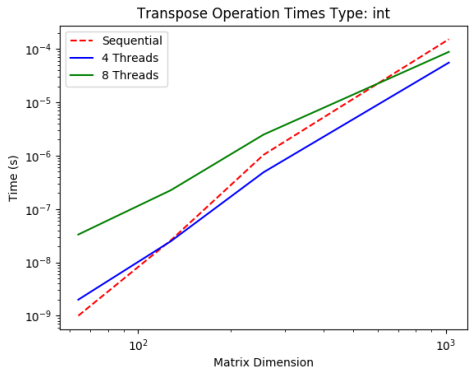
Therefore, for 4 and 8 threads an estimation for the expected maximum speed up ratios should be 2.86 and 1.29 respectively.

Similar to the scalar multiplication implementation, all three arrays are required within the parallel section, which may cause some cache misses. However, some improvements in comparison is that within the parallel section the data arrays are directly accessed and there is no copying of memory into a result matrix. Therefore, less cache misses would be expected and the speed up ratio should be similar to the estimations.

2.2 Performance Test Results

Below are the matrix dimension compared to time graphs for calculating the trace of a matrix using both integer and float data types.





Graph 5 and 6: As matrix dimensions increase the parallelism speed up ration increases.

As shown in Graphs 5 and 6, the speed up ratio for matrices of dimension 1024 are shown in the table below (see Appendix 1 for raw results).

Data Type	Integer	Float
4 Threads	2.74	2.60
8 Threads	1.72	1.70

2.3 Result Analysis and Conclusion

The results support the estimations made on the speed up ratio in section 2.1, with 8 threads even exceeding above its respective estimation for maximum speed up.

However, some improvements can still be made. Assuming that the non zero elements are evenly distributed within the matrix then using the CSR format would halve the number of computations in comparison to the COO implementation. This is can be done because the elements are stored in row-major order and by iterating through a row the trace sum can be increased if the column is the same as the row or skip to the next row if the column of the currently looked at element of the row is greater than the row number.

Since the implementation for trace already had direct access to the data arrays it was left unmodified for the final version of SMOPS. This leaves the only suggestion for improving the speed is by using the CSR format as stated above.

3 Addition

3.1 Parallelism Method

For calculating the addition of two matrices, the CSR format is used for both input matrices. The algorithim of the implementation is that a single thread (using `#pragma omp single`) spawns 2*num_of_rows tasks that add all the non-zero elements into the dense matrix.

Using the same formula from section 1.1 and estimating $f = \frac{7}{9}$ the equation for the maximum speed up ratio is:

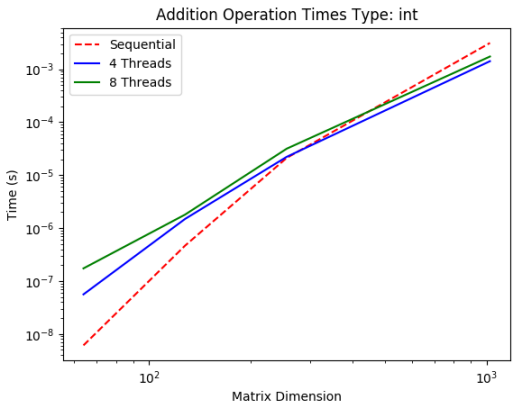
$$S \leq \frac{9N}{7N-16}$$

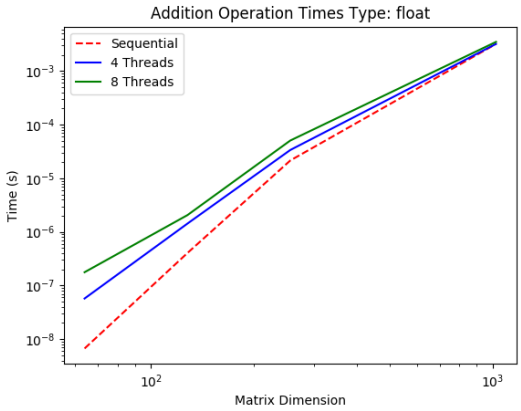
This gives us an estimated speed up ratio for 4 and 8 threads of 3 and 1.8 respectively.

However, the current implementation currently requests memory from each CSR struct, which may cause cache misses and reduce the overall performance of the program.

3.2 Performance Test Results

Below are the matrix dimension compared to time graphs for performing the addition of the same matrix using both integer and float data types.





Graph 7 and 8: Speed up occurs for integer matrices, but for float addition it is slower than the sequential performance

As shown in Graphs 7 and 8, the speed up ratio for matrices of dimension 1024 are shown in the table below (see Appendix 1 for raw results).

Data Type	Integer	Float
4 Threads	2.21	1.0
8 Threads	1.80	0.92

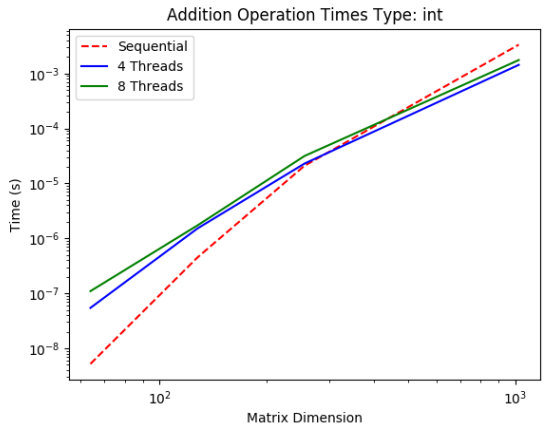
3.3 Result Analysis

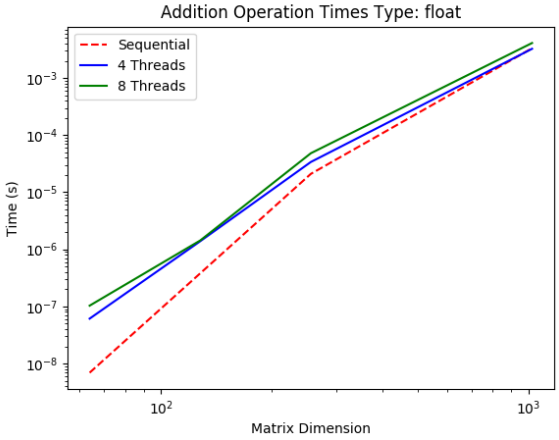
The integer results of testing are similar to the estimations made in section 3.1, however the results for float data types is bizzare. The only explaination is due the double type taking 64 bits of space, the use of the `#pragma omp atomic` directive used to avoid race conditions and that the addition is performed on the same matrix. This would mean that OpenMP is required to prevent these race conditions from occuring with the 2 tasks adding the to the dense matrix and more bytes than the integer type, significantly slowing it done.

A better implementation could be to use two `#pragma omp for` instead to minimise the race condtions instead of using tasks, but this may also cause cache misses and reduce the programs performance as well. Another alternative is to make the second task spawned from the single thread to start from the bottom row first, to reduce the likelihood of race conditions with adding the same matrix together.

3.4 Improving Results and Conclusions

The unomptimised version for Addition was modified so that the parallel regions have more direct access to the data arrays. The graphs and table below show the new matrix dimension and time graphs and speed up ratios (see appendix 2 for raw data on speed up ratios).





Graph 9 and 10: An improvement for the addition of integer type, but same issue for float types.

Data Type	Integer	Improvement: Integer	Float	Improvement: Float
4 Threads	2.33	1.05	1.02	1.02
8 Threads	1.90	1.06	0.81	0.88

The minor change of making sure the parallel regions had direct the access to the data arrays resulted in a small 5-6% performance increase for integer types, but still resulted in minimal speed up for float types (with 8 threads having a 12% performance decrease). The reason for the minimal improvements for float data types is most likely due to the race conditions discussed in section 3.3, and should be a fix for future implementations.

4 Transpose

4.1 Parallelism Method

The algorithm used for the transpose is simply swapping the number of rows and columns for the result matrix and the row and column of each non-zero element. Therefore the COO format was used for this implementation since the coordinates of each non-zero element can be easily be swapped.

Using the formula from section 1.1 and estimating that $f = \frac{5}{7}$ the equation for the maximum speed up ratio is:

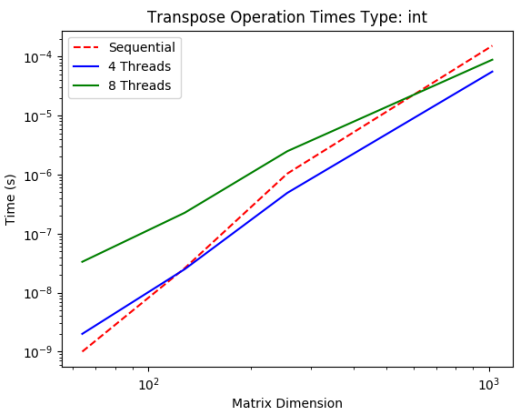
$$S \leq \frac{7N}{5N - 12}$$

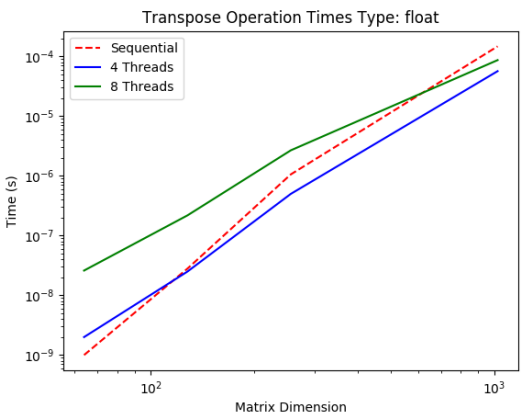
Therefore, the estimated speed up ratio for 4 and 8 threads is 3.5 and 2 respectively.

However, the transpose implementation would suffer from the same cache miss scenario as the scalar multiplication implementation (see section 1.1) due to the arrays of the COO form not being directly modified within the parallel sections. Hence a lower speed up ratio would be expected.

4.2 Performance Test Results

Below are the matrix dimension compared to time graphs for determining the transpose of a matrix using both integer and float data types.





Graph 11 and 12: Both threaded tests speed up faster than the sequential test as dimensions increased.

As shown in Graphs 11 and 12, the speed up ratio for matrices of dimension 1024 are shown in the table below (see Appendix 1 for raw results).

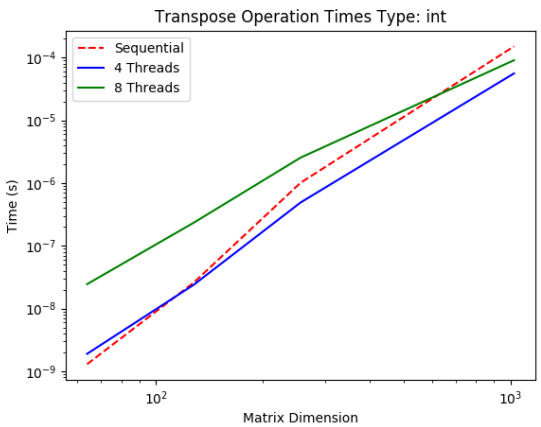
Data Type	Integer	Float
4 Threads	1.72	1.73
8 Threads	1.59	1.63

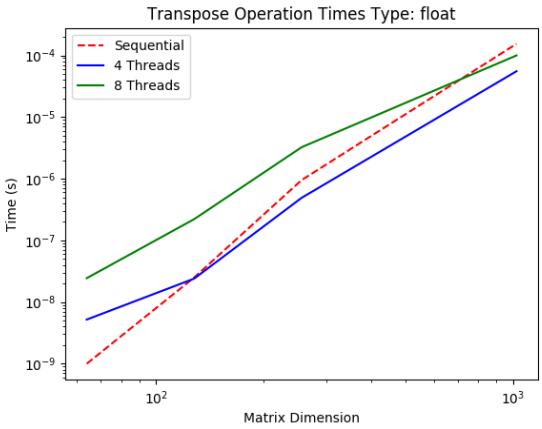
4.3 Result Analysis

As discussed in section 4.1, the condition of not having the parrallel section directly having access to the arrays inside the COO structs causes cache misses and decreases the performance of the operation. However, this is most likely the only cause for the decrease in performance, especially considering that the scalar multiplication had a below zero speed up ratio and had other design considerations that could speed it up (see section 1.3).

4.4 Improving Results and Conclusions

The unomptimised version for Transpose was modified so that the parallel regions have more direct access to the data arrays. The graphs and table below show the new matrix dimension and time graphs and speed up ratios (see appendix 2 for raw data on speed up ratios).





Graph 13 and 14: A very slight improvement for the transpose operation

Data Type	Integer	Improvement: Integer	Float	Improvement: Float
4 Threads	1.74	1.01	1.75	1.01
8 Threads	1.62	1.02	1.64	1.01

However, this change only resulted in a minimal improvement of ~1%. A different implementation of speeding up the program is to directly modify the data in the input matrix, rather than copy the results into a result matrix that increases the likelihood of cache misses.

5 Matrix Multiplication

5.1 Parallelism Method

Matrix multiplication is defined as $C_{ij} = \sum_k A_{ik} B_{kj}$, and can be implemented using the CSR format for the first matrix (A in the equation) and the CSC format for the second matrix (B in the equation). C_{ij} can then be calculated by iterating through the ith row of matrix A and adding to C_{ij} if iterating through jth column of matrix has a non-zero element in column k in A and row k in B. This can be parallelised by tasking, by having a single thread that spawns number_of_rows*number_of_columns tasks and each task performs O(number_of_rows*number_of_columns) calculations.

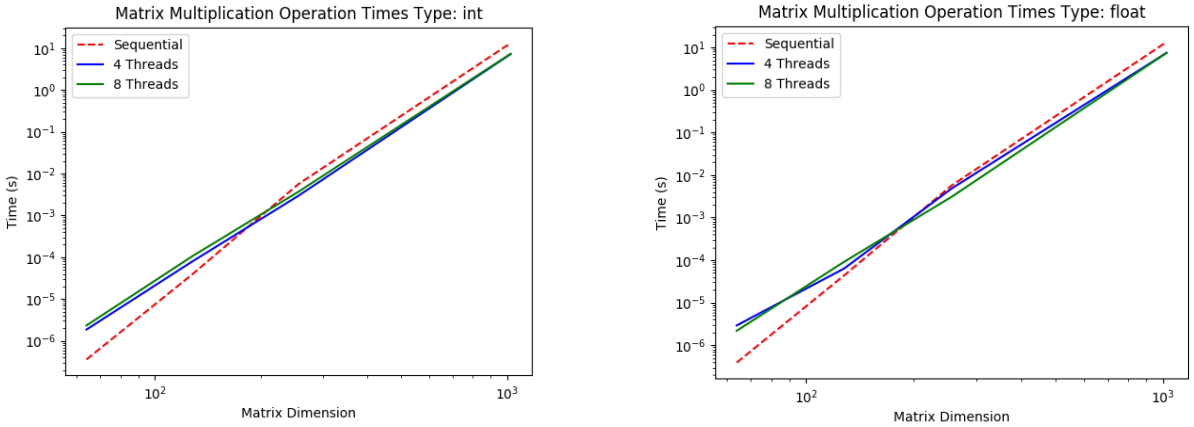
Using the formula described in section 1.1 and estimating that $f = \frac{17}{28}$, the estimated speed up ratio is:

$$S \leq \frac{28N}{17N - 45}$$

Hence, the estimated speed up for 4 and 8 threads is 4.87 and 2.46 respectively. However, the implementation would cause a number of cache misses for two reasons. The first being that the arrays stored in the CSR and CSC format are not directly accessed by each thread and secondly multiple threads may require the same row or column from either matrix A or B simultaneously. Because of this, a lower speed up ratio would be expected from the estimated values.

5.2 Performance Test Results

Below are the matrix dimension compared to time graphs for determining the transpose of a matrix using both integer and float data types.



Graph 15 and 16: Some speed up for matrix multiplication as matrix dimension increases.

As shown in Graphs 15 and 16, the speed up ratio for matrices of dimension 1024 are shown in the table below (see Appendix 1 for raw results).

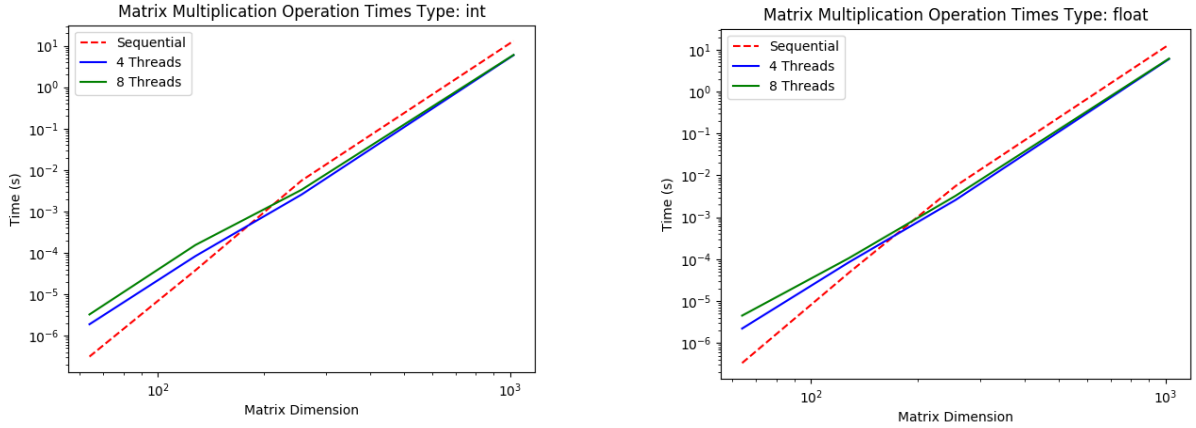
Data Type	Integer	Float
4 Threads	1.84	1.80
8 Threads	1.81	1.80

5.3 Result Analysis

As discussed in section 5.1, the matrix multiplication speed up is bottle-necked by the cache misses caused by multiple threads requiring the same row or column from a matrix. This is unavoidable with the current implementation, and the only method to circumnavigate this is to have a copy of each matrix for each matrix, which is extremely memory inefficient for large matrices. However, an improvement can be made by having each thread have more direct access to the data arrays stored in the CSR and CSC formats, this would reduce a number of cache misses.

5.4 Improving Results and Conclusions

The unomptimised version for Matrix Multiplication was modified so that the parallel regions have more direct access to the data arrays. The graphs and table below show the new matrix dimension and time graphs with speed up ratios (see appendix 2 for raw data on speed up ratios).



Graph 17 and 18: A significant improvement in performance speed

Data Type	Integer	Improvement: Integer	Float	Improvement: Float
4 Threads	2.1	1.14	2.2	1.22
8 Threads	2.18	1.20	2.17	1.21

The 14-22% improvement in performance is a significant change, implying that the fact that the matrix multiplication originally did not have direct access to the data arrays was a bottle-neck for the performance of the operation.

Appendix

1. smops_unoptimised/test_performance/results/speed_up.txt

```
ad float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9983
         8 Thread Speed Up: 0.9157
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9971
         8 Thread Speed Up: 0.9832

ad int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.2093
         8 Thread Speed Up: 1.8038
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9989
         8 Thread Speed Up: 0.9763

mm float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.8018
         8 Thread Speed Up: 1.8051
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9739
         8 Thread Speed Up: 0.9578

mm int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.8372
         8 Thread Speed Up: 1.8114
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9880
         8 Thread Speed Up: 0.9682

sm float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9997
         8 Thread Speed Up: 0.9979
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 1.0267
         8 Thread Speed Up: 1.0144

sm int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9940
         8 Thread Speed Up: 1.0082
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 1.0038
         8 Thread Speed Up: 1.0089

tr float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.6014
         8 Thread Speed Up: 1.6993
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0803
         8 Thread Speed Up: 0.0316

tr int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.7433
         8 Thread Speed Up: 1.7246
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0481
         8 Thread Speed Up: 0.0239

ts float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.7274
         8 Thread Speed Up: 1.6309
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0804
         8 Thread Speed Up: 0.0464

ts int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.7194
         8 Thread Speed Up: 1.5938
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0461
         8 Thread Speed Up: 0.0241
```

2. test_performance/results/speed_up.txt

```
ad float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.0230
         8 Thread Speed Up: 0.8099
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9880
         8 Thread Speed Up: 0.9728

ad int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.3285
         8 Thread Speed Up: 1.9029
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9887
         8 Thread Speed Up: 0.9777

mm float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.2171
         8 Thread Speed Up: 2.1681
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9798
         8 Thread Speed Up: 0.9472

mm int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.2102
         8 Thread Speed Up: 2.1763
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9784
         8 Thread Speed Up: 0.9493

sm float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 0.9982
         8 Thread Speed Up: 1.0047
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 1.0009
         8 Thread Speed Up: 1.0063

sm int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.0004
         8 Thread Speed Up: 1.0029
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 1.0161
         8 Thread Speed Up: 1.0105

tr float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.7812
         8 Thread Speed Up: 1.5426
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0761
         8 Thread Speed Up: 0.0377

tr int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 2.6941
         8 Thread Speed Up: 1.6604
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0473
         8 Thread Speed Up: 0.0254

ts float
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.7467
         8 Thread Speed Up: 1.6395
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0792
         8 Thread Speed Up: 0.0354

ts int
Operation Speed Up for Dimension 1024
         4 Thread Speed Up: 1.7382
         8 Thread Speed Up: 1.6215
Load Speed Up for Dimension 1024
         4 Thread Speed Up: 0.0462
         8 Thread Speed Up: 0.0224
```