CITS3402 Report - 21492318 Paul O'Sullivan

Operating system: OS X 10.11.6.

Matrix formats:

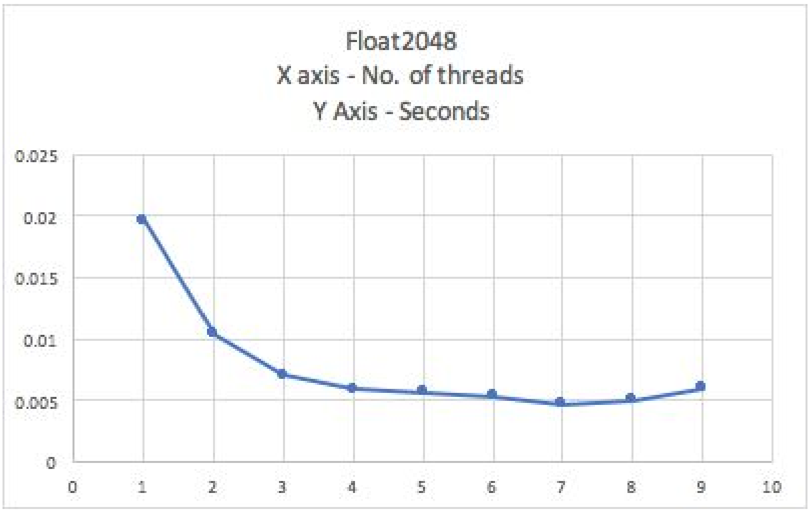
Upon investigation, it was clear that each format contains various comparable advantages. For the purpose of this project, I decided to centre my research around Coordinate Format (COO) and Compressed Sparse Row Format (CSR). Upon reviewing past literature, it became clear that COO can have fast incremental construction as well as fast conversion to CSR (Golubin 2017). However COO may have slow access and slow arithmetics (Golubin 2017). CSR on the other hand has efficient arithmetic operations, slices rows efficiently while also providing quick matrix vector products (Golubin 2017). However some cons of CSR include the fact that column slicing is slow (Golubin 2017). For these reasons, I first implemented the operations "Scalar Multiplication" ,"Addition" and "Trace" using CSR, "Transpose" was implemented using COO while "Multiplication" was implemented using a combination of CSR and COO. However upon attempts to parallelise these operations, I discovered a significant amount of loop carried dependencies within my CSR implementations. Consequently, I decided to implement these operations using COO to reduce any loop carried dependencies and ultimately improve overall performance.

Operations (`omp_get_wtime()` was used to calculate the time taken to perform each task)

Scalar Multiplication

This operation consists of a basic for loop, iterating through the "nnz" array containing all non-zero elements of the matrix. During each iteration, the current element will be multiplied by the scalar inputted by the user.

```
#pragma omp parallel for
   for(int i = 0;
i<nonZeroElements; i++){
    nnzFloatArray[i] =
nnzFloatArray[i]* inputScaler;
```



Parallelising the above for loop results in performance gains. Unfortunately it was difficult to notice any performance gains when testing this using the supplied matrices. Creating a much larger matrix of size 2048^2 allowed for noticeable performance gains to be observed. Such gains can be seen in the chart to the right.

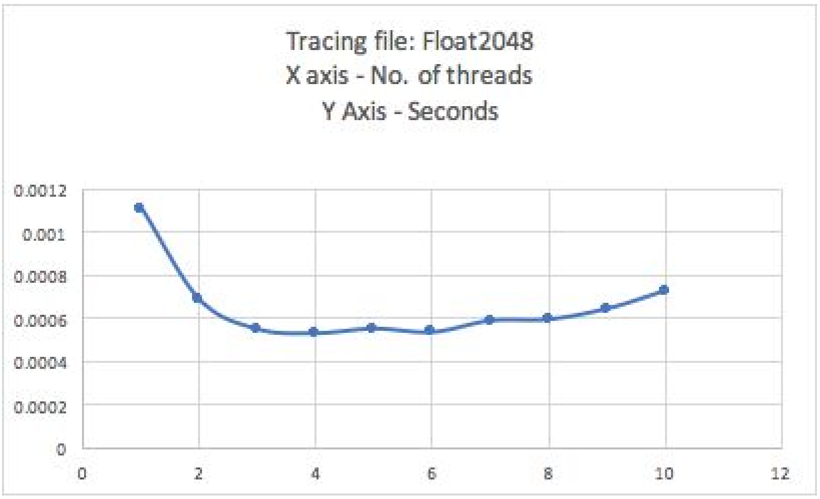
From this chart, we can observe that the fastest achieved run times falls between 4-8 threads. Performance begins to decrease when we go beyond 7 threads, which can be observed by the increasing trend for threads 8 and 9.

The estimated time complexity of this function is O(n) where n is the number of non-zero elements in the supplied matrix. Consequently, this algorithm is scalable as datasets expand.

Trace

This operation was implemented using COO format and also consists of a basic for loop. The idea behind this was to loop through the array and sum cumulative values where the row and column indices matched.

```
#pragma omp parallel for
reduction(+:sum)
   for(int i = 0; i<nonZero; i++){
           if(ja[i] == row[i]){
            sum = sum + nnz_f[i];
           }
```

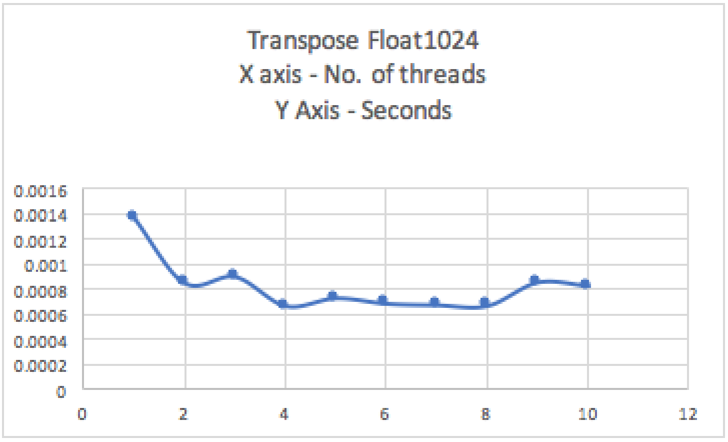


Again, it was difficult to observe any noticeable improvements in performance due to the small size of the largest matrix provided to us. However, it was possible to observe performance gains when using a matrix double the size of the largest matrix provided (float1024.in). Performance analysis using threading can be seen in the above chart.
From this chart we can observe that the fastest run times occur when 6 threads are utilised for parallel processing. Beyond 6 threads results in worsening performance. The time complexity for this operation is O(n) where n is the number of non-zero elements in the dense matrix.

Transpose

This operation was implemented using COO format and also consists of a basic for loop. The idea behind this was to loop through the array and switch the column indices with row indices.

```
#pragma omp parallel for
   for(int i = 0; i<nonZero;i++){
       int temp = column[i];
       column[i] = row[i];
       row[i] = temp;  }
```



Similar to the previous operations, it can be observed in the above chart that the most optimal threading lies around the 4-6 thread range. The time complexity of this operation is O(N) where N is the number of non-zero elements in the input matrix. While the space complexity is O(3N) (row + column + non-zero arrays).

Addition

This operation was first implemented using CSR format. However, it was decided to switch to implementing with COO due to extensive loop carried dependencies when implementing CSR. This implementation consists of three independent for loops which allows for easy parallelisation.
The first loop fills an array with zeros.

```
#pragma omp parallel for
   for(int i =0;i<rowLength*colLength; i++){
           nnzAdded[i] = 0.0;
   }
```

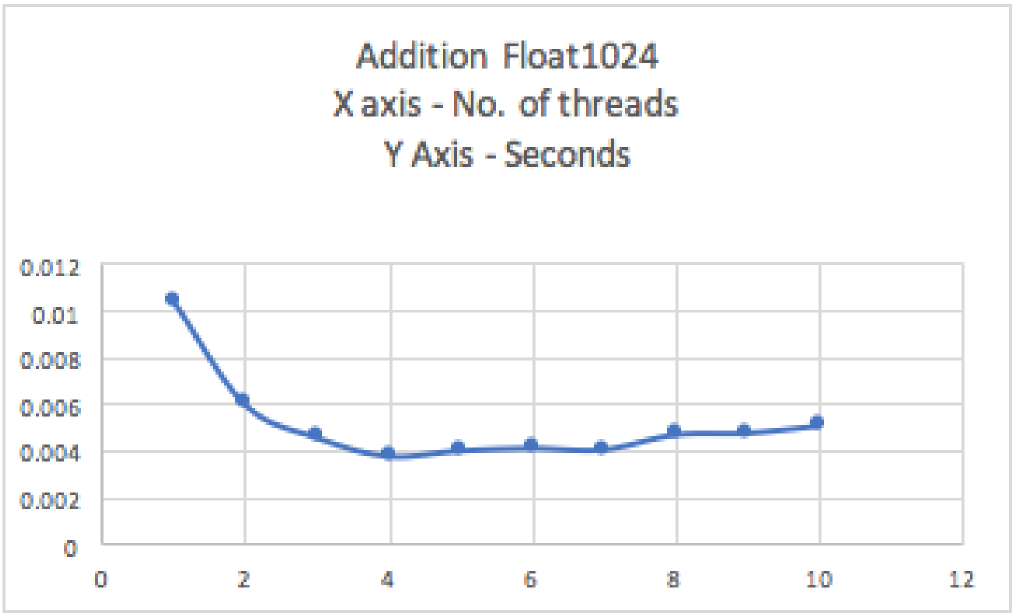
The second loop adds each non zero element in matrix 1 to the value at their position in the matrix.

```
#pragma omp parallel for
   for(int i = 0; i<nonZero; i++){
       int a = row[i]*colLength + col[i];
       nnzAdded[a] = nnz[i] + nnzAdded[a];
   }
```

The third loop adds each non zero element in matrix 2 to the value at their position in the matrix.

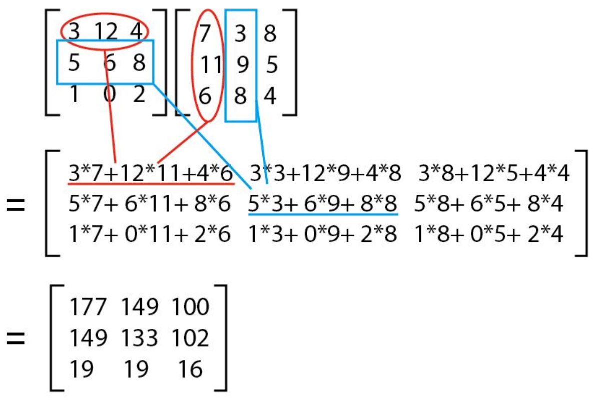
```
#pragma omp parallel for
   for(int i = 0; i<nonZero2; i++){
       int a = row2[i]*colLength + col2[i];
       nnzAdded[a] = nnz2[i] + nnzAdded[a];
   }
```

Similar to previous operations, it can be observed in the below chart that performance is optimal when using 4 - 6 threads. The space and time complexity of this operation is O(nm) where n is the number of rows and m is the number of columns. This could be implemented faster using CSR, with a potential time complexity of O(nD) where n is the number of rows and D is the number of non-zero elements.



Multiplication

This operation was calculated using the row length and non zero elements of matrix 1, and the column length and non zero elements of matrix 2. The implementation outputs in the form of CSR to be used for printing. A visual representation of how this operation works can be seen to the right.



This implementation consists of a quadruple nested loop.

1. Maintains a track of the row indices of matrix 1 represented by "i".
2. Maintains a track of the column indices of matrix 2 represented by "l".
3. Iterates through all of the non-zero elements of matrix 1.
4. Iterates through all of the non-zero elements of matrix 2. Checks to ensure that the row position of the *k'th* element in matrix 1 is equal to the current row index *i*. It checks to ensure that the column position of the *g'th* element in matrix 2 is equal to the current column index *l*. It also checks to ensure that the column position of the *kth* element in matrix 1 matches the row position of the *g'th* element in matrix 2. Once these conditions are satisfied, the *k'th* element of matrix 1 is multiplied by the *g'th* element of matrix 2. This is then cumulatively summed throughout the loop.

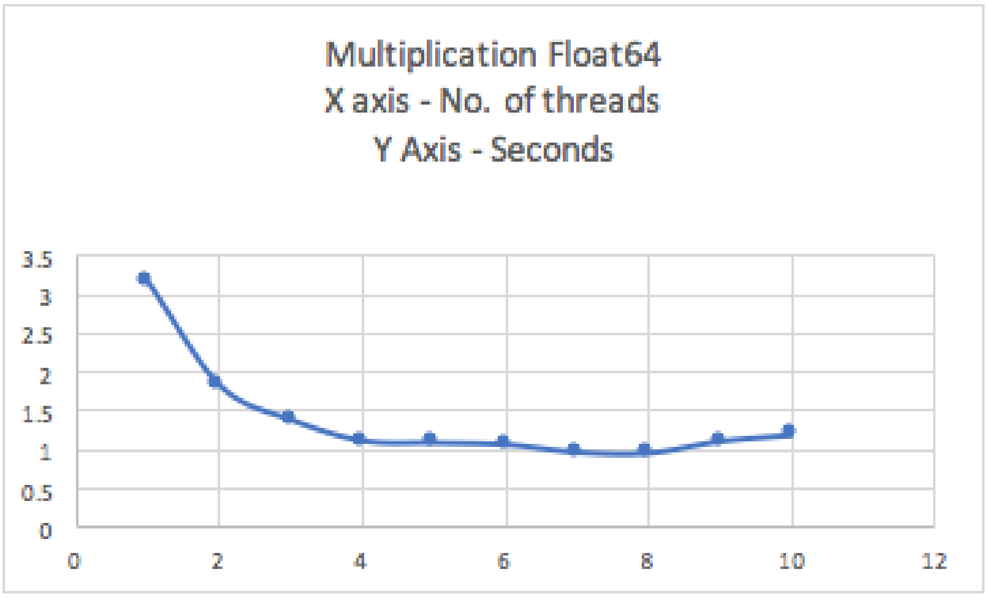
An extract of this code can be seen below.

```
float sum = 0.0;
1)   for(int i = 0; i<rowLength; i++){
2)      for(int l = 0; l<colLength2; l++){
          sum = 0;
          #pragma omp parallel for
3)          for(int k = 0; k<nonZeroMat1 ;k++){
4)              for(int g = 0; g<nonZeroMat2 ;g++ ){
                  if(row[k]==i && column2[g]==l && column[k]==row2[g]){
                       sum = sum + element1[k] * element2[g];
                       break;
                   }
                 }
              }
              if(sum != 0){
                  newArray[index] = sum;
                  index++;
              }
          }
       }

    return 0;
 }
```

Due to loop dependencies, within this implementation, it was only achievable to parallelise one of the nested loops. Regardless, this still resulted in a significant improvement in performance with an almost 50% reduction in run time when utilising two threads instead of one. Similar to the previous operations, it can be seen that the most optimal thread range lies between 4-8 threads. Anything beyond this results in decreased performance.

The time complexity of this operation is not very scalable. Performance rapidly decreases as the dataset gets larger. For this reason, I tested this using a smaller dataset *Float64.in* as the time taken to complete *Float256* or above would be too long. The time complexity is *O(r^2(n*n'))* where r is the row length of matrix 1, n is the number of non zero elements in matrix 1 and n' is the number of non-zero elements in matrix 2.



Personal Reflection

This project was quite challenging requiring extensive revision of C-language. It was frustrating switching to COO implementation after having implemented the majority of my functions using CSR only to realise that I would be unable to parallelise them. Regardless, I found this project enjoyable and very satisfying once I managed to achieve improvements in performance using threading.

References:

Golubin, A. (2017). Sparse data structures in Python. [online] Artem Golubin. Available at: https://rushter.com/blog/scipy-sparse-matrices/ [Accessed 29 Sep. 2019].