Sparse Matrices

Ethan Chin 22248878

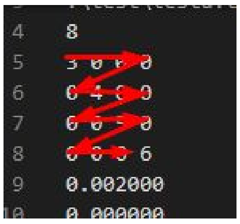
CITS3402 - High Performance Computing Project 1

Operating System used: Windows 10
https://github.com/LightXEthan/MatrixComputation

Spare matrix representations

File processing

Files are process from left to right and data is entered to arrays after looking at an element.



```
8
3 0 0 0
0 4 0 0
0 0 5 0
0 0 0 6
0.002000
0.000000
```

In an actual input file, all the elements are on one line.

<u>Coordinate Format (COO)</u>

There are 3 arrays i,j,val. In the code they are called array_i, array_j, array_val.

0 0 1	array_i = [0, 1, 1]	// Row index
3 0 2	array_j = [2, 0, 2]	// Column index
0 0 0	array_val = [1, 2, 3]	// Value

<u>Compressed Sparse Row Format (CSR)</u>

As the j and val arrays are the same in coordinate format, these arrays are reused to save time and space. This will also improve the efficiency of processing matrix files as there is less memory allocation operations needed.

array_j, array_val, array_csr

0 0 1	array_j = [2, 0, 2]	// Column index
3 0 2	array_val = [1, 2, 3]	// Value
0 0 0	csr_rows = [0, 1, 3, 3]	// Number of elements in its row + previous value

<u>Testing</u>

Operations were testing 10 times using functions that ran sequential code and 10 times using functions that ran operations in parallel. The results were put in a table and the average was then calculated. This made it easier to compare the performance of implementing parallelism into operations and also removed the variations in the results. The tests were timed on the files in example outputs that was given.

Scalar Multiplication

This was implemented by iterating over the array_val and multiplying it with the scalar value given.
Parallelism was implemented by using a parallel for loop that iterates over the array_val. Instead of sequentially going through each value and performing scalar multiplication, the values are changed in parallel. It is expected to run faster than its non-parallel version on much larger matrices.

Table 1.1 - Scalar Multiplication Operation Results

Non-parallel		Scalar	Parallel	
Processing	**Operation**	int1024	**Processing**	**Operation**
0.09800	0.00100	sm 2	0.09100	0.00200
0.13000	0.00000	4 threads	0.08800	0.00100
0.09300	0.00100		0.08600	0.00100
0.11500	0.00000		0.08700	0.00100
0.12900	0.00100		0.09100	0.00100
0.09600	0.00100		0.08600	0.00100
0.10200	0.00000		0.09400	0.00100
0.08800	0.00000		0.09900	0.00000
0.08900	0.00000		0.86000	0.00100
0.11000	0.00000		0.08400	0.00000
0.10500	0.00040	Average	0.16660	0.00090

Parallel seems to be slower but the number is so small, it doesn't seem to be a significant difference. The parallel operation is expected to run faster than the non-parallel when scaling the size of the matrix as non-parallel operation changes each value sequentially where as the parallel operation changes multiple values depending on the number of threads at the same time (in parallel).

Trace
Sum of all the diagonal elements top right to bottom left. The dimensions matrix is also checked to ensure that it is a square matrix.
Parallelism was implemented by using pragma omp parallel for reduction, to parallel iterate over all the elements to check if they are a diagonal element and add them to a results array. The results array is then summed together using reduction. This is expected to run faster on larger sparse matrices.
Using parallel seems to be slower than normal. Reduction can be implemented as it may improve efficiency.

Table 1.2 - Trace Operation Results

Non-parallel		Trace	Parallel	
Processing	**Operation**	int1024	**Processing**	**Operation**
0.09100	0.00100	22417	0.09600	0.00100
0.08400	0.00100	4 threads	0.09300	0.00000
0.00890	0.00000		0.10500	0.00000
0.09400	0.00100		0.09600	0.00000
0.08400	0.00100		0.09200	0.00100
0.08300	0.00000		0.09200	0.00100
0.09000	0.00100		0.08900	0.00100
0.08300	0.00100		0.09700	0.00100
0.08300	0.00100		0.09700	0.00000
0.08400	0.00100		0.09100	0.00000
0.07849	0.00080	Average	0.09480	0.00050

Parallel seems to be slightly faster but doesn’t seem to be a significant difference in this case. As the matrix scales it may be faster than non-parallel.

Addition
2 matrices are added together.
The operation goes through both input arrays and adds the new elements to an output array. Parallelism was implemented by having a single thread that goes through the two input arrays and generates tasks for adding the elements to the new array. This can be done as the order in which the elements are added to the new array is not dependent of other elements. It is possible that this parallel implementation may not run faster if the matrix is scaled up.

Table 1.3 - Addition Operation Results

Non-parallel		Addition	Parallel	
Processing	**Operation**	int1024	**Processing**	**Operation**
0.19100	0.00300	4 threads	0.21100	0.13600
0.19600	0.00200		0.17800	0.08200
0.18800	0.00200		0.17500	0.15800
0.18000	0.00200		0.17900	0.14200
0.18400	0.00200		0.18300	0.16900
0.18500	0.00200		0.19500	0.14100
0.19200	0.00300		0.23200	0.09300
0.24800	0.00400		0.21600	0.02800
0.20500	0.00500		0.26000	0.09400
0.19600	0.00200		0.20600	0.04900
0.19650	0.00270	Average	0.20350	0.20350

A strategy that was also tested was tasking all the comparisons then waiting for the tasks to complete to ensure that the incrementing values were in sync. This allowed an order of operations, however, it was found to be slower due to the single thread that assigns task to wait for each comparison to complete. The branch can be seen here:
https://github.com/LightXEthan/MatrixComputation/pull/20/files
The main problem with this implementation of addition was the order in which elements were to be added had to be sequential and there were two pointers that point to the position of their array.

Transpose

The transpose was implemented by then sorting the rows in COO format to be outputted by switching the columns with the rows and vice versa. The insertion sort was implemented for the as a non parallel solution but there are many sorting algorithms that are more optimal. eg. merge sort, radix sort. Insertion sort takes O(n^2) time on average. Parallel was not implemented as it was difficult to find a solution as each element's new position was determined from the results of processing previous elements.

An idea that was found but not implemented was to use parallel for to find the element with the smallest j and add it to the new array. Another idea was to use a parallel for to iterate over all the elements and perform an insertion sort to move elements to their position by comparing previous values but previous threads would have to wait if an element is being moved. This would result in multiple threads moving elements in the same array but not interfering with each other. This would be done by sharing another array of boolean values checking if they are being used by another thread. This was not tested. Many concepts were tested but failed so implementing parallel was dropped.

Table 1.4 - Transpose Operation Results

Non-parallel		Transpose	Parallel	
Processing	**Operation**	int1024	**Processing**	**Operation**
0.12300	17.68700			
0.09500	16.95300			
0.10500	15.66700			
0.13800	15.60900			
0.10500	15.70600			
0.10200	15.63700			
0.10600	15.72000			
0.12100	16.04900			
0.15000	15.79400			
0.16500	15.54100			
0.12100	16.03630	Average		

Parallel was not implemented.

Matrix Multiplication

This was implemented using the CSR format and output to a COO format. The new matrix size was calculated and each value in the new matrix was calculate by iterated over the values in the relevant row in the first column of the 1st matrix and multiplying them with the columns in the 2nd matrix. This was repeated for all the values using CSR to determine how many elements are in a row and what position they are in the array.

Parallel was implemented by having a #pragma omp parallel for reduction when iterating though the elements on the same row of the first matrix. Addition reduction was used to sum all the multiplications results of all the threads into the value of the new matrix (which is added to new arrays in COO format). This is repeated for all values to be calculated int he new array.

Essentially iterating through each element in a row in the first matrix with all the elements in each column in the second array in parallel for a single value, using the reduction to sum the multiplication results together. After performing 10 tests for both non-parallel and in parallel, running the operation in parallel is found to be significantly faster opposed to running in non-parallel which will benefit when scaling up the size of the matrix. It is expected to perform better than non-parallel when the size of the matrix increases.

Another option that was tested was to add #pragma omp parallel for, on the top for loop where each thread would calculate at each element in the new array. This was found incorrect results as adding the new calculated value into the new array needed to be in sequential order.

Table 1.5 - Matrix Multiplication Operation Results

Non-parallel		Multiplication	Parallel	
Processing	**Operation**	int256	**Processing**	**Operation**
0.012	50.27900	4 threads	0.01200	19.70900
0.01200	47.42400		0.01000	20.19600
0.01200	46.52700		0.01200	19.03100
0.01700	51.99000		0.01200	18.59600
0.01400	48.40100		0.01000	18.82100
0.01300	45.43800		0.01100	17.95100
0.01100	41.28700		0.01100	19.27600
0.01200	42.67900		0.01100	18.51900
0.01200	42.19500		0.01200	17.52600
0.01100	42.08500		0.01100	15.05500
0.01260	45.33622	Average	0.01120	18.46800

Using parallel is found to be significantly faster than without parallel. This would highly beneficial for dealing with larger matrix sizes and a larger number of threads.

Example inputs and outputs

testb.txt

```
float
4
4
1.0 0.0 0.0 3.0 0.0 1.5 0.0 0.0 0.0 0.0 2.3 0.0 0.0 0.56 0.0 1.0
```

testc.txt

```
float
4
4
0.0 0.05 0.0 0.0 1.2 1.2 0.0 0.0 5.2 0.0 0.2 0.0 0.0 0.09 0.6 0.0
```

Scalar Multiplication

Command

```
./matrix --sm 2 -l -t 4 -f testb.txt
```

Example Output

```
sm
testb.txt
4
float
4
4
2.000000 0. 0. 6.000000 0. 3.000000 0. 0. 0. 0. 4.600000 0. 0. 1.120000 0.
2.000000
0.000000
0.001000
```

Trace

Command

```
./matrix --tr -l -t 4 -f testb.txt
```

Example Output

```
tr
testb.txt
4
5.800000
0.000000
0.003000
```

Trace

Command

```
./matrix --ad -l -t 4 -f testb.txt
```

Example Output

```
tr
testb.txt
```

```
4
5.800000
0.000000
0.003000
```

Addition

Command

```
./matrix --ad -l -t 4 -f testb.txt testc.txt
```

Example Output

```
ad
testb.txt
testc.txt
4
float
4
4
1.000000 0.050000 0.0 3.000000 1.200000 2.700000 0.0 0.0 5.200000 0.0 2.500000
0.0 0.0 0.650000 0.600000 1.000000
0.001000
0.003000
```

Transpose

Command

```
./matrix --ts -l -t 4 -f testb.txt
```

Example Output - *Note that number of threads is irrelevant as transpose runs only runs in sequential. This note is also printed out.*

```
ts
testb.txt
4
float
4
4
1.000000 0.0 0.0 0.0 0.0 1.500000 0.0 0.560000 0.0 0.0 2.300000 0.0 3.000000 0.0
0.0 1.000000
0.000000
0.000000
```

Matrix Multiplication

Command

```
./matrix --mm -l -t 4 -f testb.txt testc.txt
```

Example Output

```
mm
testb.txt
testc.txt
4
```

```
float
4
4
0.0 0.320000 1.800000 0.0 1.800000 1.800000 0.0 0.0 11.959999 0.0 0.460000 0.0
0.672000 0.762000 0.600000 0.0
0.000000
0.002000
```

Helpful resources
https://kkourt.io/phd/phd-en.pdf - helped when working on matrix multiplication

Notes for the marker

- *Please note that optget was not implemented for handling flags. ie. Options won't work if they are grouped together.*
- *Make file is available, -Wall and -pedantic can be removed if they cause trouble*
- *All functions were tested and are expected to run outputting correct results including float numbers.*
- *Transpose does not have parallelism implemented.*
- *Floating points output default to 6 decimal places*