Sparse Matrices

Ethan Chin 22248878

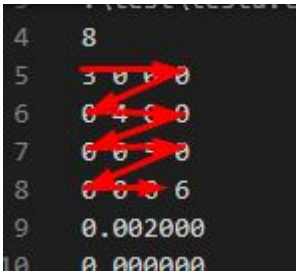
CITS3402 - High Performance Computing Project

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

Please note that optget was not implemented for handling flags. Options won't work if they are grouped together. Make file is available.

Spare matrix representations

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



Coordinate Format (COO)
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

Compressed Sparse Row Format (CSR)
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

Testing
Operations were testing 10 times using functions that had no parallelism 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.

Scalar Multiplication

Parallelism was implemented by using a parallel for loop that iterates over the array_val. It is expected to run faster than its non-parallel version on much larger matrices.
Using parallel seems to be slower than normal.
Following are tested using int1024.in, from the example_outputs given.

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 processing of files is the coded the same in both.

Trace

Parallelism was implemented by using pragma omp for, to iterated 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. 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

The operation goes through both input arrays and adds the new elements to an output array. Parallelism was implemented by having one 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, does not matter. It is excepted to run faster on larger sparse matrices than without parallel computing.

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

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. Parallel was not implemented due to time constraints.

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		

Multiplication

Implemented using the CSR format. Parallel was implemented by having a #pragma omp parallel reduction when iterating though the elements on the same row of the first matrix. Essentially multiplying a row in the first matrix with all the column in the second array in parallel for a single value, using the reduction to sum the multiplication results together. It is significantly faster running in parallel opposed to running in non-parallel which will benefit when scaling up the size of the matrix.

Table 1.4 - 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.

Helpful resources
https://kkourt.io/phd/phd-en.pdf