ELEN 4020 Lab 2: Matrix Transposition

Uyanda Mphunga - 1168101 — Darren Blanckensee - 1147279 — Ashraf Omar - 710435 — Amprayil Joel Oommen - 843463

I. Introduction

The purpose of this lab is to expose the students to the concept of parallelism, by introducing them to the parallel programming libraries of OpenMP and Pthread libraries. The task is to create code that uses the aforementioned libraries, to parallelize the process of transposing a matrix, without using any additional space. The code is to make use of different numbers of threads, which are to transpose a variety of square matrices (i.e. A[No][No]). The number of threads to be used are 4, 8, 16, 64 and 128, while the different matrix sizes that need to be tested, must be dimensioned as 128x128, 1024x1024 and 8192x8192 respectively. The entire process of creating the threads and allowing them to transpose the matrix, is to be timed and compared; this will allow the students to understand the behaviour of parallelism when applied to computing a matrix transpose.

II. OpenMP

A. Initial Approach and Problems

Initially the matrix had been declared as a one dimensional array with elements being assigned using row-major ordering. The transpose was done in a nested for loop with the outer loop going from 0 to the number of rows - 1 and the inner loop going from whatever the outer loop is + 1 to the number of columns. This is not ideal for open mp's parallel for loop decomposition.

It would be ideal to fuse the nested for loop into a single for loop which Open MP would then parallelise. This was not possible as the students could not find a way to define both nested for loop variables when there is just one loop. This is easy to do when both loop variables start from 0 and go up to a given number but becomes difficult when loop variables are defined in terms of each other which was the case here. Therefore it was decided that it would be left as a nested for loop.

The collapse(2) command for collapsing nested for loops could not be used for the same reason that the loops could not be fused and therefore only one of the two loops in the nested for loop could be parallelised. It was determined that the best loop to parallelise was the outside loop (trial and error determined that this gave the best results).

Dynamic scheduling was used with a chunk size of 1 such that each thread would work on the next available piece of data. The code was run using the various numbers of threads and on the different sized matrices and the times will be compared in the comparison section.

B. Quality of Solution

The final solution is less complex in time and storage than the basic transpose method which makes a copy of the matrix and copies element (i,j) from the first array to element (j,i) of the second array. The methods used in this lab iterates through the upper diagonal and swaps using a temporary variable and therefore no extra memory is needed and you only iterate through less than half of the matrix.

The linear and parallel methods both use the same transpose logic so as to make time comparisons. The code relies on the pragma omp parallel for compiler directive to handle the parallelisation.

III. Pthreads

The PThread algorithm developed, makes use of similar logic to the algorithm used for OpenMP, namely: the algorithm takes a square matrix (with an even-numbered dimension No), iterates through the elements of the upper half of the matrix (above the main diagonal) and transposes them with the corresponding element in the lower half of the matrix (below the main diagonal). Mathematically, the process is described by an element A[i][j] = A[j][i] and vice versa. However, unlike conventional, single-thread implementations of a transposition algorithm, the algorithm developed makes use of multiple threads which collectively operate on subdivided portions of the original matrix.

The algorithm does this, by making use of three functions, namely: $transposeMatrix$, $upperBound$ and $lowerBound$. Each thread created is defined to perform the definitions found in $transposeMatrix$, on particular subsections of the matrix. The subsections of the matrix are determined by the values returned by the functions $upperBound$ and $lowerBound$.

Specifically, the functions $upperBound$ and $lowerBound$, take in as arguments the dimension of the matrix, the total number of threads created in main, as well as the 'threadID' of a particular thread. In both functions an integer variable "binSize" is determined by using the formula:

$$binSize = matrixDimension/numberOfThreads$$

This determines an upper and lower bin of elements (alternatively stated, two sets of values, each containing a particular number of elements in the matrix) that will be transposed using the $transposeMatrix$ function.

Each thread will operate on an upper and lower bin of elements, until each subsection of the matrix has been correctly transposed, resulting in the entire matrix to be transposed. The threads are then iteratively joined back into the main thread and the final transposed matrix is printed out.

IV. Results and Analysis

For the sake of comparing the performance of the algorithms using OpenMP and PThread, both algorithms have been implemented using the same logic (namely swapping the top half of the matrix above the main diagonal, with the lower half of the matrix). The time each algorithm takes to perform the transposition, is measured using OMP's $wtime()$ functionality (due to its accuracy in measuring the time).

To have a baseline of times values, which can be used to gauge the performance of the two parallel algorithms, a typical single thread (/linear) algorithm for matrix transposition has been run for each square matrix (with dimensions No= 128, 1024 and 8192). The results of these times can be seen in Table 1.

TABLE I: Comparison of Different Methods

		Array Sizes		
		128	1024	8192
	No. of Threads	Times		
Linear	1	0.000164	0.012435	1.352024
Pthread	4	1.023914	1.020149	1.011761
	8	1.002465	1.034782	1.032099
	16	1.008585	1.014240	1.035444
	64	1.016753	1.032005	1.015308
	128	1.007373	1.023784	1.006293
OpenMP	4	0.000242	0.012308	0.611911
	8	0.000231	0.005163	0.848804
	16	0.000246	0.004093	0.914388
	64	0.000898	0.004692	0.941297
	128	0.001017	0.005042	0.925575

From the results in Table 1, it is evident that the implementation of the PThread algorithm takes longer to transpose square matrices of size No=128 and 1024, than the single-thread algorithm (even when the PThreads are increased). However, for the square matrix with No=8192, the PThread algorithm performed marginally quicker than the single-thread algorithm.

However, it is evident that the OpenMP implementation is significantly faster than the PThread implementation. For square matrices with dimension No=128 and 1024, it is evident that when 4 threads are used, that the single-thread implementation and the OpenMP implementation have similar performances; however, as the threads increase, the parallel operations of OpenMP significantly improve the performance of the time it takes to transpose the matrix. When a square matrix of size No=8192 is used, the OpenMP algorithm is performs more quickly than the single-thread algorithm.

From the above results, it is evident that the implementation of the PThread algorithm is not as efficient as the OpenMP algorithm. Clearly, the algorithm's management of the threads causes the PThreads to perform slower in most instances than the single-thread algorithm, while the OpenMP's in-built management of the threads, is significantly more efficient.

V. Conclusion

In this lab, two parallel algorithms were implemented, using the OpenMP and PThread libraries, to transpose varying sizes of matrices, using varying number of threads. It is evident that the user's management of the threads in the PThread implementation, is not as efficient as OpenMP's own management of the threads, thus resulting in the OpenMP implementation being the fastest performing algorithm.