ELEN4020A: Data Intensive Computing Laboratory Exercise No 2

Kopantsho Mathafa (849038) Chizeba Maulu (900968) James Phillips (1036603)

March 18, 2019

Abstract

This report presents a summarized description of the solutions developed to solve the problem proposed in the lab brief. The solutions developed involve the use of OpenMP and PThreads, these were used to solve the proposed problem of transposing a matrix using parallel programming methods. The computation times of all methods developed can be found in the table of results, which is presented in the report, along with pseudo code and finally a short analysis of the results.

I. Introduction

Threads are independent streams of instructions which can be scheduled to execute, independently and asynchronously, by the operating system. Optimizing performance and efficiency in high performance computing environments are the primary reasons behind threads being implemented and used.The objective of this lab is to write programs which implement matrix transposition using parallel programming methods. The preferred language used is C++.

This report discusses the use of threads in the solutions to solving the problem of matrix transposition. The analysis technique involves the use of timing sequences to compare the speed and performance of the different methods presented in this report.

II. Matrix Transposition Implementation

To implement the matrix transposition in C++, three methods are provided which achieve the required transposition technique. The methods implemented are *SerialMatrixTranspose* which transposes the matrix row by column by using a temporary variable as a place holder, a *transposeDiagonally* method which transposes the matrix by using no additional space, and then a *transposeMatrixByChunks* method which first performs a sub-matrix method on the larger matrix dividing it into a matrix of sub-matrices. The sub-matrices are then transposed using the serial method, the large matrix is then transposed resulting in the full transposition.

Serial transposition is the traditional way in which matrices are transposed, where a temporary value is used to store an element, A, thereafter placing an element B in A's position, then the temporary value is placed in B's old position[1]. This is achieved using two for-loops which iterate through the rows and columns of the matrix[1]. This implementation only requires an integer sized temporary place. Algorithm 1 presents the pseudocode to achieve this transposition.

```
Result: Transposed Matrix
Determine the size of the matrix $N$;
for $i \leftarrow 0$to $N$ do
    for $j$to $N$ do
        $temp$←matrix[i][j]matrix[i][j]←matrix[j][i]matrix[j][i]←tempend
    end
```

Algorithm 1: Serial Matrix transposition

The chunks transposition first divides the larger matrix into specified sub-matrix sizes. The pseudo code presented in Algorithm 2 demonstrates the chunking and transposition process.

The diagonal transposition can be referred to as *in place* transposition, where the transposition is achieved using two additional space[2]. This can easily be achieved for an *nxn* matrix by using the *swap* function of the C++

```
Result: $RS \times RS$ Matrix
Determine the Resulting Matrix Size (RS), Initialise a temporary matrix tempM of size RS;
if Dimensions of matrixA mod chunkSize = 0 then
    for $i \leftarrow 0$ to RS do
        for $j \leftarrow 0$ to RS do
            for $k \leftarrow 0$ to chunkSize do
                for $l \leftarrow 0$ to chunkSize do
                    $tempM[k][l]$ ←matrixA[k+(i*chunkSize)][l+(j*chunksize)]end
                end
            end
        end
    Serial Transpose Matrix
```
Algorithm 2: Sub-matrix implementation

standard library, which will use no additional space[2]. An implementation is derived from the definition found on *Wikipedia* and is presented in Algorithm 3 [2].

```
Result: Transposed matrix
Determine size of matrix to N;
for $i \leftarrow 0$ to N-2 do
    for $j$ to N-1 do
        swap $matrix[i][j]$ with $matrix[j][i]$
    end
end
```
Algorithm 3: Diagonal implementation

III. Matrix Transposition Using PThreads

The routines in transposeByChunks and finalDTranspose allow for only one argument to be passed into the pthread_create function. The structure: pthread_create(pthreads, attributes, routine, argument being passed into routine). The structs and struct arrays defined capture all the required arguments thus allowing a struct to be passed as an argument to the routine and its members being modified locally in the routine. Inside these routines, the nxn array is transposed. This done using a for-loop in which the outer loop iterations are divided evenly between the threads. Once the routine completes, pthread_exit terminates the thread, i.e frees it up.

In threadedDiagonalTranspose and threadedChunksTranspose: these functions create the specified number of threads. It creates threads with the attribute of being joinable. error modules are included in case a thread creating does not complete. once the threads are created and thus the routines called, we wait for all the threads to complete or join them. Having joined succesfully, the attribute is destroyed.

IV. Matrix Transposition Using OpenMP

OpenMP or Open Multi-Processing, is a package which forms part of the GCC compiler in Linux environments. This package supports shared memory multiprocessing[3]. Shown below is pseudo illustrating the use of OpenMP to transpose a matrix using parallelism. OpenMp is used to divide iterations of loops into threads which can be executed independently.

Implementing OpenMP is a relatively easy step when compared to that of pthreads. The implementation relies on the fact that the single threaded methods are already implemented. From the serial implementation, the *pragma* keyword is used to indicate which part of the code will be using OpenMP directives[3]. Certain directives can be called to indicate which components of OpenMP are required. To run the segment of code in parallel, the parallel directive is used. The OpenMP library will then automatically attempt to split the code into the number of specified threads[3]. Certain clauses are added to the *pragma* keyword to add certain specifications, such as *num_threads* will set the number of threads which are required[3]. OpenMP will automatically take the serial implementation of code, parallelise it and distribute that amoungst the number of threads[3].

To parallelise the implemented transposition functions, *pragma* clauses are added to each segment of code which need to be parallelised. These clauses are added above each *for* loop using the *for* clause in the *pragma*. The number of threads are specified at each clause, and the matrix which has to be transposed is specified to be shared memory.

The above mentioned clauses are used in the three serial implementations. While the implementations for *SerialMatrixTranspose* and *transposeDiagonally* remain fairly simple, the *transposeMatrixByChunks* needs some mention. There is a *pragma* clause added to the sub-matrix implementation, which will assist in the sub-matrix procedure. As this function utilises the *SerialMatrixTranspose* function to transpose the sub-matrices, no extra *pragma* is added, as there already exists one inside this function.

A. OpenMP Pseudocode

Transpose Matrix Along Diagonal: In-place transpose of matrix. The original matrix will be modified.

```
function TRANSPOSEDIAGONALLY($matrix, threads$)
    $set\_num\_threads \leftarrow threads$
    for $i \leftarrow matrix.size$ do

    end
    $j \leftarrow matrix.size$
    $tempVal \leftarrow matrix[i][j]$
    $matrix[i][j] \leftarrow matrix[j][i]$
    $matrix[j][i] \leftarrow tempVal$


end function
```

V. Critical Analysis and Future Recommendations

A main function is written which calls the methods implemented. The methods contain within them, timing sequences to compute the amount of time in seconds, that each transposition routine takes to fully transpose a given matrix. The results of these methods can be seen in Table 1. The results appear to be consistent and intuitive. Upon closer inspection it can be seen that there is a significant increase in computational time for N = 128 to N = 1024, where N is the size of the matrix. This is understandable as there is an increase in the order of magnitude of N. Thereafter there is an increase in time, in seconds, as N increases. The time increases by a constant factor with each increase in N. For example, in the case of the diagonal PThreads method, the time increased from N = 1024 to N = 2048 by a factor of 3.37. Similarly there is a time increase by a factor of 3.3 from N = 2048 to N = 4096. There is a linear increase in time as the number of elements increase. A similar pattern follows for the other methods.

This is an indication of the scalability of the solution. There is a linearly proportional relationship between the size of the matrix and the computational times. One possible explaination for this is that the low number of threads used reduces the possibility of false sharing. False sharing is a term given when threads try to access independent variables that share the same cache line[4].

If a thread gains access to a particular cache line, all other threads need to wait until that thread has completed its task, and reliquished control of the cache line, before they can gain access to that cache line and the variables in it. This means threads can directly impact the performance of other threads. The greater the number of threads used the more likely false sharing is to happen and the greater its impact on the performance of a multi-threaded program. For this reason simply increasing the number of threads used in a program may not necessarily improve the performance of a program.

OpenMP performed better than PThreads, although it is important to note that the times presented for OpenMP are strictly the time required to compute the transposition. The times recorded for PThreads on the other hand have an overhead of setting up the multi-threading environment. Regardless the Diagonal method using OpenMP gave the best result of all the methods implemented.

TABLE I
PERFORMANCE OF PTHREADS AND OPENMP METHODS

$N_0 = N_1$	**Basic**	**Pthreads**		**OpenMP**		
		Diagonal	**Blocked**	**Naive**	**Diagonal**	**Blocked**
128	0 s	0.997 ms	0.997 ms	0.998 ms	0 s	4.99 ms
1024	30.93 ms	37.89 ms	81.77 ms	13.95 ms	8.98 ms	28.9 ms
2048	136.63 ms	127.64 ms	324.13 ms	63.83 ms	41.89 ms	94.72 ms
4096	598.93 ms	420.85 ms	1.36 s	279.24 ms	194.02 ms	347.09 ms

VI. CONCLUSION

Six different methods of matrix transposition have been presented. Five of these are implemented using parallel programming libraries. The performance of each of the methods are presented and brief discussion of the results was also presented. The best performing method is stated and suggestions are given to why the performance methods performed the way that they did.

REFERENCES

[1] Learn C, "C Program to Find Transpose of a Matrix." [Online]. Available: https://www.programiz.com/c-programming/examples/matrix-transpose

[2] "In-place matrix transposition," Jan. 2019, page Version ID: 879729488. [Online]. Available: https://en.wikipedia.org/w/index.php?title=In-place$_matrix_transposition&oldid = 879729488$

[3] utexas, "Getting started with OpenMP." [Online]. Available: http://pages.tacc.utexas.edu/ eijkhout/pcse/html/omp-basics.htmlTheOpenMPmodel

[4] M. Thompson, "False sharing - dzone java," Aug 2011. [Online]. Available: https://dzone.com/articles/false-sharing