ELEN4020: Data Intensive Computing
Laboratory Exercise 2

Kavilan Nair 1076342
Christopher Maree 1101946
Iordan Tchaparov 1068874
Laura West 1084327

10/03/2018

In-place Matrix Transposition

In this lab, 3 different algorithms that performed in-place matrix transposition were tested against each other: a naive iteration, OpenMP and Pthreads algorithms. Each implementation is outlined in this report and they are then compared to find the best solution in terms of speed for in-place transposing matrices of different sizes of 128, 1024 and 8192. The pseudo code for each algorithm is given in the appendix.

1 Naive Iteration Algorithm

This algorithm is composed of two nested for loops that are used to transpose the matrix. The outer and inner loops are used to iterate through the rows and columns within the matrix, along the main diagonal. Each value within the matrix is swapped to the corresponding transposed location. This is done by atomically swapping the row and column index of two elements. This process only needs to be applied to the top (or bottom) triangular half of the matrix, as the process returns the original matrix if it is applied over the whole matrix. The diagram in Figure 1 below outlines the flow of the loops:

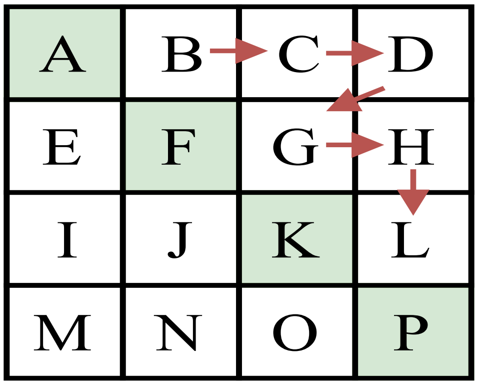

Figure 1: The matrix with its flow of access by the naive iteration algorithm loops.

Red arrows show the flow of the loops. At each location, the index values of the row and column are swapped. For example, the looping starts at B (as shown by the start of the first red arrow). This value has the location [0][1] (rows and columns, indexed at zero). It is atomically swapped with its column row counterpart, at location [1][0]. This is value E. This atomic swap uses a temporary variable to facilitate the swap.

2 OpenMP Algorithm

The OpenMP algorithm follows the same basic implementation of the naive solution, except multiple threads are used in the swapping process. A parallel array is shared across all threads, enabling them all to operate on the matrix at the same time. The matrix is broken up into a series of sections that each thread works on. OpenMP effectively uses PThreads but manages the process for the programmer. A subtlety that needs to be considered is how the matrix is split up for each thread. The implementation used worked by segmenting the matrix into equal components that are then given into each thread. The following directives are used to parallelize the nested for loop:

```
#pragma omp parallel shared(matrix) private(temp, i, j)
#pragma omp for schedule(dynamic, CHUNK_SIZE)  nowait
```

The first pragma directive instructs that the memory of the two dimensional matrix array to be shared between threads. The variables temp, i, and j are assigned to be private to each thread and separate. The second pragma directive ensures that a chunk of the loop is assigned to different threads, this chunk size is defined in code and passed in as a parameter for the scheduling. The nowait keyword informs the threads to continue after they have finished.

3 PThreads Algorithm

The Pthreads implementation requires the programmer to manually control the creation, management and execution of all threads in the application. The basic premise of the method implemented is to split the matrix into a series of segments of row-column pairs that each thread operates on. Depending on the number of threads and the size of the matrix, each thread is allocated a portion of the matrix. This notion is represented in the diagram Figure 2 below:

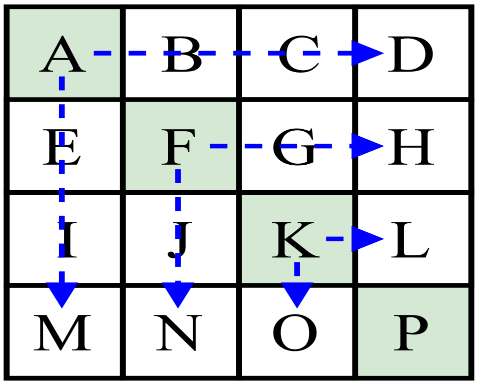

Figure 2: The matrix with its row-column segments for the PThreads algorithm.

This diagram assumes that each thread has a row and column pair, resulting in needing 1 less thread than the size of the square matrix. The implementation segments the matrix into sets of rows and columns whereby each thread operates on multiple rows and column pairs, where the segmenting process is the same in effect as outlined in the diagram. This is not the most efficient way as the first thread is required to perform the most number of swaps. Dividing the work load more evenly amongst the threads will result in a more efficient algorithm.

4 Performance comparison of different algorithms

The three algorithms were run on multiple matrix sizes of 192, 1024 and 8192. Threaded algorithms were run on each size using a varying number of threads from 4, 8 , 16, 64 and 128 to gauge the performance of each algorithm. The threaded algorithms computation times were plotted against thread number to identify performance results. The algorithms were run on a laptop with an Intel core i5 with two cores (four with hyper threading) running at 2.4 GHz. The laptop also had 8GB of DDR3 RAM. The algorithms were executed on startup of the laptop and when the CPU usage settled. No other programs were opened to prevent background processes from slowing the execution of the code. Ideally this should have been run on a Linux machine with no desktop environment however this was not available to the group. The results for the naive iteration algorithm and the OpenMp and Pthreads algorithms are shown in Table 1 & 2 respectively.

Table 1: Execution times of the naive solution

Matrix size	**Time**
128	0.000074
1024	0.017389
8192	1.510436

Table 2: Times of OpenMP and PThread transpositions for different sizes of matrices and number of threads

Matrix Size	**128**		**1024**		**8192**	
Threads	**OpenMp**	**PThread**	**OpenMp**	**PThread**	**OpenMp}**	**PThread**
4	0.000495	0.000438	0.006074	0.011141	0.928543	1.20085
8	0.000333	0.000301	0.00509	0.007396	0.911021	0.911576
16	0.000717	0.000686	0.00732	0.008052	0.709386	0.847845
64	0.001778	0.001951	0.009967	0.010091	0.66299	0.671619
128	0.003844	0.004281	0.011607	0.010748	0.613919	0.65732

5 Analysis of results

For the small sized (128) matrix, the naive iteration algorithm proved to be much faster than the threaded approaches. This was due to the additional overhead associated with the creation, management and execution of the threaded approaches, especially in OpenMp. There is no clear advantage to using a threaded architecture for this sized array. For the medium sized (1024) matrix, the difference in execution speed is closer between the three implementations. The threaded implementations were marginally faster than the naive iteration as they don't require the swaps being done sequentially, enabling the transposition of sections of the matrix to occur in parallel. For the large sized (8192) matrix, the threaded application far outperforms the speed of the naive solution as at this scale, being able to execute in parallel is very advantageous. This brings forth the statement: When processing large amounts of data, parallelism should be incorporated to achieve the results faster although there is no benefit from parallelism when the data is small.

6 Hybrid Solution

As shown by the results, a threaded application is not always the fastest. The results in Table 2 show that there is certain threshold for the size of the matrix size whereby the threaded solutions outperforms the naive solution. A hybrid solution that consists of both the naive and threaded solutions, enabling optimal execution time irrespective of matrix size would be the best solution to in-place transposition. This solution will check the size of the array before preforming the execution and use the optimum algorithm for the given size.

7 Conclusion

This report details the three different matrix transposition algorithms that were implemented using the C programming language for this lab: the naive iteration, OpenMp and PThreads. The results for each algorithm were compared and analyzed to find out how each algorithm performed in-place transposition. A hybrid solution to potentially achieve a faster transposition execution time, irrespective of matrix dimensions was also briefly discussed.

Appendix - Pseudo Code

Naive Solution

```
for i = 0 to (MATRIX_SIZE - 1)
    for j = i + 1 to MATRIX_SIZE
        temp = matrix element indexed at i and j
        matrix element at i and j position = matrix element indexed at j and i position
        matrix element indexed at j and i position = temp
    end for
end for
```

OpenMP Solution

```
set number of threads for OpenMP
create variables i, j and temp

#pragma omp parallel shared(matrix) private(temp, i, j)
    #pragma omp for schedule(dynamic, CHUNK_SIZE)  nowait
    for i = 0 to (MATRIX_SIZE - 1)
            for j = i + 1 to MATRIX_SIZE
                temp = matrix element indexed at i and j
                matrix element at i and j position = matrix element indexed at j and i position
                matrix element indexed at j and i position = temp
        end for
    end for
```

PThreads Solution

```
set matrix size
set number of threads
struct ThreadInfo {int **matrix ; int startIndex ; int endIndex}
in main:
create pointer to integer pointer called matrix
allocate memory for matrix
for each element in matrix
    allocate memory for each row of matrix to create matrix columns
end for
set count to 0
for each element in matrix
    element = count
    increment count
end for
create int variable x
create double variable matrixSize and set equal to matrix size
create pointer threadInfo that points to struct ThreadInfo
create pthread threads with number of threads defined above
for each thread
    allocate memory for threadInfo
    assign threadInfo's matrix to matrix created earlier
    assign threadInfo's startIndex to the matrix size divided by the number
    of threads and multiplied by the thread number
    assign threadInfo's endIndex to the matrix size divided by the number
    of threads and multiplied by the thread number incremented by one
    create new thread with default attributes and executing transposition on threadInfo
end for
for each thread
```

```
        suspend execution of calling thread until target thread has terminated
end for
```