ELEN 4020 Lab 2: OpenMP and Pthreads

Junaid Dawood (1094837), Xongile Nghatsane (1110680), Marissa van Wyngaardt (719804)

***Abstract*—This report discusses the parallelisation of matrix transposition algorithms in C, using OpenMP and Pthreads. The implemented algorithms include a naive element-wise swapping approach, a diagonal based row-column swap approach, and a block based approach. From a comparative analysis of the *runtimes* of the implemented algorithms, it was found that the Pthreads implementation of the block based algorithm yielded the best performance once the sizes of input matrices are sufficiently large. This is thought to be due to the even work-sharing implemented for this approach, that was not achieved elsewhere. Additional throttling has been identified in the implementations and has been linked to false sharing and cache misses.**

I. Introduction

This document discusses the implementation of three matrix transposition algorithms in C. Initially, these are implemented serially; they are subsequently parallelised through the use of OpenMP and Pthreads [1], [2]. This report also discusses tradeoffs with respect to parallelisation, in terms of computation times. In particular, the point at which the communication overhead of parallelisation is outweighed by the performance gain of said parallelisation is a key concern of this lab exercise.

II. Structs and Helper Functions

This lab exercise involved the reuse of a rank 2 tensor *struct* created for the first lab exercise. This struct is used to store matrices, and metadata i.e. the dimensions of the matrix. Additional helper functions for the initialisation of the contents of the tensor, and the freeing of memory dynamically allocated for their content are implemented.

Additionally, other *structs* are required for achieving parallelisation using Pthreads; in order to work-share between threads. That is, these *structs* contain additional data which indicates to the transposition algorithm the *segment* of the matrix which is to be parallelised by a certain thread.

III. Transposition Algorithms

This section describes the three matrix transposition algorithms implemented, which were ultimately parallelised. The algorithms implemented consist of a naive approach, a diagonal row-per-column transposition, and a block based transposition with intra-block transposition and inter-block position swapping. All of the algorithms are constrained to $O(1)$ space complexity.

It is important to note that the naive approach and the diagonal approach are very similar, and in the case of a serial implementation, are identical. Hence, differences in implementation only manifest in the way in which they are parallelised. The pseudocode for all the algorithms, including parallelisations, can be found at the end of this document.

A. Naive Approach

The naive approach involves iterating through every cell above the main diagonal. The contents of the cells are then swapped with the corresponding cells in the lower half of the matrix as shown in the diagram below.

When parallelised, there would ideally be a thread allocated to each pair of elements to be swapped; in order for the implementation to remain within the philosophy of the naive approach.

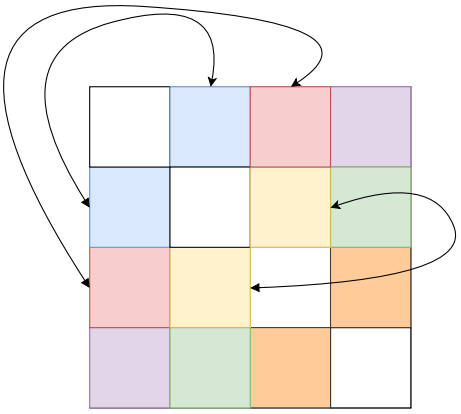

Fig. 1: Naive matrix transposition algorithm.

B. Diagonal Approach

The diagonal approach is similar to the naive approach as mentioned above. The difference lies in the manner of the iteration and thus the order of the swapping. That is, in the diagonal approach it is the diagonal which is iterated along: with a partial transposition occurring at each element on the diagonal. Each element on the diagonal naturally exists as the intersection between a column and a row: the swapping of said row and column is what constitutes the partial transposition, as shown below.

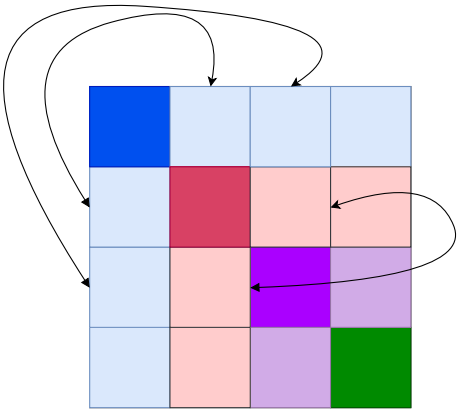

Fig. 2: Diagonal matrix transposition algorithm.

That said, within a serial implementation the distinction of iterating along each cell compared to each cell on the diagonal is lost; thus it becomes equivalent to the naive approach. In a parallel approach, the swapping of rows with columns would be handled by a single thread per diagonal; such that this method becomes distinct from the naive approach.

C. Block Approach

In this approach, the matrix to be transposed is decomposed into 2x2 sub-matrices (blocks). These blocks are individually transposed and then they are swapped with the corresponding blocks below or above the main diagonal. The blocks on the main diagonal are simply transposed, and do not go through any swapping process. Naturally, the use of 2x2 blocks is a limiting factor in that it can only partition matrices of even dimensions.

This can be seen in the diagram shown below: note the individual internal transpositions of the 2x2 blocks, and the subsequent swapping with like-coloured blocks. The 2x2 blocks which exist along the diagonal of the matrix naturally only require the internal transposition step and not the swapping step.

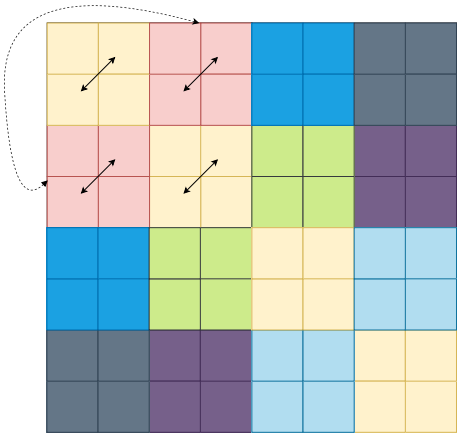

Fig. 3: Block-based matrix transposition algorithm.

A simple parallelisation of this approach would be to allocate a thread to each block pair being processed. Thus, this thread would transpose the individual blocks, as well as perform the swap of the two blocks.

IV. OpenMP Parallelisation

This section discusses the parallelisation of the aforementioned serial transposition algorithms using OpenMP. This is done entirely through the use of compiler directives, specifically those used for the parallelisation of for loops, using static scheduling.

A. Naive Approach

The parallelisation of the naive approach is achieved through essentially assigning a thread to each cell-swap operation that is required for the transposition. This is achieved by parallelising the inner *for* loop such that threads are assigned on a per swap basis. There are $\frac{N(N+1)}{2}$ swap operations in total, shared across M threads i.e. $\frac{N(N+1)}{2M}$ swaps per thread ideally.

Parallelising the outer loop, instead of the inner loop, allows for a far smaller number of fork-join operations to be used in the program's execution. Parallelising the inner loop requires fork-join operations on a per row basis, resulting in more overhead. Despite this, in order to stay within the philosophy of the naive approach, and keep it distinct from the diagonal approach, parallelisation of the inner loop is required.

In addition, the use of nested parallelism also makes for a less efficient solution: it is disabled by default in OpenMP, such that the second parallel region is ignored [3].

B. Diagonal Approach

The parallelisation of the diagonal approach corresponds to the assignment of a thread to each diagonal element of the matrix; such that a thread is tasked with all of the row-column swaps corresponding to the diagonal element it is assigned to. This is done simply by using OpenMP to parallelise the outer *for* loop of the serial implementation.

Unlike the naive approach, a thread is not assigned a pair of elements to swap, instead a single thread is responsible for all of the associated swaps of its row(s) and column(s). In addition, whilst it can be said that the number of swaps per thread is constant in the naive case, this is not true of the diagonal implementation, That is, threads allocated elements further along the diagonal will inherently have to do fewer swaps than those allocated elements earlier in the diagonal.

C. Block Approach

The parallelisation of the block approach using OpenMP would ideally involve assigning threads to an even number of block-transposition and block-swap operations. However, this is not easily achieved using OpenMP; for similar reasons to the diagonal approach.

That is, OpenMP is used to parallelise the outer *for* loop of the serial implementation. In this way, each thread is given an variable number of blocks to transpose and swap. In total, given the use of 2x2 blocks, the total number of blocks is given by $noBlocks = \frac{L(L+1)}{2}$, where $L = \frac{N}{2}$ for a $N \times N$ matrix.

V. Pthreads Parallelisation

The parallelisation of the serial implementations of matrix transposition using Pthreads is more involved than when using OpenMP. That is, the explicit manual definition of the work-sharing is required when using Pthreads, which is not the case when using OpenMP.

Generally, this involves manually splitting work into *chunks*, and assigning said chunks to individual threads. This is generally done by defining the scope of a thread's

work, placing the scope defining variables into a *struct* along with a reference to the original matrix. This *struct* is then passed to the relevant transposition method upon invocation.

A. Diagonal Approach

As per the above discussion, parallelisation of the diagonal transposition approach involves defining the manner in which work is shared amongst threads. This is achieved by assigning each thread a number of diagonal elements, based on the size of the matrix and the number of threads, such that each thread is allocated the same number of diagonal elements.

Thus, each thread is assigned a start and end row in the matrix, as per the snippet below. Each thread handles each diagonal element it is responsible for; performing the row-column swaps for each diagonal.

```
typedef struct forDiagonal
{
    rank2Tensor* srcMat;
        int start;
        int end;
} forDiag;
```

This does not achieve equal work-sharing due to different diagonal elements implying a different number of swap operations, as is the case with the OpenMP parallelisation.

B. Block Approach

The parallelisation of the block transposition using Pthreads is more complex than the parallelisation of the diagonal method. That said, the basic premise remains the same: each thread is assigned a certain number of blocks which it is responsible for; performing the necessary intra-block transpositions and subsequent block swaps, thus achieveing even work-sharing. The listing below shows the *struct* passed on a per-thread basis, to the block transposition function.

```
typedef struct forBlock
{
        rank2Tensor* srcMat;
        int startBlock;
        int noBlocks;

} forBlock;
```

The nature of the block implementation requires a coordinate mapping system which relates the rows and columns of the original matrix to a number of sequenced blocks. This is done within the called function, once per thread, and as such, this conversion overhead is not thought to be significant.

VI. Results and Discussion

Tests were conducted on a per algorithm basis, for each parallelisation method used. In addition, the serial implementations of the algorithms were also tested, for comparison. Testing consisted of supplying matrices of increasing size to the algorithms, measuring the time taken for the algorithms to complete the transpositions.

Timing was done using OpenMP's *get wtime* function; being used before and after calls to the algorithms in order to obtain their completion times. This was used in all implementations i.e. serial, parallel (Pthreads), and parallel (OpenMP), in order to ensure a fair timekeeping process across all tests.

The test machine that the code was run on had the following specifications:

- CPU: Intel i5 8350u (4C 8T) 3.6 GHz turbo frequency (256 kB L1, 1 MB L2, 6 MB L3) [4].
- RAM: 16 GB 2400 MHz

The tables and graphs shown in this section are automatically generated in LaTeXfrom the output CSV files from running the respective implementations. This is to ensure that data is not compared across multiple test runs.

TABLE I: Results for Serial Implementation (Times in seconds)

$MatrixSize$	$NaiveApproach$	$DiagonalApproach$	$BlockApproach$
128	0.000116	0.000121	0.000182
1024	0.005486	0.006021	0.006329
2048	0.021413	0.030528	0.023004
4096	0.082359	0.078588	0.091385
8196	0.343624	0.348571	0.408157
16392	1.829208	2.050170	2.425582

TABLE II: Results for OMP Parallelisation (Times in seconds)

$MatrixSize$	$NaiveApproach$	$DiagonalApproach$	$BlockApproach$
128	0.010053	0.000099	0.000138
1024	0.051545	0.001749	0.001755
2048	0.059948	0.006492	0.007013
4096	0.124011	0.030935	0.028164
8196	0.295262	0.145915	0.166874
16392	0.915424	1.321332	1.088278

TABLE III: Results for Pthreads Parallelisation (Times in seconds)

$MatrixSize$	$DiagonalApproach$	$BlockApproach$
128	0.001125	0.000787
1024	0.002731	0.002520
2048	0.007566	0.007672
4096	0.030284	0.033484
8196	0.145410	0.153985
16392	1.586490	0.977752

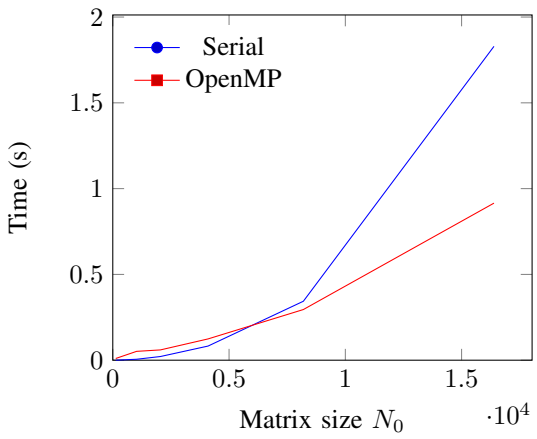

Fig. 4: Comparison of Naive Implementations of matrix transpositions.

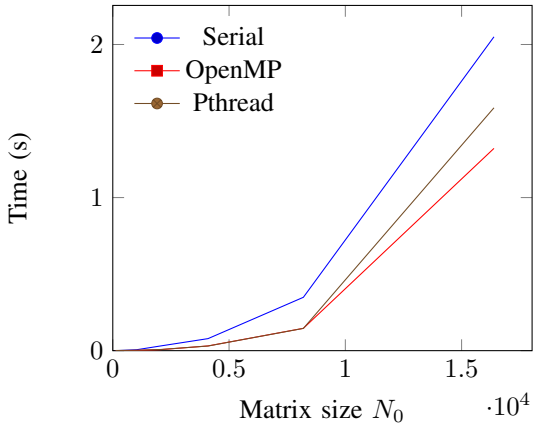

Fig. 5: Comparison of Diagonal Implementations of matrix transpositions.

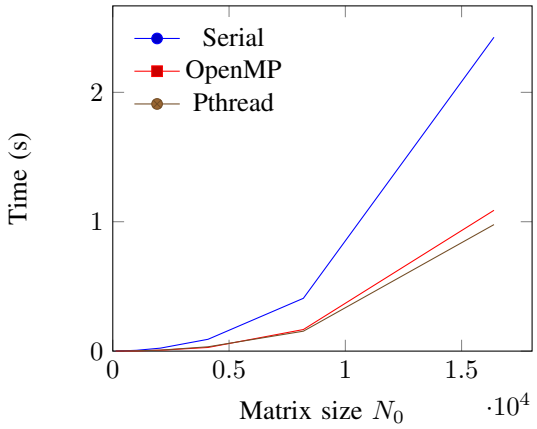

Fig. 6: Comparison of Block Implementations of matrix transpositions.

A. *Discussion*

From the results, it is clear that the overhead involved in forking and joining of threads causes the performance of serial and parallel approaches to be similar for small matrices. As far as comparisons between parallel implementations, it is clear that the block approach begins to perform better than the diagonal approach beyond a threshold. This is thought to be due to the more efficient work-sharing implied by the block approach.

In addition, there is an apparent divergence between the performance of the block implementations between the OpenMP and Pthreads implementations. This is thought to be due to the fact that the manual nature of the Pthreads implementation allowed for even work-sharing, whereas the OpenMP approach did not. This is discussed further below.

There are various issues present within the current (parallel) implementations that manifest in the trends present in the results. The first obvious fault is the lack of (near) even work-sharing between threads in some implementations; specifically, block and diagonal (OpenMP) and diagonal (Pthreads). As a result, the threads which are allocated more of the work would ultimately take the longest to complete their work. This time would naturally be longer than the time taken if work-sharing were implemented evenly.

Another issue present within the current implementations is false sharing [5]. This refers to scenarios which involve threads which access independent variables which share the same cache line. When one thread writes over their independent variable, it is required that the other thread invalidates its copy of the cache line, having to copy the new (altered) cache line in order to continue. This is especially problematic if each thread is making several alterations to the cache line. In addition, as the number of threads increases, it naturally follows that this cache invalidation could become a more common occurrence.

The current implementations offer no mechanisms with which to prevent this. Given that this exercise is mainly concerned with matrices, it follows that false sharing is a common occurrence if not accounted for. That said, these algorithms achieve $O(1)$ space complexity, as per the current implementation; achieving this whilst avoiding false sharing would be a difficult exercise.

Lastly, caching and more specifically cache misses are an important factor to consider when discussing matrix transposition implementations [6]. This is to the extent that there have been studies into cache efficient matrix transposition implementations [7], [8]. A cache miss refers to a thread failing to find some variable within the lowest cache level, causing the thread to have to fetch the data from a higher cache level, eventually reaching main memory if the data is not found at any cache level. Generally speaking, the inter-level latencies increase whilst travelling up the hierarchy e.g. from L1 to L2 cache the latency is minimal compared to the L3 cache to main memory latency.

VII. Conclusion

Three implementations of matrix transposition algorithms have been presented. This has been done with specific reference to the parallelisation of these methods, implemented in C. In doing so, the use of OpenMP and Pthreads as tools for achieving the parallelisation of the algorithms has also been presented, alongside a performance comparison of the two on a per algorithm basis. It has been shown that a naive method has a clear advantage within a serial implementation. However, when parallelised, the block approach quickly begins to outperform both the naive and diagonal approaches, as the size of the input matrices increases.

References

[1] *Home - openmp*, https://www.openmp.org/, (Accessed on 15/03/2019),

[2] *Linux tutorial: Posix threads*, https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html, (Accessed on 15/03/2019),

[3] *Chapter 3: Nested*, http://ppc.cs.aalto.fi/ch3/nested/, (Accessed on 11/03/2019),

[4] *Intel core i5-8350u processor (6m cache, up to 3.60 ghz) product specifications*, https://ark.intel.com/content/www/us/en/ark/products/124969/intel-core-i5-8350u-processor-6m-cache-up-to-3-60-ghz.html, (Accessed on 15/03/2019),

[5] *Mechanical sympathy: False sharing*, https://mechanical-sympathy.blogspot.com/2011/07/false-sharing.html, (Accessed on 15/03/2019),

[6] *Cache misses*, https://docs.roguewave.com/threadspotter/2011.2/manual_html_linux/manual_html/miss_ratio.html, (Accessed on 15/03/2019),

[7] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," Feb. 2000, pp. 195–205, ISBN: 0-7695-0550-3. DOI: 10.1109/HPCA.2000.824350.

[8] D. Tsifakis, A. Rendell, and P. Strazdins, "Cache oblivious matrix transposition: Simulation and experiment," vol. 3037, May 2004, pp. 17–25. DOI: 10.1007/978-3-540-24687-9_3.

PSEUDOCODE

Serial

Algorithm 1 Naive Approach (Serial)

```
1: procedure NAIVETRANSPOSITION(rank2Tensor t1)
2:     t1 ← first rank 2 tensor
3:     for i=0..t1.rows do
4:         for j=0..(i-1) do
5:             swap(t1[i][[j],t2[j][i])
```

Algorithm 2 Diagonal Approach (Serial)

```
1: procedure DIAGONALTRANSPOSITION(rank2Tensor t1)
2:     t1 ← first rank 2 tensor
3:     for i=0..t1.rows do
4:         for j=i..t1.cols do
5:             swap(t1[i][[j],t2[j][i])
```

Algorithm 3 Block Approach (Serial)

```
1: procedure BLOCKTRANSPOSITION(rank2Tensor t1)
2:     t1 ← first rank 2 tensor
3:     for i=0..blockCount do
4:         Transpose block i
5:         Transpose opposite block
6:         Swap block i and opposite block
```

OpenMP

Algorithm 4 Naive Approach (OpenMP)

```
1: procedure NAIVETRANSPOSITION(rank2Tensor t1)
2:     t1 ← first rank 2 tensor
3:     for i=0..t1.rows do
4:         #pragma omp parallel for
5:         for j=i..t1.cols do
6:             swap(t1[i][[j],t2[j][i])
```

Algorithm 5 Diagonal Approach (OpenMP)

```
1: procedure DIAGONALTRANSPOSITION(rank2Tensor t1)
2:     t1 ← first rank 2 tensor
3:     #pragma omp parallel for
4:     for i=0..t1.rows do
5:         for j=i..t1.cols do
6:             swap(t1[i][[j],t2[j][i])
```

Algorithm 6 Block Approach (OpenMP)

```
1: procedure BLOCKTRANSPOSITION(rank2Tensor t1)
2:     t1 ← first rank 2 tensor
        #pragma omp parallel for
3:     for i=0..blockCount do
4:         Transpose block i
5:         Transpose opposite block
6:         Swap block i and opposite block
```

PThreads

Algorithm 7 Diagonal Approach (Pthreads)

```
1: procedure WORKASSIGNMENT
2:     threads ← list of threads
3:     N_0 ← matrix row count
4:     threadCount ← number of available threads
         perThread=N_0/threadcount
5:     for i=0..threadCount-1 do
6:         threads[i].startRow=i*perThread
7:         threads[i].startRow=(i+1)*perThread
       threads[threadCount-1].endRow=N_0
8: procedure DIAGONALTRANSPOSITION(rank2Tensor t1,startRow,endRow)
9:     t1 ← first rank 2 tensor
10:    startRow ← start of thread's diagonals
11:    endRow ← end of thread's diagonals
12:    for i=startRow..endRow do
13:        for j=i..t1.cols do
14:            swap(t1[i][[j],t2[j][i])
```

Algorithm 8 Block Approach (Pthreads)

```
1: procedure WORKASSIGNMENT
2:     threads ← list of threads
3:     N_0 ← matrix row count
4:     threadCount ← number of available threads
         blockCount=(N_0/2)(N_0/2 +1)/2
         perThread=blockCount/threadcount
5:     for i=0..threadCount-1 do
6:         threads[i].startBlock=i*perThread
7:         threads[i].endBlock=(i+1)*perThread
       threads[threadCount-1].endBlock=blockCount
8: procedure BLOCKTRANSPOSITION(rank2Tensor t1,startBlock,endBlock)
9:     t1 ← first rank 2 tensor
10:    startRow ← start of thread's diagonals
11:    endRow ← end of thread's diagonals
12:    for blockNo = startBlock..endBlock do
13:        transpose(blockNo)
14:        transpose(alternateBlock)
15:        swap blockNo and alternateBlock
```