ELEN4020A: Data Intensive Computing
Laboratory Exercise 2

Lynch Mwaniki 1043475
Madimetja Sethosa 1076467
Teboho Matsheke 1157717

18/03/2019

1 In-Place Matrix Transposition

In this lab, three different In-place matrix transposition algorithms are implemented and tested against each other: naive approach, Diagonal and Block transpose algorithms. The Diagonal and Block transposition algorithms are implemented using Pthreads and the OpenMP library. The naive approach is implemented serially and using threading by implementing it using the OpenMP library.

2 Basic Algorithm

This algorithm uses two nested for loops to transpose a 2D matrix. The outer and inner loops are used to iterate through the rows and columns of the matrix respectively. They iterate along the main diagonal. Each value is swapped with its corresponding transposed location, $A[i, j]$ with $A[j, i]$. This iteration only needs to be go through either the top or bottom triangular half of the matrix.

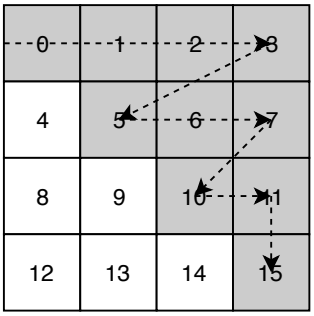

Figure 1: Traversal of elements in Upper Triangle of a 4x4 Matrix

3 OpenMP Naive Algorithm

This is a low level improvement of the basic algorithm. The OpenMP *#pragma* directives are used to parallelize the two nested for loops. The outer loop is used to iterate along the matrix's rows and the inner loop iterates along the columns.

4 Diagonal Transpose Algorithm

This algorithm transposes the matrix elements by traversing along the matrix diagonal. As with the basic algorithm, each value is swapped with its corresponding transposed location, A[i, j] with A[j, i]. The OpenMP and Pthread methods were implemented and are discussed in the sections that follow.

4.1 Pthreads

When implementing Pthreads for the Diagonal algorithm, we define a struct named *ThreadDataDiagonal*, with the following properties: diagonal Index, pointer to the matrix, and an ID. We then define or set the number of threads to be used. In order to keep track of the row and column to be transposed, a global variable, nextDiagonal, is defined. Each thread's diagonal index property is used to iterate through the columns, of the row associated to the diagonal, of the matrix and transpose the corresponding elements. Upon completion, a *Pthread Mutex Lock* section is created,

whereby, if a thread's diagonal index is less than size of the matrix, $N-1$, it is set to nextDiagonal, thereafter, nextDiagonal is incremented by one. Otherwise, the thread diagonal index is set to $N-1$ at which point the thread will exit. The pseudo code for this algorithm can be found in Appendix A Algorithm 10.

4.2 OpenMP

This implementation follows from the basic algorithm. In that it uses two nested for loops to transpose a 2D matrix. In this implementation the OpenMP *#pragma* directives are used to parallelize the for loop iterations. The outer loop is used to iterate along the matrix's diagonals and the inner loop is used to iterate along the columns. The two nested for loops are enclosed in a parallel section with a pointer to the matrix as shared and the indexing for the diagonal index variable made private. The outer for loop is parallelized with a schedule. The parameters assigned to the schedule is *kind = dynamic* and a *chunk size* = 10. These mean that each thread is given a 10 rows to iterate through. A *nowait* directive is also included, thus allowing a thread to receive the next chunk of indices without waiting for the other threads.

5 Block Transpose Algorithm

The Block Transpose Algorithm consists of three steps. The first step requires a matrix to be divided into blocks of a specified size. The second step involves transposing the elements in the blocks. The last step requires the blocks to be swapped except those on the diagonal. For this lab, a block size of 2x2 was chosen for the block transposition algorithms to use. Figure 2 shows how a 8x8 matrix is decomposed into 2x2 blocks, which are just sub matrices of size 2x2. The algorithms implemented make use of 2D indexing to assign blocks to threads and represent the 2D input matrices as 1-Dimensional arrays. The 2D indexes of the blocks are mapped to the corresponding 1D indices using the *blockElementsIndex* function which can be seen in Appendix A Algorithm 4. The swapping of the blocks is implemented in *swapBlocks* which can be found in Appendix A Algorithm 5.

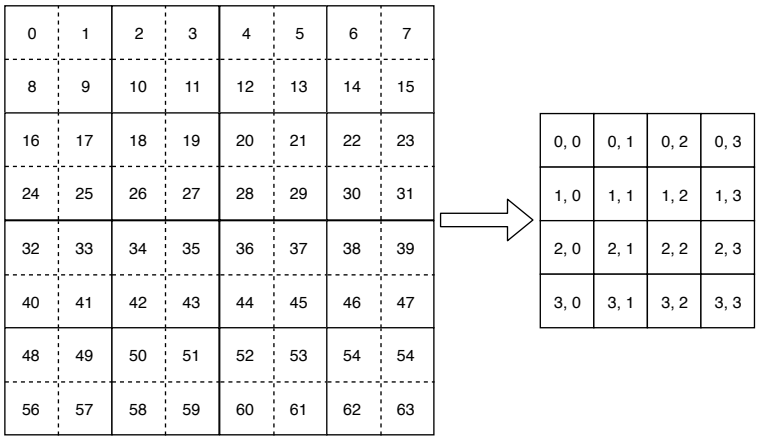

Figure 2: Figure showing a 8x8 matrix decomposed into 2x2 blocks

5.1 Pthreads

The block transposition algorithms were implemented using the *pthread* library. Two different approaches were implemented and are discussed in the sections that follow.

5.1.1 Algorithm 1

This algorithm performs block transposition by traversing through the upper triangle of a matrix, as shown in Figure 1. When creating the threads a struct called *ThreadDataBlock* is generated, with the following members: ID, row, column, matrix size and a pointer to the matrix, and assigned to each thread privately. The row and column are indices indicating the block to be worked on. Each thread is assigned a block along the path being traversed. If a thread is assigned a block on the diagonal, it will only transpose the relevant elements in that block. If a thread is assigned a non-diagonal block, it will transpose the elements in that block, interchange the indices, A[i, j] to A[j, i], and swap the corresponding block and finally swap the blocks. Once a thread has completed those steps, a

Pthread Mutex Lock section is reached.

This section is needed to update two shared global variables named, *next_block_diagonal* and *next_block_column*. These variables contain the row and column of the next block that needs to be operated on. The thread saves *row* = *next_block_diagonal* and *column* = *next_block_column*. After the thread saves the information, checks are performed to update the global indexing variables. This section checks that once the *next_block_column* reaches the boundary, of the 2D matrix representing the blocks, the *next_block_diagonal* is incremented by one and the *next_block_column* is set as the value in *next_block_diagonal*. The mutex is unlocked and the values row and column values for the thread are checked for boundary conditions. If the row and column received is equal to $matrix\_size - 1$, the thread will exit. The pseudo code code for this algorithm can be found in Appendix A Algorithm 12.

5.1.2 Algorithm 2

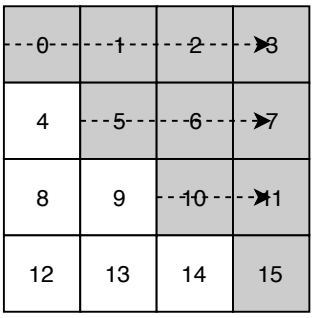

Figure 3: Allocation of Blocks to Threads as chunks

Similar to Algorithm 1, this algorithm traverses the upper triangle of the matrix, but instead of traversing the blocks lineally, they are traversed in a parallel format as shown in Figure 3. Threads are created and each is assigned to a row starting from the diagonal block. If the block assigned to the thread is on the diagonal, then the block is only transposed, else the adjacent block is calculated, transposed and then swapped. The first thread to finish the row will lock the global variable *diag_index*, copy its value and update it. For a more logical flow of the algorithm, the pseudo code is provided in Appendix A Algorithm 9.

5.2 OpenMP block

This implementation follows that of pthread algorithm 2. It makes use of two nested for loops to transpose a 2D matrix. The first for loop is for traversing the diagonal index, where else the second on is for traversing the blocks in each row. The algorithm makes use of *#progma omp parallel* with the indices of the two for loops being private. The for loop is made parallel with the schedule set to dynamic, this makes sure that each thread executes a chunk of iterations and then requires another chunk of iteration until there is no chunk remaining. The pseudo code is provided in Appendix A,9.

6 Performance Comparison of different algorithms

The seven algorithms implemented were run on matrix sizes N = 128, 1024, 2048, and 4096. The threaded algorithms were run on each size using 4 threads and 8 threads to gauge the performance of the algorithms. The algorithms were run on a desktop computer with the following specs: an Intel Core i7-6700 CPU with 8 cores running at 3.40 GHz and 8 GB DDR3 RAM. No other programs were open to ensure that no other known process will slow down the algorithms. Ideally, the algorithms should have been run on a Linux machine with no desktop environment, but this was not available to the group. The results of the implemented algorithms can be seen in Table 1 - 3.

Table 1: Results showing time taken to compute algorithms with 4 threads

No. of threads = 4						
$N_0 = N_1$	**Pthreads**			**OpenMP**		
	Diagonal	**Blocked 1**	**Blocked 2**	**Naive**	**Diagonal**	**Blocked**
128	0.001081	0.002200	0.000658	0.000483	0.000275	0.000668
1024	0.004056	0.020829	0.004910	0.005325	0.002941	0.006797
2048	0.020492	0.027299	0.028489	0.031684	0.021544	0.032312
4096	0.094026	0.111373	0.127449	0.211720	0.128334	0.145914

Table 2: Results showing time taken to compute algorithms with 8 threads

No. of threads = 8						
$N_0 = N_1$	**Pthreads**			**OpenMP**		
	Diagonal	**Blocked 1**	**Blocked 2**	**Naive**	**Diagonal**	**Blocked**
128	0.001106	0.003101	0.000669	0.000671	0.000155	0.000489
1024	0.002545	0.023252	0.003740	0.004816	0.002402	0.004610
2048	0.016938	0.029786	0.017196	0.024770	0.020085	0.022922
4096	0.081811	0.118155	0.073397	0.133002	0.115294	0.093971

Table 3: Results showing time taken to compute the Basic algorithm

$N_0 = N_1$	Basic
128	0.000320
1024	0.010438
2048	0.079379
4096	0.462447

7 Analysis of Results

Generally, the basic algorithm was out performed by all the other types of algorithms that were implemented. In table 1 where 4 threads were used for both the openMP and pthread methods, the diagonal algorithm performed better for the both implementations except for N4096 (whereby the block algorithms performed better).

8 Conclusion

This report detailed three different matrix transposition algorithms implemented in the C programming language namely: Basic or naive approach, Diagonal and Block Transpose algorithms. The Diagonal and Block transposition algorithms were threaded and implemented using two different threading libraries namely: Pthreads and OpenMP. The Naive approach was also threaded using OpenMP. The algorithms were timed and the results were analyzed. It was discovered that the pthreads algorithm 2 out performs the rest of the algorithms for N > x.

A Pseudo code

A.1 allocateMatrix

Algorithm 1: Allocate Matrix in Memory

Result: Integer pointer, pointing to first element of matrix.
Input : Total Number of Elements

```
result_ptr = (int*)calloc(number_of_elements, sizeof(int))
return result_ptr
```

A.2 getElementLocation2D

Algorithm 2: Get Location of Element in 2D matrix when represented as 1D Array

Result: Integer containing position of element in 1D matrix
Input : 1D Array containing indexes
Input : Dimension N

```
row = index[0]
column = index[1]
return (row * N) + column
```

A.3 retrieveElement2D

Algorithm 3: Retrieve Element from 2D matrix

Result: Integer value of the element in the 2D matrix co-ordinates.
Input : Pointer to matrix, 1D Array containing indexes
Input : Dimension N

```
return *(matrix_ptr + getElementLocation2D(indexes, N))
```

A.4 blockElementsIndex

Algorithm 4: Retrieve block Elements indices

Result: 4 index values of a block in the matrix
Input : blockIndex, Dimension N

```
Num_Blocks = N/2
*blockElemIndex = (blockIndex*2)+ ((int)(blockIndex/Num_Blocks))*N;
blockElemIndex[1]=*blockElemIndex+1;
blockElemIndex[2]=*blockElemIndex+N;
blockElemIndex[3]=*blockElemIndex+(N+1);
return blockElemIndex
```

A.5 swapBlocks

Algorithm 5: transpose two blocks

Result: block transposed *matrix
Input : *matrix, *blockA, *blockB

```
swapElements(matrix+indexBlockA[0], matrix+indexBlockB[0]);
swapElements(matrix+indexBlockA[1], matrix+indexBlockB[1]);
swapElements(matrix+indexBlockA[2], matrix+indexBlockB[2]);
swapElements(matrix+indexBlockA[3], matrix+indexBlockB[3]);
```

A.6 Basic Algorithm

Algorithm 6: Transpose a square 2D Matrix using Naive Approach in Serial execution

```
Result: A 2D square transposed matrix
Input  : Pointer to matrix
Input  : Matrix Size N
for i = 0 to N - 1 do
    for j = i to N-1 do
        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
```

A.7 OpenMP Naive Transpose Algorithm

Algorithm 7: Transpose a square 2D Matrix using Naive Approach in Parallel execution

```
Result: A 2D square transposed matrix
Input  : Pointer to matrix
Input  : Matrix Size N
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) nowait
for i = 0 to N - 1 do
    for j = i to N-1 do
        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
```

A.8 OpenMP Diagonal Transpose Algorithm

Algorithm 8: Transpose a square 2D Matrix using Naive Approach in Parallel execution

```
Result: A 2D square transposed matrix
Input  : Pointer to matrix
Input  : Matrix Size N
Set number of threads for OpenMP
Create variables diagonalIndex, chunk_size and temp
Assign chunk_size to 10
#pragma omp parallel shared(matrix) private(diagonalIndex, temp)
#pragma omp for schedule(dynamic, chunk_size) nowait
for diagonalIndex = 0 to N - 2 do
    for int column = diagonalIndex + 1 to N-1 do
        temp = matrix element indexed at diagonalIndex and column
        matrix element at [diagonalIndex, column] position = matrix element indexed at [column,
          diagonalIndex] position
        matrix element indexed at [column, diagonalIndex] position = temp
```

A.9 OpenMP Block Transpose Algorithm

Algorithm 9: Transpose a square 2D Matrix using Block Transpose Algorithm

```
   Result: A 2D square transposed matrix
 1 set number of threads for Pthreads
 2 in main:
 3 Allocate memory for 2D matrix of size N and fill with random integers
 4 Calculate the number of blocks (N_blocks)
 5 foreach thread do
 6     #pragma omp parallel private(i,j)
 7     #pragma omp for schedule(dynamic)  for  j = diagIndex to N_blocks do
 8         temp = matrix element indexed at diagIndex and j
 9         get blockA element indices using the blockElementsIndex function
10         transpose the elements in blockA
11         if j!=diagIndex then
12             temp = matrix element indexed at j and diagIndex
13             get blockB element indices using the blockElementsIndex function
14             transpose the elements in blockB
15             transpose blockA and blockB using the swapBlocks() function.
```

A.10 Pthread Diagonal Transpose Algorithm

Algorithm 10: Transpose a square 2D Matrix using Diagonal Algorithm

```
   Result: A 2D square transposed matrix
 1 Set number of threads for Pthreads.
 2 Create global variable nextDiagonal and set it to number of threads.
 3 struct ThreadData[ int* matrix; int diagIndex;]
 4 in main:
 5 Allocate memory for 2D matrix of size N and fill with random integers.
 6 Create 1D matrix of size number of Pthreads for storing structs of type ThreadData.
 7 Create Pthread threads with number assigned above.
 8 foreach thread do
 9     Assign threadData's matrix pointer to the 2D matrix created earlier.
10     Assign threadData's diagIndex to the ID of the thread being created.
11     while true do
12         for int column = diagIndex + 1 to N - 1 do
13             temp = matrix element indexed at diagonalIndex and column
14             matrix element at [diagonalIndex, column] position = matrix element indexed at [column,
                diagonalIndex] position
15             matrix element indexed at [column, diagonalIndex] position = temp
16         Mutex Lock
17         if nextDiagonal <= N - 1 then
18             diagIndex = nextDiagonal
19             Increment nextDiagonal by 1.
20         else
21             diagIndex = N - 1
22         Mutex Unlock
23         if diagIndex == N - 1 then
24             break from while loop
25     exit pthread
26 foreach thread do
27     Join thread and wait for other threads to exit and join
```

A.11 Pthread Block Transpose Algorithm 1

Algorithm 11: Compute next block to be worked on in Algorithm 12

Result: The row and column of the next block to be transposed.
Input : N = size of input matrix

Compute size of matrix of blocks: $N\_block = N/2$
Increment next_block_column by 1.
if *next_block_column* ***mod*** *N_block* ***equals*** *0* ***and*** *next_block_column* ***not equal to*** *0* **then**
    Increment next_block_diagonal by 1.
    Set next_block_column = next_block_diagonal.
if *next_block_column >= N_block* ***and*** *next_block_diagonal >= N_block* **then**
    Set next_block_diagonal = N_block.
    Set next_block_column = N_block.
return

Algorithm 12: Transpose a square 2D Matrix using Block Transpose Algorithm

Result: A 2D square transposed matrix.

Set number of threads for Pthreads.
Create global variables **next_block_column**, **next_block_diagonal**.
Create pthread mutex **computeNextBlockLock**.
struct ThreadDataBlock[int* matrix; int row, column, N;]
in main:
Allocate memory for 2D matrix of size N and fill with random integers.
Create 1D matrix of the size of number Pthreads for storing structs of type ThreadDataBlock.
Create pthread threads with number assigned above.
foreach *thread* **do**
    Assign threadDataBlock's matrix pointer to the 2D matrix created earlier.
    Assign threadDataBlock's row to next_block_diagonal.
    Assign threadDataBlock's column to next_block_column.
    Call computeNextBlock function.
    Calculate the number of blocks (N_blocks).
    while *true* **do**
        Set index[2] = [threadDataBlock.row, threadDataBlock.column].
        Get blockA element indices at block located at *index* using the blockElementsIndex function.
        Transpose the elements in blockA.
        if *index[0]* ***not equal to*** *index[1]* **then**
            Set index[0] = threadDataBlock.column
            Set index[1] = threadDataBlock.row
            Get blockB element indices at block located at *index* using the blockElementsIndex function.
            Transpose the elements in blockB.
            Transpose blockB using the swapBlocks() function.
        computeNextBlockLock Lock
        Assign threadDataBlock's row to next_block_diagonal.
        Assign threadDataBlock's column to next_block_column.
        Call computeNextBlock function.
        computeNextBlockLock Unlock
        if *threadDataBlock.row == N_block* ***and*** *threadDataBlock.column == N_block* **then**
            break from while loop
    Exit pthread
foreach *thread* **do**
    Join thread and wait for other threads to exit and join

A.12 Pthread Block Transpose Algorithm 2

Algorithm 13: Transpose a square 2D Matrix using Block Transpose Algorithm

```
   Result: A 2D square transposed matrix
 1 set number of threads for Pthreads
 2 create global variable nextDiagBlock
 3 struct ThreadData[ int* matrix; int diagIndex;]
 4 in main:
 5 Allocate memory for 2D matrix of size N and fill with random integers
 6 Create 1D matrix of the size of number Pthreads for storing structs of type ThreadData
 7 Create pthread threads with number assigned above.
 8 Calculate the number of blocks (N_blocks)
 9 foreach thread do
10     Assign threadData's matrix pointer to the 2D matrix created earlier.
11     Assign threadData's diagIndex to the ID of the thread being created.
12     while true do
13         for  j = diagIndex to N_blocks do
14             temp = matrix element indexed at diagIndex and j
15             get blockA element indices using the blockElementsIndex function
16             transpose the elements in blockA
17             if j!=diagIndex then
18                 temp = matrix element indexed at j and diagIndex
19                 get blockB element indices using the blockElementsIndex function
20                 transpose the elements in blockB
21                 transpose blockA and blockB using the swapBlocks() function.
22         Mutex Lock
23         if nextDiagBlock < N_blocks - 1 then
24             diagIndex = nextDiagBlock++;
25         else
26             diagIndex=N_blocks - 1
27         Mutex Unlock
28         if diagIndex == N_blocks - 1 then
29             break from while loop
30     exit pthread
31 foreach thread do
32     Join thread and wait for other threads to exit and join
```