UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

ELEN4020A: Data Intensive Computing: Lab 2

Sbonelo Mdluli(1101772), Heemal Ryan(792656), Haroon Rehman(1438756)

March 19, 2019

1 INTRODUCTION

Matrix transposition is a key operation in various scientific and mathematical applications. Matrix transposition converts a M-rows-by-N columns array to a N-rows-by-M columns array. This lab introduces students to parallel programming through the use of Pthread and OpenMP libraries. Matrix transposition of large matrices can be a time and space expensive operation. Parallelism is used to improve the performance of the operation and to improve space requirement matrix transposition is to be done in place. This report will discuss the different matrix transposition algorithms that are developed with their performance for a 2D matrix with varying dimensions.

2 MATRIX TRANSPOSE ALGORITHMS

The generated matrix is stored using row major ordering in a 1-D array with the exception of the block algorithm which uses 2-D arrays in C++ to store the matrix. This is especially useful for block-oriented transposition as it is easier to conceptualize smaller sub-matrices within a larger matrix. The use of arrays allows for element access in almost constant time resulting in relatively faster run times (compared to using C++ vector), which is the prime focus of this lab.

2.1 Basic

The basic matrix transpose algorithm sections the matrix into a lower and an upper triangle about the principal diagonal. The principal diagonal is is never transposed as it remains unchanged even when transposed. The algorithm uses two nested for loops, the outer and inner for loops specify an element in the upper triangle and the corresponding element in the lower triangle is determined based on the index of the element is the upper matrix.

2.2 Block

Initially recursion used to perform the block transpose algorithm however this implementation was later changed as recursion is memory intensive and defeated the purpose of the lab. The algorithm used here entails a twofold process. Firstly, contiguous square sub-matrices of certain length from the main matrix are extracted. These sub-matrices are transposed independently, and placed back into the main matrix. Secondly, after all the sub-matrices are transposed, the main matrix is transposed "block" wise as normal matrix transposition. These blocks are of the same length as the sub-matrices mentioned in part 1.

2.3 Diagonal

The diagonal algorithms uses the basic matrix transpose for its functionality. This algorithm also makes uses of two nested for loops, however the outer for loop starts at a specified diagonal. The diagonal is not specified using the array index but can be accessed numerically with the top left diagonal equal to zero. The transposition only takes place at the diagonal where the row and column intersect.

3 PTHREADS

POSIX threads are used to achieve parallelism in shared memory. Pthreads provide the programmer with a lot of freedom and allows one to produce portable multithreded code.

3.1 Diagonal-Threading

Initially all threads will be assigned to work on specified diagonal corresponding to its thread id. Once a thread has finished transposing it increments a global variable *next_pos* which is used to communicate the next available diagonal. To avoid race conditions *pthread_mutex_lock* is used to only allowed one thread to update this variable at a time. The threads terminate once *next_pos* is equal to the size of the matrix. Structs were used to better represent the information the threads needed to work properly. The pseudo-code for this method is shown under section A.3 in Appendix A.

3.2 Block-Oriented

Here each thread works on a certain "block" within the main matrix corresponding to its thread number. Through this, each thread extracts a sub-matrix (which is a 2-D array as well), transposes it using the basic algorithm and then puts it back into the main matrix. Then, similarly to the diagonal method, a global variable *next_pos* is used to communicate the next available block. To avoid race conditions *pthread_mutex_lock* is used to only allowed one thread to update this variable at a time. The threads terminate once *next_pos* reaches the last column of the last row block-wise. In the last part, these blocks are transposed using the basic transpose method, however now 2-D arrays are being swapped. This part does not involve any threads unfortunately due to time constraints. The pseudo-code for this method is shown under section A.7 in Appendix A.

4 OPENMP

Open specifications for Multi Processing (OpenMP) offers a high level shared memory parallelism model compared to pthreads. OpenMP is based on the fork-join execution model that is at the beginning of a program the master thread executes and forks whenever there is a parallel block. OpenMP is often easier to use compared to pthreads since the compiler does most of the low level work.

4.1 Naive

The naive implementation of the matrix transposition algorithm is heavily based on the basic algorithm. To parallelize the nested for-loops two compiler directives are used namely :
#pragma omp parallel shared(arr) private(i, j) num_threads(NUM_THREADS)
#pragma omp for schedule(static,size) nowait

The first directive parallelizes the nested loops, variable *i, j* are made private for each thread to ensure that the threads perform independent transposition. We use schedule because we have a work sharing for-loop. Nowait is used to instruct the thread to continue computing the next transpose.

4.2 Diagonal-Threading

The diagonal version of the version in OpenMP shares great similarities with that naive implementation with the exception that scheduling is dynamic and each thread works on a chunk of size one. The scheduling was changed to dynamic as the static one is less efficient and dynamic scheduling allowed

Table 5.1: Algorithm Benchmark running time

$N_0 = N_1$	**Basic**	**Pthreads**		**OpenMP**		
		Diagonal	Blocked	Naive	Diagonal	Blocked
128	0.156 ms	0.613 ms	1.557 ms	1.873 ms	3.253 ms	3.738 ms
1024	4.471 ms	2.165 ms	8.325 ms	8.554 ms	16.84 ms	11.48 ms
2048	37.70 ms	8.457 ms	27.54 ms	45.38 ms	76.67 ms	30.58 ms
4096	0.233 s	44.62 ms	0.104 s	0.283 s	0.321 s	0.105 s

a thread to move on to the next task once it finished its execution. The pseudo-code for this method is shown under section A.4 in Appendix A.

4.3 Block-Oriented

The implementation of the block-oriented transposition with OpenMP is almost identical to that of the pthread implementation. The only difference is the way threads are created. Another difference is that the second part of the transposition incorporates multi-threading whereas the pthread method did not. The second part is also almost identical to the naive OpenMP implementation, with a slight change to "block" transposition rather than element transposition. The pseudo-code for this method is shown under section A.6 in Appendix A.

5 Results and Analysis

Table 5.1 contains the results for the different algorithms discussed in the previous sections. The code for each algorithm are run on computers with the same specifications. Some of these are: Ubuntu 16.04, Intel Core i7-6700 CPU with 3.41 GHz speed and 4 cores with 8 logical processors.
The blocked algorithm was tested using square blocks of length 64. Furthermore, all the multi-threaded algorithms used a constant number of 8 threads. Besides both the block algorithm implementations (pthreads and OpenMP) which were coded in C++, the rest were coded in C. The code is intended to be as scalable as possible through allowance to change the number of threads, the size of the matrix and the length of the blocks for the blocked algorithm.

After testing, it is noticed that using larger block sizes increases the speed, however only to a certain degree. It is also observed that the size of the main matrix plays the fundamental role in the speeds, which is expected. However, looking at the results in table 5.1, the use of different algorithms and multi-threading as compared to the basic method do not necessarily increase the speed of computations. But it is also noticeable that as the size of the matrix increases, the use of other algorithms and multi-threading do begin to outperform the basic algorithm. The diagonal pthread implementation is seen to be the fastest for the largest matrix by a order of 10 compared to the others.

6 Conclusion

A variation of matrix transpose algorithms were developed and parallelized using Pthreads and OpenMp. Based on the results obtained a threaded algorithm does not necessarily mean higher performance there are various factors that should be considered when one parallelizes an algorithm such as data size. The threaded algorithms performed better when the size of the matrix was large but poorly for small matrix sizes.

REFERENCES

[1] Pthreads
https://computing.llnl.gov/tutorials/pthreads/.

[2] OpenMP
https://computing.llnl.gov/tutorials/openMP/.

[3] OpenMP: For Scheduling
`http://jakascorner.com/blog/2016/06/omp-for-scheduling.html`

A APPENDIX

A.1 General Function Used Across Algorithms

Algorithm 1: Void Swap Function

Function name: *Swap*

Input arguments: int* num1, int* num2
Initialization long int temp = 0
temp = *num
*num1 = *num2
*num2 = temp

A.2 Naive Approach

Algorithm 2: Void Function to Create and Populate Input Matrix

Function name: **CreateMatrix*

Input arguments: matrixSize
Initialization :
int *arr = (int *)calloc(matrixSize * matrixSize, sizeof(int))
val = 1

for *i=0 to size-1* **do**
  for *j= 0 to size-1* **do**
    *(arr + i*size + j) = val++
  end
end

Algorithm 3: Void Transpose Function

Function name: *Transpose*

Input arguments: int* arr, int matrixSize

for *i=0 to size-1* **do**
  for *j= 0 to size-1* **do**
    swap((arr + i*matrixSize +j), (arr + j*matrixSize +i)
  end
end

A.3 Diagonal Approach - PThreads

Global Variables:
int NumberOfThreads = 8
int nextPosition = 1
int *arr

Algorithm 4: int Function to Create and Populate Input Matrix

Function name: **CreateMatrix*

```
Initialization :
int *C = (int *)calloc(matrixSize * matrixSize, sizeof(int))
val = 1

for i=0 to size-1 do
    for j= 0 to size-1 do
        *(C + i*matrixSize + j) = val++
    end
end
return C
```

Algorithm 5: Void Transpose Function

Function name: **Transpose*

```
Initialization :
struct details* local = args
pos= 0, n= 0, nextPosition= 1

while pos < matrixSize do
    pos = local->index
    for i=pos to matrixSize-1 do
        for j = pos+1 to matrixSize-1 do
            swap((arr + pos*matrixSize + j),(arr + j*matrixSize + pos))
        end
        break
    end
    pthread_mutex_lock(&lock)
    n = nextPosition++
    local->index = n
    pthread_mutex_unlock(&lock)
end
```

Algorithm 6: Main Function - PThreads

Function name: *Main Function*

Initialization :
matrixSize=128, 1024, 2048 ,4096
struct details args[NumberOfThreads]
pthread_t threads[matrixSize]

Call to Create :
arr = CreateMatrix(matrixSize)

for *i=0 to NumberOfThreads* **do**
    args[i].index = i
    pthread_create(&threads[i],NULL, transpose,(void*) &args)
end

for *i=0 to NumberOfThreads* **do**
    args[i].index = i
    pthread_join(threads[i],NULL)
end
exit(0)

A.4 Diagonal Approach - OpenMP

Global Variables:
int NumberOfThreads = 8

Algorithm 7: int Function to Create and Populate Input Matrix

Function name: **CreateMatrix*

Input arguments: matrixSize
Initialization :
int *arr = (int *)calloc(size * size, sizeof(int))
val = 1;

#pragma omp for schedule (static, size)
for *i=0 to size-1* **do**
    for *j= 0 to size-1* **do**
        *(arr + i*size + j) = val++
    end
end
return arr

Algorithm 8: Void Transpose Function

Function name: **Transpose*

Input arguments: matrixSize, int*arr
Initialization :pos= 0,

```
while pos < matrixSize do
    pos = local->index
    #pragma omp parallel shared(pos) private( i, j) num-threads(NumberOfThreads
    for i=pos to matrixSize-1 do
        for j = pos+1 to matrixSize-1 do
            swap((arr + pos*matrixSize + j),(arr + j*matrixSize + pos))
        end
        break
    end
    #pragma omp criticalpos++;
end
```

Algorithm 9: Main Function - OpenMP

Function name: *Main Function*

Initialization :
matrixSize=128, 1024, 2048 ,4096

```
for i=0 to 3 do
    n = *(size +i)
    int *A = CreateMatrix(n)
    transpose(A,n)
end
exit(0)
```

A.5 Commonly Used Functions in Block Algorithm

Algorithm 10: Void Function to Create and Populate Input Matrix

Function name: *void createMatrix*

Initialization :long int val=1

```
for i=0 to size_mat-1 do
    for j= 0 to size_mat-1 do
        MAT [i][j] = val
        val++
    end
end
```

Algorithm 11: Void Function to get SubMatrix

Function name: *void getSubMatrix*

Input arguments: int A, int B
long int (&arr)[block_len][block_len])

```
for i=0 to block-length do
    for j= 0 to block_len do
        arr[i][j] = MAT[A+i][B+j];
    end
end
```

Algorithm 12: Void Function to set SubMatrix

Function name: *void setSubMatrix*

Input arguments: int A, int B
const long int (block)[block_len][block_len])

```
for i=0 to block_len do
    for j= 0 to block_len do
        MAT[A+i][B+j]= block[i][j]
    end
end
```

Algorithm 13: Void Function to perform normal 2x2 Transformations

Function name: *void transpose*

Input arguments: long int (arr)[block_len][block_len]

```
for i=0 to size_mat-1 do
    for j= 0 to size_mat-1 do
        swap(arr[i][j],arr[j][i])
    end
end
```

A.6 Block Approach - OpenMP

Global Variables:

const int block_len = 2
const int size_mat = 4096
long int MAT[size_mat][size_mat]
const int NUM_THREADS=8
int next-pos = NUM_THREADS

Algorithm 14: Void Function to Transpose actual bigger blocks

Function name: *void shuffle*

```
#pragma omp parallel num_threads(NUM_THREADS)
for i=0 to size_mat-1 do
    #pragma omp for schedule(static,size_mat) nowait
    for j= 0 to size_mat-1 do
        getSubMatrix(i,j,block)
        getSubMatrix(j,i,block2)
        setSubMatrix(i,j,block2)
        setSubMatrix(j,i,block)
        j+=block_length
    end
    i+=block_length
end
```

Algorithm 15: Void Transpose Function: Transposes all inner blocks of matrix

Function name: *void transpose_blocks*

Initialization :int thread_counter = 0

```
#pragma omp parallel shared (thread_counter) num_threads(NUM_THREADS)
Initialization :long local = thread_counter
thread_-counter++, int pos = 0
int posY = 0 , int tmp = size_mat/block_len
int x = 0 , long int block[block_len][block_len]
while 1 do
    pos = (block_len*local)
    x = local/tmp
    posY = x*block_len
    if posY >= size-mat then
        break
    else
        getSubMatrix(posY,pos,block)
        transpose(block)
        setSubMatrix(posY,pos,block)
        pragma omp critical
        local= next_pos
        next_pos++
    end
end
```

Algorithm 16: Main Function - OpenMP

Function name: *Main Function*

CreateMatrix()
transpose_blocks()
shuffle()
return (0)

A.7 Block Approach - PThreads

Global Variables:
const int block_len = 2
const int size_mat = 4096
long int MAT[size_mat][size_mat]
const int NUM_THREADS = 8
int next_pos=NUM_THREADS
pthread_mutex_t lock

Algorithm 17: Void Function to Transpose actual bigger blocks

Function name: *void shuffle*

Initialization :long int block[block_len][block_len]
long int block2[block_len][block_len];

```
for i=0 to size_mat-1 do
    for j= 0 to size_mat-1 do
        getSubMatrix(i,j,block)
        getSubMatrix(j,i,block2)
        setSubMatrix(i,j,block2)
        setSubMatrix(j,i,block)
        j+=block_length
    end
    i+=block_length
end
```

Algorithm 18: Void Transpose Function: Transposes all inner blocks of matrix

Function name: **void transpose_blocks(void* _ind)*

```
Initialization :long local
local = (long) _ind
int pos
int posY = 0 , int tmp = size_mat/block_len
int x = 0 , long int block[block_len] [block_len]

while 1 do
    pos = (block_len*local)
    x = local/tmp
    posY = x*block_len
    if posY >= size_mat then
        break
    else
        block = getSubMatrix(posY,pos,block)
        transpose(block)
        setSubMatrix(posY,pos,block)
        pthread_mutex_lock(&lock)
        local= next_pos
        next_pos++
        pthread_mutex_unlock(lock)
    end
end
```

Algorithm 19: Main Function - PThreads

Function name: *Main Function*

```
Call to Create :
CreateMatrix()
Initialization :int index[NUM_THREADS]
pthread_t threads[NUM_THREADS]

for i=0 to NUM_THREADS-1 do
    index[i]=i
    pthread_create(threads[i],NULL,transpose_blocks,(void*)index[i])
end
for i=0 to NUM_THREADS-1 do
    pthread_join(threads[i],NULL)
end
Shuffle()
return (0)
```