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 18, 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 C++ vectors to store the 2-D matrix. The use of vectors allows for a more efficient memory usage and most importantly to extract sub-matrices from the main matrix with ease. This is especially useful for block-oriented transposition.

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

The algorithm makes use of 8 Threads which were determined to be the most efficient when compared to other variations. Initially all 8 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.

3.2 Block-Oriented

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) $\text{num}_t hreads(NUM\_THREADS)$
$\#pragmaompforschedule(static, size)nowait$

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 Diagnoal-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 a thread to move on to the next task once it finished its execution.

4.3 Block-Oriented

In this lab, tensors are restricted to n dimensional square tensors which simplifies some aspects of the computation. The An empty 2×2 matrix of size equal to the one of the matrices being added is created. The resultant values are stored in the empty matrix. Equation describes tensor multiplication of 2×2 matrices and how each element is generated in the resultant matrix.

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	8.889 ms		3.253 ms	9.615 ms
1024	4.471 ms	2.165 ms	0.345 s		16.84 ms	0.350 s
2048	37.70 ms	8.457 ms	1.387 s		76.67 ms	1.398 s
4096	0.233 s	44.62ms	5.415 s		0.321 s	5.465 s

5 RESULTS

6 CONCLUSION

One can further verify the results using numpy for the rank2Tensor procedures. From the algorithm analysis it is evident that computationally complexity increases with the rank and dimensions of the tensor. The procedures developed in this lab are presented for $n \times n$ or $n \times n \times n$ however modifications may be made to make them more flexible to work in varying dimensions.

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
  pthreadmMutexLock(&lock)
  n = nextPosition++
  local->index = n
  pthreadMutexUnlock(&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, vector<long int> num

for *i=0 to matrixSize-1* **do**
    for *j= 0 to matrixSize-1* **do**
        num.pushBack(val)
        val++
    end
    MATRIX.pushBack(num); num.clear();
end

Algorithm 11: Void Function to get SubMatrix

Function name: *tensor getSubMatrix*
Input arguments: int A, int B
Initialization :
tensor sub(block-len,vector<long int>(block-len))

for *i=0 to block-length* **do**
    for *j= 0 to block-length* **do**
        sub[i][j] = MATRIX[A+i][B+j]
    end
end

Algorithm 12: Void Function to set SubMatrix

Function name: *tensor setSubMatrix*
Input arguments: int A, int B, tensor Block

for *i=0 to block-length* **do**
    for *j= 0 to block-length* **do**
        MATRIX[A+i][B+j]=block[i][j]
    end
end

Algorithm 13: Void Function to perform normal 2D x 2D Transformations

Function name: *void transpose*
Input arguments: int matrixSize
Initialization : size = arr.size()

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

A.6 Block Approach - OpenMP

Global Variables:

using tensor = vector <vector<long int»
int block-len=2
long int size-mat = 8
tensor MATRIX
const int NumberOfThreads=8
int nextPosition=NumberOfThreads

Algorithm 14: Void Function to Transpose actual bigger blocks

Function name: *void shuffle*

```
pragma omp parallel num-threads(NUM-THREADS)
for i=0 to matrixSize-1 do
    i+=block-length
    pragma omp for schedule(static,size-mat) nowait for j= 0 to matrixSize-1 do
        j+=block-length
        tensor block = getSubMatrix(i,j)
        tensor block2 = getSubMatrix(j,i)
        setSubMatrix(i,j,block2)
        setSubMatrix(j,i,block)
    end
end
```

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

Function name: **void transpose-blocks*

```
Initialization :int thread-counter = 0
long local = thread-counter
thread-counter++, int pos = 0
int posY = 0 , int tmp = size-mat/block-len
int x = 0 , tensor block

pragma omp parallel shared (thread-counter) num-threads(NumberOfThreads)
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)
        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*

Call to Create :

```
CreateMatrix()
transpose-blocks()
shuffle()
return (0)
```

A.7 Block Approach - PThreads

Global Variables:
int block-len=2
long int size-mat =8
tensor MATRIX
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*

```
for i=0 to matrixSize-1 do
    i+=block-length
    for j= 0 to matrixSize-1 do
        j+=block-length
        tensor block = getSubMatrix(i,j)
        tensor block2 = getSubMatrix(j,i)
        setSubMatrix(i,j,block2)
        setSubMatrix(j,i,block)
    end
end
```

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

Function name: **void transpose-blocks*
Initialization :int thread-counter = 0
long local = thread-counter
thread-counter++, int pos = 0
int posY = 0 , int tmp = size-mat/block-len
int x = 0 , tensor block

```
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)
        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
return (0)