ELEN4020 Data Intesive Computing for Data Science Lab 2 Report

Leantha Naicker (788753), Fiona Rose Oloo (790305),
Boitumelo Mantji (823869), Justine Wright (869211)

The University of Witwatersrand

March 13, 2018

Abstract

This report contains a short description of the methods used to solve the problems presented in this lab to transpose a matrix using parallel programming methods. The lab explores two ways of doing this, through OpenMP and P-Threading. The pseudo code to each method can be found within the report.

1 Introduction

The objective of this lab was to write a C language based program, which can transpose a matrix using parallel programming tools and methods.

2 Matrix Transpose Using OpenMP

Open Multi-Processing or OpenMP is an API that supports shared memory multiprocessing. The constructs for thread creation, workload distribution and thread synchronization is the core purpose of OpenMP.
The pseudo code below illustrates how OpenMP is used to parallelize a transpose matrix program. In this program, OpenMP is used to split loop iterations into threads. The variables a,b and temp are declared private within the parallel running of the program to prevent queuing delays during execution. By adopting this method we have created these variables for each thread to work on simultaneously.

2.1 Pseudocode- OpenMP

Algorithm for swap function: Interchange matrix elements for matrix transposition

swap(elem1,elem2,temp)
in: pointer to a matrix element
temp ← elem1
elem1 ← elem2
elem2 ← temp

Algorithm for transposition using threads: Create matrix, assign values, transpose with threads

Initialize Total threads to zero
Initialize temp to zero
Initialize a to zero
Initialize b to zero
in:length of N x N matrix
**testArray ← points to memory for 1 dimension of array
for (m ← to length-1)
testArray[m] ← points to memory for second dimension
end for
call populateArray(testArray,length)
in: Thread number

```
initialize clock
for (a ← 0 to length-1)
Total threads ← Thread number
    for (b ← to 0 length-1)
        call swap(testArray[a][b],testArray[b][a],temp)
    end for
end for
end clock
timeSpent← Final time-Initial time
```

3 Matrix Transpose Using P-Threads

POSIX Threads or PThreads is an API which allows the simultaneous control of multiple streams of work in a program, where each stream is referred to as a thread.
The pseudo code below illustrates how PThreads are initiated to parallelize and transpose a matrix. The program waits for each thread to reach completion then it determines the run time for each combination of threads. Pthreads differ from OpenMp in that the threads have to be manually created by the user where as in OpenMp a parallel region is created.

3.1 Pseudocode-Pthreads

Algorithm for runPThreads: This algorithm initializes the pthreads to transpose the matrix and waits for them to finish before returning the start time.

```
    runPThreads(Array* a, int n, arg_struct * ptr)
```
in: ptr to Array struct, integer number of threads and a pointer to arg_struct
out: double start time

```
    ThreadID ← PThread Array [size of MAX_THREADS]
    start ← 0.0

    for (int i ← 0 to n)
        ptr → numThreads ← n
        ptr → a ← a
        ptr → ID ← i

        if statement (i equivalent to 0)
        start ← call omp_get_wtime()
        end if

    call pthread_create(thread_id[i], NULL, tranpose, (void*) ptr)
    increment ptr
    end for

    for (int i←0 to n)
    call pthread_join(thread_id[i], NULL)
    end for

    return start
```

4 Performance

Table 1: Performance of OpenMP

Number of threads:	$N_0 = N_1 = 128$ Time elapsed (ms)	$N_0 = N_1 = 1024$ Time elasped (ms)	$N_0 = N_1 = 8192$ Time elapsed(ms)
4	1.55	5.50	650.41
8	0.41	6.52	804.77
16	0.35	7.69	816.14
64	1.53	6.25	780.26
128	1.53	8.49	776.13

Table 2: Performance of PThreads

Number of threads:	$N_0 = N_1 = 128$ Time elapsed (ms)	$N_0 = N_1 = 1024$ Time elapsed (ms)	$N_0 = N_1 = 8192$ Time elapsed(ms)
4	0.33	9,09	749.32
8	0.94	5.57	790.15
16	0.39	6.06	784.99
64	1.44	10.82	754.52
128	3.00	9.40	765.09

The OpenMP and Pthreads performance results are inconsistent and do not yield intuitive results. This is due to the fact that the code does not account for false sharing. False sharing occurs when threads impact the performance of one another in the process of modification of independent data elements which share the same cache line[1]. Increased number of threads increases the occurrence of false sharing and thus increases the processing time. Linear scalability is diminished in this case. Threads from different cores attempt to access data from the same cache line. Each thread tries to be be the first to obtain ownership of the cache line so they can perform their respective modifications. If a thread belonging to one core gains ownership of the cache it will invalidate the cache line for any other core causing a delay as the other cores have to queue to get access to the cache line. In order for the other cores have to gain ownership of the cache line, the original core has to then be invalidated.

A solution to this problem is to add padding which ensures that threads have access to different cache lines. A drawback is that the size of cache line is necessary which needs constant readjustment if there are changes as to what machine the code is executed on. Another set of solutions fall under synchronization methods such as barrier and mutual exclusion. Synchronization controls race conditions to protect conflicts of data.

5 Conclusion

Two different algorithms concerned with the application of threads to the transposition of matrices have been presented. A table of the performances of OpenMP and Pthreads has also been presented and the result as to which performed better is inconclusive, but suggestions are highlighted which give possible reasons for the outcomes obtained.

References

[1] Martin Thompson. *False Sharing.* URL: `https://dzone.com/articles/false-sharing`. (accessed: 13.03.2018).