University of the Witwatersrand
School of Electrical and Information Engineering

ELEN4020: Data Intensive Computing

Laboratory Exercise 2

Authors:

Kayla-Jade Butkow
714227

Jared Ping
704447

Lara Timm
704157

Matthew van Rooyen
706692

Date Handed In: 9th March, 2018

1. Matrix Transposition

In order to allow for simple matrix transposition, the two-dimensional square matrix is created as a one-dimensional array, populated in row-major form. The one-dimensional array is filled with integer values, where the value is equal to the $index + 1$.

To improve efficiency, only the lower left triangle of the matrix is traversed. This is sufficient as in matrix transposition the middle diagonal remains unchanged while each entry in the lower triangle is swapped with a single entry in the upper triangle. For each position in the lower triangle, the index of the entry with which that integer should be swapped is calculated using Equation 1 [1].

$$index_{old} = (index_{new} * N) \; mod(N^2 - 1) \tag{1}$$

To ensure that the matrix is transposed in place, a traditional style swap function, making use of integer pointers, is implemented.

2. OpenMP

The multi-threaded algorithm was implemented using OpenMP. Multi-threading was performed on all aspects of the process, including the populating of the matrix and the transposition of the array.

For all of the processes using OpenMP, the loops were divided into chunks of the size of the matrix divided by 256. Each function takes in a parameter called `noOfThreads`, which allows the user to specify the number of threads that should be used to execute the process.

Since the matrix is large, populating it in serial takes a large amount of time. It was thus implemented in parallel. Since each loop takes a consistent amount of time (as the same process is performed in each loop), static scheduling was used [2].

The transposition was also performed using static scheduling, as it proved to give the best performance.

3. PThreads

4. Comparison of Performance

Table 1: Performance of the algorithm when run as a serial process

$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$

Table 2: Performance of the algorithm using 4 threads

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread			
OpenMP			

Table 3: Performance of the algorithm using 8 threads

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread			
OpenMP			

Table 4: Performance of the algorithm using 16 threads

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread			
OpenMP			

Table 5: Performance of the algorithm using 64 threads

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread			
OpenMP			

Table 6: Performance of the algorithm using 128 threads

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread			
OpenMP			

REFERENCES

[1] "Inplace (Fixed space) M x N size matrix transpose." URL `https://www.geeksforgeeks.org/inplace-m-x-n-size-matrix-transpose/`.

[2] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011.

Appendix