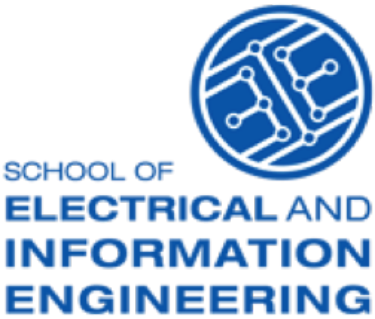

University of the Witwatersrand
School of Electrical and Information Engineering

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 ensure 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.

In order to parallelize the code, each time a parallel region was required, a new parallel section was declared using `#pragma omp parallel num_threads(noOfThreads)`, where `noOfThreads` is a variable that is used to set the number of threads that should be used for the process. Thereafter, when parallelizing a `for` loop, the parallel `for` construct is used [2]. This construct makes use of the fork-join parallel design pattern, whereby in the parallel regions, threads are created to perform work concurrently [2]. Once the work has been completed, the threads join together to form a single result [2].

For all of the processes using OpenMP, the loops were divided into chunks of the size of the matrix divided by 256. These chunks indicate the number of sequential loops that each thread will execute.

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 [3]. In static scheduling, the loop is divided into contiguous chunks of equal size [3]. The transposition was also performed using static scheduling, as it proved to give the best performance.

3. PThreads

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

The PThreads library offers a much lower level approach in implementing parallelism, offering high performance at the cost of substantial source code modification and code maintainability. This is noted when comparing the source code length of the OpenMP implementation to that of the PThreads implementation.

In comparison to the OpenMP implementation, it can be seen that required work per thread is specified when each thread is created as opposed to continuously feeding chunks of data to each thread [4]. This translates to a significant performance increase as the dataset scales. 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.

The general process of PThread implementation involves correctly dividing up the required work based on the number of specified threads, thereafter initialising each thread and assigning it the appropriate work. An important component is to always ensure the threads synchronise after completing the specified workload.

Thread functions were implemented, which are non-returning pointer functions that are provided as arguments for a thread's workload. These functions only accept a single argument and thus structs were created to pass all relevant data which the thread required to complete the assigned workload.

4. Comparison of Performance

From the tables below, it is evident that, in general, the best performance is obtained using PThreads. However, for the small matrix ($N_0 = N_1 = 128$), OpenMP provides a better performance than PThreads.

It can also be seen that OpenMP does not, in general, provide a better performance than a serially executed code. This can be explained by the large overhead that is present whenever a parallel region is started or stopped [3]. This overhead manifests since threads must be created at the start of each region [3]. These threads must then be assigned tasks, and at the end of the for loop, the threads must wait at the barrier to be synchronised [3].

Figure 1 provides a graphical description of the data in Tables 1 to 6. In the figure, the times for each matrix size were averaged, and then plotted against the matrix size.

Table 1: Performance of the algorithm when run as a serial process (measured in seconds)

$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
0.000270	0.010812	1.609882

Table 2: Performance of the algorithm using 4 threads (measured in seconds)

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread	0.000175	0.004886	0.766117
OpenMP	0.000138	0.011894	1.649183

Table 3: Performance of the algorithm using 8 threads (measured in seconds)

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread	0.000184	0.003155	0.532490
OpenMP	0.000207	0.010809	1.665707

Table 4: Performance of the algorithm using 16 threads (measured in seconds)

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread	0.000418	0.002654	0.476413
OpenMP	0.000168	0.012843	1.623063

REFERENCES

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

[2] T. Sterling et al. *High Performance Computing. Modern Systems and Practices*. Morgan Kauffman Publishers, 2018.

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

[4] M. Kerrisk. "PThreads - POSIX threads." URL `http://man7.org/linux/man-pages/man7/pthreads.7.html`.

Table 5: Performance of the algorithm using 64 threads (measured in seconds)

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread	0.001417	0.002739	0.482428
OpenMP	0.000573	0.012632	1.655354

Table 6: Performance of the algorithm using 128 threads (measured in seconds)

	$N_0 = N_1 = 128$	$N_0 = N_1 = 1024$	$N_0 = N_1 = 8192$
PThread	0.002326	0.003241	0.448408
OpenMP	0.001360	0.011724	1.620541

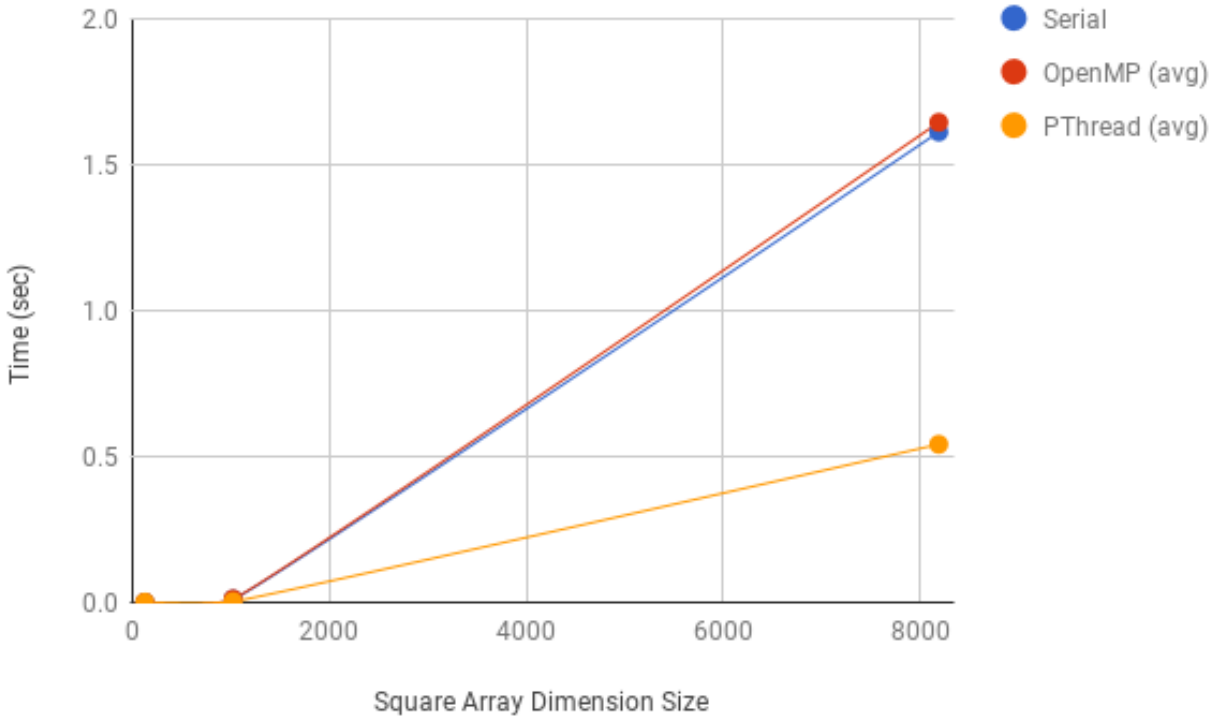

Figure 1: Performance benchmark of transposition algorithm.

Appendix

A Pseudo-code

Algorithm 1 Serial

```
function generateSquareMatrix(dimensions)
    size ← dimension * dimension;
    created_squareMatrix ← malloc((size) * sizeof(int))
    if created_squareMatrix is equal to null then
        catch error
    end if
    for i ← 0 to _size do
        created_squareMatrix[i] ← i + 1
    end for
    return created_squareMatrix
end function

function swap(i, j)
    temp ← i
    i ← j
    j ← temp
end function

function transpose(squareMatrix, dimension)
    size ← dimensions * dimensions
    for index ← 1 to dimensions do
        for j ← 0 to index do
            currentIndex ← index * dimension + j
            newPosition ← (currentIndex * dimension) % (size - 1)
        end for
    end for
    swap(squareMatrix[currentIndex], squareMatrix[newPosition])
end function

function printMatrix(squareMatrix, dimension)
    count ← log10(squareMatrix[dimension * dimension - 1]) + 2
    for i ← 0 to dimension * dimension do
        if i % dimension is equal to 0 then
            print newline
        end if
    end for
end function

function callFunctions(dimension)
    squareMatrix ← generateSquareMatrix(dimension)
    transpose(squareMatrix, dimension)
end function

function main
    dimension[3] ← [128, 1024, 8192]
    for i ← 0 to 3 do
        callFunctions(dimension[i])
    end for
    return 0
end function
```

Algorithm 2 OpenMP

```
function generateSquareMatrix(dimensions, noOfThreads)
    size ← dimension * dimension;
    created_squareMatrix ← malloc((size) * sizeof(int))
    if created_squareMatrix is equal to null then
        catch error
    end if
    #pragma omp parallel

    for i ← 0 to _size do
        created_squareMatrix[i] ← i + 1
    end for
end function
return created_squareMatrix

function swap(i, j)
    temp ← i
    i ← j
    j ← temp
end function

function transpose(squareMatrix, dimension, noOfThreads)
    size ← dimensions * dimensions
    #pragma omp parallel
    for index ← 1 to dimensions do
        for j ← 0 to index do
            currentIndex ← index * dimension + j
            newPosition ← (currentIndex * dimension) % (size - 1)
        end for
    end for
    swap(squareMatrix[currentIndex], squareMatrix[newPosition])
end function

function printMatrix(squareMatrix, dimension)
    count ← log10(squareMatrix[dimension * dimension - 1]) + 2
    for i ← 0 to dimension * dimension do
        if i % dimension is equal to 0 then
            print newline
        end if
    end for
end function

function callFunctions(dimension, noOfThreads)
    squareMatrix ← generateSquareMatrix(dimension, noOfThreads)
    transpose(squareMatrix, dimension, noOfThreads)
end function

function main
    dimension[3] ← [128, 1024, 8192]
    threads[5] ← [4,8,16,64,128]
    for i ← 0 to 3 do
        for j ← 0 to 5 do
            callFunctions(dimension[i], threads[j])
        end for
    end for
    return 0
end function
```

Algorithm 3 PThread

```
function generateValuesForMatrix(ThreadData)
    start ← data.start
    stop ← data.stop
    array ← data.array;
    for i ← start to stop do
        array[i] ← i + 1
    end for
    return null
end function

function generateSquareMatrix(dimensions, noOfThreads)
    size ← dimension * dimension
    created_squareMatrix ← malloc((size) * sizeof(int))
    tasksPerThread ← (size + noOfThreads - 1) / noOfThreads
    if created_squareMatrix is equal to null then
        catch error
    end if
    for i ← 0 to noOfThreads do
        data[i].start ← i * tasksPerThread
        data[i].stop ← (i+1) * tasksPerThread
        data[i].array ← created_squareMatrix
    end for
    data[noOfThreads - 1].stop ← size
    for i ← 0 to noOfThreads do
        pthread_create(thread[i], null, generateValuesForMatrix, data[i])
    end for
    for i ← 0 to noOfThreads do
        pthread_join(thread[i], null)
    end for
end function
return created_squareMatrix

function swap(i, j)
    temp ← i
    i ← j
    j ← temp
end function

function threadTranspose(ThreadTransposeData)
    start ← data.start
    stop ← data.stop
    dimension ← data.dimension
    size ← data.size
    array ← data.array
    for index ← start to stop do
        for j ← 0 to index do
            currentIndex ← index * dimension + j
            newPosition ← (currentIndex * dimension) % (size - 1)
            swap(array[currentIndex], array[newPosition])
        end for
    end for
    return null
end function
```

Algorithm 4 PThread Continued

```
function TRANSPOSE(squareMatrix, dimension, noOfThreads)
    size ← dimensions × dimensions
    pthread_t thread[noOfThreads]
    tasksPerThread ← (dimension + noOfThreads - 1) / noOfThreads
    for i ← 0 to noOfThreads do
        data[i].start ← i * tasksPerThread
        data[i].stop ← (i+1) * tasksPerThread
        data[i].dimension ← dimension
        data[i].size ← size
        data[i].array ← squareMatrix
    end for
    data[noOfThreads - 1].stop ← dimension
    for i ← 0 to noOfThreads do
        pthread_create(thread[i], null, threadTranspose, data[i])
    end for
    for i ← 0 to noOfThreads do
        pthread_join(thread[i], null)
    end for
    return squareMatrix
end function

function PRINTMATRIX(squareMatrix, dimension)
    count ← log10(squareMatrix[dimension × dimension - 1]) + 2
    for i ← 0 to dimension × dimension do
        if i % dimension is equal to 0 then
            print newline
        end if
    end for
end function

function CALLFUNCTIONS(dimension, noOfThreads)
    squareMatrix ← generateSquareMatrix(dimension, noOfThreads)
    transpose(squareMatrix, dimension, noOfThreads)
end function

function MAIN
    dimension[3] ← [128, 1024, 8192]
    threads[5] ← [4,8,16,64,128]
    for i ← 0 to 3 do
        for j ← 0 to 5 do
            callFunctions(dimension[i], threads[j])
        end for
    end for
    return 0
end function
```