Data Intensive Computing - ELEN 4020
Lab2

William Becerra Gonzalez 789146
Sailen Nair 1078491
Kyle Govender 571133

March 16, 2018

Introduction

This report documents the procedure taken, as well as the reasons behind the proposed solution within the laboratory. The purpose of this laboratory is to become familiar with the Pthread and OpenMP libraries through the design and implementation of a multi-threaded algorithm which computes the transpose of an MxM matrix. The main problem posed by this laboratory was adhering to the constraint of transposing the matrix 'in-place', and to design the algorithm so that it is concurrently friendly, in order to run in parallel.

1 Proposed Solution

There are two main approaches when computing the transpose of a matrix. The first requires copying the rows of the original matrix into the columns of a second matrix, of the same dimensions. This algorithm would result in a space complexity of $O(2n)$ and a linear time complexity of $O(n)$, where n represents the number of elements in the MxM matrix. The second approach eliminates the need for double the memory space capacity. Performing the transpose 'in place' requires traversal of the matrix element wise, a temporary variable is created and the appropriate elements are swapped. The second method has a time complexity $O(n^2)$ and a space complexity of $O(n)$ where n represents the number of elements of the MxM matrix. Having a quadratic time complexity can make the algorithm computationally slow if the matrix is very large. However, by making use of parallel programming the computational performance of the algorithm can be improved by a factor proportional to the number of threads running the task. The pseudo code for the proposed serial solution is found in section section 3

It is important to note that matrix transposition is inherently difficult to optimise due to the constant access of non-contiguous memory blocks making the transposition speed dependent on the memory copy bandwidth. Non-contiguous memory blocks are placed in different cache lines .This is further explained by Andrey Vladimirov's paper [2].

1.1 OpenMP

OpenMP is a high-level multiprocessing programming interface for use with C/C++ and Fortran. It allows programs to be readily modified to run in parallel. The OpenMP framework is able to do this by creating threads within the computers architecture. The threads are then run in parallel, which allows for a programs runtime to be significantly reduced. One of the main advantages of OpenMP is its ease of use. By using a simple single line command, a section of code can be run in parallel.

For the purpose of this lab, two methods of OpenMP was executed. Both of which use the 'in-place' matrix transposition algorithm. The pseudo code for the two methods used in conjunction with OpenMP can be found in section section 3. The first method is to just multithread the nested for loop of the main serial code, thus letting the OpenMP framework choose the best method to multithread the program. When using OpenMP, it is important to declare variables with the appropriate scope within the parallel section of the code, this ensures that threads do not

all queue to access and edit the same variable. By ensuring correct privatisation of variables, the programs execution time can be significantly reduced.

The execution time of the OpenMP programs can be seen in section section 2.

1.2 Pthread

Pthreads also known as POSIX (Portable Operating System Interface) threads, is an architecture that is independent of any particular language. It allows for programs to be multithreaded. Pthreads allows a programmer to create threads in a computer's architecture and then assign specific code for which the corresponding thread is to execute. This is similar to OpenMP, however the allocation of threads and the code that each thread executes needs to be defined and coded by the programmer, without the assistance of an API. It is a lower level of multithreading as compared to OpenMP, and is much more involved.

When a thread is created using Pthreads, it needs to be assigned a certain set of code to execute, this code will then be executed independently. As is the same with OpenMP, if used correctly, Pthreads can significantly reduce execution time of a program. For this lab, Pthreads was used to decrease the time taken to perform 'in-place' matrix transposition. The original serial algorithm was used, however it was multithreaded using Pthreads, this reduced the execution time as seen in the results section of this report. To create a Pthread the function "$pthread\_create()$" needs to be called. This function accepts the name of the thread being created, the function that will be executed by the thread, and any additional arguments needed by the function itself.

2 Results

3 Pseudo code

D	#	#	#	#	#	#	#
	D	#	#	#	#	#	#
		D	#	#	#	#	#
			D	#	#	#	#
				D	#	#	#
					D	#	#
						D	#
							D

Matrix 1: Matrix showing upper diagonal

The following subsections, outlining the pseudo code, will make reference to the "upper triangle" of a matrix. The figure above shows what is meant by that through cells filled in with a '#' making up the upper triangle and the cells filled in with a 'D' making up the main diagonal.

3.1 Main Serial Algorithm

```
create 2D matrix[matrixSize][matrixSize]

for i = 0 to matrixSize -1
    for j = i + 1 to matrixSize
      temp = matrix[i][j]
        matrix[i][j] = matrix[j][i]
        matrix[j][i] = temp
```

The above algorithm shows the main algorithm that is used throughout all the implementations. It traverses the entire upper triangle and swaps it with its opposite cell along the main diagonal. This algorithm is only valid for a square matrix.

3.2 OpenMP Single Loop Algorithm

```
create 2D matrix[matrixSize][matrixSize]

#pragma omp parallel private (temp, i, j)
    #pragma omp for schedule(dynamic) nowait
        for i = 0 to matrixSize -1
            for j = i + 1 to matrixSize
                temp = matrix[i][j]
                matrix[i][j] = matrix[j][i]
                matrix[j][i] = temp
```

The above pseudo code shows the implementation of the OpenMP single loop algorithm, this splits up the main for loop into parallel threads. As mentioned above, ensuring certain variables are private is very important. In the above code, the variable 'temp' was made private, this ensures that each thread's stack will have its own copy of 'temp', thus no queueing to access this variable will take place.

Another key aspect to take note of in the above code is the 'nowait' clause. This clause tells the compiler that if a thread is done with its execution, it does not need to wait for the completion of any of the other threads. For certain application this is not desirable and will not work efficiently, however for this particular application of transposing a matrix, the threads do not need to be synchronized.

3.3 OpenMP Blocked Algorithm

```
create 2D matrix[matrixSize][matrixSize]

#pragma omp parallel private (temp, i, j)
    #pragma omp for schedule(dynamic) nowait
        for i = 0 to matrixSize/2 -1
            for j = i + 1 to matrixSize/2
                temp = matrix[i][j]
                matrix[i][j] = matrix[j][i]
                matrix[j][i] = temp

#pragma omp parallel private (temp, i, j)
    #pragma omp for schedule(dynamic) nowait
        for i = matrixSize/2 to matrixSize -1
            for j = i + 1 to matrixSize
                temp = matrix[i][j]
                matrix[i][j] = matrix[j][i]
                matrix[j][i] = temp

#pragma omp parallel private (temp, i, j)
    #pragma omp for schedule(dynamic) nowait
        for i = matrixSize/2 to matrixSize
            for j = 0 to matrixSize/2
                temp = matrix[i][j]
                matrix[i][j] = matrix[j][i]
                matrix[j][i] = temp
```

The algorithm above is another algorithm for OpenMP for transposing a matrix. It is very similar to the first OpenMP algorithm, however it breaks up the upper diagonal into three pieces to be multi-threaded. As is seen in the results section, this implementation of OpenMP executes marginally faster than the single loop OpenMP implementation.

3.4 Pthreads' algorithm

```
create struct ThreadData{
i
matrixSize
matrix
StartofArraySegment
EndOfArraySegment
}


create matrix[matrixSize][matrixSize]
//Declaring the amount of threads that is going to be created
    pthread_t threads[numberOfThreads]

    for i = 0 to numberOfThreads
     ThreadData->i = i
        ThreadData->matrix = matrix
        ThreadDate->matrixSize = matrixSize
        ThreadData->StartofArraySegment = (matrixSize/numberOfThreads)*i
        ThreadData->EndOfArraySegment = (matrixSize/numberOfThreads)*(i+1)

        //create pthread using the function call pthread_create()
        pthread_create(&thread[i],NULL,Swap,(void *)ThreadData


//Swap function that each thread will execute
void swap(ThreadData)
for i = ThreadData->StartofArraySegment to ThreadData->EndOfSegment
     for j = i + 1 to matrixSize
         temp = matrix[i][j]
            matrix[i][j] = matrix[j][i]
            matrix[j][i] = temp
```

The algorithm above shows the Pthreads implementation to transpose a matrix. As can be seen above, threads are explicitly created by use of the "*pthread_create*()" function which takes in arguments: the thread name, the function it will execute as well as a pointer to the arguments of the function that it is going to execute. In the above algorithm, each thread transposes a small section of the matrix. The section size is determined by the number of threads created in the algorithm. Each thread will then transpose its allocated number of rows. This algorithm however is not the most efficient, as the first few threads will have much more cells to transpose as compared to the last few threads. This is due to the algorithm only traversing the upper diagonal.

3.5 Results

The tables of results are shown in the following three pages. The results document the running of the different algorithms through different sizes of matrices, whilst the thread count was varied.

The various implementations were executed on a core i5-4210U CPU with 2 cores and 4 threads, with a base frequency of 1.7 GHz. The environment used was Ubuntu 16.04 LTS.

As can be seen in the following tables, as expected the serial code runs the slowest. This is due to it being executed on a single thread. The 'OpenMP Blocked Algorithm' is seen to be the fastest implementation, however is very close in comparison to the normal OpenMP implementation. The Pthread implementation performs similar to the two OpenMP implementations however, is consistently slower. This shows that OpenMP did a better job in multithreading the programs,

as compared to explicitly creating threads using Pthreads. Thus letting OpenMP manage and create threads are shown to be faster in this case.

The tests conducted on the algorithms required varying the number of threads. As can be seen from the results, all times for the parallel implementations did not change greatly when the thread count increases. This is due to the computer that executed the programs. Since the CPU is made up of 4 threads, the largest amount of threads that can run in parallel is 4. More threads can be created however they will run concurrently. Thus, the creation of threads that exceeds the amount the computer's architecture supports, will not appreciably speed up the execution time. It can be seen from the results that running more threads than the computer has does not better the performance. Rather it slightly decreases the performance, due to the overhead of creating all the extra threads.

One of the key points to take note of from the results is that all parallel implementations of the code executes faster than a serial implementation, apart from the 128x128 sized matrix as this . This illustrates the benefit of parallel computing.

3.6 Matrix Size 128:

Table 1: Table showing results for matrix of size 128*128 using 4 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	128	1	0.000143
Pthread	128	4	0.000703
OpenMP Single loop	128	4	0.000424
OpenMP Blocked Algorithm	128	4	0.000514

Table 2: Table showing results for matrix of size 128*128 using 8 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	128	1	0.000143
Pthread	128	8	0.000304
OpenMP Single loop	128	8	0.000675
OpenMP Blocked Algorithm	128	8	0.000868

Table 3: Table showing results for matrix of size 128*128 using 16 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	128	1	0.000143
Pthread	128	16	0.001516
OpenMP Single loop	128	16	0.00107
OpenMP Blocked Algorithm	128	16	0.001381

Table 4: Table showing results for matrix of size 128*128 using 64 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	128	1	0.000143
Pthread	128	64	0.004435
OpenMP Single loop	128	64	0.003175
OpenMP Blocked Algorithm	128	64	0.004294

Table 5: Table showing results for matrix of size 128*128 using 128 threads

	Matrix Size	No. of Threads	Time (s)
Serial	128	1	0.000143
Pthread	128	128	0.00609
OpenMP Single loop	128	128	0.006012
OpenMP Blocked Algorithm	128	128	0.008153

3.7 Matrix Size 1024:

Table 6: Table showing results for matrix of size 1024*1024 using 4 threads

	Matrix Size	No. of Threads	Time (s)
Serial	1024	1	0.00587
Pthread	1024	4	0.003396
OpenMP Single loop	1024	4	0.005076
OpenMP Blocked Algorithm	1024	4	0.00552

Table 7: Table showing results for matrix of size 1024*1024 using 8 threads

	Matrix Size	No. of Threads	Time (s)
Serial	1024	1	0.00587
Pthread	1024	8	0.00377
OpenMP Single loop	1024	8	0.005589
OpenMP Blocked Algorithm	1024	8	0.003941

Table 8: Table showing results for matrix of size 1024*1024 using 16 threads

	Matrix Size	No. of Threads	Time (s)
Serial	1024	1	0.00587
Pthread	1024	16	0.003741
OpenMP Single loop	1024	16	0.005096
OpenMP Blocked Algorithm	1024	16	0.004495

Table 9: Table showing results for matrix of size 1024*1024 using 64 threads

	Matrix Size	No. of Threads	Time (s)
Serial	1024	1	0.00587
Pthread	1024	64	0.004024
OpenMP Single loop	1024	64	0.005621
OpenMP Blocked Algorithm	1024	64	0.005703

Table 10: Table showing results for matrix of size 1024*1024 using 128 threads

	Matrix Size	No. of Threads	Time (s)
Serial	1024	1	0.00587
Pthread	1024	128	0.004981
OpenMP Single loop	1024	128	0.008253
OpenMP Blocked Algorithm	1024	128	0.006457

3.8 Matrix Size 8192

Table 11: Table showing results for matrix of size 8192*8192 using 4 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	8192	1	0.606175
Pthread	8192	4	0.391196
OpenMP Single loop	8192	4	0.287986
OpenMP Blocked Algorithm	8192	4	0.278515

Table 12: Table showing results for matrix of size 8192*8192 using 8 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	8192	1	0.606175
Pthread	8192	8	0.364242
OpenMP Single loop	8192	8	0.297446
OpenMP Blocked Algorithm	8192	8	0.284069

Table 13: Table showing results for matrix of size 8192*8192 using 16 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	8192	1	0.606175
Pthread	8192	16	0.361604
OpenMP Single loop	8192	16	0.303974
OpenMP Blocked Algorithm	8192	16	0.290174

Table 14: Table showing results for matrix of size 8192*8192 using 64 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	8192	1	0.606175
Pthread	8192	64	0.361683
OpenMP Single loop	8192	64	0.302052
OpenMP Blocked Algorithm	8192	64	0.291543

Table 15: Table showing results for matrix of size 8192*8192 using 128 threads

	Matrix Size	**No. of Threads**	**Time (s)**
Serial	8192	1	0.606175
Pthread	8192	128	0.371873
OpenMP Single loop	8192	128	0.301045
OpenMP Blocked Algorithm	8192	128	0.287934

3.9 Performance analysis of multithreaded algorithms

Figure 1 shows the performance times of the three presented parallel solutions. It can be observed that Pthreads is on average 31% slower than the blocked OpenMP solution.

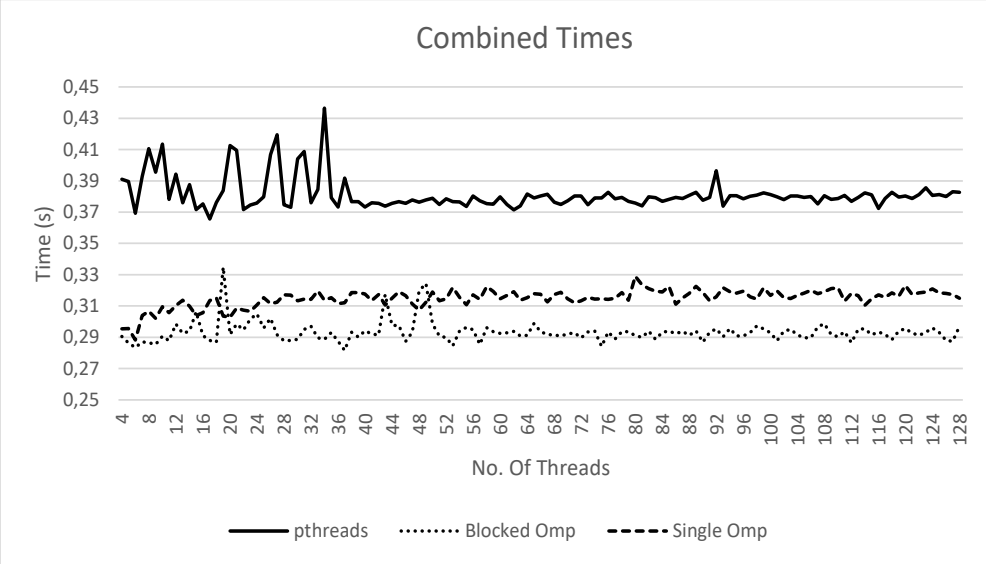

Figure 1: Graphical illustration of the performance times of the three parallel implemented solutions at different thread allocations, for an array size of 8192*8192

4 Evaluation of Solutions

The results show that a parallel solution is not always less time intensive and efficient compared to a traditional serial solution. In Section 3.6 it can be seen that the serial algorithm is faster than all the multithreaded algorithms. This can be because the array is relatively small and there is a decrease in bandwidth of the processing speed allocated to different threads. However, when the array size is increased the benefits in performance of multithreading are more evident. Table 7 shows that the OpenMP blocked algorithm is 32% faster than the serial algorithm, for Section 3.8 OpenMP blocked algorithm is 53% faster. The Pthreads algorithm is the slowest of all the parallel algorithms. This is because the Pthreads library is more flexible and requires greater depth of knowledge and experience into parallel computing to efficiently implement compared to OpenMP. For example, the Pthreads implementation does not take into account queue management or synchronisation. The OpenMP library does more work behind the scenes abstracting functionality away from the programmer. One critical observation is that more threads does not necessary translates into greater performance. The benefits of having multiple threads can be greatly reduced, sometimes nullified, by the overheads incurred through the management of the threads. Having more threads requires better designed thread management implementation. Figure 1 shows the time performance of the parallel algorithms and how this performance changes with an increasing number of threads.

5 Conclusion

The laboratory demonstrates the benefits of multithreading algorithms by comparing three different parallel solutions. OpenMP library was seen to be a more efficient library for parallel algortihms than Pthreads. OpenMP's abstraction allows the programmer to worry less about smaller details and focus more on the solution. The results showed that 8 threads and the OpenMP library yields the best performance for this particular system configuration. One very important observation is the fact that more threads do not directly translate to faster performance. The number of threads should be chosen by evaluating the overheads associated with the thread management.

References

[1] CPrograming,$https://cboard.cprogramming.com/c-programming/133513-traversing-n-dimensional-array.html$ Last accessed February 2018.

[2] Andrey Vladimirov, Colfax Research, August 12, 2013 $https://colfaxresearch.com/multithreaded-transposition-of-square-matrices-with-common-code-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors/$ Last accessed 07 March 2018