PDC Task 1: Parallel Programming Using OpenMP

1. Task Description

The assigned task was to implement two sorting algorithms at choice, from the possible set of: Mergesort, Quicksort, Bubblesort, and Radix Sort. The implementation of the algorithms had to be done in a parallel fashion using OpenMP, an API employed to explicitly direct Multi-Threaded, Shared Memory Model parallelism in applications. Additionally, the performance of these algorithms needs to be gauged and compared for varying problem sizes, and thread number. For this particular paper, the Mergesort, and Quicksort algorithms have been chosen to be implemented, and analyzed. The analysis was done on an 'Intel Core i7-3520M' CPU running at 2.9Ghz. It is to be noted that this processor has **two physical cores** and **four logical cores**. Thus, it is important to say that the optimal number of threads for this processor is **four**. This is quite important, because, as it will be seen later, performance will rarely increase past four threads, and, generally speaking, more often than not, it will actually decrease.

2. Algorithm Analysis

2.1 The Quicksort Algorithm

The idea behind the Quicksort algorithm is to recursively divide the array to be sorted into gradually smaller partitions. These partitions are created relative to a so called pivot element, i.e. an element chosen from the current partition, which will separate the current partition into two partitions, one containing the elements smaller than the current pivot, and the other containing those greater. The algorithm is then called recursively on the aforementioned partitions, and so on. The recursive calls will end when a partition can no longer be divided. Ideally, once partitions fall below a certain size threshold they are sorted using a direct sorting algorithm, like Insertion Sort, or Selection Sort, since they yield better performance for smaller sized arrays. However, this is not implemented here, since it was considered that analysis should be done on a pure version of the Quicksort Algorithm.

The following is the Quicksort algorithm containing a number of OpenMP Directives which are used to create parallelism. The algorithm also uses a random pivot selection feature, as to avoid the worst case of the Quicksort Algorithm, which is $O(n^2)$:

```
void quickSortSequential(int* a, int l, int r)
{
	if (l < r)
	{
		// Generate the proper partitions
		int index = randomPivotPartition(a, l, r);

		// Call the algorithm recursively on the two partitions
		quickSortSequential(a, l, index);
		quickSortSequential(a, index + 1, r);
	}
}

void quickSortParallel(int* a, int l, int r, int splits)
{
	// If there are as many or more spawned threads than the allowable
	// number of threads, continue without generating other threads
	if (splits >= threads)
		quickSortSequential(a, l, r);
	else
	{
		int index = randomPivotPartition(a, l, r);

	// ‘splits’ is the number of threads created so far
	// ‘threads’ is the maximal number of threads allowed
#pragma omp parallel sections num_threads(2) if (threads > splits)
		{
#pragma omp section
			quickSortParallel(a, l, index, splits * 2);
#pragma omp section
			quickSortParallel(a, index + 1, r, splits * 2);
		}
	}
}
```

Fig. 1: Quicksort Code, exemplifying the OpenMP Directives

The above code represents a portion of the implementation of the Quicksort Algorithm. As seen above there are two functions: ‘quickSortSequential’, and ‘quickSortParallel’. The function ‘quickSortParallel’ contains three OpenMP directives. The first one, i.e. the PARALLEL SECTIONS directive will explicitly try to create two threads, through the ‘num_threads(2)’ clause, where each is intended to execute a different section from the two defined by the SECTION directives. It is to be noted that this directive also contains an IF clause which will control whether or not the parallel block will truly be executed in parallel, or sequentially. If the IF clause evaluates to a non-zero value, then the following block will be executed in parallel, otherwise it will be executed sequentially. In this case however the IF clause is added just as a precaution, since the ‘if’ condition at the beginning of the ‘quickSortParallel’ function will test whether or not the maximal number of threads has been achieved. If it has, the thread will continue its execution sequentially, without trying to ever create a new team of threads, since it will call ‘quickSortSequential’, a function which relies solely on sequential execution. The algorithm has been implemented in such a way, due to the fact that, through testing and evaluation it has been discovered that better execution times can be obtained without having to force OpenMP to sequentially execute a block which is intended to be executed in parallel. As such, this overhead will be avoided by relying on the 'if' condition to test if the maximal number of allowed threads is lower than, or equal to the number of spawned threads.

The SECTION directives will distribute the work to the two threads in the thread team created by the SECTIONS directive. One thread will recursively call 'quickSortParallel' on the left partition, whilst the other thread will call the same function on the right partition.

As can be seen above this parallel model requires nested threads. This is enabled in the code by supplying the 'omp_set_nested' function a non-zero value. Furthermore, dynamic thread control will be disabled by passing a zero to the 'omp_set_dynamic' function call:

```
 // Enable nested multithreading
omp_set_nested(1);

 // Disable dynamic feature
omp_set_dynamic(0);
```

Fig. 2: Enabling nesting, and disabling dynamic threads

Further details regarding the 'randomPivotPartition' function can be obtained by taking a look at the associated source code for the Quicksort algorithm, which is amply documented. The complete code has not been added here, for brevity reasons.

The following are measurements obtained from the performance analysis of the Quicksort algorithm:

Problem Size		Actual Time (seconds)			
		Thread Number			
		1	2	4	8
1000000	1st Run	0.060004	0.048875	0.037786	0.038247
	2nd Run	0.061272	0.060418	0.040906	0.052732
	3rd Run	0.059908	0.037096	0.039497	0.040825
2000000	1st Run	0.120372	0.108743	0.096796	0.072824
	2nd Run	0.120747	0.077681	0.076428	0.068711
	3rd Run	0.120346	0.101331	0.068826	0.073699
4000000	1st Run	0.242054	0.244414	0.178725	0.160673
	2nd Run	0.237735	0.139384	0.113003	0.115729
	3rd Run	0.232971	0.223191	0.185497	0.186916
8000000	1st Run	0.47969	0.265746	0.250201	0.273243

	2nd Run	0.479701	0.309529	0.26432	0.301879
	3rd Run	0.474811	0.28554	0.257598	0.294173
16000000	1st Run	0.952551	0.900449	0.687443	0.70208
	2nd Run	0.942828	0.866135	0.763981	0.760372
	3rd Run	0.985197	0.700609	0.473004	0.490434

Fig. 3: Execution times(seconds) obtained for different problem sizes and different number of threads

By averaging the times presented above, and computing the speedup, using the formula:

$$\frac{sequential\_time}{parallel\_time\_4\_threads} * 100$$

Fig. 4: Speedup formula

We obtain the following data:

Problem Size	Average Time (seconds)				Speedup (%)
	Thread Number				
	1	2	4	8	
1000000	0.060394	0.048796	0.039396	0.043935	153%
2000000	0.120488	0.095918	0.080683	0.071745	150%
4000000	0.237587	0.20233	0.159075	0.154439	150%
8000000	0.478067	0.286938	0.257373	0.289765	185%
16000000	0.960192	0.822398	0.641476	0.650962	150%

Fig. 5: Average execution times and speedups obtained through parallelization

As can be seen above the parallelization of the Quicksort Algorithm yields an average speedup of 150%, no matter the problem size. It is worth mentioning that the average execution times do increase the more the size of the input array increases, especially for sequential execution, where it is quite clear, from the obtained data, that doubling the problem size, will result in doubling the execution time as well. Nonetheless parallelization yields better results than the sequential execution, and as such is recommended. Below, a graphical representation of the above data is presented:

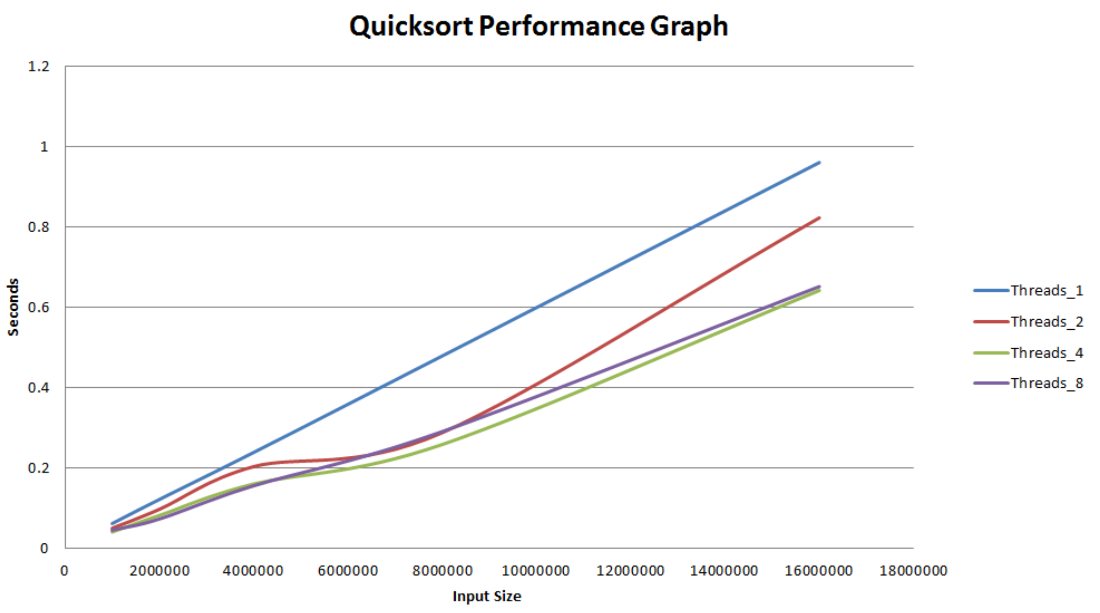


Fig. 6: Graph of the average execution times of the Quicksort Algorithm

As previously mentioned the processor on which the tests were run has two physical cores, and four logical ones, meaning that the ideal number of threads, in terms of parallelization is four, given that the workload of the threads is equal. Thus, as can be seen above, running the algorithm on eight threads yields no real improvement to the execution times. In fact it produces slightly worse results. This is, however, not always the case, as more threads can yield better results, in those cases when the workload is unbalanced, as is the following case:

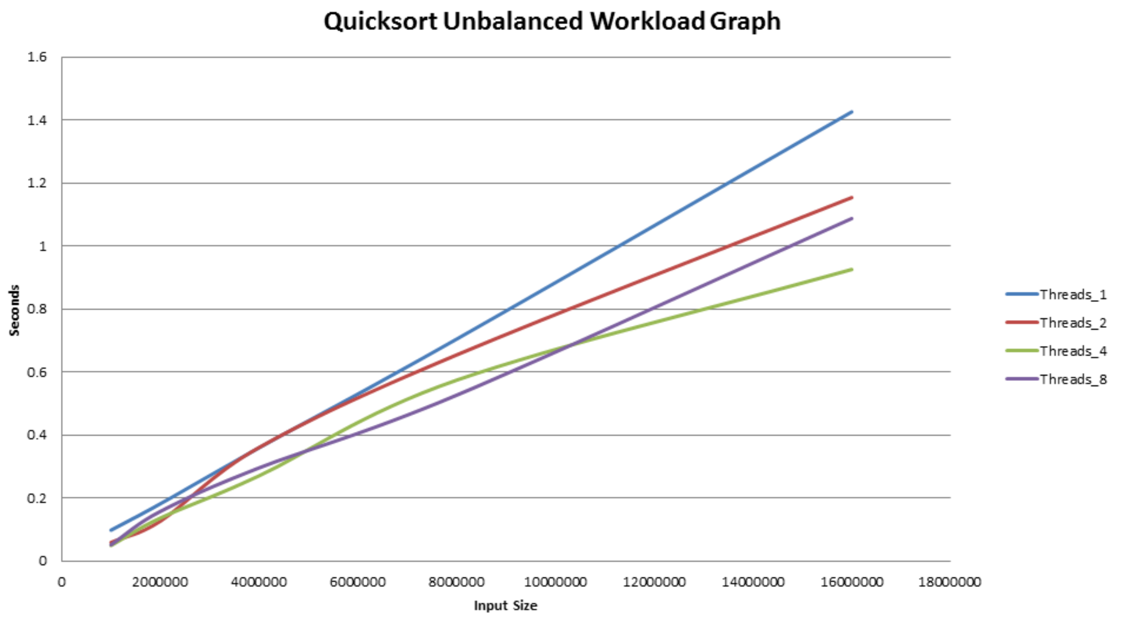
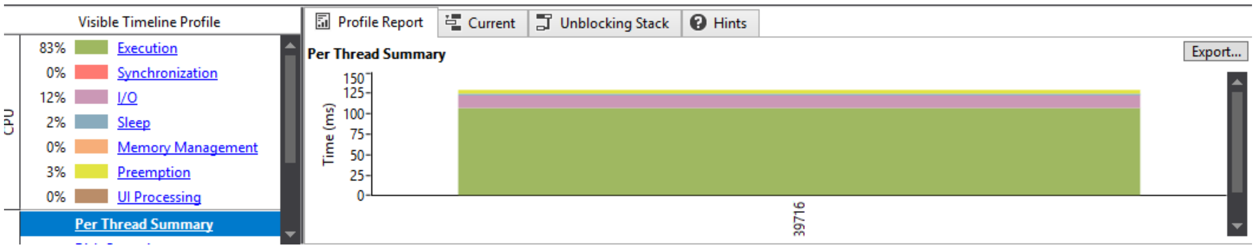


Fig. 7: Unbalanced case for the Quicksort Algorithm

To correct this problem, work-balancing mechanisms are required to make sure that each thread receives relatively the same amount of work. However these mechanisms need to be highly efficient. If not, they might overburden the system, and do more harm than help, as they might increase execution times.

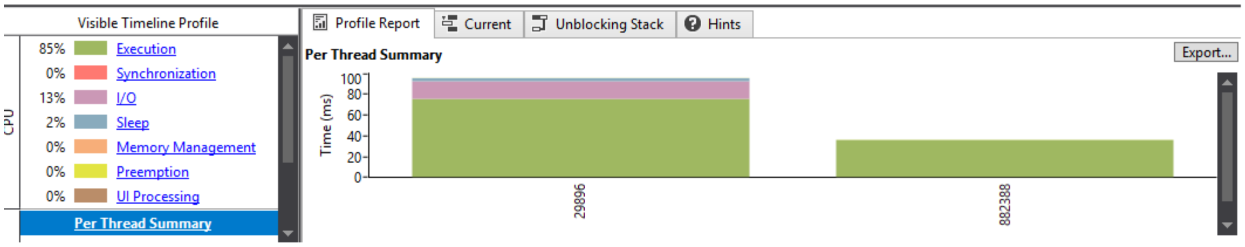
Snapshots of the analysis provided by the Concurrency Analyzer are presented below. These snapshots present the 'Per Thread Summaries' for executions on the same problem size, but for different numbers of threads:



Visible Timeline Profile

83%	🟩	Execution
0%	🟥	Synchronization
12%	▨	I/O
2%	▨	Sleep
0%	▨	Memory Management
3%	▨	Preemption
0%	▨	UI Processing

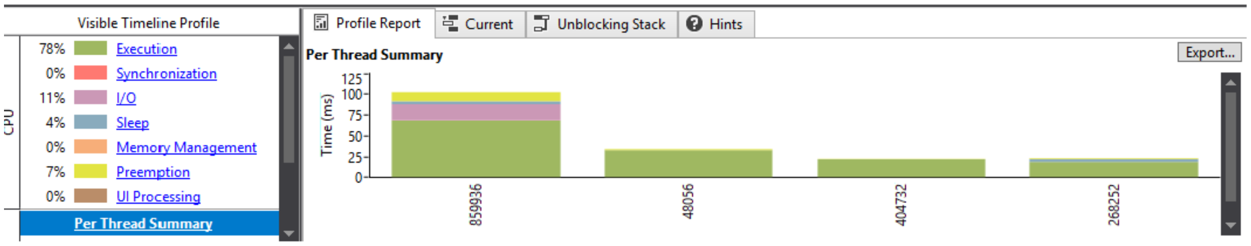
Fig. 8: The 'Per Thread Summary' of the Sequential execution



Visible Timeline Profile

85%	🟩	Execution
0%	🟥	Synchronization
13%	▨	I/O
2%	▨	Sleep
0%	▨	Memory Management
0%	▨	Preemption
0%	▨	UI Processing

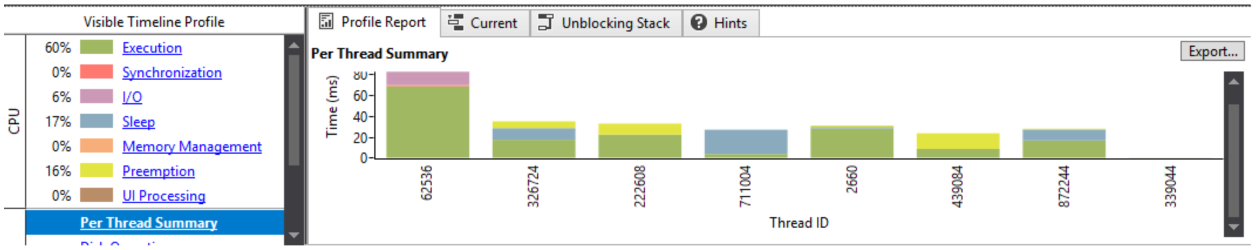
Fig. 9: The 'Per Thread Summary' of the two threaded execution



Visible Timeline Profile

78%	🟩	Execution
0%	🟥	Synchronization
11%	▨	I/O
4%	▨	Sleep
0%	▨	Memory Management
7%	▨	Preemption
0%	▨	UI Processing

Fig. 10: The 'Per Thread Summary' of the four threaded execution

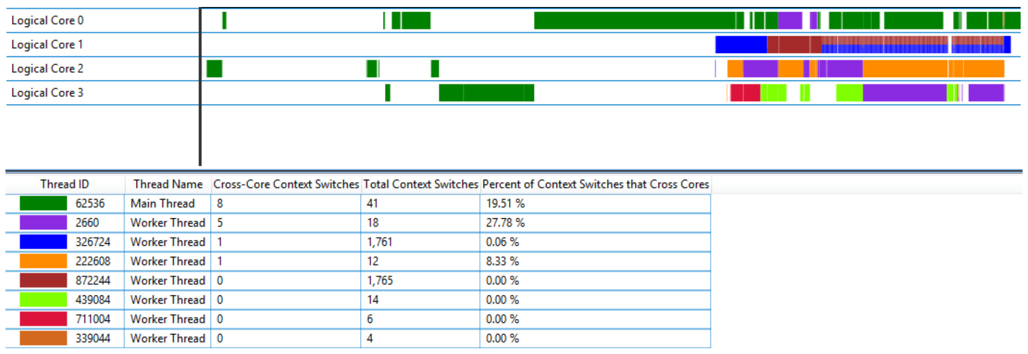


Visible Timeline Profile

60%	🟩	Execution
0%	🟥	Synchronization
6%	▨	I/O
17%	▨	Sleep
0%	▨	Memory Management
16%	▨	Preemption
0%	▨	UI Processing

Fig. 11: The 'Per Thread Summary' of the eight threaded execution

As seen above, the threads will stall very little when there are at most four, since that is the maximal number of threads which can run in parallel on this particular processor. However, once the number of threads increases past the number of logical cores, they begin competing with one another for the processor's attention, forcing some threads to stall. As such, the best results are most often obtained by spawning as many threads as there are logical cores, and assigning them equal work-loads. Below is a graphic, generated by the Concurrency Analyzer, showing the frequency of the context switches in the case of the eight threaded execution:



Thread ID		Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
🟩	62536	Main Thread	8	41	19.51 %
🟪	2660	Worker Thread	5	18	27.78 %
🟦	326724	Worker Thread	1	1,761	0.06 %
🟧	222608	Worker Thread	1	12	8.33 %
🟫	872244	Worker Thread	0	1,765	0.00 %
🟩	439084	Worker Thread	0	14	0.00 %
🟥	711004	Worker Thread	0	6	0.00 %
🟧	339044	Worker Thread	0	4	0.00 %

Fig. 12: Context switching in the eight threaded execution

2.1 The Mergesort Algorithm

The idea behind the Mergesort Algorithm is to divide the array to be sorted recursively into smaller and smaller partitions. The division process will end when partitions can no longer be divided due to their size. At that point, partitions will begin to be gradually merged together in order to sort the array.

```
void mergeSort(int* a, int l, int r, int splits)
{
       if (l < r)
       {
              // Find the middle index
              int middle = (r + l) / 2;

              // Create a new team of two threads
#pragma omp parallel sections num_threads(2) if (threads > splits)
              {
#pragma omp section
                     mergeSort(a, l, middle, splits * 2);
#pragma omp section
                     mergeSort(a, middle + 1, r, splits * 2);
              }

              // Use the inplace_merge since it's likely more efficient
              std::inplace_merge(a + l, a + middle + 1, a + r + 1);
       }
}
```

Fig. 13: Mergesort Code, exemplifying the OpenMP Directives

Once more, the code presented above will make use of the SECTIONS directive. This directive will attempt to create a team of two threads, where each thread is intended to execute a different SECTION from the available two, defined within the SECTIONS block. The block of code defined by the SECTIONS directive will only be executed in parallel if the IF clause evaluates to a non-zero value, in this case, if the number of spawned threads is less than the maximal number of threads. Otherwise, the block will be executed sequentially.

As seen above, this will require nested multithreading, and as such this feature needs to be enabled in the OpenMP API, by calling '`omp_set_nested`' with a non-zero parameter, as shown below:

```
 // Enable nested multithreading
omp_set_nested(1);

 // Disable dynamic feature
omp_set_dynamic(0);
```

Fig. 14: Enabling nesting, and disabling dynamic threads

In order to make sure that dynamic thread control is not enabled, and that the parallel behavior can be explicitly controlled during execution, the '`omp_set_dynamic`' function will be called with a zero parameter.

The following are measurements obtained from the performance analysis of the Mergesort Algorithm, performed on various problem sizes, and thread numbers:

Problem Size		Actual Time (seconds)			
		Thread Number			
		1	2	4	8
1000000	1st Run	0.375926	0.279158	0.210717	0.241261
	2nd Run	0.364861	0.194532	0.155912	0.205665
	3rd Run	0.358285	0.224837	0.210974	0.226399
2000000	1st Run	1.16578	0.61282	0.373258	0.403552
	2nd Run	0.744621	0.385428	0.415744	0.416415
	3rd Run	1.13111	0.665809	0.482829	0.438043
4000000	1st Run	1.87518	1.0903	0.700701	0.671458
	2nd Run	1.87218	1.4528	0.77554	0.77711
	3rd Run	1.8634	1.49666	0.767458	0.720087
8000000	1st Run	3.33918	1.7739	1.32526	1.39002
	2nd Run	3.38373	2.53092	1.42272	1.4225
	3rd Run	3.34973	2.20414	1.42154	1.42607
16000000	1st Run	6.25524	3.69293	2.60178	2.64557
	2nd Run	6.33818	4.0716	2.78296	2.83367

	3rd Run	6.4394	3.9294	2.69777	2.69157

Fig. 15: Execution times(seconds) obtained for different problem sizes and different number of threads

By averaging the times presented above, and computing the speedup, using the formula presented in **Fig. 4**, we will obtain the following data:

Problem Size	Average Time (seconds)				Speedup (%)
	Thread Number				
	1	2	4	8	
1000000	0.366357	0.232842	0.192534	0.224442	190%
2000000	1.013837	0.554686	0.423944	0.419337	240%
4000000	1.870253	1.346587	0.7479	0.722885	250%
8000000	3.357547	2.169653	1.38984	1.412863	240%
16000000	6.344273	3.897977	2.69417	2.723603	235%

Fig. 16: Average execution times and speedups obtained through parallelization

As can be seen above, the parallelization of the Mergesort Algorithm will yield great results, obtaining speedups of up to 250%. From the data, it is clear to see that sequential execution will greatly increase, time-wise, as the size of the problem increases. As such, parallelization is highly recommended in order to obtain better performances.

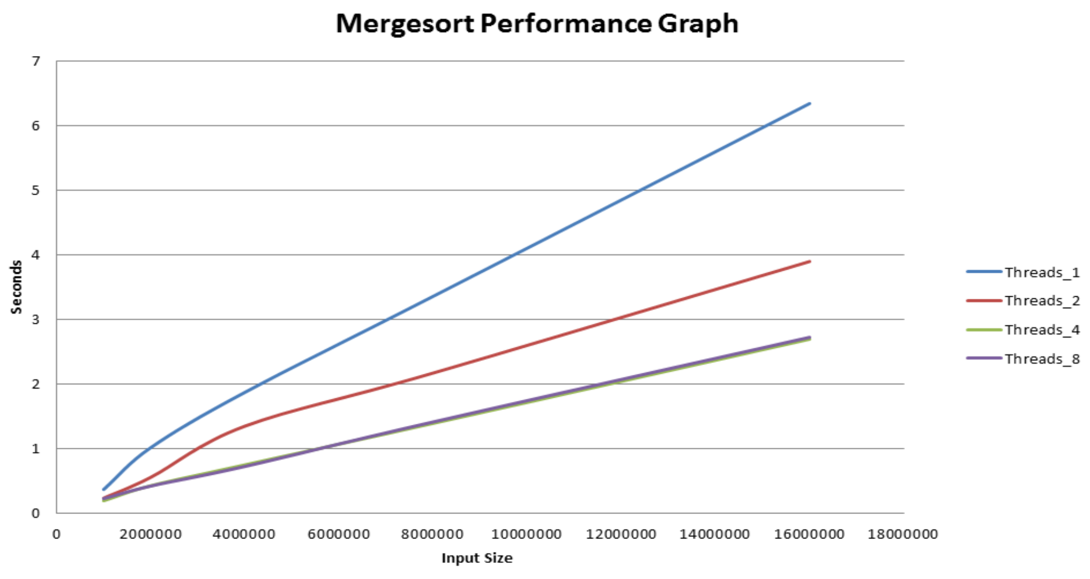


Fig. 17: Graph of the average execution times of the Mergesort Algorithm

As previously mentioned, the processor on which the tests were run supports at most four parallel threads. This can be clearly observed, by studying the above graph. As can be seen above, the performance of the four threaded execution, and of the eight threaded execution is nearly identical. By taking a closer look at the graph, however, we'll see that the four threaded execution yields slightly better results. The sequential execution is far behind the parallel one, and it's clear to see the performance improvements brought on by parallelism.

Snapshots of the analysis provided by the Concurrency Analyzer are presented below. These snapshots present the 'Per Thread Summaries' for executions on the same problem size, but for different numbers of threads:

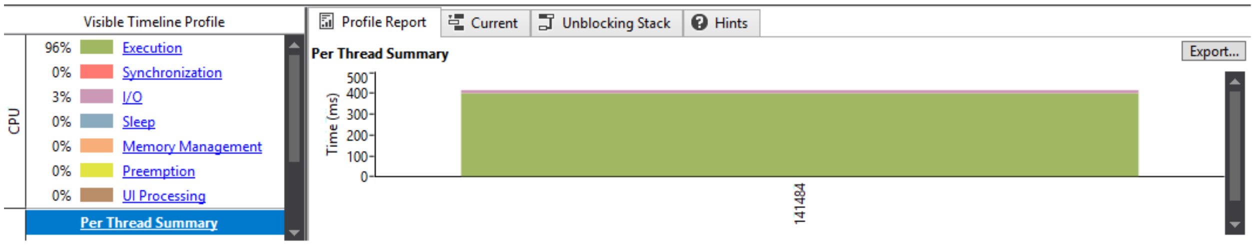


Fig. 18: The 'Per Thread Summary' of the Sequential execution

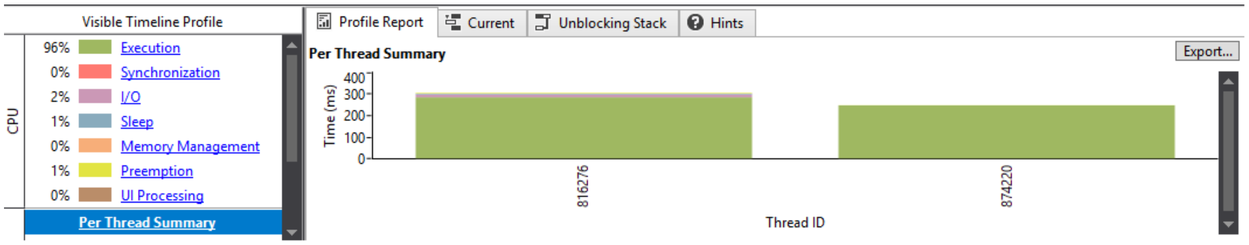


Fig. 19: The 'Per Thread Summary' of the two threaded execution

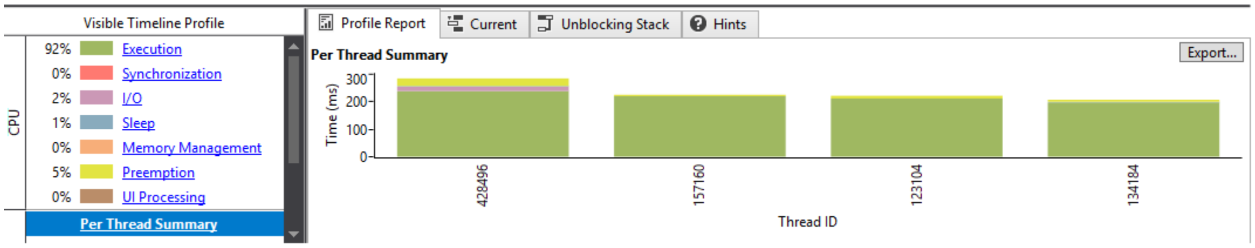


Fig. 20: The 'Per Thread Summary' of the four threaded execution

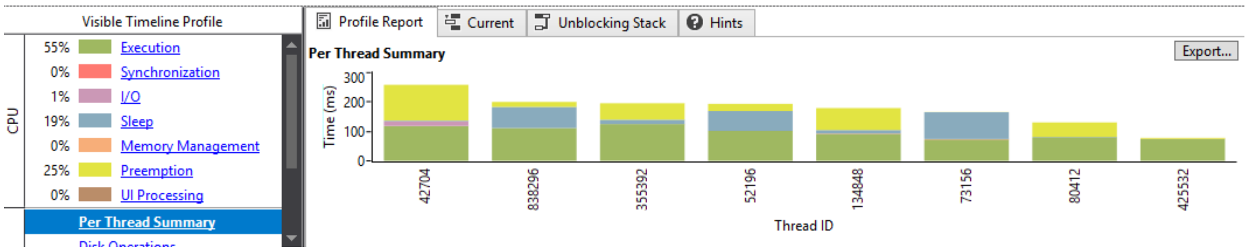
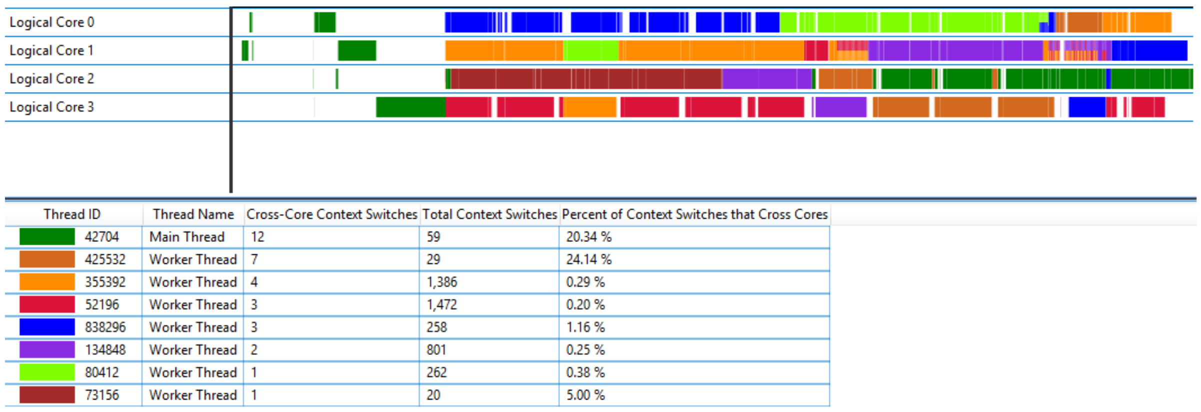


Fig. 21: The 'Per Thread Summary' of the eight threaded execution

The threads will execute with very little stalling, if any, when there are at most four of them. However, once the number of threads increases past the number of logical cores, which in this case is four, threads will be forced to stall, as they will compete with one another for a core on which to execute. As such, the best results are most often obtained by spawning as many threads as there are logical cores, and giving them equal work-loads. Below is a graphic, generated by the Concurrency Analyzer, showing the frequency of the context switches in the case of the eight threaded execution:



Thread ID	Thread Name	Cross-Core Context Switches	Total Context Switches	Percent of Context Switches that Cross Cores
🟩 42704	Main Thread	12	59	20.34 %
🟫 425532	Worker Thread	7	29	24.14 %
🟧 355392	Worker Thread	4	1,386	0.29 %
🟥 52196	Worker Thread	3	1,472	0.20 %
🟦 838296	Worker Thread	3	258	1.16 %
🟪 134848	Worker Thread	2	801	0.25 %
🟩 80412	Worker Thread	1	262	0.38 %
🟥 73156	Worker Thread	1	20	5.00 %

Fig. 22: Context switching in the eight threaded execution

2.2 Quicksort and Mergesort Performance Comparison

As previously presented the best performance gains were obtained on four threads. However, by comparing the execution times of the two sorting algorithms, it is clear to see that the parallel Quicksort implementation does much better than the parallel Mergesort implementation. This is most likely due to the workload brought on by the merge step in the Mergesort Algorithm. Below is a comparison, in tabular format, between the average execution speeds of the two algorithms, on varying problem sizes, and on two or four threads:

Problem Size	Average Time (seconds)				Speedup (%)
	Mergesort		Quicksort		
	Thread Number				
	2	4	2	4	
1000000	0.232842	0.192534	0.048796	0.039396	490%
2000000	0.554686	0.423944	0.095918	0.080683	525%
4000000	1.346587	0.7479	0.20233	0.159075	470%
8000000	2.169653	1.38984	0.286938	0.257373	540%
16000000	3.897977	2.69417	0.822398	0.641476	420%

Fig. 23: Comparison table between the Quicksort and Mergesort execution speeds

The speedup was calculated using the following formula:

$$\frac{mergesort\_4\_threads}{quicksort\_4\_threads} * 100$$

Fig. 24: Speedup formula

The speedup obtained from employing the Quicksort algorithm, over the Mergesort algorithm is quite significant, yielding performance improvements of up to 500%. As such it is likely better to favor a parallel Quicksort implementation over a parallel Mergesort implementation, since the results are much better, and so is the scalability. Below is a graphical representation of the tabular data from above:

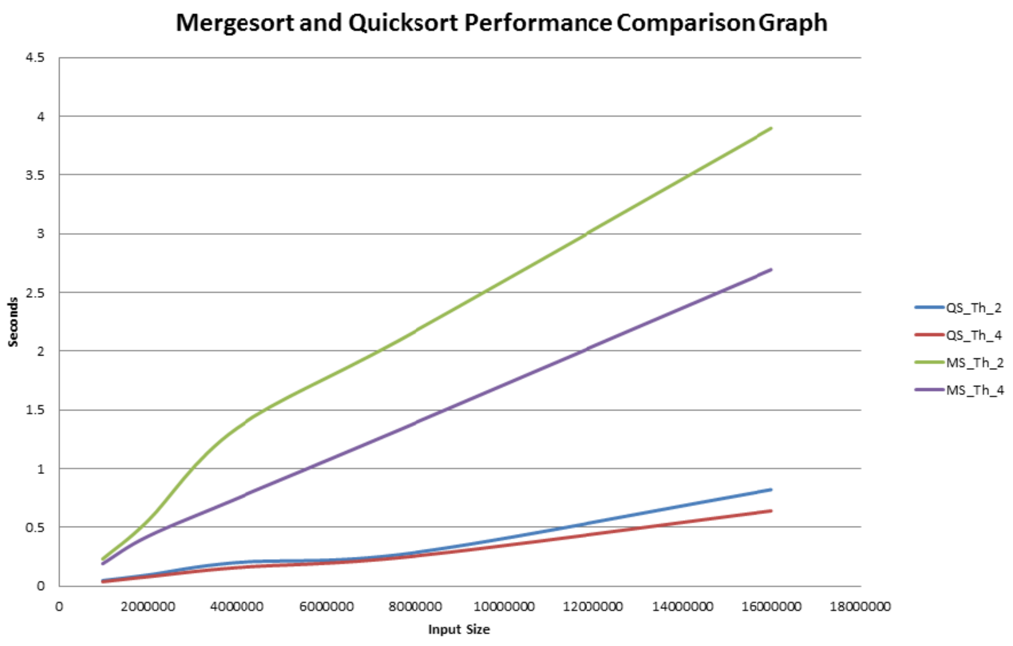


Fig. 25: Comparison graph between the Quicksort and Mergesort execution speeds

3. Conclusion

The analysis conducted so far, has revealed that a parallel execution yields much better results than its sequential counterpart. As such, it is clear that if we seek to improve the performance of a certain algorithm or application, parallelism provides us with one of the best ways to achieve this task. In addition, parallelism improves scalability, as it is clearly seen in the above data, being a much more suitable replacement for sequential implementations, in terms of large scale problems.

In terms of comparing the parallel Mergesort Algorithm with the parallel Quicksort Algorithm, it has been revealed that Quicksort does much better than Mergesort, although both are algorithms with an average complexity of $O(n * \log(n))$. As such, Quicksort is a better candidate when it comes to sorting, as it is faster, and scales much better.