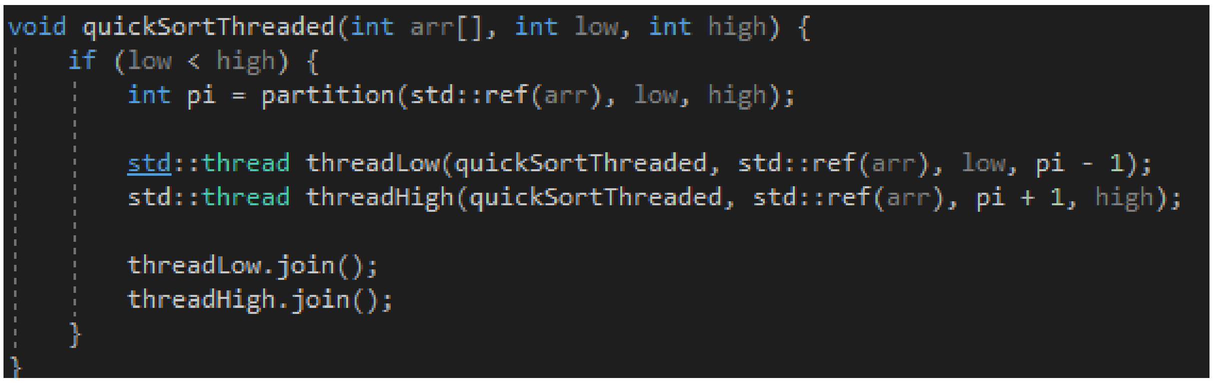
Program Decomposition and Analysis of the Sequential Algorithm

The sequential quicksort algorithm I'll be focusing on is the recursive implementation with the last array element always selected as the pivot during the partitioning stage.

The ***expected*** runtime of this algorithm is already quite fast with a complexity of O(nlogn). However in the worst case, i.e. when the pivot selected is ***always*** the highest or lowest element in the partitioning stage, the algorithm becomes O(n^2) which has drastic performance drawbacks for larger datasets. This task will aim to focus on the ***expected*** complexity.

Similar to Task-1P, it is important to recognise that in order for a parallel solution to be more efficient than its sequential counterpart, a significant amount of work needs to be done within each thread to negate the effects of the overhead caused by initialising, starting, and joining each thread.

Initially, I thought I could just use this snippet inside my quickSort recursive function:



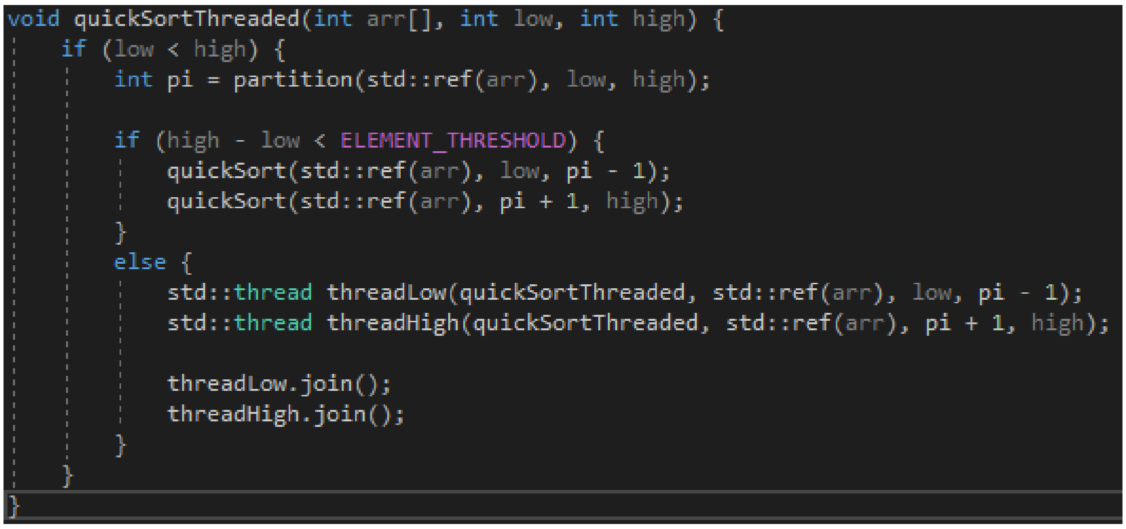
```
void quickSortThreaded(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(std::ref(arr), low, high);

        std::thread threadLow(quickSortThreaded, std::ref(arr), low, pi - 1);
        std::thread threadHigh(quickSortThreaded, std::ref(arr), pi + 1, high);

        threadLow.join();
        threadHigh.join();
    }
}
```

This will create two new threads at each level of recursion to sort it's respective datasets, and join the together once both sides are done. At first glance this seems like the optimal and simplest solution. However, this runs into many problems no matter how large the dataset is. At each level of recursion, the dataset is reduced to be half in the best case or only drop 1 element in the worst case. As we make our way down the recursion tree, the difference between 'low' and 'high' will only be a couple of elements, and we are still creating new threads to handle these, wasting resources and generating unnecessary overheads.

Instead, I decided to include an ELEMENT_THRESHOLD defined as NUM_ELEMENTS / 2, which will create a total of 6 threads in the best case (2 in the first pass, 2 more in first threadLow call, 2 more in the first threadHigh call).

Taking inspiration from the well-known Timsort algorithm - a hybrid sorting algorithm that uses merge sort until the remaining unsorted dataset becomes too small, at which point it reverts to using insertion sort – in my algorithm, once the number of elements at the current step (defined by high – low) is below the element threshold, the algorithm abandons the threaded quicksort and uses the sequential quickSort method for the remainder of its recursion.

Due to the nature of the pivot selection, this will yield some varied results as far as execution time goes, because different threads will potentially be handling very different loads. The new code looks like the following:



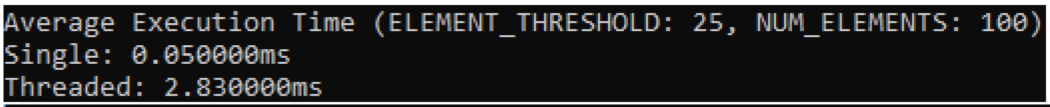
```
void quickSortThreaded(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(std::ref(arr), low, high);

        if (high - low < ELEMENT_THRESHOLD) {
            quickSort(std::ref(arr), low, pi - 1);
            quickSort(std::ref(arr), pi + 1, high);
        }
        else {
            std::thread threadLow(quickSortThreaded, std::ref(arr), low, pi - 1);
            std::thread threadHigh(quickSortThreaded, std::ref(arr), pi + 1, high);

            threadLow.join();
            threadHigh.join();
        }
    }
}
```

Now, threads will only be created to handle datasets with more than or equal to ELEMENT_THRESHOLD elements in them, ensuring that there will be enough work to negate the overheads if ELEMENT_THRESHOLD is properly configured and the original dataset is large enough.

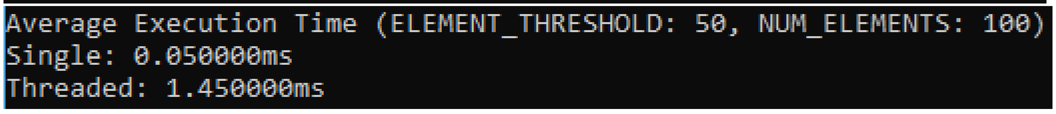
Benchmarking and Metrics

The program is executed 100 times (defined by NUM_EXECUTIONS) and the average execution time of all runs is calculated for both the sequential and threaded implementations.

Firstly, I tested a dataset with a relatively small amount of elements (100), and element thresholds of 25 and 50. As the element threshold increases, less threads are used, and the threaded implementation becomes more alike to the sequential implementation, which shows in the execution time differences.

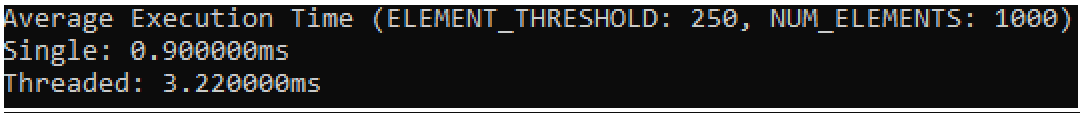


```
Average Execution Time (ELEMENT_THRESHOLD: 25, NUM_ELEMENTS: 100)
Single: 0.050000ms
Threaded: 2.830000ms
```

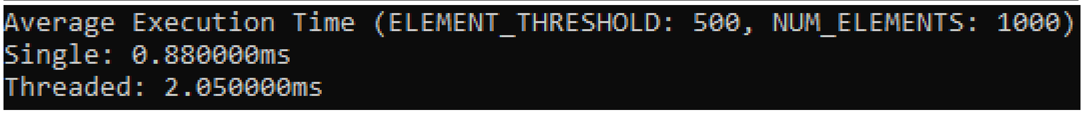


```
Average Execution Time (ELEMENT_THRESHOLD: 50, NUM_ELEMENTS: 100)
Single: 0.050000ms
Threaded: 1.450000ms
```

Next, 1000 element random datasets were tested. Still not enough work to be done by threads even at higher element thresholds, but the gap between the sequential and threaded execution times has already significantly closed.

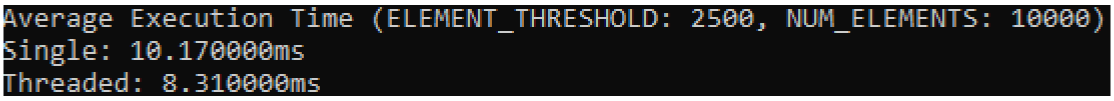


```
Average Execution Time (ELEMENT_THRESHOLD: 250, NUM_ELEMENTS: 1000)
Single: 0.900000ms
Threaded: 3.220000ms
```

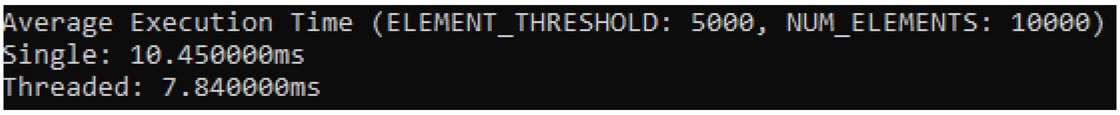


```
Average Execution Time (ELEMENT_THRESHOLD: 500, NUM_ELEMENTS: 1000)
Single: 0.880000ms
Threaded: 2.050000ms
```

The threaded implementation overtakes the sequential implementation somewhere in between the 1000 – 10,000 element mark, but is clearly shown at 10,000. A lower threshold of 2500, which creates more threads, is still slower than a higher threshold of 5,000, indicating that the amount of work is still quite low below 5,000 elements.

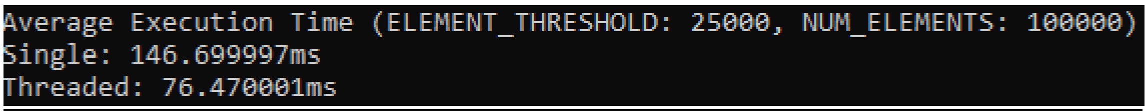


```
Average Execution Time (ELEMENT_THRESHOLD: 2500, NUM_ELEMENTS: 10000)
Single: 10.170000ms
Threaded: 8.310000ms
```

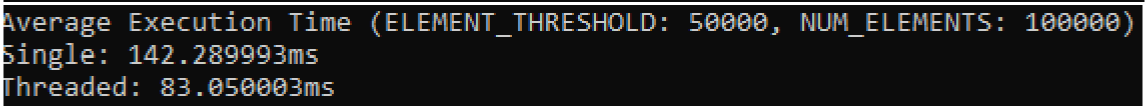


```
Average Execution Time (ELEMENT_THRESHOLD: 5000, NUM_ELEMENTS: 10000)
Single: 10.450000ms
Threaded: 7.840000ms
```

At 100,000 elements the sequential algorithm is almost 100% slower than the threaded version, and a noticeable improvement at a lower threshold of 25,000 compared to 50,000 can be noticed.

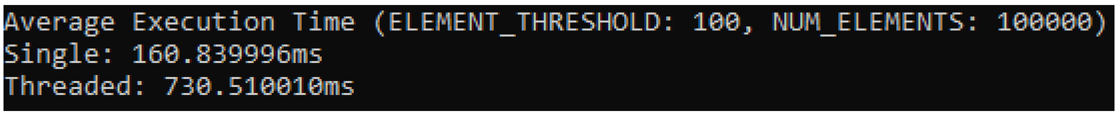


```
Average Execution Time (ELEMENT_THRESHOLD: 25000, NUM_ELEMENTS: 100000)
Single: 146.699997ms
Threaded: 76.470001ms
```



```
Average Execution Time (ELEMENT_THRESHOLD: 50000, NUM_ELEMENTS: 100000)
Single: 142.289993ms
Threaded: 83.050003ms
```

Finally, as a demonstration, using a very low threshold of 100 in a 100,000 element dataset yields very poor results from the threaded implementation – 454% slower than the sequential implementation.



```
Average Execution Time (ELEMENT_THRESHOLD: 100, NUM_ELEMENTS: 100000)
Single: 160.839996ms
Threaded: 730.510010ms
```

This task has demonstrated the importance of carefully selecting how many threads should be created to provide the most optimal performance boost. It is essential that each thread has enough work to do to considerably outweigh the performance impact of the overheads generated by the threads in the first place. Threads can be very useful when a large amount of work needs to be done quickly, as shown in the datasets above 10,000, but sequential implementations are far better when this is not the case, as the overheads provide more delay than the parallel execution can counteract.