CSC2002S
Assignment 1 – Parallel Sorting

Nkosingiphile Gumede
GMDNKO003

1.5) Report

1.5.1) Introduction:

This report aims to explore the effects of parallel computing in the java programming language with the built-in java fork/ join framework. This framework will be applied to sorting one dimensional arrays with a number of sorting methods namely; merge-sort, quick-sort and radix-sort. The fork-join framework is essentially required to split the work required to be computed into a number of processes. This algorithm had its foundations rooted in the necessity to run a number of process. Without which, things like operating systems and graphical user interfaces could not exist.

1.5.2) Methods:

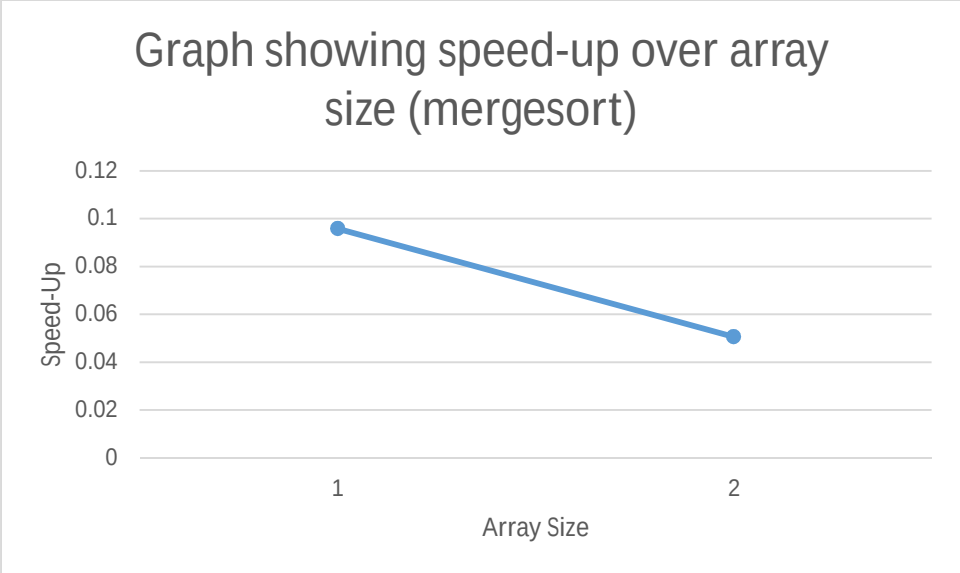
A sequential solution will consistently be pitted against a parallel one to derive a speed-up value. This speed-up value equates to the ratio of the times given to compute the sequential and parallel algorithms respectively, given by the equation: time of parallel algorithm divided by time of sequential algorithm. The variables of change include array size, sequential cut-off and the desired type of sort (either merge-sort, quick-sort or bubble-sort). The number of processors/ cores will also be a variable of interest in understanding the comparison between sequential and parallel algorithms.

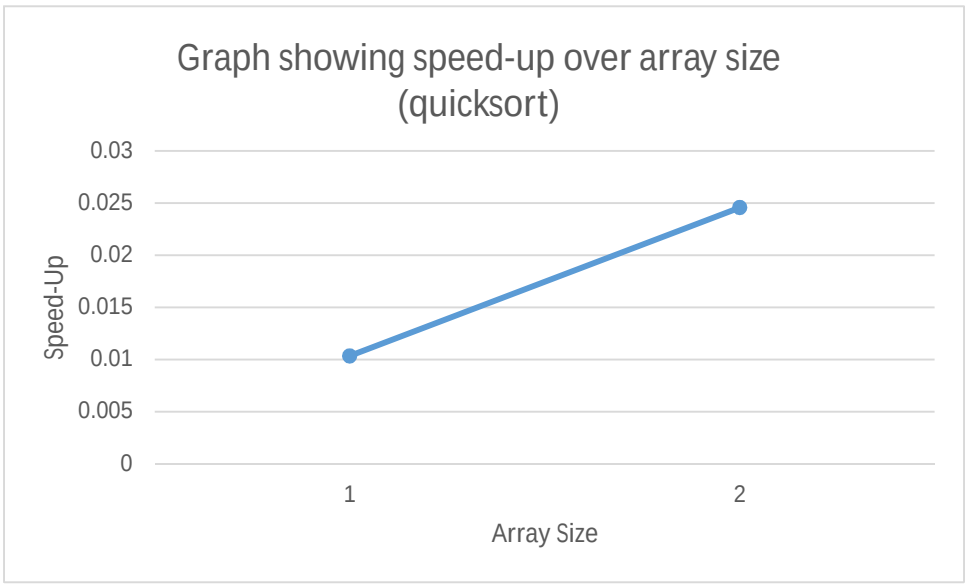
The sequential cut-off represents the size that the sub-array is required to be to compute sequentially. If this array is greater in length than the sequential cut-off, the original array will split at the mid-point index using the fork-join framework. The sub-arrays will follow the same rules and so on. This implies the value of the sequential cut-off will represent the number of threads the algorithm will generate. A sequential cut-off value equal to the size of the original/ input array is essentially just the sequential method. This yields only one thread. Any sequential cut-off less than the size of the original array will yield more than one thread, the point at which parallelization occurs. This can be best understood via the following code extract:

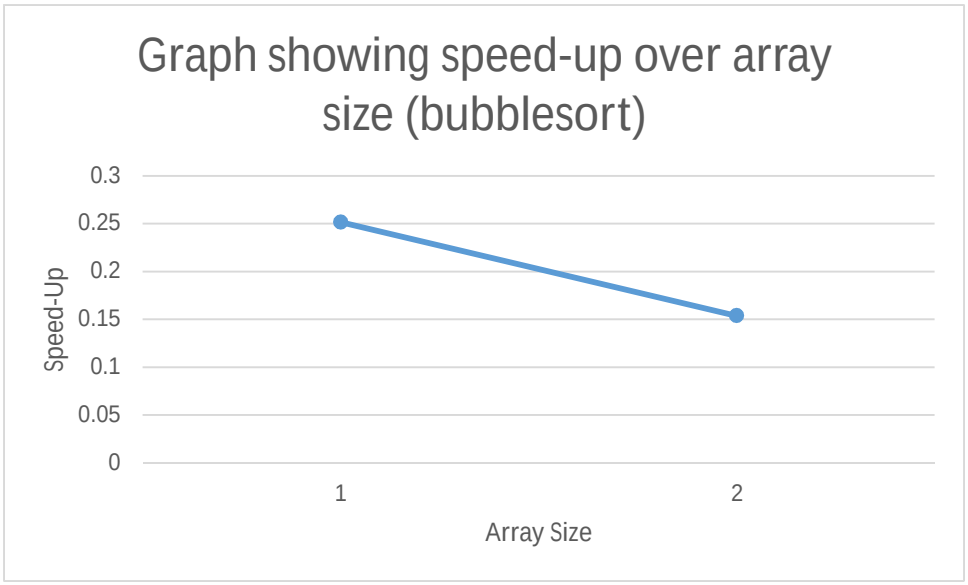
```
if ((hi-lo) < SEQUENTIAL_CUTOFF) { //the size of the sub-array is less than the sequential cut-off
        //median filter the array sequentially...
}
else {
        ParallelSolution left = new ParallelSolution(arr,lo,(hi+lo)/2,filterSize);
        ParallelSolution right= new ParallelSolution(arr,(hi+lo)/2,hi,filterSize);
        left.fork();
        right.compute();
        left.join();
 }
```

1.5.3) Results and Discussions:

By running the relevant tests numerously over a variation of changing variables, it is possible to establish any observation trends in which variables relate to the output data values we derive. The outputs of a number of relevant relationships have been graphed and discussed below.







In these graphs; array size '1' was used to represent the set of arrays ranging from size 100 to 1000 and array size '2' was used to represent the set of arrays ranging from size 1000 to 10000.

As visualized in the graphs above, there is big difference between the speedup times of sequential and parallel algorithms. From the given speed up values multi-threading is clearly efficient and well worth implementing to speed up the computation of programs in cases where the array size is considerably large and the given algorithm takes long to

compute single task, as was mostly the case in the bubble sort method.

Generally speaking, parallel programming performs superiorly with the minimum amount of subsequent threads and on significantly large array sizes. Based on the data from the Excel spreadsheet, the merge-sort (O(n log n)) algorithm performed best in this regard, followed by the quick-sort (O(n log n)) and bubble-sort (O(n^2)) algorithms respectively.

As given by the data above, the best average speedup obtainable was 0,01 (by the quick-sort algortithm). There were no identifiable trends in the maximum speedup obtainable, therefore from this output data it is not possible to establish an actual maximum given that this variable is so varied. Theoretically speaking, doubling the number of threads should half the time taken to compute the parallel solution.

1.5.4) Conclusion:

From the above-mentioned findings it is possible to conclude that parallel programming is effective in producing more efficient algorithms which aid in the implementation of relatively lengthy sequential programs and innovation in the field of computer science. The number of processors a computer possesses further enhanced the efficiency of parallel algorithms. This is especially useful with large data sets. Variables such as sequential cut-off, which dictated the number of threads/ processors; and data set sizes have also had an observable effect on the effectiveness of parallelization.