Dean Chong San (40061194)
Étienne Bérubé (40052212)

Quick Sort with Multi-threading

Introduction

The goal was to implement the divide and conquer quick sort algorithm with multi-threading. This algorithm sorts numbers by choosing a pivot and swapping the numbers based on whether they are greater or smaller than the pivot. When this is done, it redoes this operation but on smaller subsets of the data. This process could be implemented by making use of multi-threading. Doing so, will create a thread when re-partitioning the subsets. The resulting threads can then be run in parallel in the hopes of increasing efficiency.

Procedure

The implementation is written in Java and composed of two particular classes. The first one is the Main class, which also happens to be the runnable class. It receives from the program argument the location of the Input file from which the numbers are stored, reads it, creates the array with the right length and fills the latter with the numbers. It then calls the sort method from the QSort class before creating the output text file and writing the resulting numbers from the sorted array there.

The usual algorithm sorts the numbers by randomly choosing a pivot and swapping the numbers on whether they are bigger or smaller than the pivot. The partitioning of a subset of the array is done using the Hoare partitioning scheme. Once this complete, the QSort class will recursively run this algorithm again but with the upper half and lower half of the array with respect to the previous pivot.

The algorithm was implemented by adding the multi-threading components to it. A thread is created right before the recursive call is executed. Consequently, the partitioning of the subsets are performed in parallel, which in theory, should reduce the computation time. The algorithm checks whether a recursive call is required on the left subset and right subset; if so, it creates a thread and runs it. Once the two threads have been created and initialized, they will be joined by the parent thread. The following code is used to create the threads for the recursive calls on the subsets of data.

```
Thread threadLeft = null;
Thread threadRight = null;

//Call for the first part of the array
if (pivotPos - startingIndex > 1) {
    threadLeft = new Thread(() -> sort(array, startingIndex, pivotPos - 1));
    threadLeft.start();
}


//Call for the second part of the array
if (lastIndex - pivotPos > 1) {
    threadRight = new Thread(() -> sort(array, pivotPos + 1, lastIndex));
    threadRight.start();
}

if(threadRight != null){
    threadRight.join();
}
if(threadLeft != null){
    threadLeft.join();
}
```

Discussion

Surprisingly, running a multi-threaded program like this one can be very expensive on processing and increases the computation time by multiple numbers of magnitude. This happens because creating threads is very expensive on performance and this implementation does not have any limit on the amount of threads it is allowed to create. This means that it takes more time to initialize a thread than actually running the computation. The following numbers are the time taken to perform the algorithm on 10000 integers.

Time taken for the sorting without Threading is : 4606902 in Nano seconds
Time take for the sorting with Threading is : 1418544451 in Nano seconds

The test was also performed on a data set of one million integers. After 5 minutes the program was still not finished.

Conclusion

Overall, implementing a multi-threaded system is not hard. However, if not done properly it can be orders of magnitude slower than a typical sequential program. If managed, it can have very powerful applications such as video or photo rendering or web-app endpoint management (one thread per call).