Parallelized Sorting

Project Report

Submitted by :
Eti Chaudhary, CS17MTECH11029

Sorting is a very fundamental problem that finds its applications in a great many areas such as database and web based applications, search engines etc. With the advent of multithreading and parallel programming, a lot of research has been put into parallelizing the traditional (sequential) sorting algorithms as well as proposing new ones suited particularly to this goal. This project involved the implementation of two of the most popular traditional sorting algorithms, namely, merge sort and quicksort and their parallel counterparts that have been proposed in literature. These implementations were then analysed under different datasets, both targeted and random.

The report is divided into 2 sections. The initial two sections, provide a brief description of the algorithms implemented and a detailed analysis of the corresponding results. The third section, logs the strengths and weaknesses of these sorting schemes and the conclusions drawn based on the results.

1 Parallel Quicksort

Quicksort, is a very versatile sorting algorithm having complexity $O(n \log n)$ for an average case and $O(n^2)$ in the worst case for almost sorted sequences. This project included the implementation of 2 parallel sort algorithms from the available literature that have been explained in the following subsections.

1.1 Multi-threaded Quicksort using Buckets

This sorting scheme can be explained in 3 phases, namely,

- Pre-processing
- Partitioning and Write-Back
- Sorting the partitions

The **pre-processing phase** begins by partitioning the array into $t = 2^{i-1}$ partitions, where i is inputted by the user. This partitioning involves the calculations of the *initial splitters*, that are basically the terminal indices for these partitions. This is followed by selection of s randomly chosen elements from the unsorted input sequence, and sorting them. This is followed by selecting k evenly spaced elements from the s sorted ones. These k elements are termed as *intermediate splitters*.

The **partitioning and write-back phase**, is executed by threads in parallel. Each thread traverses its own initial partition and divides the elements into *buckets* depending on the intermediate partition values chosen. While, the elements are being put into their respective buckets, atomic *counts* associated with each bucket are incremented, marking the size of the final partitions. Each of these threads, then write the elements in the buckets back to the original array concurrently.

The final phase, **parallel sorting** involves each thread then sorting its own *final partition*, using sequential quicksort, thereby rendering a sorted array.

1.2 Map Sort

This sorting algorithm also shares the same basic idea of partitioning the input data and then sorting parallely. However, the partitioning and the write back processes of different threads donot interfere with each other. This is achieved by maintaining an 2D array of counts. This algorithm takes 2 parameters as inputs, the number of subsets S and number of intervals, L. The *Counts[][]* array comprises of S rows (1 per partitioning thread) and L columns(1 per interval).

The **pre-processing stage** requires selection of L randomly chosen elements from the input array, which when sorted act as the intervals in which the input data will be divided.

In the **partitioning stage**, each thread traverses through its own subset and depending upon the element value, the interval it belongs to is decided. Accordingly, the thread then increments the count value for that interval and the current thread by 1. At the end of the partitioning phase, thread start writing the input elements in another array to their respective positions. Note that during this phase as well no thread interferes with the write of any other thread. This is made sure of by calculating the indices in the following manner:

$$\sum_{j'=1}^{j-1} \sum_{i'=1}^{S} Counts[i'][j'] + \sum_{i'=1}^{i-1} Counts[i'][j]$$

The above equation calculates the starting index of writing the values for a particular thread i for a particular interval j. Once the preceding stage is over, each of the intervals can then be sorted concurrently yielding a fully sorted sequence.

1.3 Result Analysis

Both the algorithms in addition to sequential quicksort were tested against a number of inputs, both random and targeted. For random inputs, the input sizes were varies from 10^3 to 10^6 and the execution times for the 3 schemes were recorded and have been plotted in **Figure 1**. It was observed that due to limited sized data sets, sequential quicksort performed better. However, the performance gap was minute and wouldn't be observed for larger data sets. Further, the execution times of Map Sort were also seen to be closer to sequential

version as compared to parallel quicksort. This could be attributed to the lack of contention effects in Map Sort as each thread only works on its portion of the data structure free from any interference. This in contrast to parallel quicksort in which buckets and atomic count values act contribute to the higher execution time. Further, it was found that the best results were obtained when the number of threads were kept as 2 (system dependent) and for higher number, the cost of thread handling led to poorer results.

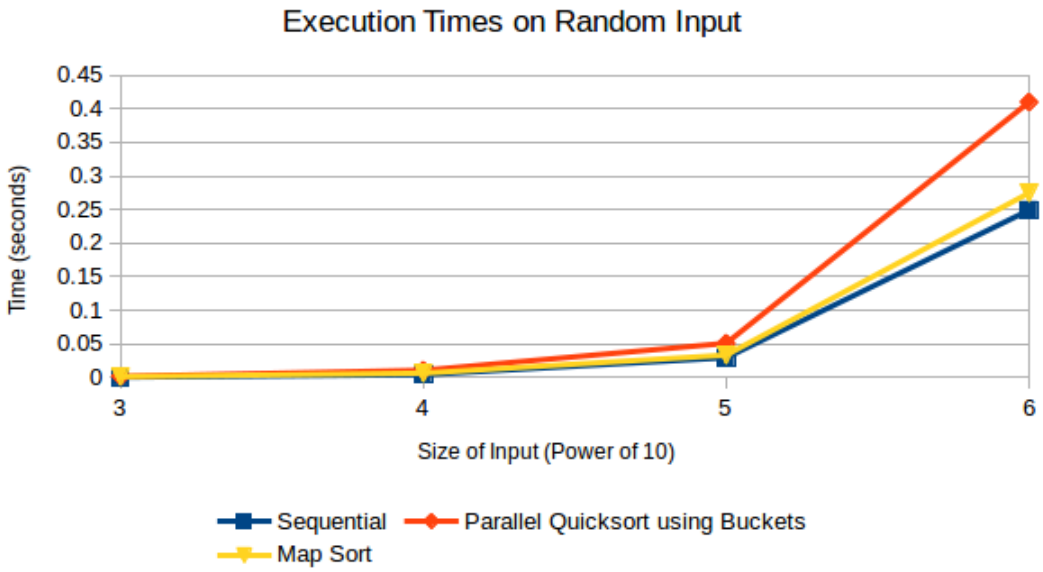

Figure 1: Execution Times of the 3 schemes on Random Inputs

Figure 2, depicts the running times of the 3 algorithms on sequential input. This being the worst case for quick sort, can very clearly be seen from the graph. The speedup obtained by parallel counterparts is massive, which clearly reap the benefits of multi-threading.

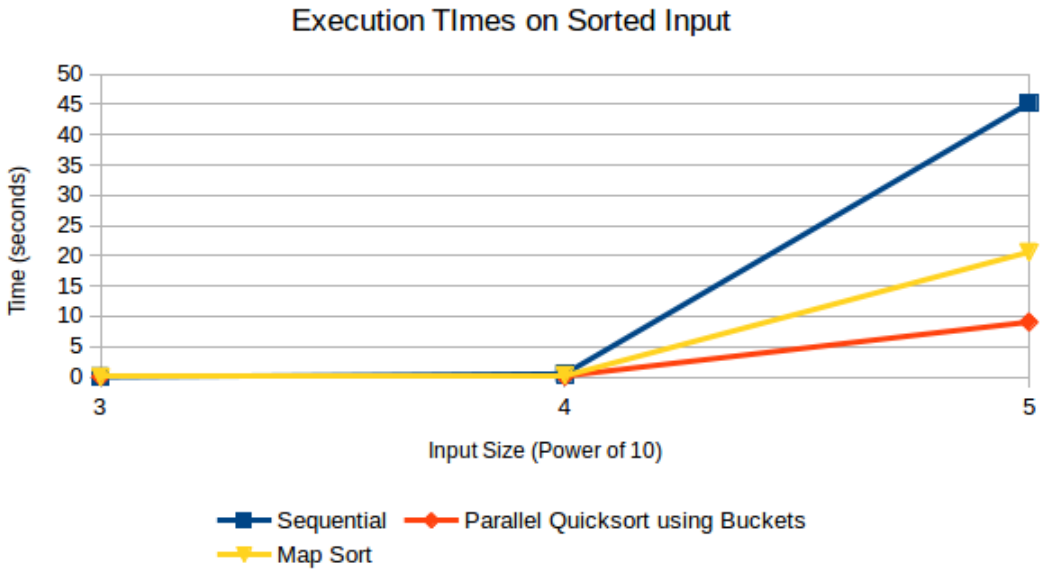

Figure 2: Execution Times of the 3 schemes on Sequential Input

The sequential case, although gives a good insight into the power of parallelism, is not a very practical case. Hence, to get a better insight on the performance of these parallel algorithms, they were tested on almost sorted data. This data comprised of the sequential elements only with some percentage of elements swapped out of their places. **Figures 3 and 4**, portray the results of these tests on the two algorithms. It can be seen that at a displacement as high as 40% for these data set sizes, the parallel and sequential algorithms perform almost equally. Further, as this percentage goes down, the parallel algorithms very clearly overtake the sequential counterpart. Further, a comparison between the two parallel algorithms reveals that parallel quicksort performs better than map sort. The reason for this improvement is low contention in case of almost sorted data, such that the extra pre-processing required for map sort now leads to its decline.

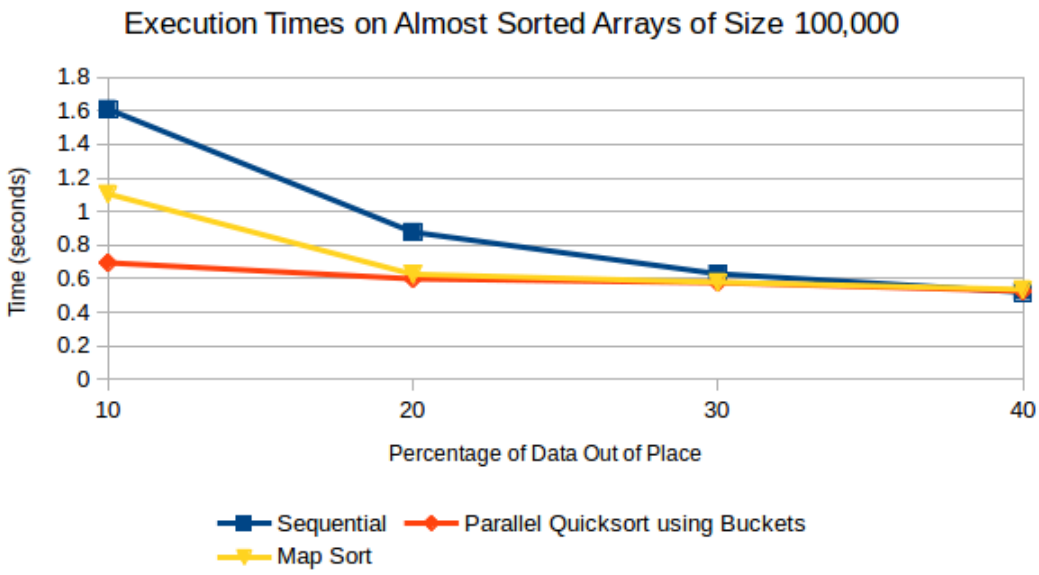

Figure 3: Execution Times of the 3 schemes on Almost Sorted Input

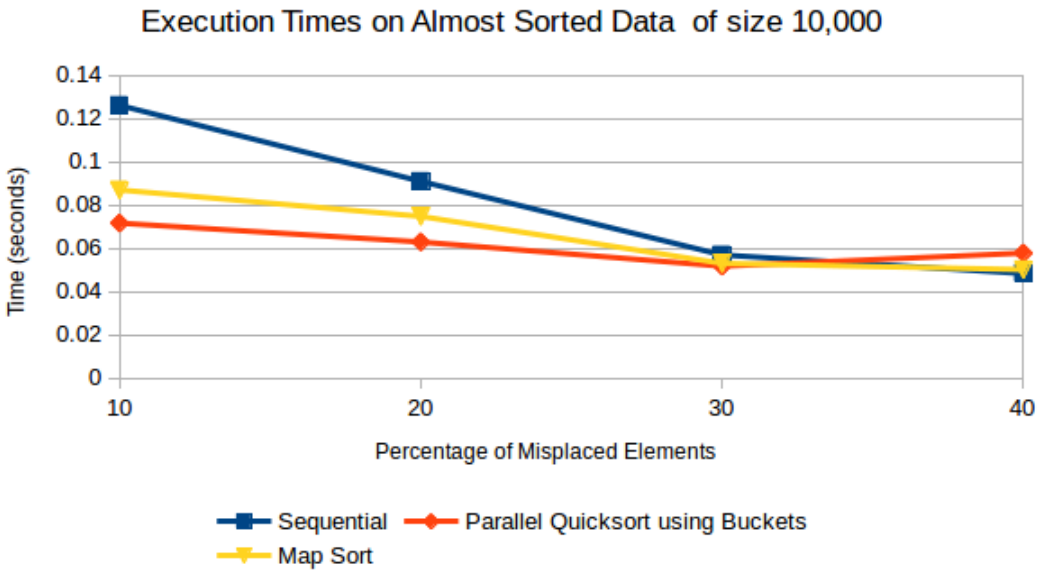

Figure 4: Execution Times of the 3 schemes on Almost Sorted Input