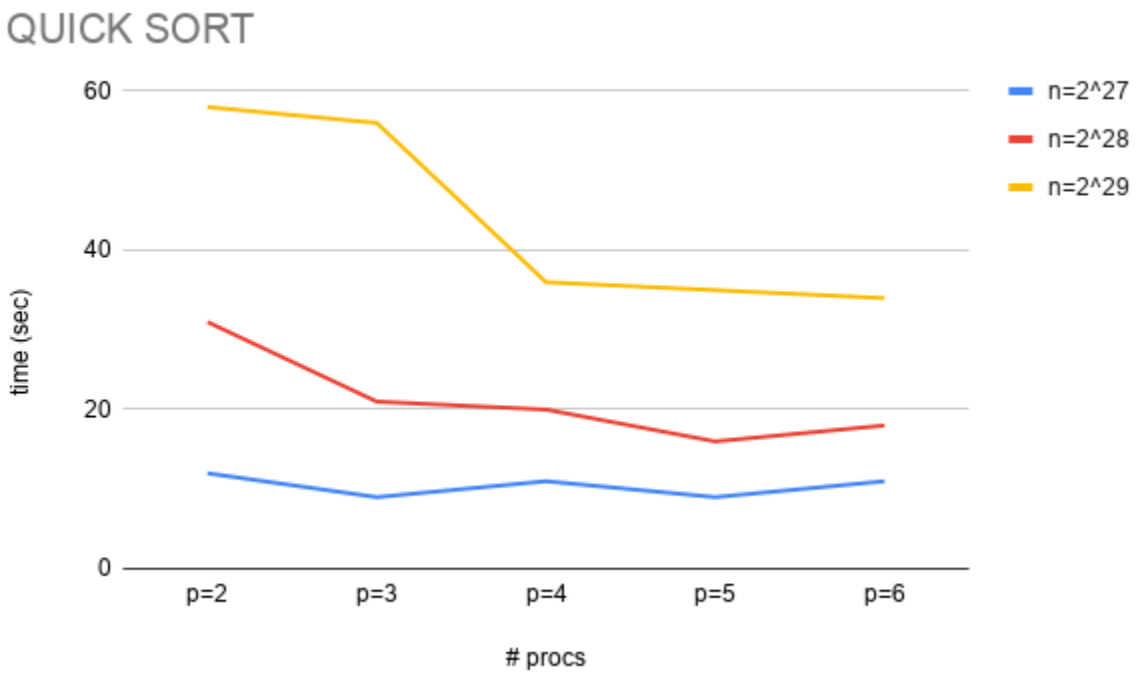
COL730 - Assignment 2
OpenMP Implementation of various Parallel Sorting Algorithms
Piyush Kansal - 2017EE30539

Let's begin by looking at some numbers from a small experiment.

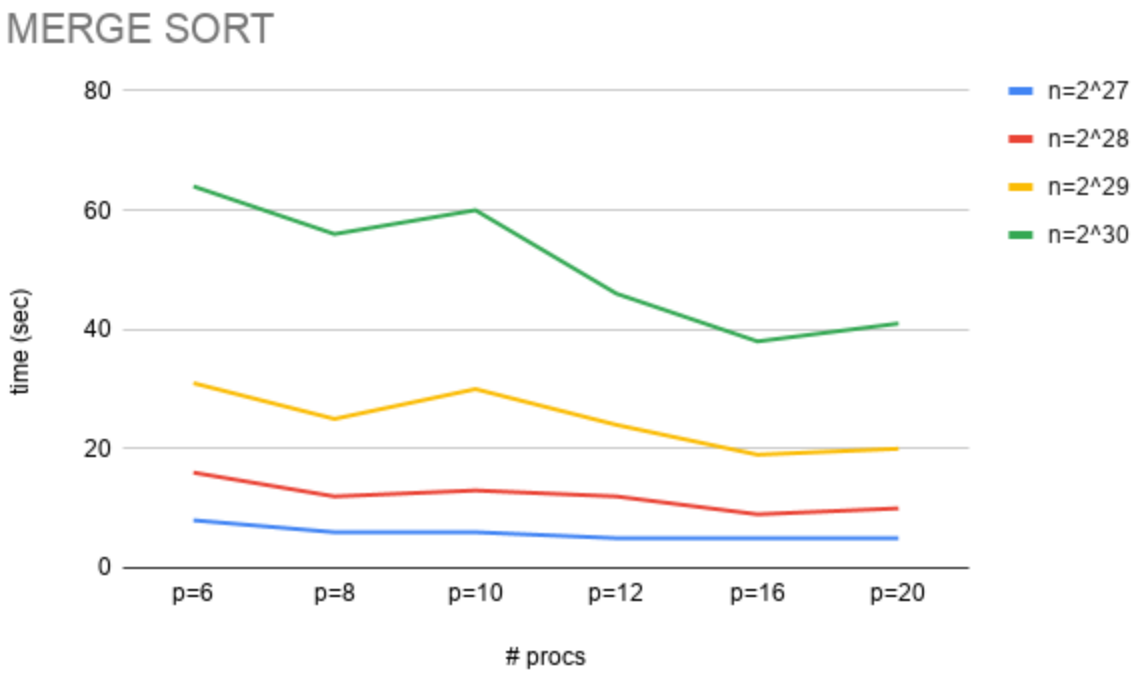
Experiment setup:
I evaluate the parallel quicksort algorithms on different numbers of records distributed across 1 to 20 processors on a single node. These algorithms use serial radix sort while sorting the data for a single processor/thread.

The goal is to show how the algorithm scales till 20 processors.

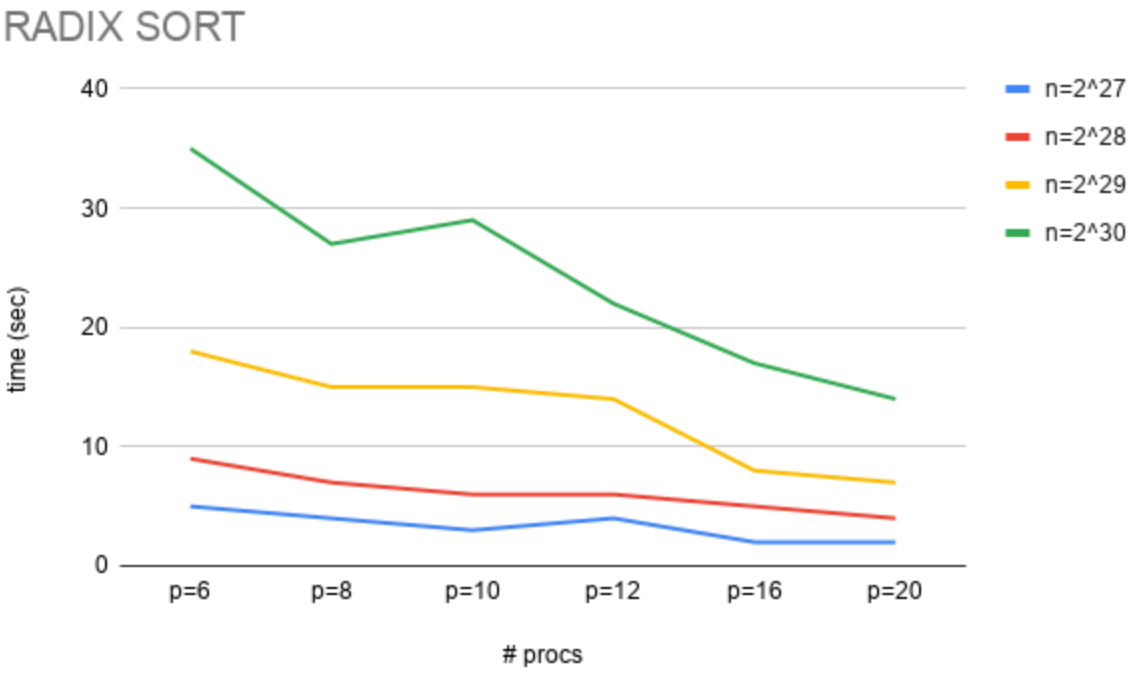
To view the exact numbers, click here.



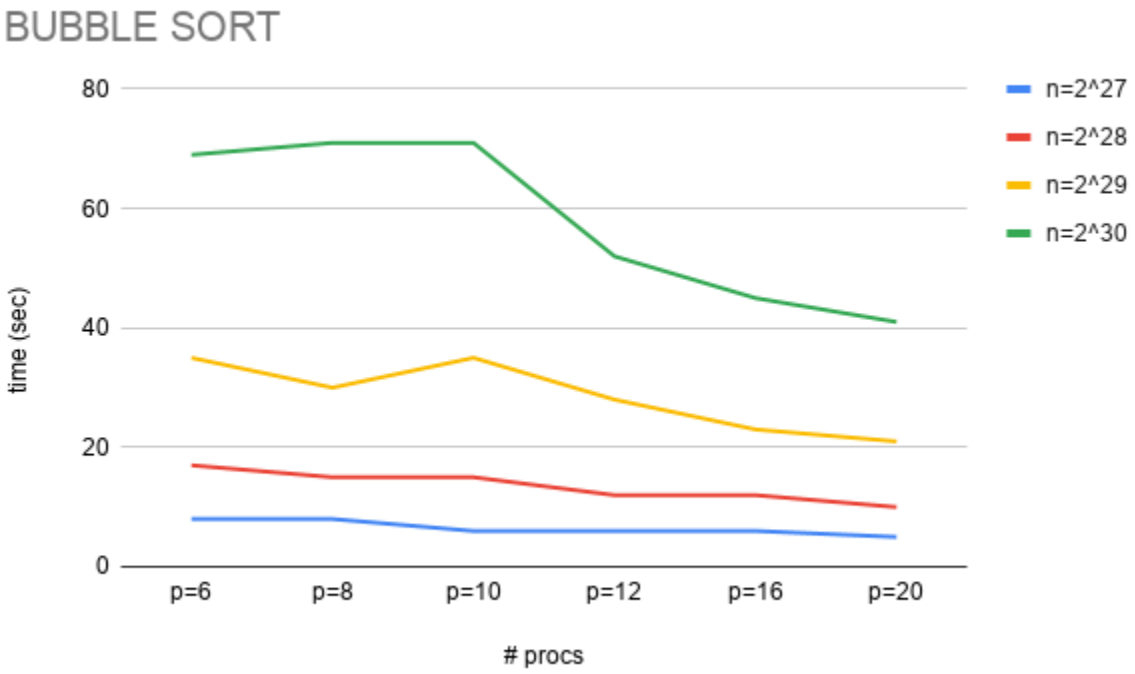
The graph reported above is from my personal laptop instead of HPC unlike all other graphs. My quicksort algorithm was slow and couldn't scale on HPC. The reason I believe for this is the use of barriers which are apparently very costly on HPC and with increasing number of processors on HPC, the cost of omp barriers subdue the benefit from more processors. However, my own laptop quicksort is definitely weekly scaling for at least 6 processors since the top most series in the graph above shows positive benefits of increasing the processors. To make the implementation even more efficient I have made sure that if a particular thread that gets less number of elements and finished earlier doesn't sit idle but is used again and again.



Merge sort is performing very well for smaller data sizes but may not scale to a very large number of processors. The reason for this being that the work of merging two sorted arrays during the end iteration falls upon just 1 or 2 processors and ultimately becomes a bottleneck for a very large number of records on a large number of processors.



Just like the message passing assignment done earlier, radix sort beats all other algorithms in this case. It is very fast like merge sort as well as scales very well (the graph above shows it to be even strongly scaling for the processor counts we are worried about). It has another benefit over the radix sort in the message passing implementation that the imbalance in the data distribution during the iterations doesn't hamper the performance because it doesn't cause any large overheads from the network effects unlike the message passing case.



To my surprise, unlike the standard performance results we know of serial bubble sort compared to other algorithms, parallel bubble sort in the shared memory setting performs very well and is at least weakly scalable and can be said as strongly scalable around the number of processors we care about. (1-20). In the message passing setting, parallel bubble sort incurs huge overheads from the network because a lot of data needs to be passed around but in shared memory even those overheads are not incurred.