Introduction

Quicksort is an excellent testbed for parallelization methods. There are two reasons for this. First, quicksort is embarrassingly parallel thus all tasks can be run in parallel with little worry regarding race conditions. Second, quicksort is a relatively well researched and optimized algorithm with many implementation that can be used as benchmarks. The aim of this report is to find how well parallel implementations of quicksort and regular sampling parallel sort (RSPS) parallelize and the speedup of this parallelization. Comparisons will be made between quicksort and RSPS to see which algorithm performs better and OpenMP and MPI to see which framework has better performance. All algorithms will be tested for strong and weak scaling.

Serial quicksort

Description

The quicksort used was as *vanilla* as possible. This was done so that all algorithms would be compared fairly, in terms of choosing pivots and recursion versus iteration. That being said sequential quicksort was not put at a disadvantage in anyway and I cannot think of a way to improve the code other than better pivot selection and implementing it iteratively. However these two inefficiencies were left in by design to keep tests controlled. To be clear pivot selection is important as it determines where the array is split, thus intelligent pivot selection would improve the splits of the array, however intelligently picking pivots would of course come at a cost. Therefore I am unsure if all intelligent methods would be useful.

I hypothesize that this will be the slowest of all the sorts tested since it is sequential. It's runtime is known to be O(nlogn)

Implementation

The notable implementation details for this algorithm is how it selects its pivot and the insertion sort threshold. The pivot was selected as the lowest element in the given section of the array. Then partitioning was done in O(n) time. Partitioning for all parallel types of quicksort was done exactly the same, however RSPS uses a different method as detailed in its algorithm. The insertion sort threshold was chosen as 100 after some testing. Here is the code snippet of the quicksort.

```
void qs(int v[], const int &thresh, long l, long h)
{
    if (h - l < thresh)
        utils::insertionsort(v, l, h);
    else
    {
        int pivot = utils::partition(v, l, h);
        qs(v, thresh, l, pivot - 1);
        qs(v, thresh, pivot + 1, h);
    }
}
```

Validation

Validation was the same for all implementations of this algorithm. It loops through the array and check that the current element is less than the one that came before it. Here is the code snippet.

```
int is_sorted(int *arr, long size)
{
    for (int i = 1; i < size; ++i)
    {
        if (arr[i - 1] > arr[i]) return 0;
    }
    return 1;
}
```

Parallel algorithms OpenMP

Description

Both RSPS and quicksort were implemented in OpenMP. I implemented quicksort, however RSPS was acquired from https://github.com/Fitzpasd/Parallel-sort-by-regular-sampling/.

I hypothesize that this quicksort will be faster than sequential and mostly similar to the MPI quicksort. Similarly this PSRS will be similar to the MPI implementation, however since this sort is specifically designed for parallelism I hypothesize that it will be be considerably faster than any quicksort implementation. I hypothesize that both will have a best case runtime of O(n/p logn) where p is the number of processors. This is because quicksort is O(n logn) and since both of these algorithms use quicksort then with no overhead they would sort in O((n logn)/p).

Implementation

I implemented three different types of quicksort in OpenMP only two of which ended up being tested. First, the naive method I initially wrote was to create a task whenever I would have made a recursive step. This is of course inefficient as it will likely create a huge number of threads causing massive overhead. I did attempt to add in some form of thread management, however all attempted methods resulted in a slow down. The most promising attempted method was to stop creating tasks once the number of tasks exceeded the number of threads however I hypothesize that this created uneven work for the threads (because of bad pivot selection). This results in some threads being starved causing a slowdown.

The second method implemented was using sections. There was no thread control implemented here and as such this method did not work for large array sizes. I decided to abandon it since tasks seem to be better suited to this problem compared to sections.

The third method implemented used tasks and is a slightly more intelligent version of quicksort. This version divides up the array evenly and allows each process to quicksort the array individually, it then performs an O(n) merge once all the processors has completed. I hypothesize that this will perform better than the naive version since it has less overhead and divides the array ideally. However an obvious disadvantage of this method is that it needs to combine the results at the end adding in another O(n), combine this with the fact that the main thread has to wait until all the threads are finished sorting before it can merge the array means that this algorithm may take longer than the naive version.

The RSPS code, which I did not write seems to perform to the algorithms specifications. The only modifications I made to this code is changing data types to allow for longer arrays and added in a timing and validation method.

Validation

The same validation was as in sequential.

Parallel algorithms: MPI

Description

Both RSPS and quicksort were implemented in OpenMPI. I implemented quicksort, however RSPS was acquired from https://github.com/shao-xy/mpi-psrs/.

I hypothesize that both these algorithms will run similarly to their OpenMPI counterparts and that the RSPS will be faster than quicksort. However I estimate that this quicksort will be very slightly faster than the OpenMP. Estimated run times will be the same as mentioned in the OpenMP discussion section namely O(n/p logn).

Implementation

MPI quicksort was implemented the same was my *intelligent* version of OpenMP. Namely quicksort equal parts of the array on each thread and then perform an O(n) merge on the final data. This MPI version is slightly more optimized than the OpenMP version, since instead of waiting for all the threads to be completed this version merges as soon as the main thread and the first other thread are completed. Then all arrays are merged into a global array on the master thread as soon as they are received. This is a huge advantage of MPI as I do not know if there is a way to do this in OpenMP. This now means that the only bottleneck in the code code is the main thread finishing. A better way to do this may be to allow the initial finishing threads to merge their arrays and continuously merge arrays in available threads until all arrays are merged.

The RSPS code, which I did not write seems to again perform to the algorithms specifications. I made similar modification as I did to the previous PSRS code.

Validation

The same validation was as in sequential.

Hypothesis

The table below details my hypothesized speedups, with 1 having the best speedup and 5 haing the worst.

1	MPI Regular Sampling Parallel Sort
2	OMP Regular Sampling Parallel Sort
3	MPI Quicksort
4	OMP Quicksort balanced
5	OMP Quicksort naive

Table 1: Hypothesized ordering of sorts from most to least speedup

Benchmarking

Methods

Timinge was done using the wtime functions from OpenMP and MPI. The OpenMP wtime was used for timing sequential to keep timing methods consistent.

No input files were used, testing was done with a generated array using a seed of 100 each time. This ensures that all arrays are the same.

Testing was done by varying the number of threads, nodes and dataset size. Nodes varied from between 1, 2 and 4, while threads varied between 2, 4 and 8 and dataset size varied between one thousand and one hundred million. All these variations were done for each algorithm. Each test was run 20 times to obtain a fair average. Tests were done using the included bash scripts, which use loops and arguments to control for the number size of the arrays.

System Architecture

From http://hex.uct.ac.za/system.html the architecture is 64 cores on 4 CPUs running at 2300MHz with 128Gb of RAM.

Results and Discussion

In figures 1-4 and 6 it can be seen that for small array sizes (<100 000) it is not worth implementing a parallel algorithm. This is because

From figure 1 and 2 we can see that MPI had the top two performing sorts when using only two threads, with MPIs quicksort having the best speedup and MPIs RSPS having the second best speedup. To be clear for both two and four threads both of the MPI implementations not only achieved better speedup than their OpenMP counterpart, but MPIs quicksort performed better than even OpenMPs RSPS. This is interesting as it shows that for low core counts MPI is possibly more efficient than OpenMP. Another interesting factor in figure 1 is that MPIs quicksort seems to perform better than MPIs RSPS, I hypothesize that this is because RSPS is made to perform well on highly parallel systems thus quicksort is able to perform better with only two cores.

When comparing the two different OMP quicksort implementations figure 1 and 2 show that the naive implementations is clearly worse, but in figure 3 it is clearly better. Thus the balanced method performs better for two and four cores, but worse for 8 cores. This can be clearly seen in figure 6 where the speedup for the naive method is better than the balanced method at 8 cores, but the 4 core balanced is better than the 4 core naive method. Thus the balanced method does not scale as well as the naive method for higher core counts. This could be due to the overhead of the combination method when 8 arrays need to be merged, although this is not definitely the case it is the only plausible reason I can come up with.

As can be seen from figure 4 OMP quicksort is performing as expected with higher core counts getting a better speedup than lower core counts. While in figure 5 it can be seen that all algorithms achieved strong scaling since the speedup increased linearly on each core for a fixed problem size. However there are only 3 data points and as such one cannot be sure if this trend will continue for many more than 8 cores. Especially since in figure 5 MPI quicksort already seems to be slightly less than linear.

The performance on two nodes is as expected. As can be seen in figures 7 and 8, the two node versions of each algorithm performed notably worse than the single node counterparts. From figure 9 one can see that there is a large drop off in speedup between 1 and 2 nodes, however the speed between 2 and 4 nodes is relatively constant. This is likely because inter-node communication takes much longer than intra-node communication, but once a single extra node is added any extra nodes do not severely impact performance. One would of course want to avoid using multiple nodes if possible, but if high thread counts are required this is often impossible.

Conclusions

My hypotheses are never entirely correct my hypothesised ordering from best to worst speedup is seen in table 1. The closest is seen in figure 3 with 8 threads, where the only incorrect ordering is OMP balanced sort. In both figure 2 and 3 it can be seen that MPI is simply better than OMP for small thread counts. It is so much better that MPIs quicksort performs better than OMPs RSPS, which is of course highly unexpected.

The most surprising finding is that the balanced OpenMP quicksort only performed worse than the naive quicksort at 8 threads, but better otherwise. Other notable findings are that MPI is better than OpenMP for smaller thread counts, to the point where MPIs quicksort was faster than OpenMPs, along with the order of best to worst speedup was never consistent among differing thread counts. Finally all algorithms seem to have achieved strong scaling, however, this cannot be stated with 100% certainty since there are not enough data points to confirm the trend.

Figures

Please note that in all graphs where array size is on the x-axis there are always 3 points for each series that seem to have an array size of 0. These values are actually 1000, 10 000 and 100 000. I left these in to show that speedup increases for large enough array sizes and for small arrays overhead often outweighs the benefits of parallelism

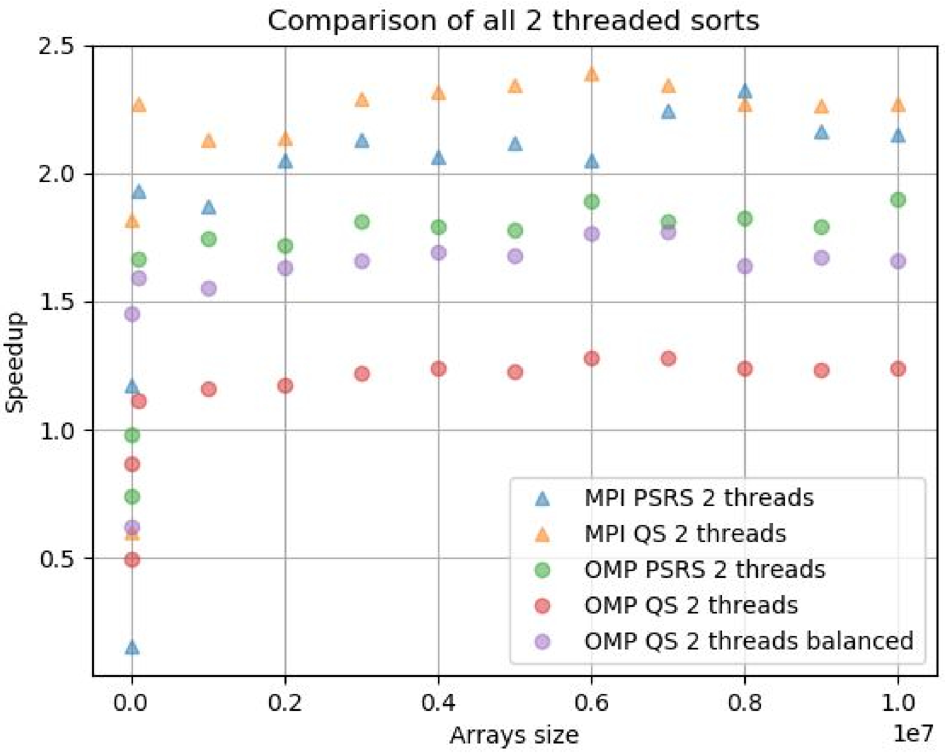

Figure 1

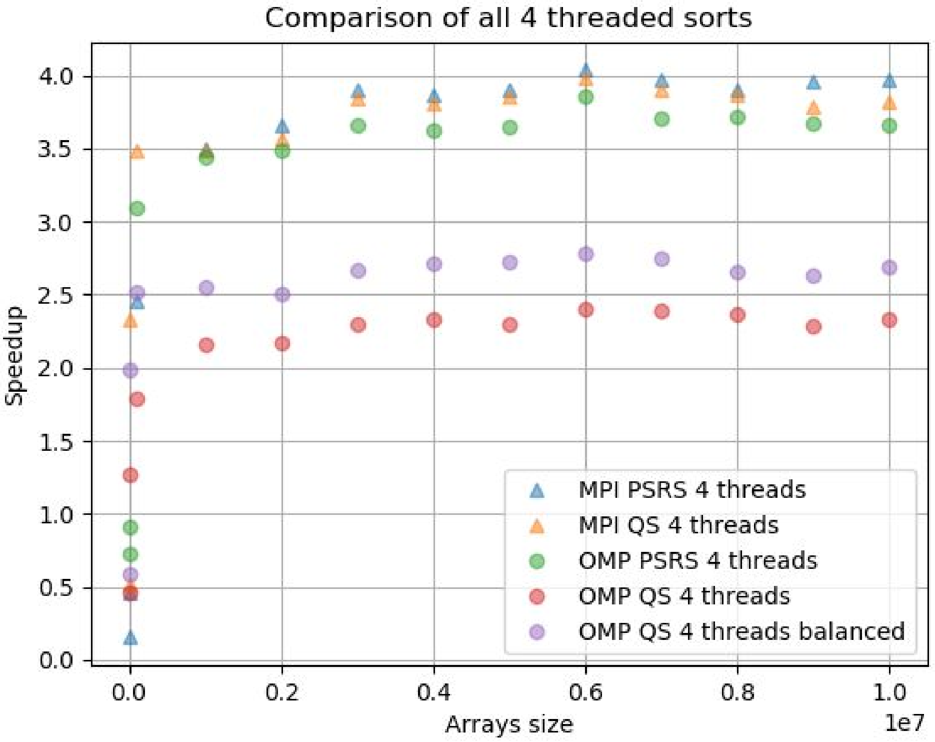

Figure 2

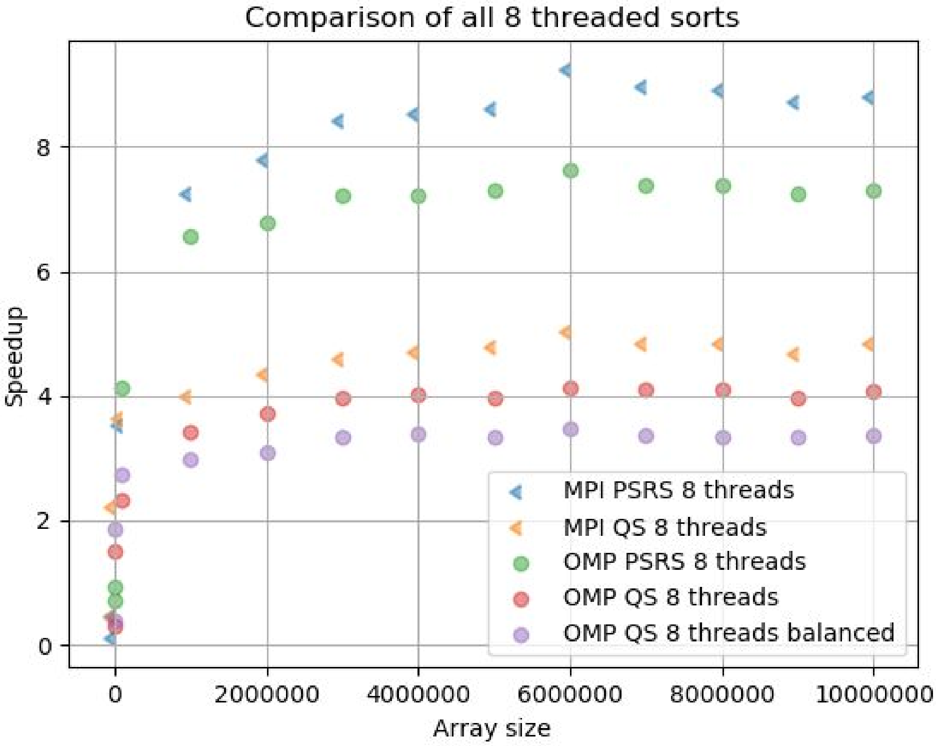

Figure 3

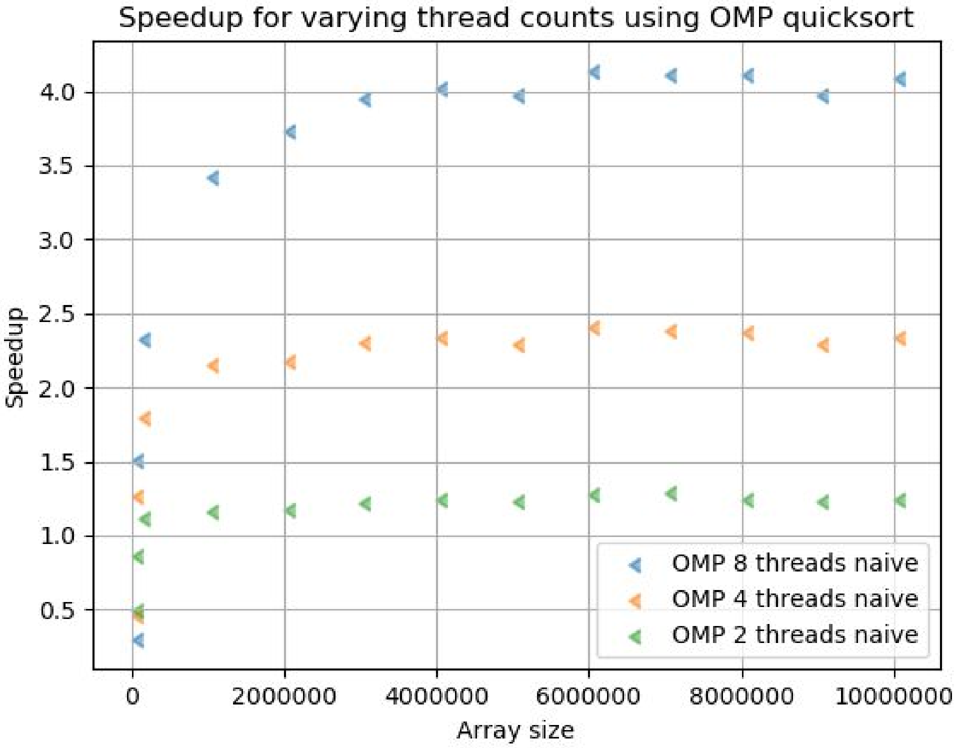

Figure 4

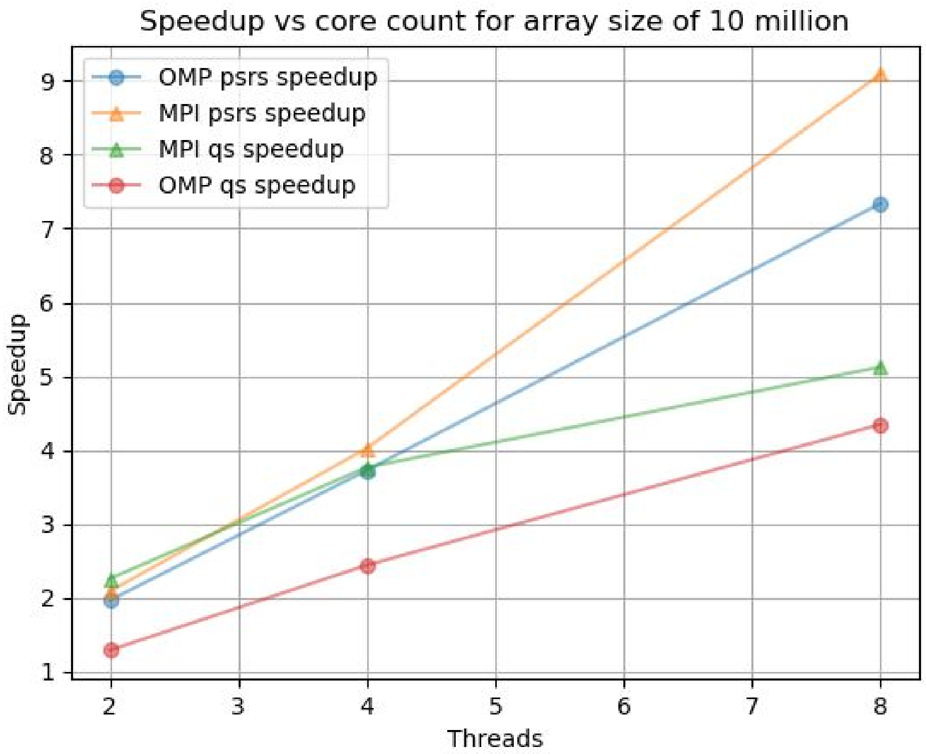

Figure 5

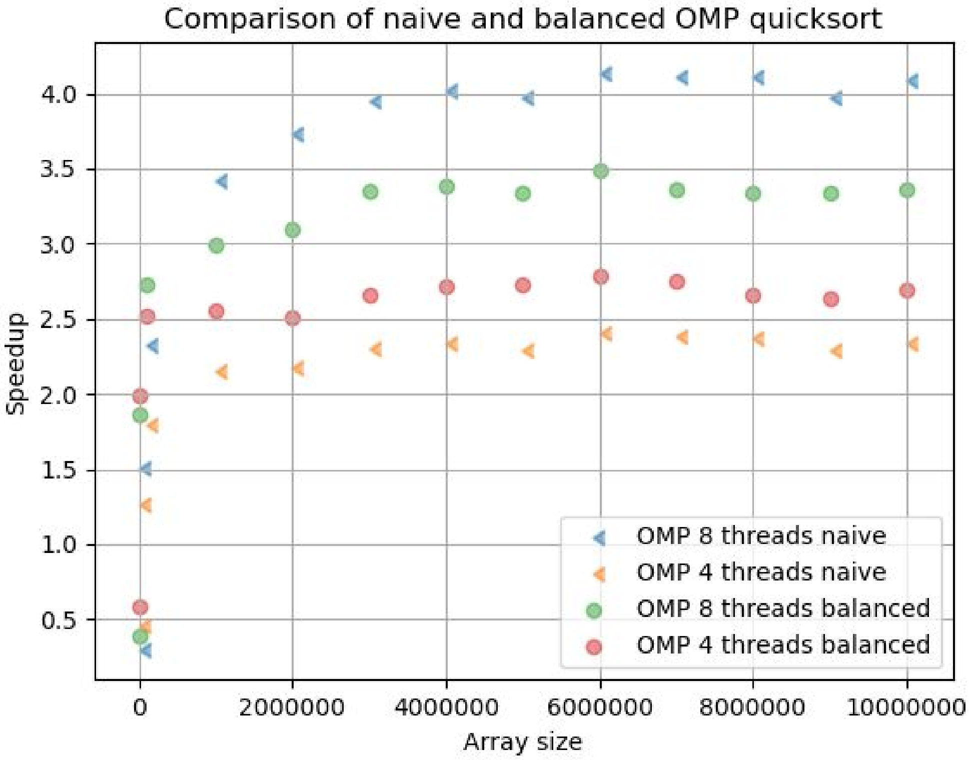

Figure 6

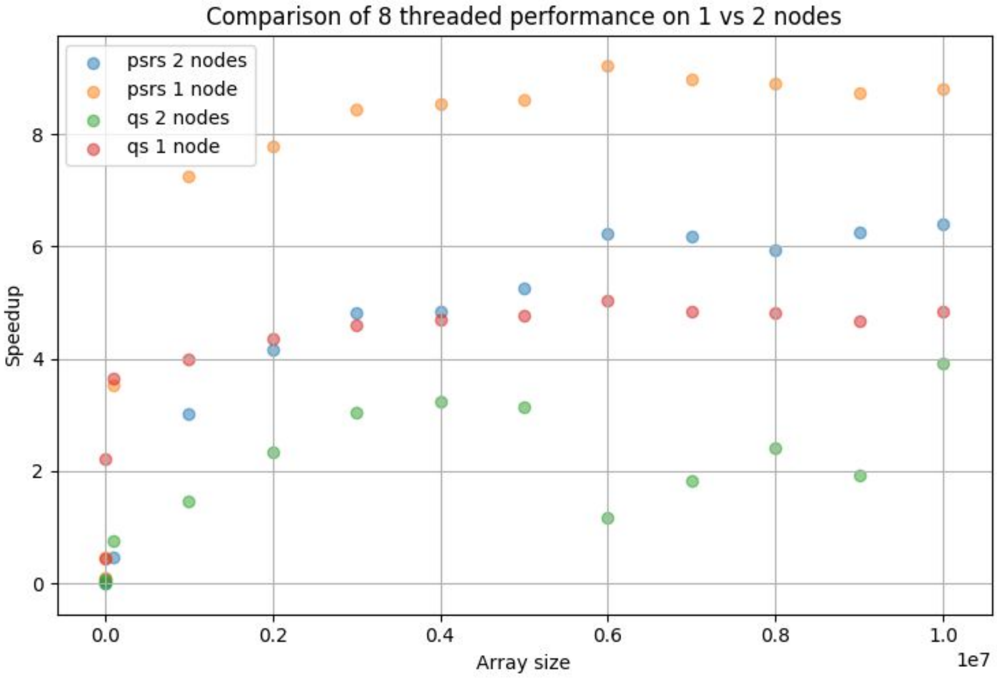

Figure 7

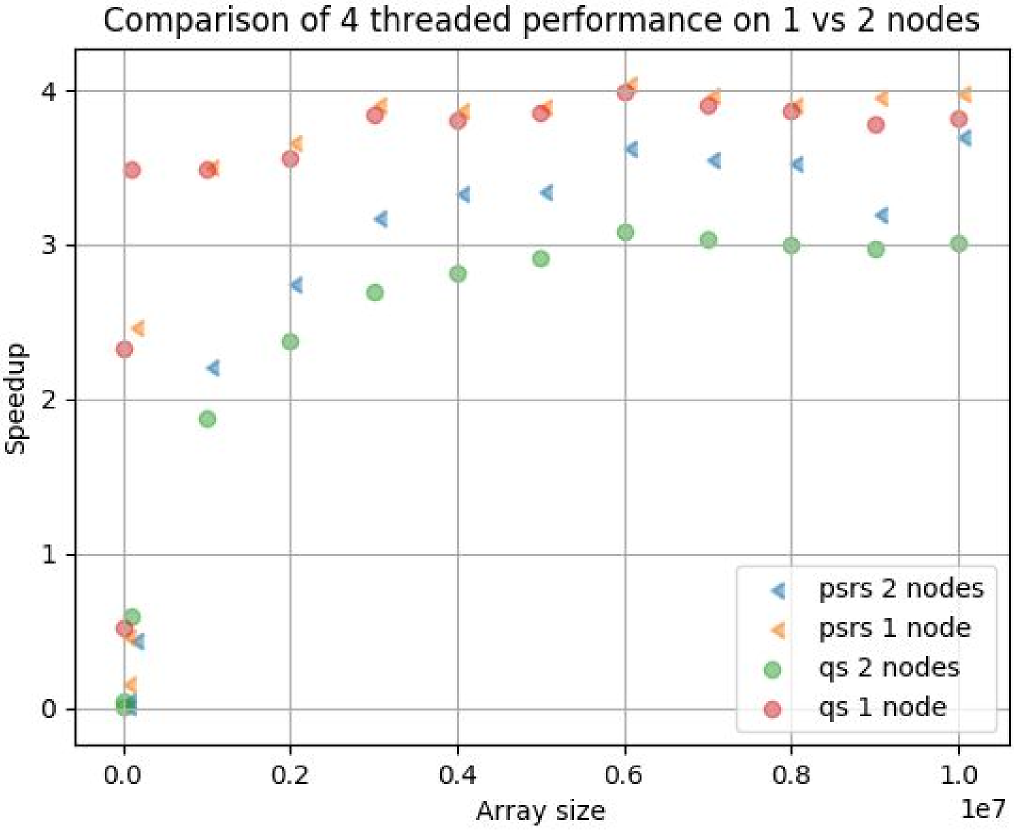

Figure 8

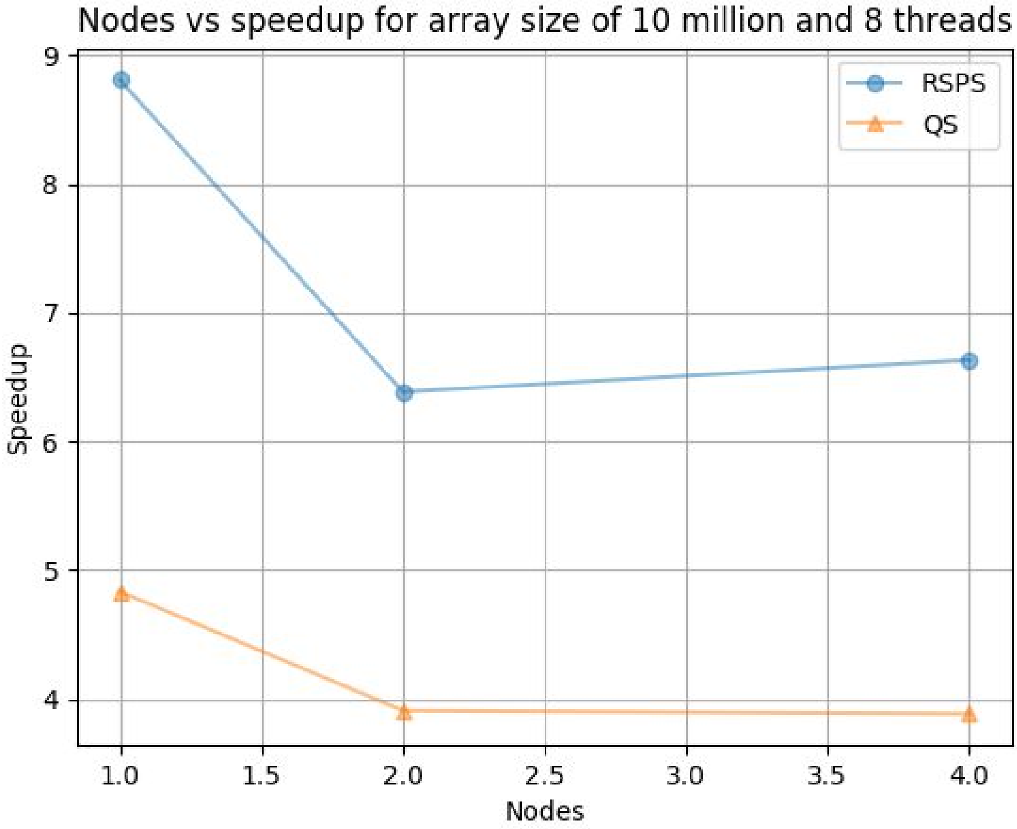
Nodes vs speedup for array size of 10 million and 8 threads
RSPS
QS
Speedup
9
8
7
6
5
4
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Nodes

Figure 9