Parallel Sorting Assignment

CSC4028Z
Liron Toledo
TLDLIR001

Introduction:

Quicksort is an extremely useful algorithm for the purpose of testing parallelization methods. This can be attributed to two core reasons. Firstly, Quicksort has been researched extensively over the years and as such has a wealth of information on how to best benchmark and optimise it. More importantly however, quicksort is known to be embarrassingly parallel, meaning that all tasks can run the algorithm in parallel with little to no chance of getting a race condition.

The two main goals of this report are to determine how well the parallel implementations of quicksort and regular sampling parallel sort (RSPS) perform and to ascertain the speedup that these implementations achieve against the serial implementation of quicksort. Quicksort and RSPS will be parallelised using two different frameworks, namely, OpenMP and MPI. Tests will be run to distinguish which algorithm performs the best as well as which parallelism framework returns better results.

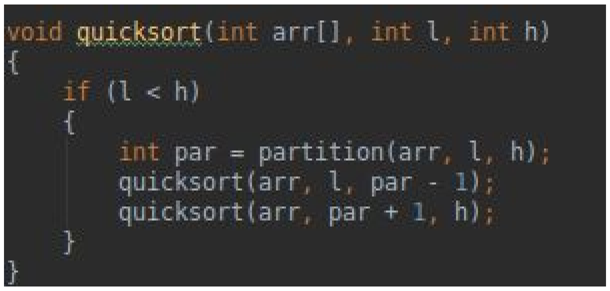
Serial Quicksort:

Description:

The serial quicksort implementation was kept as simple as possible. Serial quicksorts runtime has been recorded as O(nlogn). There are many ways to optimise serial quicksort such as intelligent pivot selection or through the implementation of tail recursion or certain iterative methods. However, in order to perform fair and controlled tests and comparisons, the algorithm was left in its vanilla form. I hypothesise that serial quicksort, due to its sequential nature, will be the slowest sorting method.

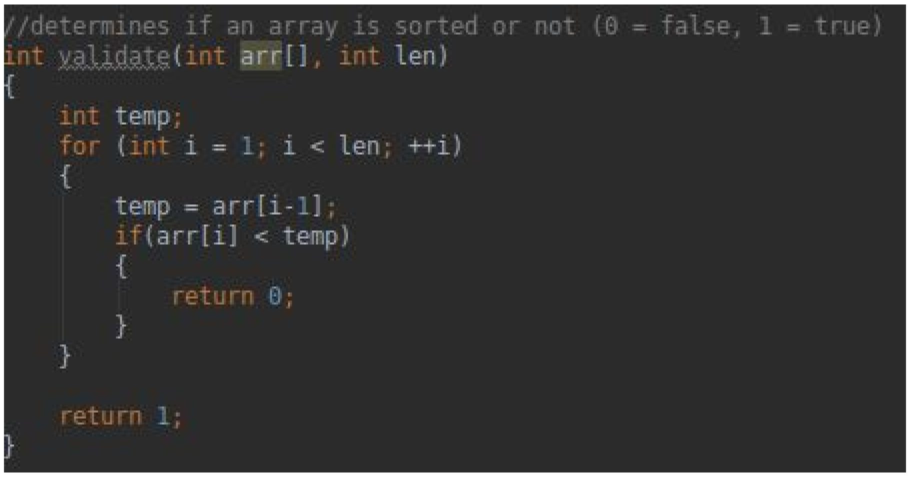
Implementation:

My implementation of serial quicksort utilises the traditional quicksort algorithm with nothing explicitly notable about it. It always picks the last element in the array as the pivot and uses recursive calls to split up the array and partition each sub-array separately. The method used for partitioning is kept constant for all all my parallel implementations of quicksort. Partitioning was recorded to take O(n) time. Below is an image of my implementations recursive calls:


```
void quicksort(int arr[], int l, int h)
{
    if (l < h)
    {
        int par = partition(arr, l, h);
        quicksort(arr, l, par - 1);
        quicksort(arr, par + 1, h);
    }
}
```

Validation:

Validation was performed in the same way for all sorting algorithm implementations. A method named validate iterates through the array and checks to see that the current element is bigger than the element that came before it. The code can be found below:


```
//determines if an array is sorted or not (0 = false, 1 = true)
int validate(int arr[], int len)
{
    int temp;
    for (int i = 1; i < len; ++i)
    {
        temp = arr[i-1];
        if(arr[i] < temp)
        {
            return 0;
        }
    }

    return 1;
}
```

Parallel algorithms: OpenMP

Description:

Both quicksort and RSPS were implemented in OpenMP. Quicksort was implemented on my own however the code for RSPS was taken from:
https://github.com/Fitzpasd/Parallel-sort-by-regular-sampling/blob/master/psrs_sort.c

I believe that both the OpenMP quicksort and RSPS implementations will perform better than serial quicksort but have similar performance to their respective MPI implementations. However, due to the fact that RSPS is an algorithm specifically designed to use parallelism, I suspect that it will outperform the parallel quicksort algorithm.

Implementation:

I attempted two different strategies to parallelize quicksort in OpenMP however only one managed to achieve speedup. The first strategy involved using OpenMP sections but because no thread control was being used, this method did not achieve speedup for large array sizes. Instead, I swapped to using tasks, which seemed to better fit the problem. Next, I ensured that only one thread could run the algorithm at a time and allowed for each task to quicksort each sub-array individually. Using this method, I managed to significant speedup against the serial implementation. However, it is still possible for this algorithm to underperform due to the overhead caused by having to merge all sub-arrays as well as the fact that that the main thread has wait until all tasks are finished sorting before it can merge the array.

The RSPS code found online performs the algorithm effectively. However, due to the fact that it is not my code, i cannot comment to much on its implementation. My modifications to the code involved adding my own testing and validation methods.

Validation:
Validation was done the same way as in the serial implementation

Parallel algorithms: MPI

Description:
Both quicksort and RSPS were additionally implemented in MPI. However, neither were implemented by myself. The MPI quicksort code was sourced from http://monismith.info/cs599/examples/quicksortMPI.c and the MPI RSPS code was taken from https://github.com/shao-xy/mpi-psrs/blob/master/PSRS.c

I believe that both MPI implementations will receive similar performance to their OpenMP counterparts. However, as mentioned earlier, due to the fact that RSPS is an algorithm specifically designed for parallelism, I suspect that i will outperform quicksort.

Implementation:
Unfortunately, because both implementations were not written by me. I have little so say about their implementation. Both implementations follow their respective algorithm specifications and execute without any problems. Once again modifications were made in order to include my own testing and validation functions.

Validation:
Validation was done the same way as in the serial implementation

Benchmarking:

Methods:

Arrays were generated inside the code using a random number generator. Numbers generated range from zero to two million. However, should it be required, a method was created to generate a file populated with random numbers given a size of your choosing as well as a method to populate an array from a file.

Both quicksort and RSPS were timed using the w_time functions from OpenMP and MPI. In order to keep timing methods consistent, the serial quicksort implementation was timed using the OpenMP w_time.

A number of tests were performed by altering the number of threads and nodes used as well as varying the array dataset size. For all parallel implementations, tests were performed on 6 different array dataset sizes ranging from ten thousand to two million. Additionally, tests were performed on 2, 4 and 8 threads. For the MPI implementations specifically, tests were also performed for 1, 2 and 4 nodes with 2, 4 and 8 threads used for each node. Each test was run 10 times to receive a fair average. The number of threads and cores was controlled through the use of multiple different bash scripts.

System Architecture:

All tests were performed on the hpc (curie) cluster. According to what was found on http://hex.uct.ac.za/system.html. The system architecture has a total of 64 cores with 4 CPUs (16 cores per CPU). The CPUs used are AMD Opteron 6376. The system runs at 2300MHz with 128GB of RAM.

Results and Discussion:

Figures 1-3 compare how the number of threads being used effects speedup. We can see in figures 1-3, that both implementations of RSPS vastly outperformed both quicksort implementations on all threads. The OpenMP RSPS out performed the MPI RSPS for two threads but was bested by its MPI counterpart for both 4 and 8 threads. However, it is worth it to note OpenMP RSPS did receive slightly better speedup for the array size of 2 million on 8 threads which is surprising considering MPI RSPS beat it for every other data size. The two implementations of parallel quicksort achieved very similar performance to each other. OpenMP quicksort performed marginally better on two and 8 threads but was bested by MPI quicksort on 4 threads.

Figures 4-6 compare the speedup achieved by different threads on the same node. Note that only the MPI algorithms were tested on separate nodes. Not surprisingly, given figures 1-3, RSPS mostly outperformed quicksort across all nodes. For both 1 and 4 nodes, RSPS using 4 and 8 threads got significantly better speedup compared to the other algorithms. Interestingly though, RSPS using 2 threads performed the worst for 2 and 4 nodes and second worst for 1 node. Another interesting result is that quicksort on 8 threads performed far better than the other algorithms for 2 nodes. However, for the most part quicksort across all threads and nodes achieved very similar speedup.

From figures 1-6 we can see most of these parallel implementations perform relatively poorly for smaller dataset size (<100 000). The only exception being the MPI implementation of RSPS which seems to consistently beat the other algorithms for smaller datasets (such as 100 000). I hypothesise this is because the parallelism overhead can cause the algorithms to underperform for smaller data sizes.

Finally, in Figure 7 we can witness a comparison between algorithms running on different amounts of nodes. RSPS once again outperforms the other algorithms with good speedup across all nodes and threads except for RSPS with 2 nodes with got worse speedup for 8 threads. Quicksort with 1 and 4 nodes performed similarly with the 4 node variation slightly outperforming 1 node across all threads. Interestingly, quicksort with 2 nodes not only outperformed the 1 and 4 node variations but actually achieved similar speedup to RSPS for 8 threads, this supports what we see in figure 5.

Conclusion:

In conclusion, my hypotheses and assumptions were not entirely correct. I was correct in assuming that RSPS would perform the best and that quicksort's OpenMP and MPI implementations would get similar speedup but was wrong in thinking that would also be the case for RSPS. MPI RSPS got the best results overall when using 4 or 8 threads whilst OpenMP RSPS performed better for 2 threads. MPI RSPS consistently beat out both its OpenMP implementation as well as both parallel quicksort algorithms for higher thread and node counts. I Believe this is because RSPS was designed to perform better on highly parallel systems.

The most surprising thing found from the data is that MPI quicksort on 2 nodes and 8 threads returned results on par if not better than RSPS for higher data sizes. Additionally, MPI was found to be better than OpenMP for dealing with smaller data sizes. Finally, I believe that all parallel algorithms achieved strong scaling but it is difficult to confirm that given that so few data points were tested.

Figures:

Please note that for all graphs, you will see three points around 0 on the x-axis. These points are actually 10 000, 50 000 and 100 000 respectively. Despite the fact that they make the graphs look rather cluttered, they are necessary to show how the algorithms perform on smaller datasets vs larger ones as well as how parallelism overhead can cause algorithms to underperform for smaller data sizes.

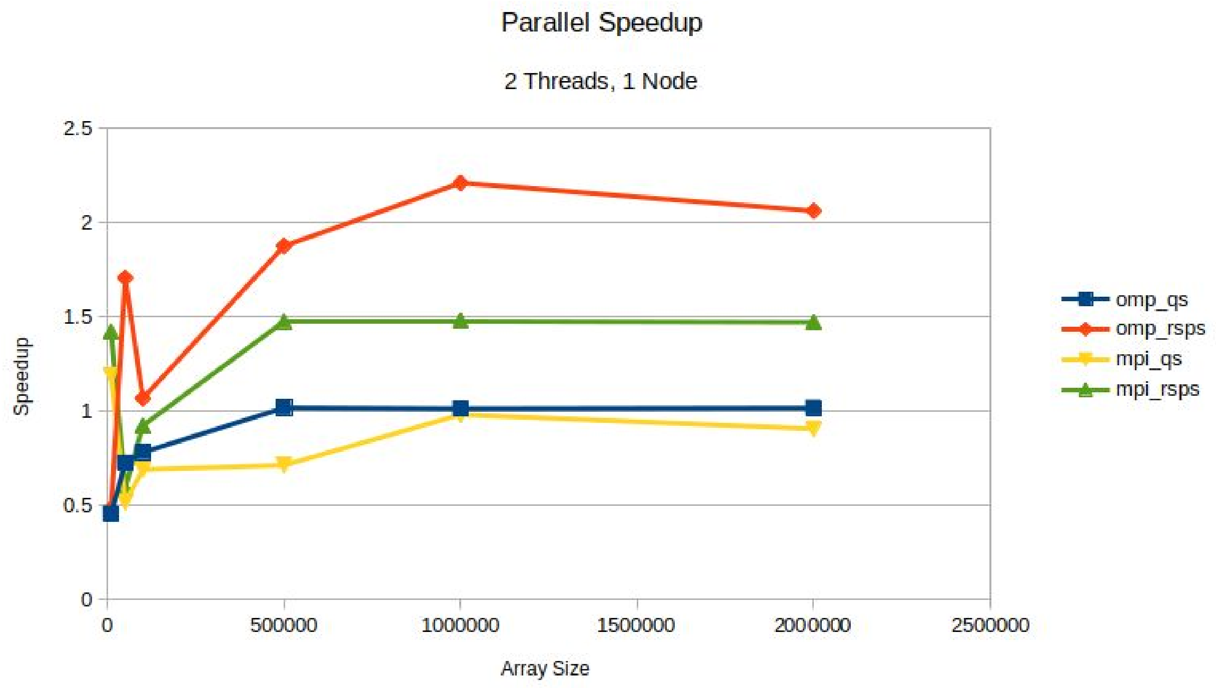


Figure 1: Parallel speedup comparison using 2 threads

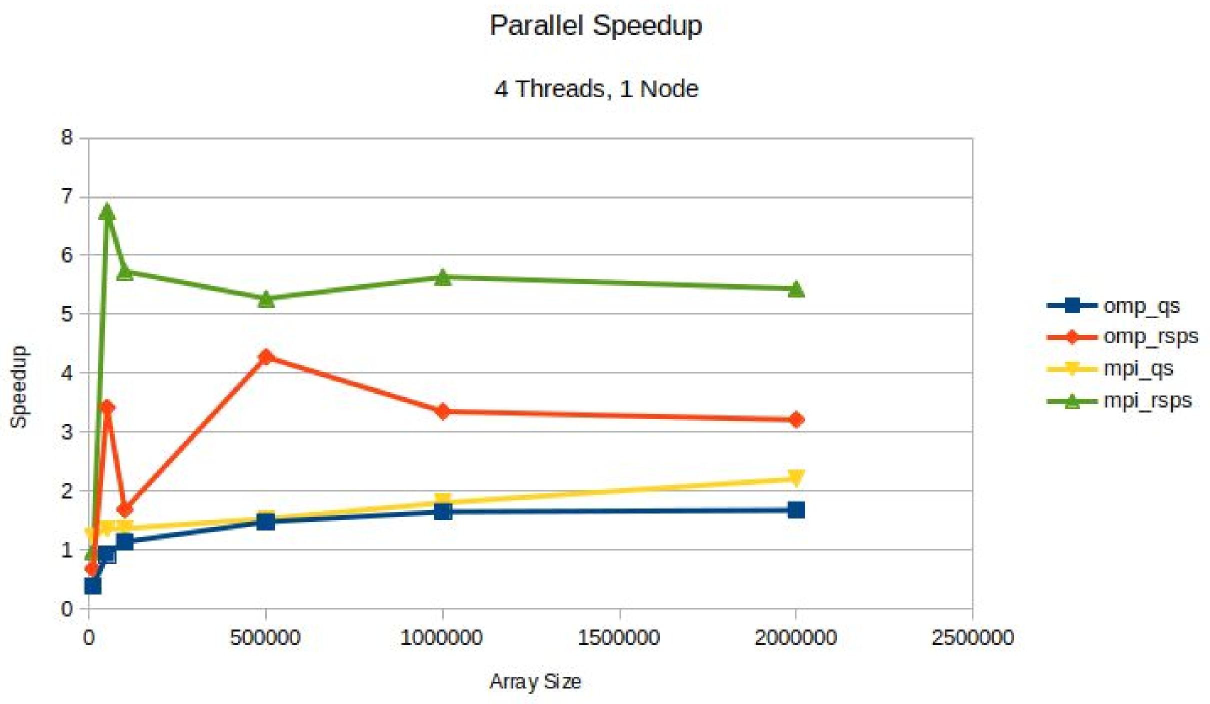


Figure 2: Parallel speedup comparison using 4 threads

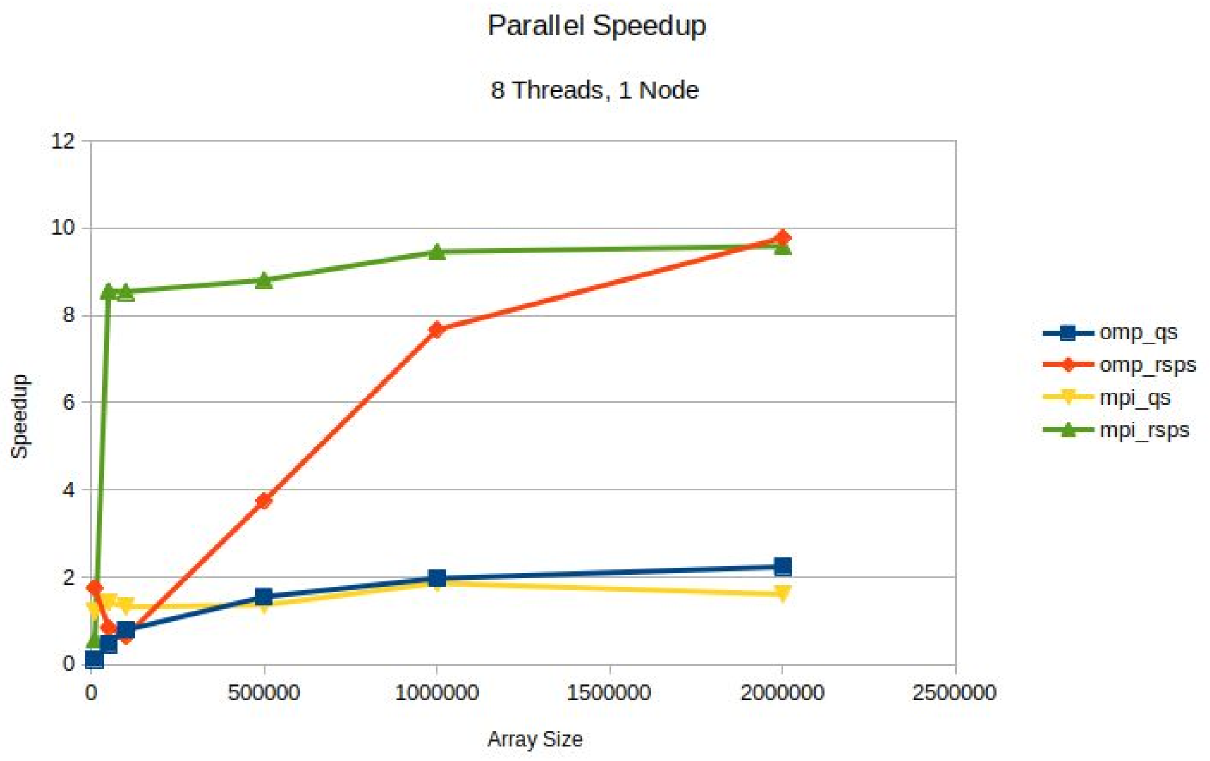


Figure 3: Parallel speedup comparison using 8 threads

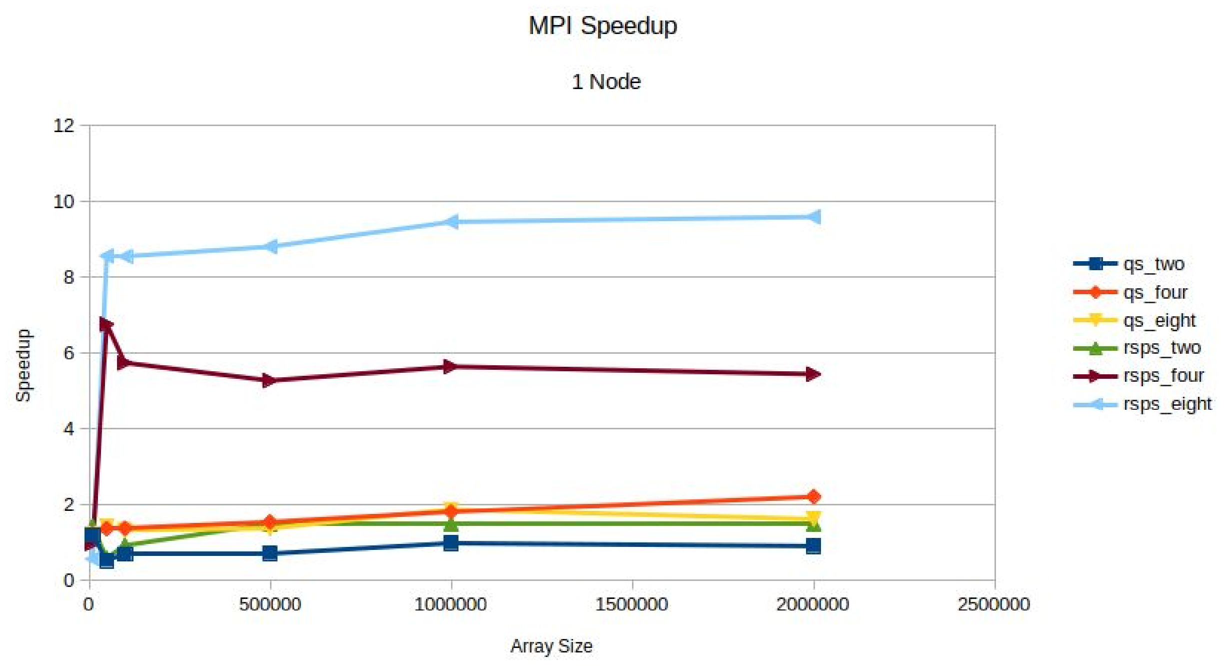


Figure 4: Parallel speedup comparison of MPI implementations on 1 node between 2, 4 and 8 threads (Note: qs_two refers to quicksort with two threads used)

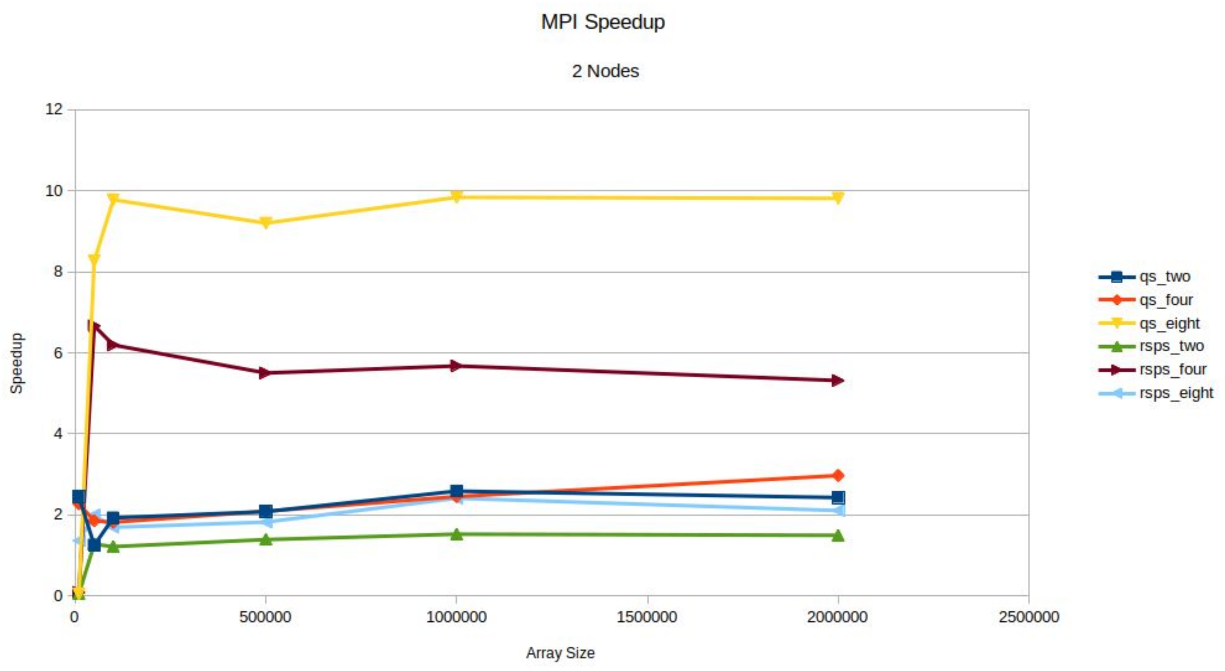


Figure 5: Parallel speedup comparison of MPI implementations on 2 nodes between 2, 4 and 8 threads (Note: qs_two refers to quicksort with two threads used)

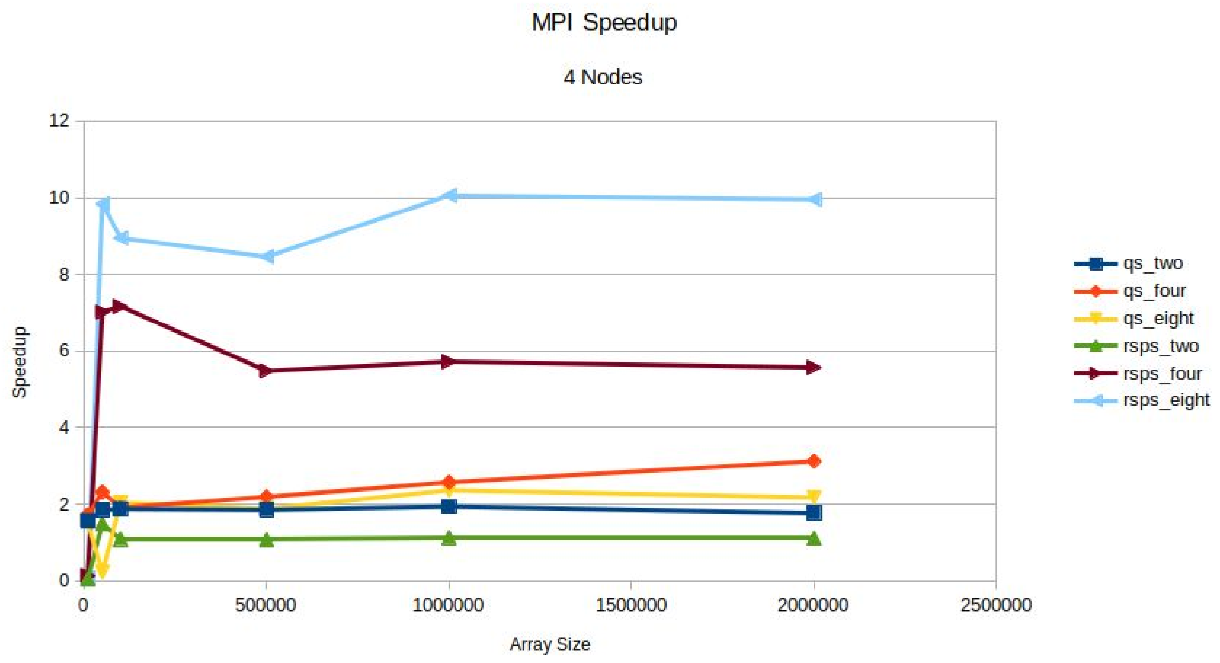


Figure 6: Parallel speedup comparison of MPI implementations on 4 nodes between 2, 4 and 8 threads (Note: qs_two refers to quicksort with two threads used)

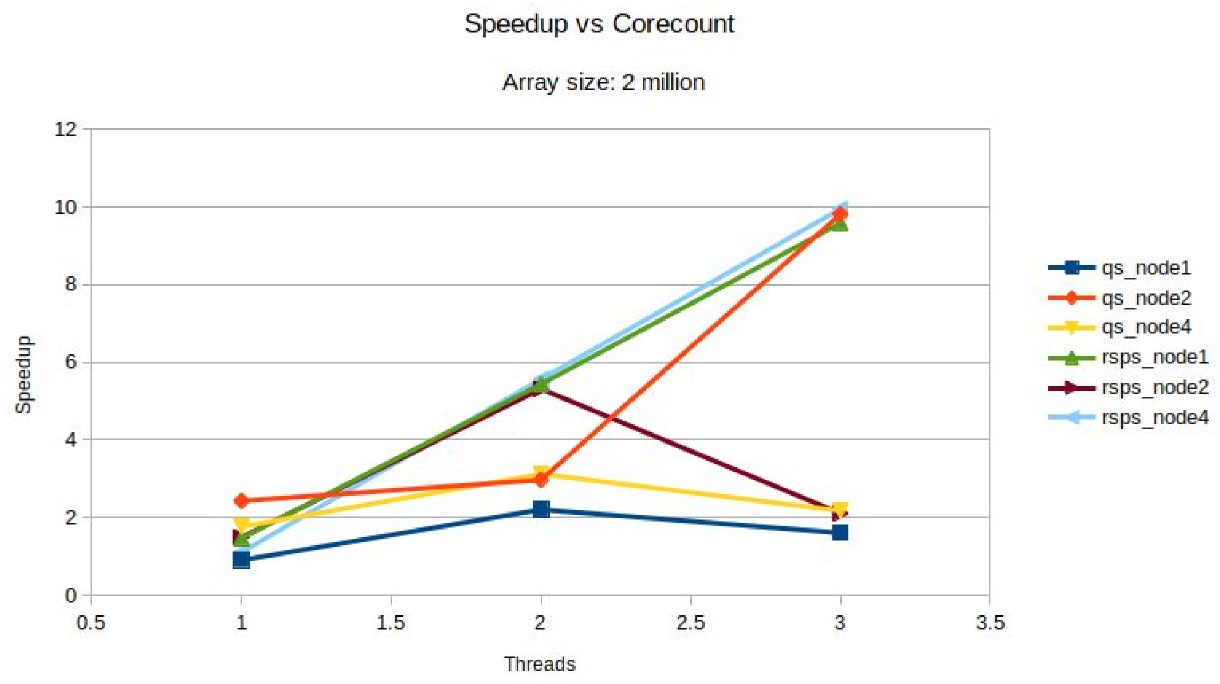


Figure 7: Parallel speedup comparison between algorithms running on different amount of nodes