Parallel Sorting

Brandon Gower-Winter (GWRBRA001)

May 31, 2019

1 Introduction

For this assignment, we were tasked with creating implementations of the Quicksort and Regular Sampling Parallel sort in both OpenMP and MPI and comparing their performances to a serial Quicksort. The Quicksort is a divide and conquer algorithm that is capable of sorting all items that have a "less-than" relationship. Figure 1 shows the pseudocode for a serial Quicksort implementation. While better than the Bubblesort or the Insertionsort, the Quicksort performs $O(nlogn)$ comparisons in the average case and $O(n^2)$ comparisons in the worst case. This means that as n gets very large so do the processing times of the Quicksort. This is unsuitable for high performance computing(HPC) where processing large datasets in a time efficient manner is a requirement. Fortunately, due to it being a divide and conquer algorithm, the Quicksort is naturally concurrent and thus highly parallelizable. By computing the partitions serially and passing the two halves of the array to other threads/processes to be further sorted improves the performance of the Quicksort to $O(logn)$ if properly implemented. Figure 2 highlights the benefit this yields for an increasing n.

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Figure 1: Pseudo-code for Serial Quicksort Algorithm (https://en.wikipedia.org/wiki/Quicksort)

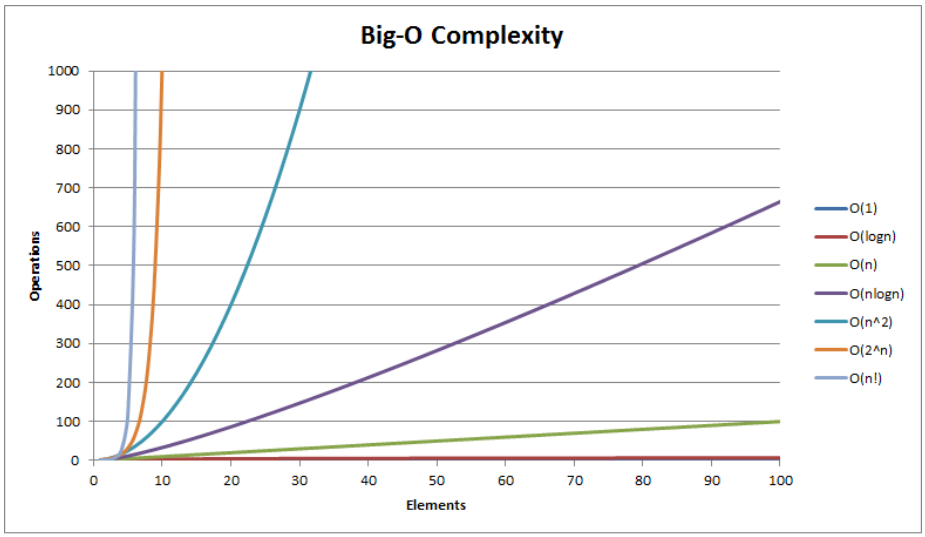

Figure 2: Line graphs showing the number of operations for various Big-O notations for an increasing n (https://www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/)

2 Serial Quicksort

2.1 Description

This Quicksort runs on on a single thread to sort n elements. The time it should take to sort elements should increase by $O(nlogn)$ for increasing n. The quicksort takes an array of items, chooses a pivot, partitions the array so that the pivot is in its correct position and then performs the quicksort on the left half of the array and the right half of the array independently, where the left half of the array = arr[lo,pivot-1] and the right half of the array = [pivot+1,hi].

2.2 Implementation

My implementation for this assignment is largely similar to figure 1. However, there are numerous optimizations one can make to improve the performance of the quicksort. A number of which have been implemented for this assignment. Firstly, for smaller $n < 50$ an insertion sort is used as it is faster. Secondly, in regular Quicksort implementations the initial pivot in the partition function is the last element in the list/sublist to be partitioned. This does little to partition the lists equally and as such I implemented a three-way comparison pivot that selects the first, last and middle element in the sublist and selects the the median value of the three. This is more useful for parallel load balancing but does improve the sequential implementation slightly. I chose to use Lomouto's partitioning scheme as it allowed me to integrate other sorts into the Quicksort. The alternative to Lomouto's scheme is Hoare's scheme and, while more efficient, does not necessarily partition pivots to their correct position and thus does not allow for the use of other sorting algorithms on parts of the array.

2.3 Validation

I implemented a optional parameter to validate sorting functions once they have been run. The validation simply ensures that all values in position i are less than or equal to the values in position j where $j = i - 1$ for all elements in the array.

3 Parallel algorithms: OpenMP

3.1 Description

OpenMP is a C/C++ library that allows a programmer to parallelize their program. In the case of the Quicksort it is capable of reducing its complexity to $O(logn)$ with some scheduling overhead. This is done by passing on recursive work to idle threads and performing the paritions in serial. It also allows for the introduction of new algorithms designed specifically for parallel computing. The regular sampled parallel sort(PSRS) is one such algorithm. It revolves around the idea of sorting chucks of data independently and then merging said data by sampling points. Shi and Schaeffer do an excellent job of describing their algorithm (See references).

3.2 Implementation

All the optimzations made to the serial Quicksort have been made to the OpenMP Quicksort. With special attention to the pivot choice, the use of a three-way pivot comparison increases the probability of efficiently load balancing ie: ensuring each thread is constantly working. I used OpenMP task directives on the recursive steps in the serial Quicksort to make the code parallel and implemented a configurable cutoff value that, instead of executing parallel code, runs the serial Quicksort. This is useful for smaller data sizes that occur later on in the algorithm as the OpenMP Quicksort, due to overhead, tends to be slower for smaller values of n.

For the PSRS, I closely followed Shi and Schaeffer's pseudo-code. Wherever they indicated parallel work, I ensured it was within an omp parallel directive. I made use of omp single to ensure that certain phases of the code could be completed. The code uses a lot of omp barrier directives as each phase requires the previous to have finished. This slows the program down significantly.

3.3 Validation

The validation of both the Quicksort and PSRS OpenMP implementations are done in exactly the same way as the serial version mentioned in section 2.3.

4 Parallel algorithms: MPI

4.1 Description

MPI is a message passing interface that allows parallel code to be executed on mulitple CPUs over a network. With MPI there are some performance overhead issues one must take into consideration. Nodes and tasks need to communicate

in order to share resources. This takes time and for small n it is unwise to use MPI because the deficit of the overhead outweighs the benefit of running in parallel. The time complexity of the Quicksort using MPI is hard to measure but it stands to reason that it should be be something similar to $O(n/plogn/p)$ where n is the number of elements and p is the number of tasks. The PSRS is a little more trickier to measure but its time complexity should be the same as the Quicksort as the most expensive operation in the PSRS is the Quicksort.

4.2 Implementation

My implementation of the Quicksort in MPI is flawed. It takes p tasks and gives them n/p data to sort. The master thread then merges those lists. This limited my MPI Quicksort to the time complexity of a serial Mergesort $O(nlogn)$. I tried to modify the solution to create sorted pivots and then use those pivots to sort the data among p tasks but quickly realized that I was essentially implementing a simpler PSRS. A better solution would've been to, like the OpenMP solution, perform the partition is serial and then send the recursive step to be completed by separate tasks.

I used an existing implementation of the PSRS as I did not have the technical knowledge to implement it efficiently/successfully. This solution follows Shi and Shaeffer's pseudo-code where all parallel work is done by the tasks/processes.

4.3 Validation

The validation of the Quicksort is done in exactly the same way as the serial version mentioned in section 2.3. The PSRS has checks put in there by the original developer. Unlike my implementation it uses MACROS to determine if the result sould be checked or not.

5 Benchmarking

5.1 Methods

To collect data I implemented a '-reps' flag and a '-avg' flag that allows a user to specify how many trials a user would like to perform and outputs the average time taken for all trials. For the sake of this assignment, I used a '-reps' value of 10 as I felt that 10 was sufficient to remove any outliers in the data. For each different test I used data sizes of 10^x where x=[1,8]. To measure time taken to complete a sort I used clock() for serial, omp_get_wtime() for OpenMP and MPI_Wtime() for MPI. All time was measured in millisceonds and rounded off to the nearest millisecond when recorded. For all OpenMP implementations I tested 2,4,8,16 threads and for MPI I tested 2-2, 2-4, 2-8, 4-4, 4-8 where the first number is the number of nodes used and the second number is the number of tasks. A mersenne twister was used to randomly generate the array to be sorted at runtime.

Once the data had been recorded it was important to measure the speedup of the parallel implementations versus the serial Quicksort. The speedup was

calculated using the following formula:

$$speedup = \frac{T_{(S)}}{T_{(P)}}$$

where S is serial and P is parallel and T is the time taken to complete the given function.

5.2 System Architecture

For this assignment we had access to the UCT HPC curie cluster. We had access to 4 nodes. Each node is a Dell PowerEdge C6145 AMD Based Rack Server. Each node has 64 cores and 128GB of RAM. We used SBATCH to schedule jobs as well as manage nodes and tasks for MPI.

5.3 Results

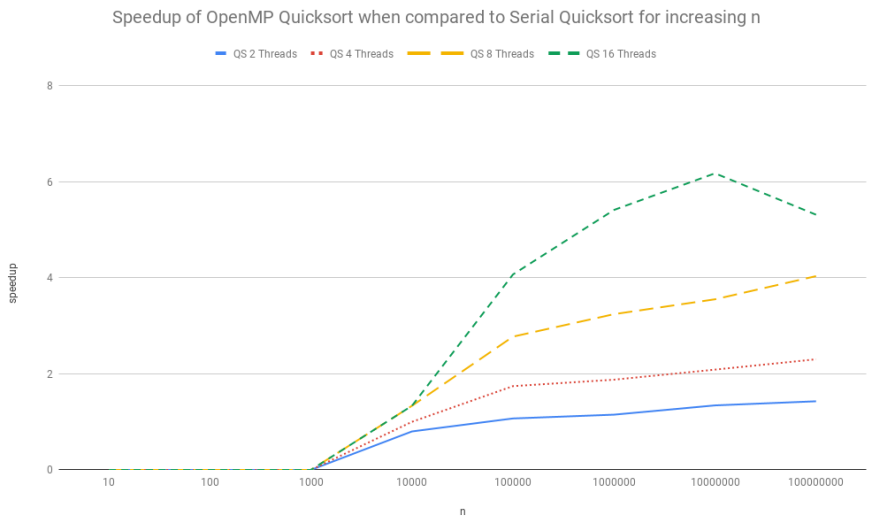

Figure 3: A line graph showing the speedup of the OpenMP Quicksort relative to the serial Quicksort for a varying number of threads.

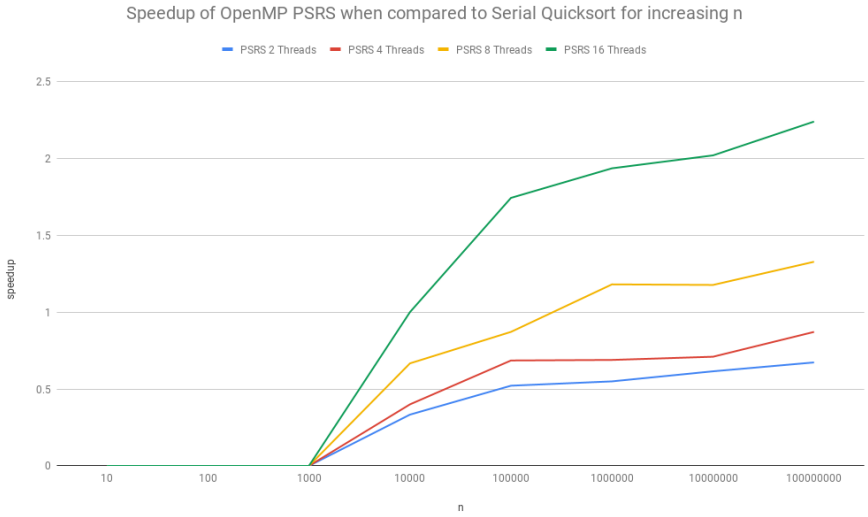

Figure 4: A line graph showing the speedup of the OpenMP PSRS relative to the serial Quicksort for a varying number of threads.

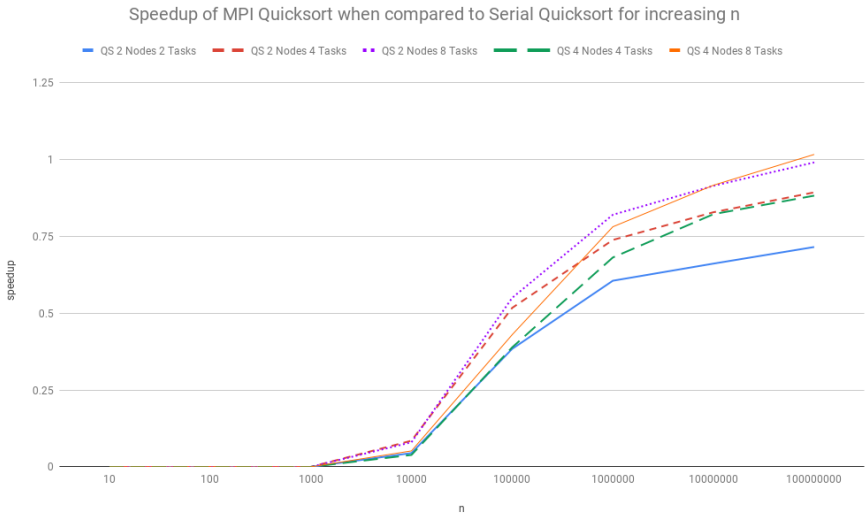

Figure 5: A line graph showing the speedup of the MPI Quicksort relative to the serial Quicksort for a varying number of nodes and tasks.

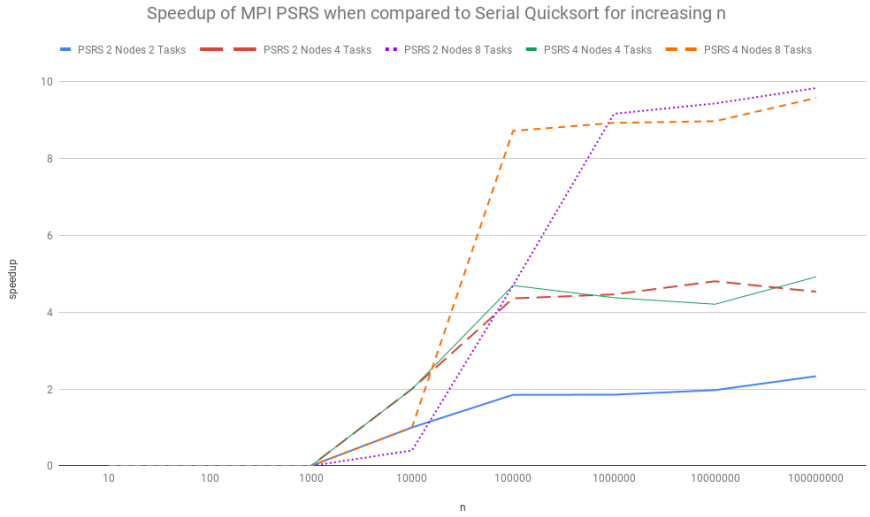

Figure 6: A line graph showing the speedup of the MPI PSRS relative to the serial Quicksort for a varying number of nodes and tasks.

5.4 Discussion

Figure 3 shows the speedup obtained by the OpenMP Quicksort implementation. This figure indicates that it is not appropriate to use the OpenMP Quicksort implementation for $n < 10000$. This is most likely due to the overhead associated with parallelization ie: It takes longer to create the threads and run the quicksort than it does to just run it sequentially. However, this figure indicates that for $n > 10000$ speedup occurs on all number of threads tested. This is especially true for large n where both the 8 thread and the 16 thread Quicksorts showed speedups of over 4 for $n = 100000000$. It is also evident from this figure that increasing the thread count greatly improves speedup.

Figure 4 shows the speedup obtained by the OpenMP PSRS implementation. This figure indicates that for $n > 100000$ speedup occurs. This figure also shows that increasing the thread count greatly increases the speedup so much so that for a thread count < 8 speedup does not occur. This seems to indicate that the PSRS heavily relies on having multiple (> 8) threads available. This makes sense because of how the PSRS divides up work equally amongst the threads rather than giving work to the first available thread in the case of the parallel Quicksort. When comparing these results to the results of figure 3, it seems that the Quicksort performs better than the PSRS in OpenMP. This may be true or it may be because there is some bottleneck in the PSRS implementation preventing it from achieving speedups like the Quicksort. However, it is clear from both figure 3 and figure 4 that using either OpenMP implementation with a thread count > 8 for $n > 1000000$ yields significant speedup over the serial Quicksort.

Figure 5 shows the speedup of the MPI Quicksort. This figure indicates no speedup for any value of n. I do not believe this to be an accurate representation of the Quicksort in MPI but rather a fault in my implementation. As mentioned

in section 4.2 my implementation of the Quicksort divides the work equally amongst some number of tasks and then merges the results together. This essentially turns the implementation into a mergesort with MPI overhead and thus possibly explains why no speedup occurred. The one thing that can be noted is that increasing the number of nodes and tasks available to MPI brings the speedup closer to one for very large n. This could possibly be an indication that increasing the number of available nodes and tasks has a large impact on the speed of the implementation as ,for very large n, the speedup was nearly one indicating that the parallel component of the algorithm executed extremely quickly.

Figure 6 shows the speedup of the PSRS MPI implementation. It also shows that for $n < 10000$ speedup does not occur. This is again most likely do due the performance overhead that comes with using MPI. However, for very large n, speedups of almost 10 occurred. This indicates that the PSRS is vastly more efficient for very large n. An interesting point of discussion is between 2 Nodes and 8 Tasks and 4 Nodes and 8 Tasks. It is clear from the figure that 2 Nodes and 8 Tasks is slightly better than 4 Nodes and 8 Tasks. This is most likely to due with the overhead that one incurs when communicating with a seperate CPU (Node) over a network as opposed to shared memory.

I do not believe it is fair to compare the MPI Quicksort to the MPI PSRS or the OpenMP Quicksort due to the reasons stipulated above. It is however fair to compare the OpenMP Quicksort (Best OpenMP implementation) and the MPI PSRS (Best MPI implementation). Whilst the OpenMP Quicksort may be easier to code, it was not able to to produce a speedup close to that of the MPI PSRS. This makes sense because OpenMP implementations are strictly limited to shared memory and one cpu while MPI implementations can use any number of cpus. This means that MPI implementations have access to a significant amount of computational power than OpenMP implementations do not. This does not mean to say that MPI is always the best solution. As evident by the figures, if you do not have access to many nodes and tasks then the MPI implementations are slow and the OpenMP implementations are better.

6 Conclusions

I was tasked with comparing the performance of a serial Quicksort to that of the Parallel Quicksort and the PSRS. By utilizing the UCT HPC Cluster, the time it took for an implementation to be correctly completed was recorded and the speedup relative to the serial Quicksort was calculated. It is clear from the results that for very large datasets ($n > 10000$)), it would be be beneficial to make use of one of the parallel implementations if performance is a major concern. For a very large n and access to only 1 cpu or only a few nodes and tasks, it is recommended that you use a OpenMP implementation of the Quicksort. For very large n and access to multiple cpus, it is recommended that you use an MPI implementation of the PSRS.

References

[1] H.Shi and J.Schaeffer, *Parallel sorting by regular sampling,Journal of Parallel and Distributed Computing,Volume 14, Issue 4,1992,361-372. https://doi.org/10.1016/0743-7315(92)90075-X.*