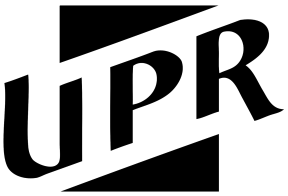
Universidad de Puerto Rico
Recinto Universitario de Mayagüez
Facultad de Ingenieria
Departmento de CIIC e INSO



Benchmarking Parallel Algorithms

Eduardo O'Neill Valerio
801155476
Adahid Galán Rivera
802142544

I. Introduction:

In this project, we visited various algorithms that we are all familiar with and we tried to formulate parallel implementations for them using various tools to see which is better suited for each one. Here we will specifically be testing with OpenMP and MPI. These are two very different tools to apply parallelism techniques. OpenMP works with threads, while MPI handles processes. They both have their advantages and disadvantages. OpenMP is a good and efficient way to perform parallel tasks that don't require much communication, while MPI is more resource intensive but provides an interface for communication between processes so that synchronisation is as cost effective as it can be. This makes it so that depending on what you are doing it may be better to go with one or the other. Sometimes it might not be better to even parallelize an algorithm at all because the overhead introduced by required communication or by the cost of creating threads might make the execution time worse than to simply apply it sequentially. The algorithms that were tested are the following:

1. QuickSort
2. Linear Search
3. MergeSort
4. Binary Search

All of these will be implemented using OpenMP and MPI and their execution times will be compared with each other and with the sequential algorithm.

II. Technical Approach:

1. QuickSort:
    - For quicksort, the parallel approach was to give each processor or thread a piece of the partition to sort recursively. In MPI every process sorts its partition and they send each other their partition to merge. In OpenMP a tree of threads with tasks are made to sort recursively each partition.

2. Linear Search:
    - In parallel there is no guarantee which thread or process will find an element first. To make sure that the element that we find is the first occurrence, each thread or process will record its first occurrence of the element if any and then a single thread or process will determine which of those found was the actual first occurrence in the search.
3. MergeSort:
    - For merge sort, the approach was to give each thread or process a given amount of data of the big array so that they can handle them in parallel. The key factor to this problem was to try to partition the data in the correct line of code. The final implementation was to use sections for every partition.
4. Binary Search:
    - In binary search the approach was to find one element from 0-1k in every iteration of the code. Because when you parallel the code, you

don't have the guarantee that the first element was found in which thread or process, the approach was to see in which process or thread was found first.

III. Experimental Settings:

The experimental settings that were used during the process of making this project were:

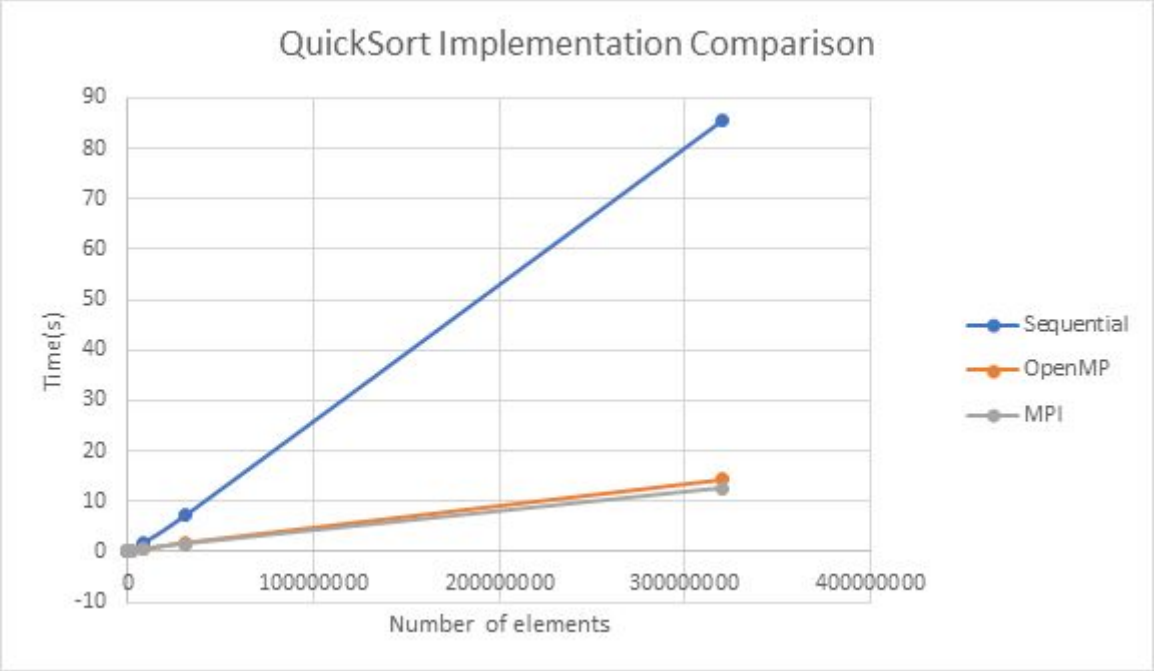
- Chameleon Cloud
- Ubuntu 16.04
- CentOS7
- OpenMP with 48 threads
- MPI with 32 Processes

IV. Results:

After developing the algorithms in the respectives parallel platforms, they were run on the Chameleon cloud clusters and this were the results:

- QuickSort:

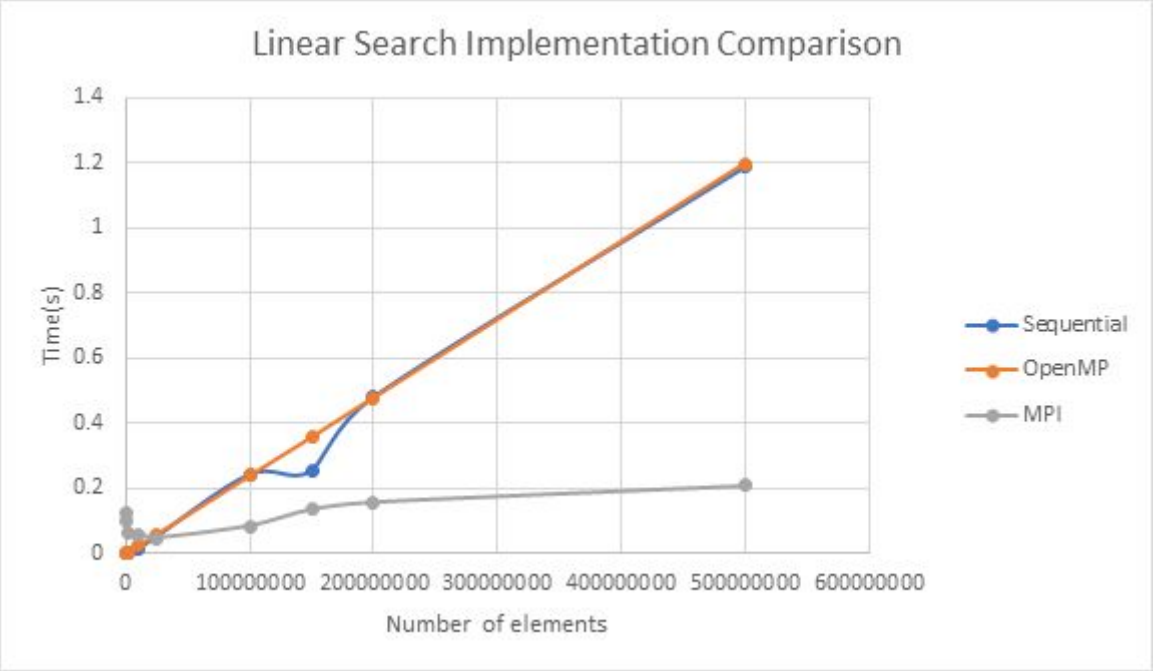
Number of elements	Sequential	OpenMP	MPI
1600	0.000175	0.003078	0.175639
16000	0.00219	0.020745	0.233255
32000	0.004761	0.04022	0.26599
336000	0.059246	0.1234	0.217766
849536	0.160625	0.16032	0.240329
1482272	0.289982	0.183476	0.355726
8377248	1.838675	0.540124	0.483124
30475648	7.200761	1.849684	1.488508
319999776	85.606167	14.44764	12.60068



Here we see that both OpenMP and MPI are significantly better than the sequential algorithm by more or less the same margin. The reason for MPI to be a bit better than OpenMP might be the cost of starting threads as opposed to having processes that are all running simultaneously.

- Linear Search:

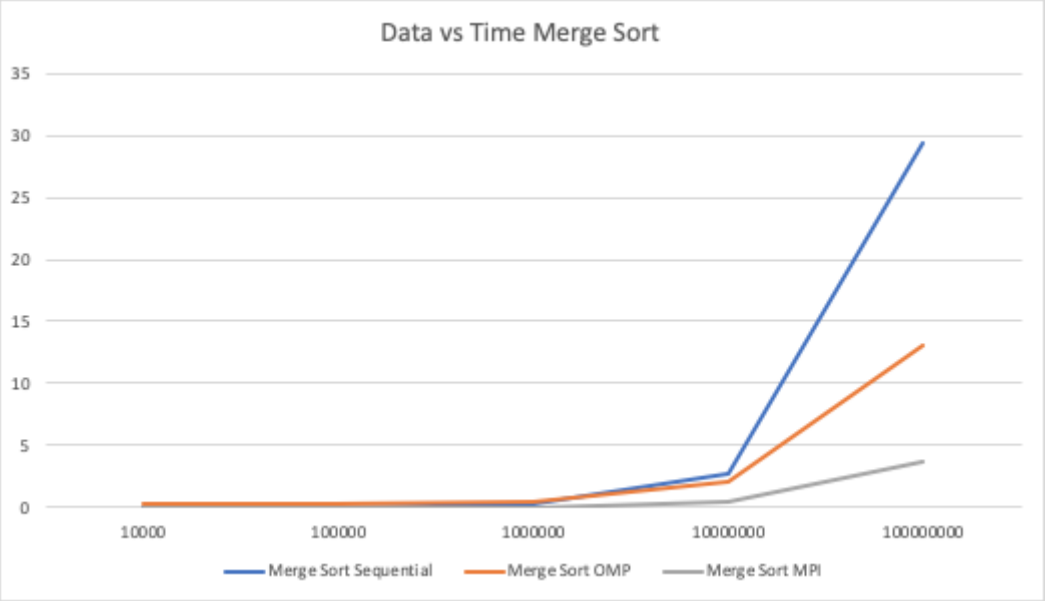
Number of elements	Sequential	OpenMP	MPI
100000	0.000168	0.00024	0.127004
500000	0.001012	0.001155	0.102952
1500000	0.002145	0.003453	0.061964
10000000	0.016158	0.023428	0.059236
25000000	0.053644	0.059082	0.047748
100000000	0.244464	0.239171	0.084939
150000000	0.255287	0.357904	0.135876
200000000	0.481257	0.478966	0.158147
500000000	1.187504	1.197864	0.208792



Here we see that MPI is significantly better than the sequential run and the OpenMP implementation. The reason is due to the fact that this algorithm requires a certain amount of synchronisation once the element was found in each thread or process, but each process can find the element faster because of the smaller chunk it has to go through. The results for the OpenMP run were not the ones expected. There might be a problem with implementation.

- MergeSort:

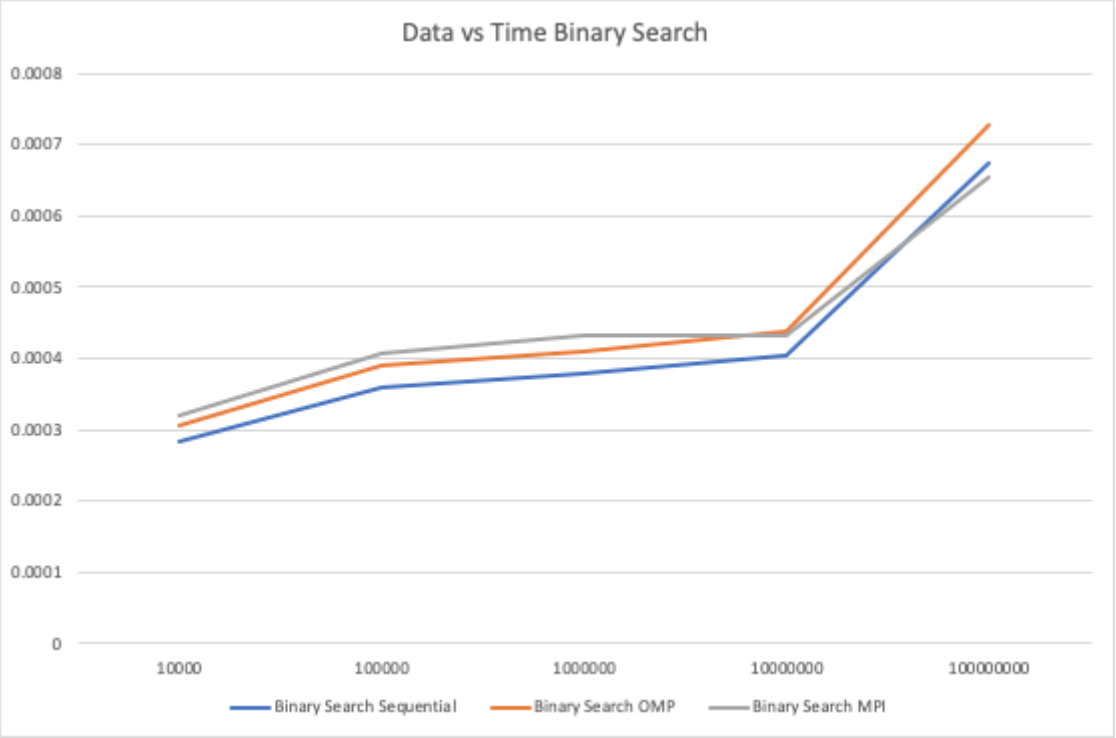
Data	Merge Sort Sequential	Merge Sort OMP	Merge Sort MPI
10000	0.00487	0.320085	0.000368
100000	0.042452	0.348198	0.005315
1000000	0.336553	0.417903	0.035822
10000000	2.724085	2.015327	0.412927
100000000	29.342124	13.073045	3.646986



As you can see through this results, all three of the different platforms perform mostly equals for array of sizes < 1 million. It is when we cross the 10 million size that the parallel algorithms tend to take advantage of their respective platforms, MPI being the winner.

- Binary Search:

Data	Binary Search Sequential	Binary Search OMP	Binary Search MPI
10000	0.000283	0.00030564	0.000320922
100000	0.00036	0.0003888	0.00040824
1000000	0.00038	0.0004104	0.00043092
10000000	0.000405	0.0004374	0.000433026
100000000	0.000673	0.00072684	0.000654156



The results on the search were a little broad. Because we did not have a certain way to sent all processes or threads to stop when the number was found, but we managed to see which one was the first to find it. More or less, this could affect the final results in regard of the performance.

V. Conclusion:

As we can see some of the results are a little mixed, but some were a success. For the sorting algorithms the most successful runs where from the MPI implementation due to the high degree of synchronization that is required for those types of algorithms. OpenMP gave good results in QuickSort and MergeSort but was ultimately less efficient than MPI due to inefficient synchronisation. And the results of the searching algorithms were not the ones expected. Turns out that the nature of the problem is too random. The missing element could be very near the beginning of the array or it could be at the end. Those resources used to parallelize the algorithm might not be necessary and may produce unwanted overhead in certain scenarios. In conclusion, sorting algorithms favor parallelization with MPI due to synchronization advantages, and searching algorithms are not really worth the cost of parallelization.