WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A Assignment 1 – Report

Wesley Earl Stander - 1056114 - Coms Hons

March 13, 2020

1 Introduction

The algorithm being studied in this paper is the k-Nearest-Neighbours algorithm (kNN). The algorithm is widely used for classification and regression in the fields of machine learning and data mining [Deng *et al.*, 2016]. The algorithm takes a set of reference points and query points and classifies the query points according to the median group of the k nearest neighbours or another heuristic in the reference set. [Zhang *et al.*, 2017]. The brute force implementation of kNN computes all distances of the reference points for a query point, sorts the computed distances and then selects the k closest reference points and uses a selection metric such as median of group to apply a group to the query point. This process is then repeated for all query points.

2 Foster's parallel algorithm design for KNN

The method involves 2 computationally expensive parts that need to occur in sequence hence 2 designs will be created. The partitioning step of the distance calculation is to partition each distance calculation into a primitive task. These primitive tasks need to broadcast their distance calculation to a single task that does the final calculation on the sum of all the individual distances. The tasks can be agglomerated into p tasks dividing the state space by p. Each agglomerated task sends it's distance to the main one and the final distance is calculated. The Mapping is linear with the p tasks and p processors. Assume log(p) communication steps as there is a binary reduction hence the worst case computational speed is $O(n/p + log(p))$.

The partitioning step of the sorting calculation is to partition each key into a primitive task that communicates with the task before it and after it. The first communication step compares every even index with the value subsequently after it and the second communication step compares every odd index with the value subsequently after it. The comparisons swap values if needed. The list is sorted after $\theta((n/2)^2)$ iterations. The tasks are agglomerated into p larger tasks that communicate with each other comparing the largest value of each larger list with the smallest value of the subsequent larger list. Hence each step is $\theta(n/p)$ and communication is constant therefore each step is $\theta(n/p+1)$. After n iterations the list is sorted hence the final computational speed is $\theta(n^2/p+n)$. Each larger task is mapped to a processor.

3 Methodology

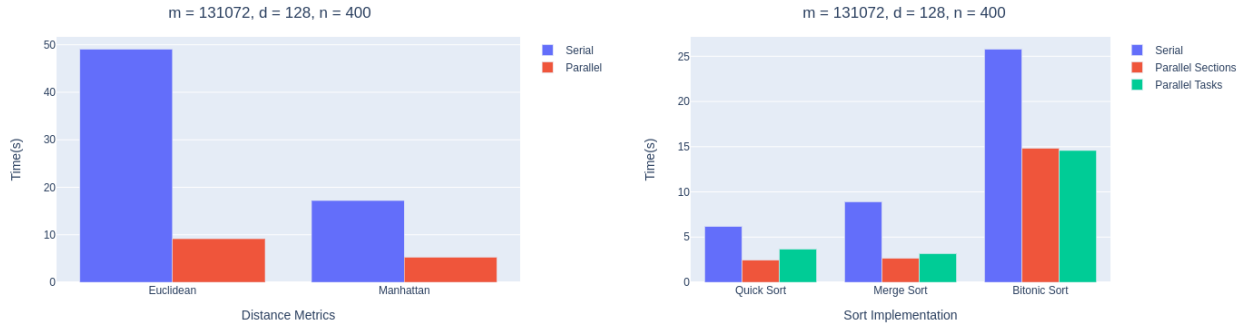
Euclidean distance and Manhattan distance were implemented for a vector of arbitrary dimensions. Quick sort, merge sort and bitonic sort were implemented. All algorithms were designed to return the time taken for completion for easy usage in a main clause and easier analysis.To calculate the Euclidean distance each point in the reference array had to

have their distance relative to the query point calculated. The distance is calculated by taking the sum of the squared distances between each dimension and square rooting that distance. Manhattan distance is calculated by taking the sum of the distances along each dimension. The computationally expensive part is proportional to the amount of dimensions and the amount of points in the reference array and hence $\theta(d*m)$ where d is the amount of dimensions of the points and m is the amount of Points in the reference array. The Quick sort algorithm uses divide and conquer to sort through array elements [Sedgewick, 1978]. It creates two arrays and recursively sorts the sub arrays. The quick sort finds the middle value as a pivot and compares all values above and below moving them to the correct place relative to the pivot point. The array is then subdivided and the two halves are sorted independently. The computationally expensive part here is the split and the sort at each split. This means that at the worst case the algorithm can take $O(n^2)$ time and is the slowing factor of the algorithm. The merge sort similarly to quick sort uses a divide and conquer method to sort array elements [Cole, 1988]. The merge sort splits the array up into 2 halves recursively until it can evaluate a swap with 2 elements and then re-merges the array cleverly. The computationally expensive part is the split and the merge hence creating a worst case complexity of $O(n\log(n))$ which is less than quick sort due to the clever merging algorithm.Bitonic merge sort is similar to merge sort but is more suitable for parallel implementations as the amount of comparisons is known in the beginning. The bitonic sort sorts the array into recursive bitonic sequences then merges the sequences into the same direction [Ionescu and Schauser, 1997]. The sort is the computationally expensive part of the code and the merge although the sort can be made faster using parallelism. The worst case complexity of bitonic sort is $O(n\log^2(n))$ which takes more time than merge sort but is possible to be coded using a for loop to make for a better parallel implementation.

The initial for loop is implemented in parallel using a for structure and the schedule(dynamic) clause was used over schedule(static) as it provided a computational speed boost for both Euclidean and Manhattan distance metrics. The quick sort has a split in the recursive function calling which is where the parallelism was placed. Both the sections construct and the task construct was placed over this split. The merge sort has a recursive split on the middle of the array which is where the parallelism was placed [Cole, 1988]. Both the sections construct and the task construct were placed over the split. The bitonic sort has a recursive split and this is where the parallelism was placed [Ionescu and Schauser, 1997]. Both the sections construct and the task construct were placed over the split. Nested parallelism was turned on for all sorts. The task construct created new tasks on each recursive split using a single construct and the sections construct added a section at each recursive split for all sorts except the bitonic sort which only split on the first layer for the sections construct.

4 Experiment

The data was represented using a struct called Point. Each data point had an integer value corresponding to the group that it belonged to, a float distance for the distance to be stored between it and another point as well as a float pointer to store an array of arbitrary length for the dimensions of the data point. The data is then generated by allocating memory to 2 arrays (reference and query) and randomizing two seeds then generating random group values for the reference array. The dimension values for each are randomized up to the maximum of a float for the arbitrary number of dimensions. Another array is then set to be the original reference array and gets the values of reference. The reference array then copies the values from the original reference array before any sort so that the starting state is always the same. The machine that the testing was done on contained a 2.6 GHz Intel Core i7-9750H 6-Core 12-thread and 64GB of RAM running on Ubuntu 18.04. After all sorts were run a validate sort function was run to ensure that the parallelism did not break the sorting algorithm. All parallel implementations were experimented on to find optimal parameters for the parallel construct as well as for the algorithm. The original array was used for all sorting calculations as it was copied over to prevent best case performance on subsequent sorts. The first test run was to calculate the comparisons between the distance metrics and the different sort implementations. The reference array was run at a constant size of 131072, the dimensions of the points was kept constant at 128 dimensions and the query array was kept constant at 400 query points. Figure 1a graphs the results of the distance metrics. It can be seen that the manhattan distance metric is significantly faster than the euclidean distance metric with a slight loss of accuracy. The euclidean distance metric was selected for all further calculations as it provided a higher accuracy to the actual distance and would most likely be used in real world applications. In figure 2 the time taken for the serial implementations and the parallel implementations of the solutions



(a) Distance Metrics

(b) Sort compute times

Figure 1: Distance and Sort Compute times

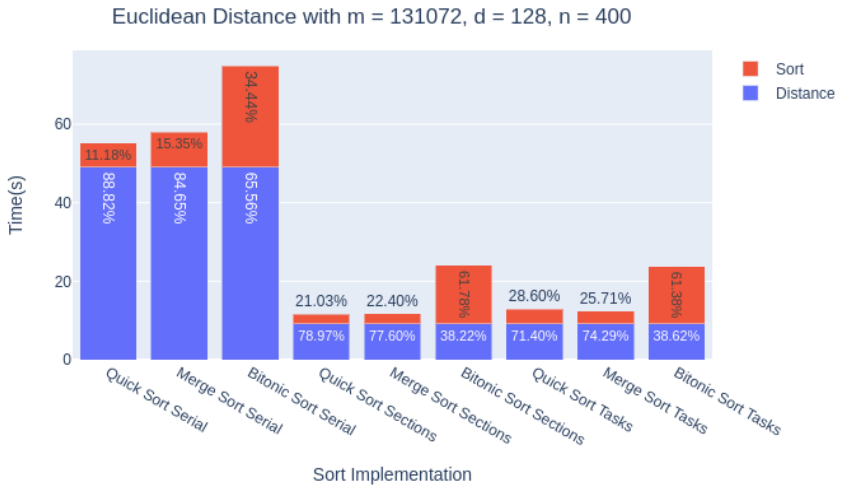


Figure 2: Final implementations with percentage of distance and sort compute time

is shown using the different sorting implementations and their respective percentage of compute time with respect to sorting calculation and distance calculation. It can be seen that the parallel distance calculation is significantly faster than the serial implementation. The bitonic sort follows a trend of having significantly longer compute times than the other sorts and in the parallel implementations has a significantly longer compute time than the distance calculation. All sort time see an increase in percentage of compute time in parallel implementations from serial implementations and this is due to a larger increase in speed up with respect to distance calculations averaging at around 5 times speed up whereas the speed ups for the sorting algorithms were averaging half of the distance speed ups. The remaining compute time of the algorithm goes into allocating the group of the query point but is constant and therefore negligible. Figure 1b shows the individual compute times of the various implementations of the sort algorithms. Quick sort was the fastest serial implementation with merge sort as a close second and bitonic sort as the slowest by a significant amount. The parallel implementations gained large increases in speed up and reduced the respective computational time by significant amounts. The quick sort gained a speed up of three on average for the two constructs but did see a larger speed up on the sections construct. The merge sort gained significant speed up with both implementations and continued the trend from

quick sort of having slightly better gains with the sections construct. Bitonic sort gained less significant speed up than the other two sorts although broke the trend of more gains with the section construct and shows the tasks construct may be better for this particular sort. Further testing is required to determine the best construct for each sorting algorithm.

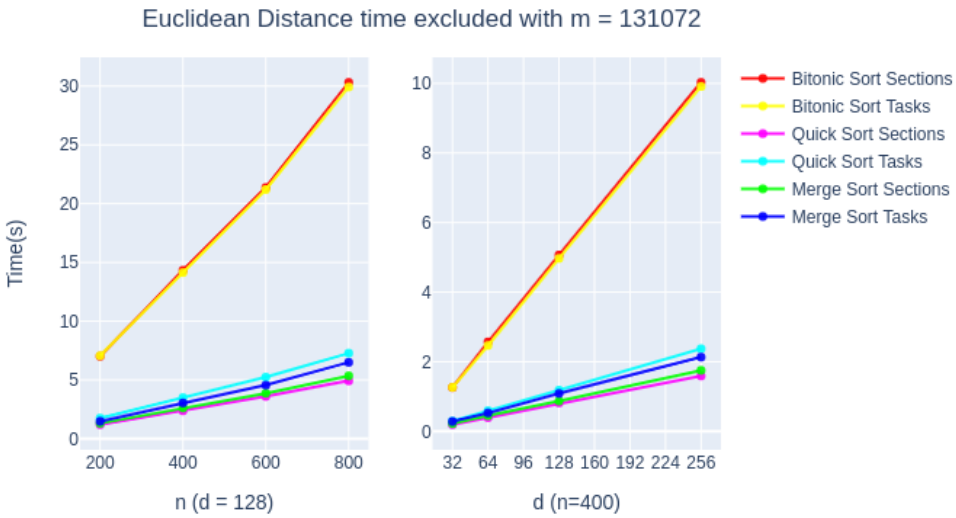


Figure 3: Varying n and d

The second test run goes through the two constructs with varying sizes of n and the third test run goes through the two constructs with varying sizes of d. The m value is kept constant at 131072 and for second test run d is kept at 128 and for the third test run n is kept at 400. In figure 3 the varying n and d are visible for implementations to see the trend of changing n and d. The trend follows that bitonic sort takes the longest and has a relatively steep increase trend as compared to the other sorts. The quick sort sections construct narrowly has the lowest compute time and it's trend is the lowest of all the sorts and hence is the best implementation. The merge sort constructs and the tasks construct of quick sort are slightly slower than the quick sort sections construct and hence are relatively optimal selections. The bitonic sort constructs are very similar in compute time although the tasks construct is slightly faster.

5 Conclusion

The most optimal solution is to use quick sort with the sections construct and manhattan distance with the parallel for construct if compute time is of great concern. The most optimal construct with respect to both compute time and accuracy is the quick sort with the sections construct and the euclidean distance with a parallel for construct. The bitonic sort is limited by it's inability to work on reference arrays that are not a size of a power of 2. The bitonic sort is also the slowest so it is not recommended for this problem. Merge sort is relatively close to quick sort and hence with further testing may prove to be equivalent or better but for the value ranges tested in this document quick sort with the tasks construct is the optimal solution. A challenge that was encountered was the sorting of the array after it was sorting creating best case scenarios on all subsequent sorts. For the results to be accurate the original reference array had to be kept and copied to be sorted at each sort in it's original state to have the same start case for each sort. The implementations of the various sorts changed throughout experimentation as the original implementations did not get speed up. Recommendations for further work are to test other sorting algorithms and to test the optimal algorithms at larger state spaces.

References

Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

Zhenyun Deng, Xiaoshu Zhu, Debo Cheng, Ming Zong, and Shichao Zhang. Efficient knn classification algorithm for big data. *Neurocomputing*, 195:143 – 148, 2016. Learning for Medical Imaging.

Mihai F Ionescu and Klaus E Schauser. Optimizing parallel bitonic sort. In *Proceedings 11th International Parallel Processing Symposium*, pages 303–309. IEEE, 1997.

Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Ruili Wang. Efficient knn classification with different numbers of nearest neighbors. *IEEE transactions on neural networks and learning systems*, 29(5):1774–1785, 2017.