WITS UNIVERSITY

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

COMS4040A & COMS7045A Assignment 1 – Report

Marc Marsden
1437889
BDA Hons

March 18, 2020

1 Introduction

The general form of the k-Nearest Neighbours (kNN) problem is defined as follows: Given a set of reference points, R, in any dimension *d*, the k nearest neighbours must be found for each query point in a set Q. The distances will be stored in a 2-D array, and then sorting will be implemented. This algorithm is used widely for classification and regression problems found in machine learning and data mining. This report looks at the simplest version of kNN, Brute Force (BF) kNN, where the points of the sets R and Q are disjoint, and how parallelization can be implemented to improve the algorithm.

2 Methodology

2.1 Overview

Two sections make up the kNN algorithm: the distance calculation and the sorting. Both of these are computationally taxing, especially when the dimensions, *d*, are very large. The distance metrics used in this paper will be the Euclidean distance formula and the Manhattan distance formula, both which will use Quicksort, Mergesort and Bitonic Sort to sort the distances in ascending order.

2.2 Foster's Design Methodology

2.2.1 Calculation of Distance

Partitioning: This part is perfectly parallel and ideal partitioning can be achieved by using "Domain Decomposition" with only one computation of the distance between two points as the "primitive task".

Communication: No inter-task communication is required, since the distance calculations are independent.

Agglomeration: With the independence of the distance calculations, task can be arbitrarily agglomerated.

Mapping: By assigning the same number of distance calculations to each thread, mapping can be done trivially.

2.2.2 Computation of Distance

Partitioning: Again "Domain Decomposition" will be used. The primitive task will be the sorting of a length two list, and so the problem can be partitioned by separating the list of distances.

Communication: All the tasks may access and change the 2-D distance array when sorting. Therefore there needs to be coordination between the tasks.

Agglomeration: Agglomeration can be achieved by combining each task's data.

Mapping: Mapping can be attained by separating the lists into sub-lists of approximately the same size and then assigning the tasks to each thread.

2.3 Parallelization

OpenMP and C will be used to parallelize the problem.

2.3.1 Calculation of Distance

The calculation of distance was parallelized by parallelizing a for loop, making the iteration variable private and sharing the 2-D array, which stores the calculated distances, to each thread.

2.3.2 Quicksort

OpenMP sections and tasks constructs were used to parallelize the sorting algorithm, meaning that each recursive call to the sorting function acts as a section or task. There was no need to parallelize the partitioning algorithm of Quicksort.

2.3.3 Mergesort

Similarly to Quicksort, Mergesort was parallelized using the section and tasks constructs of OpenMP. Parallelization of the merge function was not implemented.

2.3.4 Bitonic Sort

Bitonic sort was parallelized similarly to Quicksort and Mergesort. The algorithm was modified to accept any size array.

3 Experiment

3.1 Experimental Setup

3.1.1 Data

C's random number generator was used to generate to float points for both the reference and query points.

3.1.2 Specification of Machine Used

- CPU: Intel Core i5-4210M @2.60GHz(4 Cores)
- RAM: 4GB DDR3
- GPU: Intel Corporation 4th Gen Core Processor Integrated Graphics Controller (rev 06)

3.1.3 Experiment Details

Multiple tests were conducted by varying *m,n* and **d**, where *m* is the number of reference points, *n* is the number of query points and *d* is the number of dimensions. Two large numbers, 50 000 and 100 000, were chosen for *m*; *n* varied between 200, 400, 800, 1600 and *d* varied between 32, 64, 128, 256 and 512. A low limit of 5000 will also be used so that the overhead for parallelization is minimised for small data.

4 Results

See Appendix for figures

4.1 Discussion

In figures 1-6, we can see that as *n* grows, the amount of time taken for the sorting algorithms to sort the data, increases linearly. Even though that is the case, due to parallelization of the algorithms, there was a significant decrease in time taken to sort the data in all three of the algorithms. Even with the overhead of using OpenMP, the overhead becomes less significant the greater the data set. In figure 11, it can be seen that the sorting algorithms parallelized by OpenMP sections construct outperformed the serial code, but under-performed compared to parallelization by the task construct, which was to be expected. Overall, Quicksort performed the best in all three tests, followed by Mergesort and then Bitonic sort which did exceptionally worse. The parallelization of the Euclidean distance calculation outperformed the serial calculation for both small and large data, as seen in figure 7 and 8. The same can be said for the Manhattan distance calculation when looking at figure 9-10. Overall the Manhattan distance calculation outperforms the Euclidean calculation so much so, that the Manhattan's serial, whether big or small data, is better than the Euclidean's parallel(figure 12).

5 Summary

In this report, a parallel BF kNN algorithm was designed using Foster's Design Methodology. To implement this algorithm, an API known as OpenMP was used in conjunction with C. The three sorting algorithms were parallelized using OpenMp's sections and tasks constructs, while the metric distance calculations were parallelized using the for construct. As expected, the tasks construct outperformed both the sections constructs as well as the serial. The parallel versions of the metric calculations outperformed their respective serials, but it was also found that Manhattan is a lot less computationally taxing than Euclidean. Due to the limitations of both hardware and time, bigger amounts of data points could not be processed to see how effective the parallelisation truly is. However, this report has achieved its goal of showing how parallel computing can improve the BF kNN algorithm.

6 Appendix

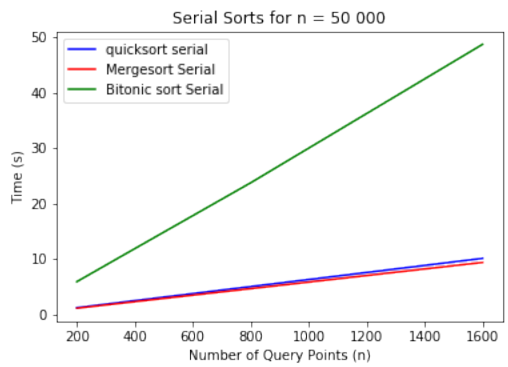

Figure 1: Serial Sorts for m = 50 000

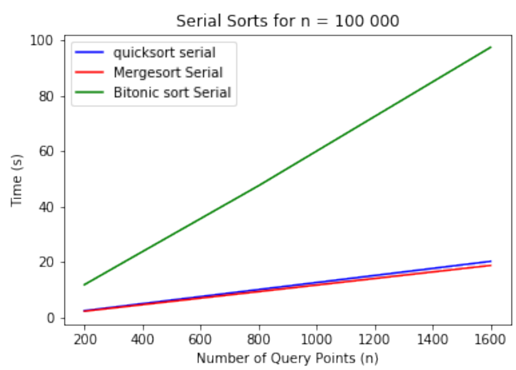

Figure 2: Serial Sorts for m = 100 000

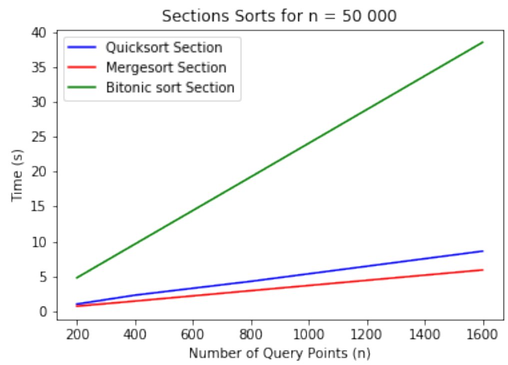

Figure 3: Section Sorts for m = 50 000

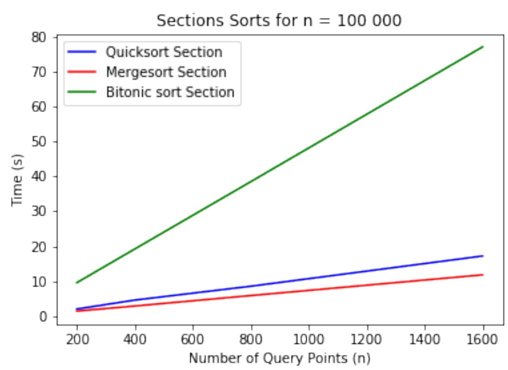

Figure 4: Section Sorts for m = 100 000

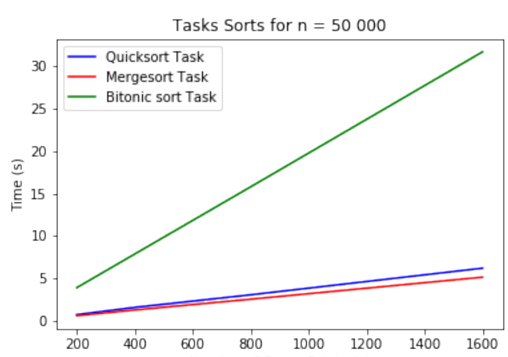

Figure 5: Tasks Sorts for m = 50 000

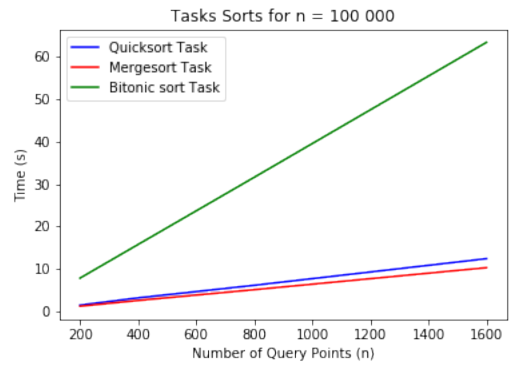

Figure 6: Section Sorts for m = 100 000

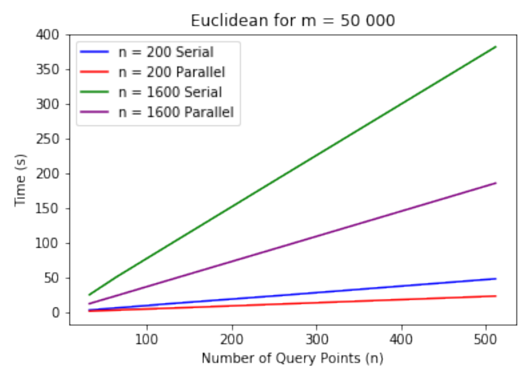

Figure 7: Euclidean for m = 50 000

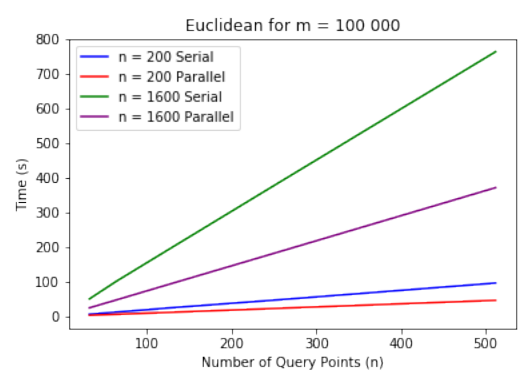

Figure 8: Euclidean for m = 100 000

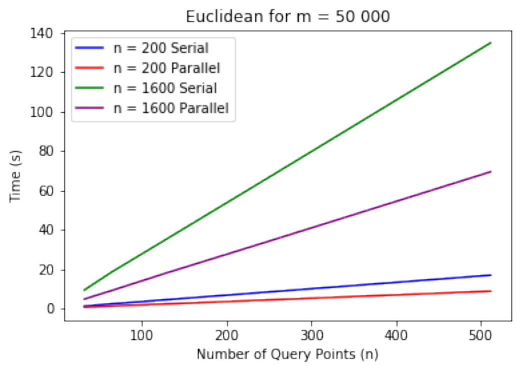

Figure 9: Manhattan for m = 50 000

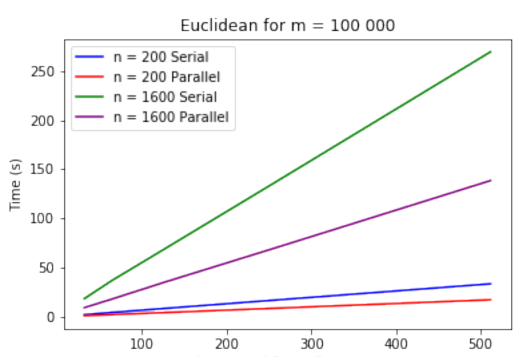

Figure 10: Manhattan for m = 100 000

Sort	S or P	m	n	Time
Quick	Serial	50000	200	1,276262
Quick	Section	50000	200	1,049593
Quick	Task	50000	200	0,780281
Merge	Serial	50000	200	1,181396
Merge	Section	50000	200	0,741127
Merge	Task	50000	200	0,650625
Bitonic	Serial	50000	200	5,956117
Bitonic	Section	50000	200	4,808678
Bitonic	Task	50000	200	3,945989
Quick	Serial	50000	400	2,548587
Quick	Section	50000	400	2,321328
Quick	Task	50000	400	1,635534
Merge	Serial	50000	400	2,357791
Merge	Section	50000	400	1,474471
Merge	Task	50000	400	1,322419
Bitonic	Serial	50000	400	11,906683
Bitonic	Section	50000	400	9,619410
Bitonic	Task	50000	400	7,892660
Quick	Serial	50000	800	5,081660
Quick	Section	50000	800	4,299649
Quick	Task	50000	800	3,107842
Merge	Serial	50000	800	4,702942
Merge	Section	50000	800	2,967339
Merge	Task	50000	800	2,591752
Bitonic	Serial	50000	800	23,756806
Bitonic	Section	50000	800	19,231492
Bitonic	Task	50000	800	15,777326
Quick	Serial	50000	1600	10,160490
Quick	Section	50000	1600	8,619137
Quick	Task	50000	1600	6,231983
Merge	Serial	50000	1600	9,423553
Merge	Section	50000	1600	5,926502
Merge	Task	50000	1600	5,175385
Bitonic	Serial	50000	1600	48,670716
Bitonic	Section	50000	1600	38,488672
Bitonic	Task	50000	1600	31,615626

Sort	S or P	m	n	Time
Quick	Serial	100000	200	2,682771
Quick	Section	100000	200	2,275667
Quick	Task	100000	200	1,577763
Merge	Serial	100000	200	2,489450
Merge	Section	100000	200	1,464458
Merge	Task	100000	200	1,366534
Bitonic	Serial	100000	200	13,196952
Bitonic	Section	100000	200	9,030946
Bitonic	Task	100000	200	8,644303
Quick	Serial	100000	400	5,372803
Quick	Section	100000	400	4,439511
Quick	Task	100000	400	3,196413
Merge	Serial	100000	400	4,930597
Merge	Section	100000	400	2,908089
Merge	Task	100000	400	2,753430
Bitonic	Serial	100000	400	26,395999
Bitonic	Section	100000	400	17,972935
Bitonic	Task	100000	400	17,397463
Quick	Serial	100000	800	10,884589
Quick	Section	100000	800	9,686132
Quick	Task	100000	800	6,814064
Merge	Serial	100000	800	9,874263
Merge	Section	100000	800	5,823721
Merge	Task	100000	800	5,460650
Bitonic	Serial	100000	800	52,907545
Bitonic	Section	100000	800	35,935640
Bitonic	Task	100000	800	34,810399
Quick	Serial	100000	1600	21,510151
Quick	Section	100000	1600	18,290551
Quick	Task	100000	1600	12,668864
Merge	Serial	100000	1600	19,723461
Merge	Section	100000	1600	11,613924
Merge	Task	100000	1600	10,945361
Bitonic	Serial	100000	1600	105,351250

Figure 11: An extract of the table of data for the times for the sorting algorithms

Metric	S or P	m	n	d	Time
Euclidean	Serial	50000	200	32	3,155037
Euclidean	Parallel	50000	200	32	1,540469
Manhattan	Serial	50000	200	32	1,138069
Manhattan	Parallel	50000	200	32	0,598420
Euclidean	Serial	50000	200	64	6,165024
Euclidean	Parallel	50000	200	64	2,985475
Manhattan	Serial	50000	200	64	2,265247
Manhattan	Parallel	50000	200	64	1,146525
Euclidean	Serial	50000	200	128	12,405062
Euclidean	Parallel	50000	200	128	5,855940
Manhattan	Serial	50000	200	128	4,362639
Manhattan	Parallel	50000	200	128	2,226522
Euclidean	Serial	50000	200	256	23,972252
Euclidean	Parallel	50000	200	256	11,669425
Manhattan	Serial	50000	200	256	8,514408
Manhattan	Parallel	50000	200	256	4,389054
Euclidean	Serial	50000	200	512	48,132861
Euclidean	Parallel	50000	200	512	23,215145
Manhattan	Serial	50000	200	512	16,826677
Manhattan	Parallel	50000	200	512	8,718675
Euclidean	Serial	50000	400	32	6,287648
Euclidean	Parallel	50000	400	32	3,071530
Manhattan	Serial	50000	400	32	2,271884
Manhattan	Parallel	50000	400	32	1,179054
Euclidean	Serial	50000	400	64	12,337514
Euclidean	Parallel	50000	400	64	5,948865
Manhattan	Serial	50000	400	64	4,527136
Manhattan	Parallel	50000	400	64	2,254779
Euclidean	Serial	50000	400	128	24,364927
Euclidean	Parallel	50000	400	128	11,737868
Manhattan	Serial	50000	400	128	8,713762
Manhattan	Parallel	50000	400	128	4,437229
Euclidean	Serial	50000	400	256	48,243196
Euclidean	Parallel	50000	400	256	23,272823

Metric	S or P	m	n	d	Time
Euclidean	Serial	100000	200	32	6,276675
Euclidean	Parallel	100000	200	32	3,075949
Manhattan	Serial	100000	200	32	2,277492
Manhattan	Parallel	100000	200	32	1,173727
Euclidean	Serial	100000	200	64	12,376283
Euclidean	Parallel	100000	200	64	5,956520
Manhattan	Serial	100000	200	64	4,540785
Manhattan	Parallel	100000	200	64	2,256501
Euclidean	Serial	100000	200	128	24,238994
Euclidean	Parallel	100000	200	128	11,730761
Manhattan	Serial	100000	200	128	8,697045
Manhattan	Parallel	100000	200	128	4,437291
Euclidean	Serial	100000	200	256	47,923759
Euclidean	Parallel	100000	200	256	23,350028
Manhattan	Serial	100000	200	256	17,023613
Manhattan	Parallel	100000	200	256	8,724964
Euclidean	Serial	100000	200	512	95,331567
Euclidean	Parallel	100000	200	512	46,482637
Manhattan	Serial	100000	200	512	33,750149
Manhattan	Parallel	100000	200	512	17,295394
Euclidean	Serial	100000	400	32	12,536665
Euclidean	Parallel	100000	400	32	6,136239
Manhattan	Serial	100000	400	32	4,541666
Manhattan	Parallel	100000	400	32	2,361541
Euclidean	Serial	100000	400	64	25,451703
Euclidean	Parallel	100000	400	64	11,900511
Manhattan	Serial	100000	400	64	9,129871
Manhattan	Parallel	100000	400	64	4,529456
Euclidean	Serial	100000	400	128	49,759393
Euclidean	Parallel	100000	400	128	23,464331
Manhattan	Serial	100000	400	128	17,389859
Manhattan	Parallel	100000	400	128	8,864958
Euclidean	Serial	100000	400	256	97,412792
Euclidean	Parallel	100000	400	256	46,646587

Figure 12: An extract of the table of data for the times for the metric