K-NEAREST NEIGHBOUR (KNN) SERIAL VS PARALLEL
ASSIGNMENT 1

HIGH PERFORMANCE COMPUTING

SCHOOL OF COMPUTER SCIENCE AND APPLIED MATHEMATICS
UNIVERSITY OF THE WITWATERSRAND

KYLE WEIHER
1116087

MARCH 8, 2018

Introduction

Given a dataset, with *m* rows of datapoints, each point in $\mathbb{R}^d$. Given *n* query points that fall within the domain of the dataset, the k Nearest Neighbours (kNN) search problem involves finding the *k* nearest neighbours in a dataset to each query point, with regards to a specific distance metric. This problem appears in a number of areas, such as kNN classification and regression in machine learning.

In this report an algorithm will be described, namely the bruteforce solution to the problem, which has the following basic structure:

1. For a specific query point, compute the distance from said point to every point in the dataset.
2. Sort the distances in ascending order.
3. Take the first k points with the shortest distances.
4. Repeat for each point in the query set.

The main focus of this experiment is to compare the runtimes of the algorithm, differing search algorithms and distance metrics, with regards to serial or parallel execution.

Methodology

A number of different distance metrics are available, and as such two will be tested in this experiment, namely Euclidean distance, and Manhattan distance. Furthermore, different search algorithms will be compared, namely quick-sort, merge-sort, and bubble-sort.

The experiment will cover a variety of variations to the above distance metrics and search algorithms, as well as testing between parallel and serial versions. There will be no mixing of parallel and serial, however, as holistically the kNN algorithm will be entirely parallel or entirely serial, and not a mix of both. Furthermore, both task and section constructs will be tested.

A number of considerations will be made when testing the different set-ups that the kNN algorithm can use:

- As the distance calculation and the sorting are not reliant on each other in terms of which algorithm is picked for both parallel and serial tests, when testing the different distance algorithms, the sorting algorithm used will remain the same.
- The fact that the amount of query points used, *n*, is somewhat of a global multiplier on run time, one test focussing on how *n* influences runtime will be performed, and then for the rest of the tests *n* will remain constant.
- As said above, the algorithm will be tested as either entirely parallel, or entirely serial, the will be no mixing of this parameter.

Experimental Setup

The data used in these tests is randomly generated at run time. Before any computations take place, both the reference dataset and the query dataset are created, and then tests begin. The data is of type double, the code is written in C, and the parallel framework used is OpenMP.

The specifications of the machine that ran the tests are as follows:

- **OS**: Ubuntu 17.10 64bit
- **Kernel**: 4.15.7-041507-generic
- **Compiler**: gcc 7.2.0

- **CPU**: Intel i7-4720HQ (8) @ 3.600GHz
- **Memory**: 16GiB

Results

In this section multiple aspects of the experiment will be discussed, with accompanying tables to support the text.

Effect of *n*: - Table 1 shows the results of varying n, the amount of query points, on a Quick-Sort with Euclidean Distance used. As can be seen in the table, varying n simply increases the run-time of both the sorting and searching, and keeps their percentage ratios similar. This can be attributed to the fact that the implementation of kNN used in this experiment, whereby each query point is run one after each other, with no parallelisation, and as such acts as a scaler coefficient to the run-time. Therefore, in the rest of the tests, it's value is set at 3 and does not change.

n	1	5	10	15
Serial				
Distance%	0.402066	0.356454	0.356817	0.351973
Sorting%	0.597934	0.643546	0.643183	0.648027
Total Time	0.032916	0.146645	0.290281	0.432257
Parallel				
Distance%	0.349746	0.313459	0.320546	0.315975
Sorting%	0.650254	0.686541	0.679454	0.684025
Total Time	0.019778	0.083631	0.163993	0.241572

Table 1: Effect of varying n , $m = 200000$, $d = 30$
Quick-Sort, Euclidean Distance

Effect of d: - Table 2 shows the results of varying d, the number of dimensions of each datapoint in the dataset. As can be seen, the effect is localised to the run time of the distance calculation functions, whereby in both series and parallel they are responsible for more and more of the combined run-time as the value of d increases. Therefore it was decided that it would only vary in this test and the tests for the distance metrics, and in other tests it is kept constant.

d	3	30	300	3000
Euclidean				
Distance%	0.251236	0.391525	0.785063	0.968736
Sorting%	0.748764	0.608475	0.214937	0.031264
Total Time	0.073966	0.091819	0.249631	1.695478
Manhattan				
Distance%	0.129495	0.339821	0.831825	0.979623
Sorting%	0.870505	0.660179	0.168175	0.020377
Total Time	0.162400	0.124531	0.319219	2.325372

Table 2: Effect of varying d , $m = 200000$, $n = 3$
Quick-Sort

Sections vs Tasks: - Tests involving sections and tasks resulted in tasks performing better on average, as seen in Table 3, however when compared to the run time of a test running in serial, the relative different is much smaller, and as such in other tests the term parallel is used to denote results from either sections or tasks. For sections, nested parallelism needed to be enabled to make use of the full power of the machine running the test.

Serial vs Parallel: - In general there is a large run-time decrease when running the kNN algorithm in parallel compared to serial, as seen in Table 4. There are however great differences in the effectiveness of parallelism on

m	10000	50000	100000	500000
Sections				
Distance%	0.213616	0.144095	0.080476	0.055651
Sorting%	0.786384	0.855905	0.919524	0.944349
Total Time	0.023404	0.062275	0.125248	0.537504
Tasks				
Distance%	0.310401	0.106238	0.125474	0.066694
Sorting%	0.689599	0.893762	0.874526	0.933306
Total Time	0.013735	0.044038	0.079928	0.451017

Table 3: Comparison of Sections vs Tasks on varying m, n = 3, d = 30
Merge-Sort, Euclidean Distance

the specific sorting algorithms. For example, while Quick-Sort shows a noticeable decrease in run-time, Merge-sort shows a greater improvement, and Bubble-Sort shows an extreme improvement. This can be attributed to the method in which Bubble-Sort was parallelised, whereby the dataset was split up into equal amounts for each core, and each then worked on by an individual serial Bubble-Sort, and then when all sections are sorted, they are merged together in the same fashion a merge sort would merge two sets of data. This results in allowing more cores of the host machine to work on the otherwise unparallelisable sorting method, and since each section has fewer data points to work on, the $O(n^2)$ nature of the method is subdued dramatically, leading to a massive reduction in run time.

Sort	Quick-Sort	Merge-Sort	Bubble-Sort
Serial			
Distance%	0.403915	0.075407	0.000344
Sorting%	0.596085	0.924593	0.999656
Total Time	0.044641	0.217152	53.944891
Parallel			
Distance%	0.333057	0.080476	0.004566
Sorting%	0.666943	0.919524	0.995434
Total Time	0.032820	0.125248	1.330655

Table 4: Sorting comparisons of serial vs parallel, m = 100000, n = 3, d = 30
Euclidean Distance

Euclidean vs Manhattan Distance: - Table 5 shows the comparison between the two distance metrics tested for the kNN algorithm. As can be seen, and as mentioned above, the distance metrics are sensitive to changes in the value of d, since the run-time contribution for both Euclidean and Manhattan distance metrics is around 96%, in both serial and parallel cases when d = 3000. A possible reason for the Manhattan distance metric taking longer than the Euclidean metric is that the Manhattan metric involves the use of an absolute value operator, which may be a more expensive operation than a the power operator. The parallelisation that is used for distance metrics makes use of the *$pragma omp for* tag, and as such there are neither tasks nor sections used.

Summary and Discussion

In this report the kNN algorithm was introduced, and algorithmic approach of solving the problem was laid out and implemented, tested with and without parallelisation on its fundamental internal workings, and the results were tabulated and discussed.

Some challenges encountered in over the course of the experiment were:

- Difficulty with getting OpenMP to behave in a predictable manner
- Finding a way to tabulate and convey the relevant parts of the results gathered from tests, as there are many

Distance Calculation	Euclidean	Manhattan
Serial		
Distance%	0.968736	0.979623
Sorting%	0.031264	0.020377
Total Time	1.695478	2.325372
Parallel		
Distance%	0.967413	0.959599
Sorting%	0.032587	0.040401
Total Time	0.747117	0.801413

Table 5: Euclidean vs Manhattan Distance, $m = 200000$, $n = 3$, $d = 3000$
Quick-Sort

combinations of variables that can be tested and finding the useful ones requires planning.

- The algorithm was implemented in C, and as such makes heavy use of pointers. This can lead to difficulty if not everything is kept track of.