Παράλληλα και Διανεμημένα Συστήματα

Implement in `MPI` a distributed brute-force all-KNN search algorithm for the k nearest neighbors (k-NN) of each point x∈X

Τσιράκης Ορέστης, ΑΕΜ: 9995
Github: https://github.com/Orestistsira/auth-parallel-and-distributed-systems-ex2

Σύντομη περιγραφή του προβλήματος:

Στη δεύτερη εργασία μας ζητήθηκε να υλοποιήσουμε έναν αλγόριθμο ο οποίος θα βρίσκει τους k κοντινότερους γείτονες κάθε σημείου x, όπου το x ανήκει σε ένα σύνολο σημείων X, και τις αποστάσεις από αυτούς.

Δομή knnresult:

```
// Definition of the kNN result struct
typedef struct knnresult{
        int    * nidx;    //!< Indices (0-based) of nearest neighbors [m-by-k]
        double * ndist;   //!< Distance of nearest neighbors        [m-by-k]
        int      m;       //!< Number of query points               [scalar]
        int      k;       //!< Number of nearest neighbors          [scalar]
} knnresult;
```

V0. Σειριακή υλοποίηση: (*Η υλοποίηση είναι Row-Major)

Στήν σειριακή υλοποίηση η συνάρτηση **‘kNN’** καλείται από τη main, η οποία δέχεται έναν πίνακα X[m-by-d] και έναν πίνακα Y[n-by-d], όπου d η διάσταση του χώρου των σημείων και επιστρέφει ένα struct τύπου knnresult όπως αναφέρεται επάνω, όπου για κάθε σημείο του X έχουμε τους κοντινότερους γείτονες του στο Y.

Αρχικά υπολογίζουμε τον πίνακα D[n-ny-m], ο οποίος περιέχει την ευκλείδια απόσταση του κάθε σημείου από όλα τα υπόλοιπα σημεία του χώρου από τη σχέση:

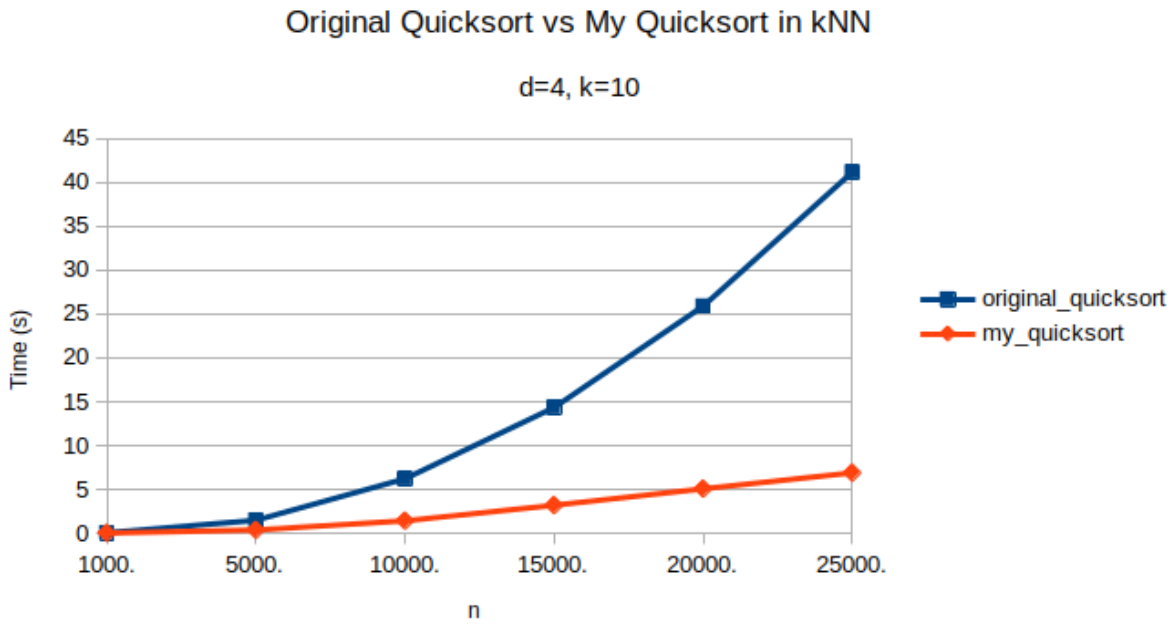
$$D=(X\odot X)ee^T-2XY^T+ee^T(Y\odot Y)^T$$

Χρησιμοποιούμε τη βιβλιοθήκη *openblas* ώστε να κάνουμε γρήγορες πράξεις μεταξύ πινάκων και τον υπολογισμό των μέτρων X και Y.

Έπειτα με τη χρήση της quicksort βρίσκουμε γιά κάθε σημείο τις k κοντινότερες αποστάσεις και ταυτόχρονα παίρνουμε τα ids των k κοντινότερων γειτόνων. Καθώς χρειαζόμαστε μόνο τις k μικρότερες αποστάσεις έχει χρησιμοποιηθεί μία παραλλαγή της quicksort, στην οποία όταν το pivot βρίσκουμε να είναι μεγαλύτερο του k (δηλαδή όλα τα στοιχεία αριστερά του pivot περιέχουν τα μικρότερα k στοιχεία του πίνακα), τότε δε χρειάζεται να συνεχίσουμε την quicksort στο δεξί κομμάτι αφού ξέρουμε πως δεν θα μας χρειαστούν. Αυτή η παραλλαγή μειώνει πολύ τον χρόνο εκτέλεσης της ταξινόμισης ειδικά για μεγαλύτερους πίνακες.

Ο υπολογισμός των k κοντινότερων σημείων θα μπορούσε να γίνεται και ταυτόχρονα για όλα τα σημεία με χρήση παράλληλου προγραμματισμού (opnemp, opencilk), καθώς η ταξινόμιση των γειτόνων είναι ανεξάρτητη για το κάθε σημείο, και θα μας έδινε αρκετά καλύτερους χρόνους.

(Οι χρόνοι είναι με τοπική εκτέλεση του προγράμματος)



V1. Ασύγχρονη MPI υλοποίηση:

Στην V0 υλοποίηση ήταν δύσκολο να υπολογίσουμε το knn για ένα πολύ μεγάλο σύνολο σημείων καθώς οι απαιτήσεις για μνήμη ήταν μεγάλες για να μπορέσει ένας υπολογιστής να ανταπεξέλθει.

Για την ασύγχρονη υλοποίηση, το κάθε task παίρνει το υποσύνολο τών σημείων που του αναλογεί (π.χ. για n=4 tasks, η κάθε εργασία θα πάρει το ¼ των σημείων) μειώνοντας έτσι τις χωρικές απαιτήσεις για την κάθε διεργασία. Έπειτα καλείται η συνάρτηση **‘distAllKnn’** η οποία επιστρέφει τους k κοντινότερους γείτονες (από ολόκληρο το σύνολο των σημείων) για το υποσύνολο του κάθε task.

Τα tasks σχηματίζουν ενα ¨δαχτυλίδι”, όπου σε κάθε επανάληψη το κάθε process δίνει στο επόμενο τα corpus points Y του και αντιακθιστά τα δικά του από αυτά πού το στέλνει ο προηγούμενος, κρατόντας τα query points X ίδια με τα αρχικά.

Γιά να γίνει η ανταλλαγή των corpus points μεταξύ των διεργασιών, χρησιμοποιείται ασύγχρονη επικοινωνία με τη βοήθεια των συναρτήσεων **‘MPI_Isend’** και **‘MPI_Ircv’**. Για να “καλύψουμε” τον χρόνο που ανταλλάσεται η πληροφορία, όση ώρα πραγματοποιείται η μεταφορά τρέχουμε τη συνάρτηση **‘kNN’** για τα σημεία του κάθε task και κρατάμε τα k κοντινότερα σημεία από όλα τα σύνολα που περνάνε.

Αφού περάσουν n – 1 επαναλήψεις, από το κάθε task θα έχουνε περάσει όλα τα σημεία, οπότε σταματάμε την επικοινωνία και επιστρέφουμε το τελικό σύνολο των κοντινότερων σημείων για κάθε task. Τέλος συγκεντρώνουμε σε μία διεργασία όλα τα αποτελέσματα με τη συνάρτηση **‘MPI_Gather’**.

Ορθότητα υλοποιήσεων:

Και οι δύο υλοποιήσεις δίνουν ίδια και σωστά αποτελέσματα για 2d, 3d και 4d regular grids και άλλα μικρά datasets. Έγιναν κάποιες δοκιμές στα MNIST hand-written digits και φάνηκε να παίρνουμε επίσης σωστά αποτελέσματα (http://yann.lecun.com/exdb/mnist/).

Αποτελέσματα: (Όλοι οι χρόνοι είναι από εκτέλεση στο HPC)

V0:

d = 4, k = 10

n	**Χρόνος εκτέλεσης (s)**
100	0.001066
1000	0.032294
10000	2.114836
15000	4.603953
20000	7.854658

n = 10000, k = 10

d	**Χρόνος εκτέλεσης (s)**
100	2.426142
300	3.507884
500	4.223797
1000	6.486485
5000	24.631921

Παρατηρούμε πως προφανώς όσο μεγαλώνει ο αριθμός των σημείων (n) αυξάνεται και ο χρόνος εκτέλεσης του αλγορίθμου. Βλέπουμε όμως πως η διάσταση του χώρου (d) έχει πολύ μεγαλύτερη επιρροή, όσο αυτή αυξάνεται, στον χρόνο υπολογισμού τον γειτόνων.

V1:

n = 50000, d = 1000, k = 100

p (διεργασίες)	**Χρόνος εκτέλεσης (s)**
4	54.963926
16	18.019774
32	11.651933
64	8.042196

Παρατηρούμε πως όσο αυξάνουμε τις διεργασίες (p) μειώνεται κατά πολύ ο χρόνος εκτέλεσης. Στις 64 εργασίες δεν υπάρχει και τόσο μεγάλη αύξηση στη ταχύτητα καθώς το δείγμα 50000 σημείων δεν είναι επαρκώς μεγάλο και υπάρχει μεγάλη καθυστέρηση στην επικοινωνία μεταξύ τους.

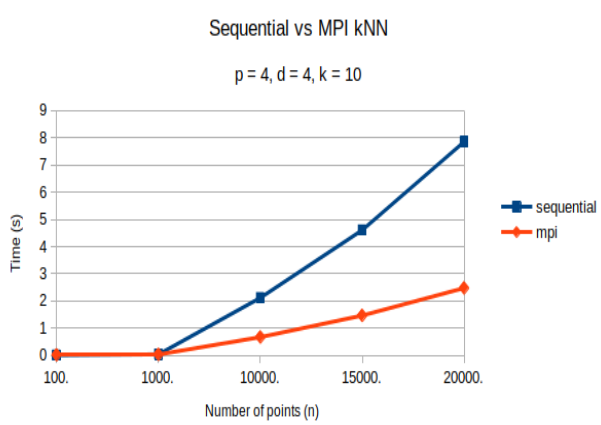
p = 4, d = 4, k = 10

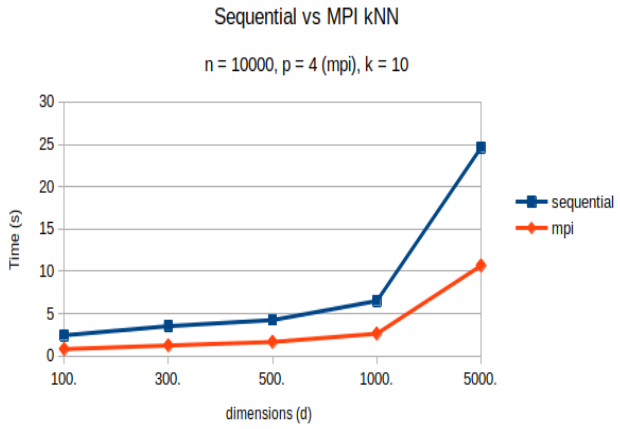
n	Χρόνος εκτέλεσης (s)
100	0.012705
1000	0.028110
10000	0.665643
15000	1.462031
20000	2.471986

n = 10000, p = 4, k = 10

d	Χρόνος εκτέλεσης (s)
100	0.788104
300	1.223308
500	1.636567
1000	2.619268
5000	10.659137

Σύγκριση σειριακής με mpi υλοποίησης:





Συμπεράσματα:

Βλέπουμε πως όσο ο αριθμός των σημείων μεγαλώνει η παράλληλη mpi υλοποίηση γίνεται αισθητά πιο γρήγορη από της σειριακή, καθώς βλέπουμε πως για n = 20000 το V1 είναι σχεδόν 4 φορές πιο γρήγορο από το V0 με τη διαφορά αυτή να αυξάνεται όσο το n μεγαλώνει.

Επιπλέον παρατηρούμε πως για διάστηαση χώρου μικρότερη απο 1000, η διαφορά στη ταχύτητα είναι σχεδόν ίδια, ενώ για μεγάλες διαστάσεις (d > 1000) η mpi υλοποίηση είναι πάνω από 2 φορές γρηγορότερη από την σειριακή.