

KNN Search with MPI  
Παράλληλα και Διανεμημένα Συστήματα  
*Assignment 2*

Φίλης Χάρης

January 20, 2021

- Δείκτης στο αποθετήριο της εργασίας : <https://github.com/harryfilis/Parallel-and-Distributed-Systems-Assignments/tree/master/KnnSearch>

## 0 Το πρόβλημα all-KnnSearch

Στην παρούσα εργασία υλοποιείται ένας κατανεμημένος αλγόριθμος all-KNN search ο οποίος παραλληλοποιείται με την βοήθεια της διεπαφής MPI. Ο αλγόριθμος βρίσκει τους **k** κοντινότερους γείτονες (**k-NN**) από κάθε σημείο ενός set **X**.

Είσοδος του Αλγορίθμου είναι τα εξής:

- πίνακας-set **X**
- **n** πλήθος σημείων του πίνακα **X**
- **d** αριθμός διαστάσεων των σημείων
- **k** αριθμός γειτόνων **k** για το οποίο θα γίνει το knnSearch

Κάθε MPI διαδικασία  $P_i$  θα υπολογίζει την απόσταση των δικών της σημείων από όλα τα άλλα σημεία και καταγράφει τις αποστάσεις (**distances** || **dmatrix** || **ndist**) και τους δείκτες του **k** πλησιέστερου για κάθε ένα από τα δικά του σημεία (αποθηκευση σε minarr στο searchVPT).

## 1 v0.c Sequential - Ανάλυση αλγορίθμου

Αρχικά Υλοποιήθηκε μια έκδοση του v0 σε matlab που παρατίθεται παρακάτω σε σχόλια εξηγείται η διαδικασία. **v0.m**

Όσον αφορά τον κώδικα σε c αυτός ακολουθεί το αρχείο matlab. Αρχικά ορίζεται η δομή δεδομένων **knnresult** με τα εξής μελη :

- **\*nidx** -> Indices (0-based) of nearest neighbors [m-by-k]
- **double \*ndist** -> Distance of nearest neighbors [m-by-k]
- **int n** -> Αριθμός των σημείων του ερωτήματος **Y**
- **int k** -> Αριθμός των **k** κοντινότερων γειτόνων

Στόχος να κρατήσουμε τα **k** κοντινότερα σημεία του **Y** στον **X** πίνακα.

## 1.1 Ανάλυση συνάρτησης knnresult kNN(double \*X, double \*Y, int n, int m, int d, int k)

Αρχικά αρχικοποιείται κατάλληλα η επιστρεφόμενη δομή **knnresult**.<sup>1</sup>

Θέλουμε να υπολογίσουμε τον πίνακα διαστάσεων **distances** βάση του τύπου:

```
D = sqrt(sum(X.^2,2)- 2 *X*Y.' + sum(Y.^2,2).');
X matrix
Y.' transpose Y matrix
```

1. Δεσμεύονται οι πίνακες **xx,yy** όπου υπολογίζονται και υλοποιούνται τα τετράγωνα των στοιχείων των πινάκων.

2. Ταυτόχρονα υπολογίζονται οι πίνακες

**y\_sum, x\_sum**

οι οποίοι είναι σε rowmajor format όπως ολοι οι 2d πίνακες.

3. μέσω της βιβλιοθήκης **cblas.h** γίνεται η πράξη -  $2*X*Y'$  και αποθηκεύεται σε rowmajor format στον πίνακα **distances** με την συνάρτηση

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans,
```

4. Στην συνέχεια αθροίζονται τα **x\_sum** και **y\_sum** στοιχεία στα στοιχεία του **distances** (για αυτό χρειάστηκε να είναι σε rowmajorformat)
5. Τώρα ο **D matrix** είναι έτοιμος και εκτελούμε **quik\_select** για να βρούμε τα **k** μικρότερα στοιχεία μέσω της **kthSmallest** (εδώ γίνεται μια πιο απλή υλοποίηση).
6. γίνεται αποθηκευση των **min distances** στο **knnresult.ndist** σε row\_major και ομοίως και των **indices**.
7. Επιστρέφεται η δομή **knnresult result**.

## 1.2 v0.c/main

Εδώ αρχικά παίρνονται από τα command line arguments τα **n,d,k,m** για τυχαίο πείραμα και αρχικοποιείται η δομή **knnresult** (malloc, etc.)

Στην συνέχεια ορίζονται ψευδώς τυχαίοι **X, Y** πίνακες με την βοήθεια της **randomBounded(double lower\_limit, double upper\_limit)**

Καλείται η **kNN** και μετράται ο χρόνος εκτέλεσης της.

<sup>1</sup>(To ndist έχει μέγεθος  $m*d$  doubles)

## 2 v1.c Asynchronous - Ανάλυση Αλγορίθμου

Γίνεται μια υλοποίηση σε matlab παλι **v1.m** Δεν χρησιμοποιώ την knnresult distrAllkNN(double \* X<sub>1</sub>, int n, int d, int k); αλλά την συνάρτηση του v0.c. Περνάω τα δεδομένα μέσω του MPI περιβάλλοντος σε ένα δαχτυλίδι όπου τσεκαρονται μεταξύ τους οι αποστάσεις και αλλάζει ο k\_nearest

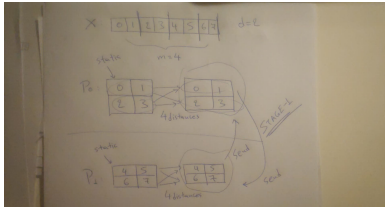


Figure 1: Ring\_Schematic

Q: How many times do we need to iterate to have all points checked against all other points?

Ans: Every process has processed its own x\_i\_data, now they have to pass around data to each other in a ring mode That has to repeat until each process has processed all possible data => p times

Η αντιγραφή (memory) που γίνεται γίνεται σε θέσεις μνήμης. \*Αναλυτικά σχόλια υπάρχουν στον κώδικα στο πως διαχειρίζομαι τα processes και τα μηνυματα που στέλνω.

## 3 v2.c Vantage Point tree - Ανάλυση αλγορίθμου

Εδώ χρησιμοποιήθηκαν οι εξής ψευδοκώδικες σε matlab:

```

1 function T = makeVPT(P)
2 [n,d] = size(P);
3 T.vp = P(1,:);
4 if n == 1
5     return
6 else
7     d = sqrt(sum(P-T.vp)^2,2);
8     T.md = median(d);
9     T.int = makeVPT(P(d<T.md,:));
10    T.ext = makeVPT(P(d>T.md,:));
11
12 end
13 end

1 function p=searchVPT(T,P)
2 p = work(T.vp,p)
3 if ~ (isfield(T,'int') | isfield(T,'ext'))
4     return

```

```

5 else
6     norm(T.rp-p.coor) < T.md
7     p = searchVPT(T.int,p)
8     if checkintersection
9         p = searchVPT(T.ext,);
10 end
11 end

```

Όσον αφορά τον κώδικα σε c Ορίζεται minArray struct που αποθηκεύονται απο κάθε process τα οι μικρότερες αποστάσεις και οι δείκτες αυτών. Ορίζεται επίσης η κλάση **node** που είναι η βασική δομική μονάδα του vp-Tree.(εχει pointers σε left child και right child(τύπου node)).

### 3.1 Συνάρτηση:node \*vpTree\_create(double \*x\_i\_data, node \*root, int m, int d)

Δημιουργεί ανδρομικά το δένδρο και επιστρέφει την ρίζα του. Συγκεκριμένα:

1. Αρχικοποιεί τον πίνακα vp του root με σημεία x\_i\_data(vp).
2. Υπολογίζει την μέση τιμή των αποστάσεων αυτών των σημείων της ρίζας
3. Αν οι αποστάσεις είναι μικρότερες απο την μέση τιμή της απόστασης των αποστάσεων της ρίζας τότε δημιουργείται αριστερό παιδί και αποθηκεύονται τα x\_i\_data σε row\_major format εκεί.
4. Αλλιώς γίνεται δεξί παιδί κτλ.
5. Τέλος καλείται η vpTree\_create για το αριστερό και το δεξί παιδί και αναδρομικά δημιουργείται το δένδρο.
6. επιστρέφεται το root.

### 3.2 Συνάρτηση:void searchVPT(double \*x\_query, node \*root, int d, int k, minArray\* min\_arr)

Αναδρομική συνάρτηση που αρχικά υπολογίζει την απόσταση των x\_query από το vantage point του root και τα αποθηκεύει στον min\_arr. Η radius παίρνει την απόσταση του γείτονα k. Αν η απόσταση είναι μικρότερη απο την μέση τιμή των αποστάσεων της τωρινής ρίζας + της radius καλείται η searchVPT για το ίδιο x\_query αλλά με root το αριστερό παιδί της root και αυτό κάνει την αναδρομή. Αντίθετα η αναδρομή γίνεται για το δεξί

υποδένδρο μέχρι και στις δύο περιπτώσεις να φτάσουμε σε φύλλο.

Η αναδρομή τελειώνει αν  $root == NULL$  δηλαδή το προηγούμενο root είναι φύλλο (τερματικός κόμβος).

Η main χρησιμοποιεί MPI και έχει αναλυτικά σχόλια για την κατανόηση και το πώς στο `s_knn_result` αποθηκεύονται εν τέλει οι  $kNN$ .

## Διαγράμματα και Προβλήματα

### Προβλήματα

1. Όταν έφτιαξα τρόπο να διαβάζω τους πίνακες που μας δώσατε πετούσε segmentation fault και δεν κατάφερα να τρέξω κανονικά για τους πίνακες παρόλο που έχω έτοιμα **ShellScripts** για το slurm που τρέχουν όλους τους πίνακες και κατέθεσα στην υπολογιστική μονάδα batches. (το λέω για να είμαι ειλικρινής.)
2. Η υπολογιστική μονάδα δεν ήταν προσβάσιμη λόγω του ότι δεν μας έδινε access στο vnp και επίσης είχαν κατατεθεί πολλές εργασίες οπότε η δικιά μου δεν προλαβε να ολοκληρωθεί.

### Διαγράμματα

Έγιναν 4 διαγράμματα από αρχικά batches που είχα θέσει στο hpc για τυχαίους πίνακες.

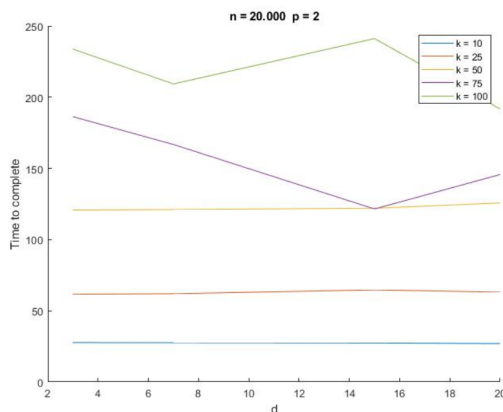


Figure 2: ./v1 n=20.000 p = 2

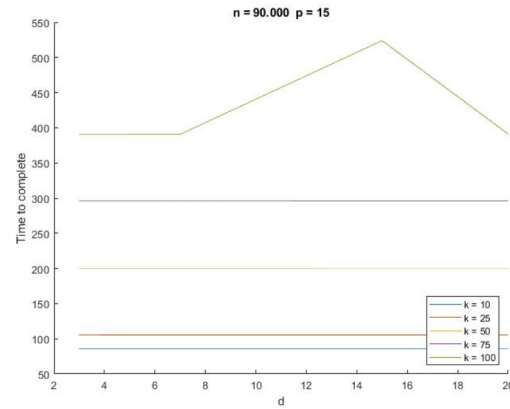


Figure 3: ./v1 n=90.000 p = 15

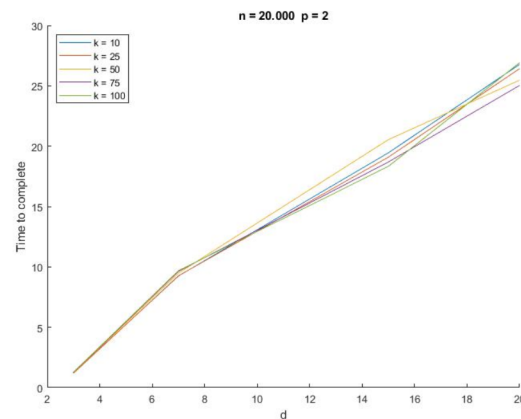


Figure 4: ./v2 n=20.000 p = 2

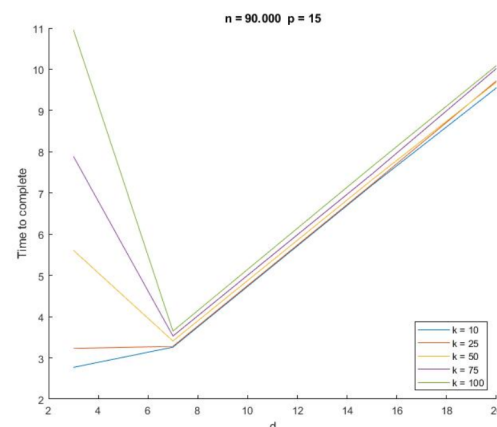


Figure 5: ./v2 n=90.000 p = 15

### Παρατηρήσεις

Βλέπουμε σαφή βελτίωση με την χρήση του vnpTree.

**ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ**