K-NN with Vantage-Point tree

Fotiou Dimitrios (AEM 9650)
Andreas Eleutheriadis (AEM 9649)

Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
github: https://github.com/dimfot3/K-NN-with-vantage-point-tree

February 26, 2022

***Abstract*—**Το παρόν έγγραφο αποτελεί την αναφορά στη τέταρτη και τελική εργασία του μαθήματος Παράλληλα και Διανεμημένα συστήματα**.** Διεκπεραιώθηκε από δύο άτομα που δεν έχουν συνεργαστεί στο παρελθόν στα πλαίσια αυτού του μαθήματος**,** όπως ορίζεται από τους κανονισμούς**.** Σκοπός της εργασίας είναι να δημιουργήσουμε ένα **Vantage point tree** και να το χρησιμοποιήσουμε για τον υπολογισμού **k** κοντινών γειτόνων**.** Καθώς η δημιουργία του προσφέροντος δυαδικού δένδρου μπορεί να αποδειχθεί αρκετά δαπανηρή για μεγάλο αριθμό σημείων**,** καλούμαστε να χρησιμοποιήσουμε διάφορες τεχνικές για την παραλληλοποίηση του**.**

***Index Terms*—OpenMP, MPI, Vantage Point tree, KNN**

I. Introduction

Ένα αρκετά γνωστό πρόβλημα της στατιστικής είναι η αναζήτηση k κοντινότερων γειτόνων ενός πολυδιάστατου σημείου από ένα σύνολο σημείων. Το πρόβλημα αυτό είναι χρήσιμο σε πληθόρα εφαρμογές όπως θεωρία γράφων [1], [2], γεννετική και βιολογία [3], [4], γενικά σε εφαρμογές μηχανικής μάθησης που έχουν να κάνουν με αναγνώριση προτύπων [5], [6]. Φαίνεται λοιπόν η αναγκαιότητα ο υπολογισμός των γειτόνων να γίνεται γρήγορα και αποτελεσματικά.

Για τον σκοπό αυτό, έχουν προταθεί διάφορες δομές δεδομένων που επιτρέπουν την γρήγορη αναζήτηση των γειτόνων. Χαρακτηριστικά παραδείγματα είναι τα kd-tree που δημιουργούμε δυαδικά δένδρα χωρίζοντας τον πολυδιάστατο χώρο με βάση μια διάσταση, τα R-tree που κατηγοριοποιουν κοντινά σημεία και έχουν αρκετά κοντινή μορφή με B-tree. Στην εργασία αυτή θα μας απασχολήσουν τα vantage-point tree, μια από τις πιο αποτελεσματικές δομές για την αναζήτηση k κοντινότερων γειτόνων.

Τα vantage-point tree είναι δυαδικά δέντρα που χωρίζουν τον πολυδιάστατο χώρο σε υποχώρους. Πιο συγκεκριμένα επιλέγεται ένα αρχικό σημείο γνωστό ως vantage-point, μετριούνται οι αποστάσεις από όλα τα άλλα και τα σημεία μοιράζονται στο αριστερό και δεξί υποδέντρο ανάλογα με τη ενδιάμεση απόσταση. Έτσι δημιουργούνται ισοσταθμισμένα δέντρα, στα οποία η αναζήτηση k κοντινότερων σημείων γίνεται αποτελεσματικά, όπως θα δούμε παρακάτω.

Το επόμενο θέμα που προκύπτει και αποτελεί καίριο κομμάτι της εργασία είναι η αποτελεσματική δημιουργία τέτοιων δυαδικών δέντρων. Για τον σκοπό αυτό καλούμαστε να υλοποιήσουμε εκτός από την σειριακή εκδοχή και ορισμένες παράλληλες με χρήση τόσο μιας CPU όσο και περισσότερων ή αξιοποίηση της GPU.

Στη συνέχεια της αναφορά παρουσιάζουμε τις υλοποιήσεις σειριακή και παράλληλες, πληροφορίες για το σύστημα πάνω στο οποίο αναπτύχθηκε και δοκιμάστηκε η εργασία, το κώδικα, τα tests και το dataset που χρησιμοποιήθηκε. Στο τέλος παρουσιάζονται και σχολιάζονται τα αποτελέσματα και οι συγκρίσεις των υλοποιήσεων.

II. Implementation

A. Serial version

Ας ξεκινήσουμε με την σειριακή εκδοχή του αλγορίθμου. Παρακάτω στο 1 φαίνεται ο βασικός αναδρομικός αλγόριθμος για τη δημιουργία του vantage-point tree. Ουσιαστικά δίνονται σαν είσοδος τα πολυδιάστατα σημεία από τα οποία θα δημιουργηθεί το δέντρο. Επιλέγεται ένα σημείο σαν vantage-point, και μετρούνται όλες οι αποστάσεις των υπολοίπων από αυτό. Έπειτα βρίσκεται η ενδιάμεση απόσταση και χωρίζονται τα σημεία με βάση την απόστασή τους, σε αυτά που έχουν μικρότερη ή ίση με την ενδιάμεση και σε αυτά που έχουν μεγαλύτερη της ενδιάμεσης. Τέλος δημιουργείται ένας κόμβος του δέντρου που περιέχει σαν πληροφορία την ενδιάμεση απόσταση και το δείκτη του vantage point, και για αριστερό και δεξί παιδί καλείται αναδρομικά ο αλγόριθμος δημιουργίας δέντρου αλλά αυτή τη φορά με σημεία τα δύο νέα σύνολα που προέκυψαν από το διαχωρισμό με την ενδιάμεση απόσταση.

Algorithm 1 Sequential VP-tree creation

Input: N points with dimension d A
Output: root to a VP-tree

```
if N == 0 then
    return NULL
vp = points[0]
points = points[1,end]
dists = calculate_distance(points,vp)
median = find_median(dists)
left,right = split_points(points,dists,median)
node = New_node(idx[vp],dist)
node.left = vp_create(left)
node.right = vp_create(right)
return node
```

Το vantage-point που εκλέγεται σε κάθε αναδρομή είναι το πρώτο. Ο υπολογισμός των αποστάσεων γίνεται με την ευκλείδεια απόσταση και η μέση απόσταση βρίσκεται με τη χρήση του quick-select αλγορίθμου. Επιπρόσθετα ο διαχωρισμός των αποστάσεων δεν δημιουργεί καινούργιους πίνακες με πολυδιάστατα σημεία, καθώς κάτι τέτοιο θα ήταν υπολογιστικά ασύμφορο για τη μνήμη. Αντιθέτως ο διαχωρισμός γίνεται με διανύσματα των δεικτών των σημείων στον αρχικό πίνακα και διαγράφονται απευθείας όταν δεν χρειάζονται πλέον. Με αυτό τον τρόπο αυξάνουμε μεν ελάχιστα τον εκτελέσιμο χρόνο μιας και απελευθερώνεται συχνά μνήμη, αλλά από την άλλη δεν έχουμε θέματα κορεσμού της μνήμης.

B. Parallel version

Η παράλληλη υλοποίηση λειτουργεί με τον ίδιο τρόπο με την σειριακή, αλλά με μερικές αλλαγές προκειμένου να

έχουμε παραλληλοποίηση με την χρήση OpenMP. Αρχικά, χρησιμοποιείται η συνάρτηση calculateDistancesParallel στην θέση της calculateDistancesParallel. Η μόνη διαφορά μεταξύ τω δύο είναι η προσθήκη της γραμμής "#pragma omp parallel for" στην πρώτη ώστε ο βρόχος for να εκτελεστεί παράλληλα. Στην συνέχεια, προστέθηκαν οι εντολές "#pragma omp parallel", "#pragma omp sections nowait" και "#pragma omp section", προκειμένου να τρέξουμε τις αναδρομικές κλήσεις της συνάρτησης παράλληλα.

C. Mixed version

Η παρούσα υλοποίηση εναλλάσει μεταξύ παράλληλων και σειριακών αναδρομικών κλήσεων για να επιτευχθούν βέλτιστες επιδόσεις. Η παράλληλη υλοποίηση έχει καλύτερες επιδόσεις από την σειριακή μόνο για επαρκώς μεγάλο αριθμό σημείων. Καλούμε αναδρομικά τις συναρτήσεις "*_vp_create" χωρίζοντας το πρόβλημα σε όλο και μικρότερα κομμάτια. Συνεπώς, ακόμη και αν το πρόβλημα είναι αρχικά επαρκώς μεγάλο για να επωφεληθεί από την παραλληλοποίηση, μετά από έναν αριθμό αναδρομικών κλήσεων, είναι προτιμότερη η σειριακή εκτέλεση του κώδικα. Πάνω σε αυτό βασίζεται η συγκεκριμένη υλοποίηση. Με την ενός ελέγχου που συγκρίνει το μέγεθος του προβλήματος με ένα κατωφλι που προέκυψε από benchmarks, επιλέγεται αν η εκάστοτε αναδρομική κλήση θα εκτελεστεί παράλληλα ή σειριακά, ώστε να έχουμε καλύτερες επιδόσεις.

Επιπλέον, σε αυτή την υλοποίηση περιορίζεται και ο μέγιστος αριθμός ενεργών thread που δημιουργούνται καθώς ο ανεξέλεγκτος αριθμός ενεργών thread μπορεί να δημιουργήσει σοβαρή καθυστέρηση στο σύστημα ή και σφάλμα εκτέλεσης αν ξεπεραστεί το όριο threads που μπορεί να διαχειριστεί το σύστημα. Ο περιορισμός γίνεται με μια μεταβλητή που μετράει τα παράλληλα recursion και όταν ξεπεράσουν ένα όριο επιλέγεται σειριακή εκτέλεση.

Για να εντοπιστούν τα όρια εναλλαγής από παράλληλο σε σειριακό εξετάζουμε τα plots με αυξανόμενο αριθμό σημείων, διάσταση σημείων και γινομένου αυτού και βλέπουμε πότε το σειριακό υπερτερεί. Για τον ανώτερο αριθμό ενεργών thread δοκιμάζουμε διάφορες τιμές παρακάτω στο εδάφιο των αποτελεσμάτων και εξετάζουμε την επιρροή του ορίου.

D. Hybrid MPI version

Σαν τελευταίο υποερώτημα όσον αφορά τη δημιουργία των vantage-point trees μας ζητούνταν να διευρύνουμε την υλοποίησή μας σε περισσότερες CPU με χρήση Message Passing Interface (MPI) ή GPU. Αν και μελετήσαμε πιθανές υλοποιήσεις και με τους δύο τρόπους, μας φάνηκε πιο πιθανό να έχουμε σημαντική βελτίωση χρόνου με MPI. Κι αυτό γιατί η υλοποίηση δημιουργίας του VP δέντρου είναι αναδρομική και αναδρομικές κλήσεις είναι αρκετά περιορισμένες και δεν ενδείκνυται για Nvidia GPUS. Καθώς σκοπός μας ήταν να υλοποιήσουμε την εφαρμογή ώστε να τρέχει σε ένα εύρος συστημάτων αν θέλαμε να χρησιμοποιήσουμε τη GPU θα τη χρησιμοποιούμε μόνο για τον υπολογισμό των αποστάσεων.

Μια άλλη εκδοχή αν θέλαμε να χρησιμοποιήσουμε GPU θα ήταν να κάνουμε τον αλγόριθμο επαναληπτικό. Αυτό βέβαια έχει άλλα μειωνεκτήματα όπως ότι θα έπρεπε να υλοποιήσουμε ένα είδος stack που να αποθηκεύει τα node όδο διατρέχουμε το δέντρο. Σε τέτοια υλοποίηση ο αριθμός των αποτυχημένων αναζητήσεων στη cache των kernels θα ήταν μεγάλος και ουσιαστικά θα αξιοποιούνταν αποκλειστικά η global μνήμη που έχει σημαντικά μεγαλύτερο delay.

Από την άλλη η MPI υλοποίηση μας επιτρέπει να αξιοποιήσουμε και άλλες παράλληλες μέθοδοι όπως OpenMP και να λάβουμε τα πλεονεκτήματα μιας παράλληλης και διανεμημένης εφαρμογής. Η υλοποίηση που σκεφτήκαμε είναι να ξεκινούν δύο processes και στον πρώτο διαχωρισμό των σημείων από τον leader οι δεξιά δείκτες να μεταφέρονται στο δεύτερο process. Στη συνέχεια εφαρμόζεται η mixed υλοποίηση OpenMP με εναλλαγή σε σειριακό όπου χρειάζεται. Στο τέλος ενώνονται στο leader αριστερό και δεξί υποδέντρο.

Η μεταφορές μνήμης μεταξύ των διεργασιών ουσιαστικά είναι στην αρχή οι δείκτες, δηλαδή ένα αριθμός απλών int32. Ενώ στο τέλος μεταφέρεται το δέντρο σε preorder μορφή. Αυτό είναι λίγο πιο δαπανηρό μιας και μεταφέρονται και τα thresshold και οι δείκτες του κάθε κόμβου, άρα ουσιαστικά μια σειρά από int32 και μια από float32. Εδώ κοστολογείται και ο μετασχηματισμός από δέντρο σε preorder και από preorder σε δέντρο.

E. KNN search

Η εκφώνηση της εργασία ζητάει και την υλοποίηση του KNN αλγορύθμου. Συγκεκριμένα έχοντας σαν είσοδο το VP-tree θα πρέπει να βρίσκουμε τους k κοντινότερους γείτονες. Μία γενική μορφή της υλοποίησης που επιλέξαμε μπορείτε να δείτε στο 2.

Algorithm 2 KNN algorithm

Input: VP-tree node, d-D point, int K, threshold τ, Queue q

1: **if** $node == NULL$ **then**
2: return
3: $dist = calculate\_distance(node.point, point)$
//if it is closer than τ add the current node in queue
4: **if** $dist < \tau$ **then**
5: **if** $q.size() == k$ **then**
6: $q.pop()$
7: $q.push(point, dist)$
8: **if** $q.size() == k$ **then**
9: $\tau = q.top().dist$
10: **if** $node.left == NULL$ and $node.right == NULL$ **then**
11: return
//recursive call of knn search
12: **if** $dist < node.threshold$ **then**
13: **if** $dist - \tau \leq node.threshold$ **then**
14: $knn\_search(node.left, point, K, \tau, q)$
15: **if** $dist + \tau \geq node.threshold$ **then**
16: $knn\_search(node.right, point, K, \tau, q)$
17: **else**
18: **if** $dist + \tau \geq node.threshold$ **then**
19: $knn\_search(node.right, point, K, \tau, q)$
20: **if** $dist - \tau \leq node.threshold$ **then**
21: $knn\_search(node.left, point, K, \tau, q)$

Στα περισσότερα ερευνητικά έργα όπως [8] δείνεται έμφαση στην αναζήτηση vp-tree μόνο του πρώτου γείτονα. Για να καταστεί δυνατό να βρεθούν παραπάνω από ένα γείτονα θεωρήθηκε απαραίτητη η υλοποίηση και η χρήση μιας ουράς προτεραιότητας, όπου σημεία με μεγαλύτερη απόσταση βγαίνουν από την ουρά σε περίπτωση που γεμίσει. Η υλοποίησή της έγινε με συνδεμένες λίστες και πολύτιμη βοήθεια στην υλοποίηση τους υπήρξε το [9].

Όσο αναφορά τον αλγόριθμο αξιοποίησαμε τον αλγόριθμο που παρεχόταν στο [8] και το διευρύναμε για περισσότερους γείτονες. Παρόλα αυτά δόθηκε ιδιαίτερη προσοχή ώστε να γίνονται ακριβώς οι απαραίτητες αναδρομικές κλήσεις και να πάρουμε τον ελάχιστο εκτελέσιμο χρόνο (κυρίως στη γραμμή 12 του αλγορίθμου). Το όριο τ αρχικοποιείται σε μια πολύ μεγάλη τιμή προκειμένου το εύρος αναζήτησης να είναι μεγάλο και σταδικά αυτό μικραίνει.

Στην εκφώνηση δεν ζητούνταν κάποια παραλληλοποίηση αυτού του αγλορίθμου. Παρόλα αυτά καθώς ζητούσε να

βρίσκει τους γείτονες απόλα τα σημεία και ουσιαστηκά τρέχει ο παραπάνω αλγόριθμος για κάθε σημείο, αποφασίσαμε να χρησιμοποιήσουμε OpenMP με dynamic scheduler που μοιράζει δίκαια το φόρτο εργασίας και εφρμόζει work stealing τεχνικές. Η παραλληλοποίηση αυτή ήταν προαιρετική αλλά αναγκαία καθώς για μεγάλα dataset ήταν μια δαπανειρή διαδικασία.

F. Code and Validation

Η υλοποίηση έγινε σε C. Σαν building tool προτιμήθηκε το CMake, αντί χαμηλότερου επιπέδου Makefile, καθώς είναι cross-platform. Για την επαλήθευση ορθότητας όσον αφορά τη δημιουργία των δένδρων δημιουργήθηκε ο κώδικας πρώτα σε python και εξετάστηκε αν παράγει παρόμοια αποτελέσματα με αυτόν σε Matlab που μας δόθηκε. Στη συνέχεια δημιουργήθηκε σε C η σειριακή μορφή και έγινε έλεγχος για διάφορα dataset αν παράγει ίδια αποτελέσματα με την Python υλοποίηση. Για γρηγορότερο έλεγχο αποθηκεύεται και εξετάζονταν μόνο η προδιατεταγμένη μορφή των δέντρων καθώς εφόσων αποθηκεύεται και η κενή θέση αποτελεί μια $1-1$ απεικόνιση του δέντρου.

Εφόσον επιβεβαιώθηκε η ορθότητα του σειριακού κώδικα για διάφορο αριθμό σημείων, δημιουργήθηκαν οι παράλληλοι κώδικες και εξετάζεται αν παράγουν σε κάθε δοκιμή ίδιο αποτέλεσμα με το σειριακό.

Επιπλέον για την επιβεβαίωση των βασικών συναρτήσεων χρησιμοποιήθηκαν και Unit Tests με το repo της Google [7]. Η τεχνική που ακολουθήθηκε είναι black box.

G. Dataset

Χρησιμοποιήσαμε πληθώρα δεδομένων με διαφορετικό αριθμό σημείων, αριθμό διάστασης και κατανομή. Πιο συγκεκριμένα ο αριθμός σημείων κυμαίνεται από 10 εώς και 60000, ο αριθμός διάστασης από 10 εώς 40000 και οι κατανομές που χρησιμοποιήθηκαν είναι ομοιόμορφη, εκθετική, κανονική αλλά και πολυδιάστατα σημεία παραγόμενα από το dataset mnist.

Ορισμένα δείγματα για testing μπορούν να κατέβουν από συνδέσμους που παρέχονται στο *README.md* του φακέλου data.

III. Results

Όλα τα αποτελέσματα πάρθηκαν από τη συστοιχία του ΑΠΘ, προκειμένου να συγκριθούν πάνω σε κοινή βάση με τις εργασίες των υπόλοιπων συμφοιτητών. Σε όλα τα πειράματα χρησιμοποιήθηκαν 18 cores και στην MPI υλοποίηση μόνο 2 nodes των 18 cores το καθένα.

Για κάθε πείραμα εκτός από ένα μεγάλο αριθμό δεδομένων, επαναλαμβάνουμε το testing για κάθε dataset ξεχωριστά και παίρνουμε μία μέση μέτρηση. Αυτό σε συνδυασμό με ορισμένες δοκιμαστικές εκτελέσεις πριν την βασική χρονομέτρηση, γίνεται για να έχουμε μια αντικειμενική εικόνα του εκτελέσιμου χρόνου και να μην επηρεάζονται τα αποτελέσματά μας από προσωρινά φαινόμενα όπως cold cache.

Για τα αποτελέσματα θα ξεκινήσουμε με την σύγκριση του σειρικού με του απλού παράλληλου. Αρχικά για να εξετάσουμε την επίδραση της κατανομής θα χρησιμοποιήσουμε τρεις κατανομές, ομοιόμορφη, κανονική και εκθετική. Επίσης θα εξετάσουμε τη επιρροή του αριθμού των σημείων και της διάστασης των σημείων στον εκτελέσιμο χρόνο.

Όπως βλέπουμε στα σχήματα 1 και 2 υπάρχει σαφής επικράτηση του παράλληλου έναντι του σειριακού για αυξανόμενο αριθμό σημείων. Επίσης γίνεται εύκολα αντιληπτό ότι η κατανομή των σημείων δεν έχει άμεση επιρροή στον χρόνο εκτέλεσης, αν και για το παράλληλο βλέπουμε μια μείωση χρόνου στην

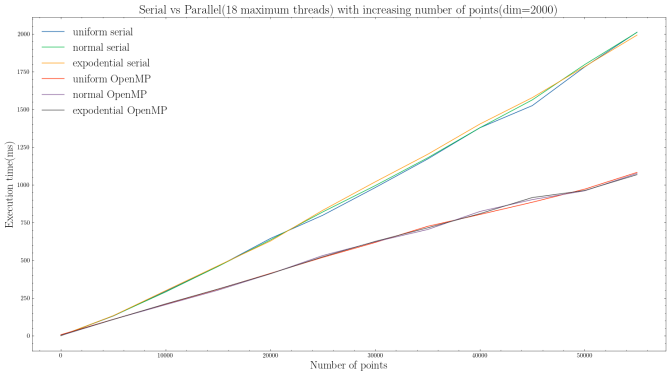

Figure 1. Comparison of serial and simple parallel in different distributions and increasing number of points

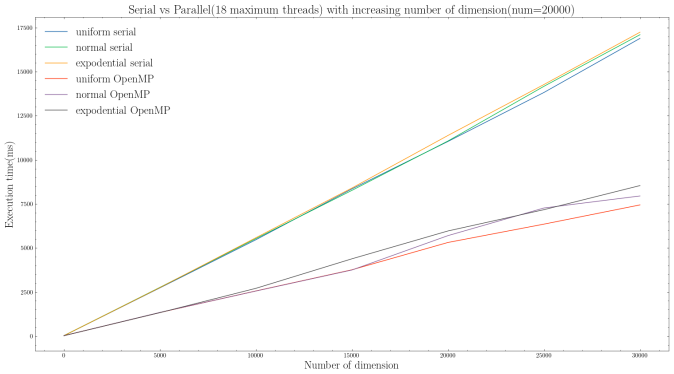

Figure 2. Comparison of serial and simple parallel in different distributions and increasing dimension of points

ομοιόμορφη κατανομή. Τέλος αν συγκρίνουμε την κλήση των δύο αυξήσεων συμπεραίνουμε ότι η διάσταση των σημείων έχει μεγαλύτερη επιρροή από την αύξηση των σημείων.

Στη συνέχεια θα εξετάσουμε την δεύτερη παράλληλη υλοποίηση που στην ουσία εναλλάσουμε σειριακές και παράλληλες αναδρομές με βάση τα σημεία και περιορίζουμε τον μέγιστο αριθμό των ενεργών threads. Για να βρούμε το βέλτιστο σημείο εναλλαγής χρησιμοποιούμε τον γινόμενο αριθμού σημείων και διάστασης μιας και τα δύο έχουν επιρροή όπως είδαμε προηγουμένως. Παρατηρούμε στο 3 ότι το σημείο βρίσκεται κοντά στο γινόμενο με αριθμο 13500 επομένως ορίζουμε αυτό ως σημείο εναλλαγής.

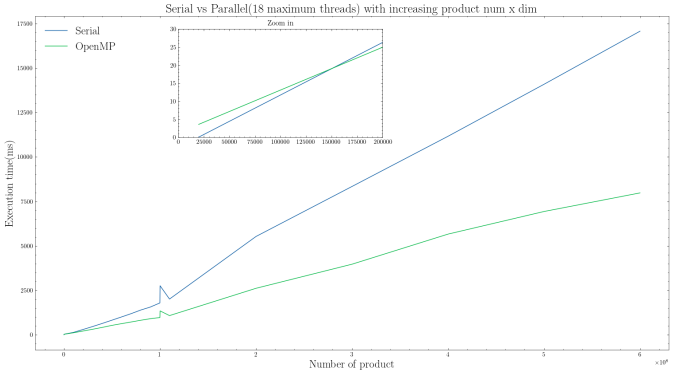

Figure 3. Parallel implementation with increasing product of number and dimension of points

Για όριο ενεργών threads δοκιμάζουμε ένα μεγάλο εύρος τιμών και βλέπουμε την επίδραση που έχει σε τρια dataset διαφορετικών διαστάσεων. Στο 4 παρατηρούμε ότι όλα τα dataset έχουν βέλτιστη τιμή το 4 οπότε προχωράμε με αυτή τη τιμή. Πιθανότατα για αρκετά μεγαλύτερο αριθμό σημείων ή σύστημα με περισσότερους πραγματικούς πυρήνες αυτή η τιμή να ήταν μεγαλύτερη.

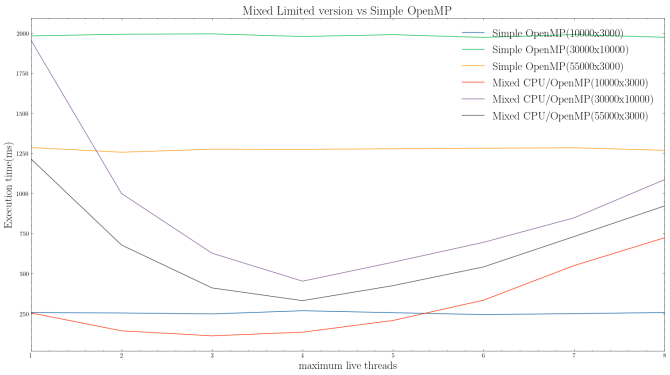

Figure 4. Tuning the maximum number of live threads

Στη συνέχεια βλέπουμε τη σύγκριση όλων των εκδοχών για την δημιουργία vanta_point δέντρων σε 5 dataset με ποικιλομορφία στον αριθμό των σημείων και της διάστασης.

Table I
BEST AVERAGE EXECUTION TIMES(MS) COMPARISONS ON DIFFERENT DATASETS ALONGSIDE WITH THE BEST SPEEDUP

Dataset	Sequential	Simple OpenMP	Limited OpenMP	Hybrid MPI	SpeedUp
800x500	**7.513**	8.1114	13.74	25.47	1
10000x2000	318.792	213.8572	**142.5**	143.9814	2.24
30000x5000	2040.361	1226.187	299.9248	**260.965**	7.82
20000x20000	4991.596	2635.13	591.3892	**401.258**	12.44
50000x20000	14062.98	7387.35	1437.052	**871.81**	16.13

Στο I βλέπουμε μια εικόνα της συνολικής σύγκρισης των διαφόρων μεθόδων. Σε πολύ μικρά dataset δεν ωφελεί καθόλου η παραλληλοποίηση, ενώ βλέπουμε ότι η περιορισμένη εκδοχή του OpenMP που εναλλάσει σε σειριακή για μικρό αριθμό σημείων είναι αρκετά πιο αποδοτική από την απλή. Σε γενικές γραμμές όμως η τρίτη υλοποίηση με τη χρήση ενός επιπλέον CPU και τη χρήση του MPI είναι λίγο καλύτερη από την περιορισμένη OpenMP σε μέτριου μεγέθους dataset αλλά σε αρκετά μεγάλα dataset βλέπουμε ότι η επιτάχυνση της ξεφεύγει.

Τέλος εξετάζουμε την αναζήτηση knn σε διάφορα dataset στο 5. Εδώ για εκπαιδευτικούς σκοπούς γίνεται σύγκριση με τον αλγόριθμο που χρησιμοποιεί το sklearn της python για την αναζήτηση των γειτόνων όπως και της C υλοποίησης με και χωρίς χρήση OpenMP. Συγκεκριμένα αναζητούνται max 256 γείτονιες για κάθε σημείο ξεχωριστά όπως αναφέρεται και στην εκφώνηση σε δεδομένα με διαφορετικό αριθμό σημείων.

Μπορούμε να πούμε ότι και η αναζήτηση σε KD tree που χρησιμοποιεί η python είναι αποτελεσματική, και ενδέχεται να χρησιμοποιούνται παράλληλες υπορουτίνες εσωτερικά. Όμως με χρήση απλής OpenMP ρουτίνας με δυναμικό προγραμματιστή σε C, που μοιράζει δίκαια τη εργασίες και εφαρμόζει τεχνικές work stealing, της αναζήτησης KNN σε Vantage Point-trees είναι αρκετά καλύτερη.

IV. Conclusion

Κλείνοντας θα λέγαμε ότι τα vantage-point δέντρα αποτελούν μια ιδιαίτερα χρήσιμη δομή για την αναζήτηση κοντινών γειτόνων. Σε μεγάλα σύνολα σημείων με πολλές διαστάσεις η δημιουργία

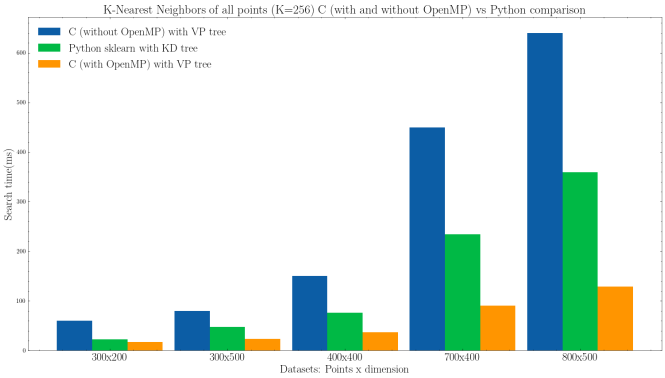

Figure 5. C's implementation of KNN with and without OpenMP and sklearn's in Python

τέτοιων δέντρων μπορεί να αποτελέσει μια αρκετά δαπανηρή διαδικασία και η παραλληλοποιηση της κρίνεται αναγκαία. Έγινε αρκετά ευδιάκριτο ότι άκριτη παραλληλοποίηση δεν ωφελεί και για μικρό αριθμό σημείων η σειριακή υλοποίηση υπερτερεί. Ένω για πολύ μεγάλα dataset μια μονάχα CPU ίσως δεν είναι αρκετή να δώσει ικανοποιητικά αποτελέσματα και κρίνεται αναγκαίο να διαμειράσουμε την εργασία σε περισσότερου υπολογιστικούς πόρους.

Acknowledgment

Τα αποτελέσματα που παρουσιάζονται έχουν παραχθεί αξιοποιώντας την Υπολογιστική Συστοιχία και τις παρεχόμενες υπηρεσίες υποστήριξης του Κέντρου Ηλεκτρονικής Διακυβέρνησης του Α.Π.Θ..

References

[1] P. Balister and B. Bollob as Percolation in the k-nearest neighbor graph
[2] Wei Dong, Moses Charika, Kai Li, K-Nearest Neighbor Graph Construction for Generic Similarity Measures, Department of Computer Science, Princeton University
[3] Dang, Yan Zhang, Yulei Zhang, Dongmo Zhao, Liping. (2005). A KNN-Based Learning Method for Biology Species Categorization. Lecture Notes in Computer Science. 3610. 956-964. 10.1007/11539087_127.
[4] Parry, R., Jones, W., Stokes, T. et al. k-Nearest neighbor models for microarray gene expression analysis and clinical outcome prediction. Pharmacogenomics J 10, 292–309 (2010). https://doi.org/10.1038/tpj.2010.56
[5] Guo, Gongde Wang, Hui Bell, David Bi, Yaxin. (2004). KNN Model-Based Approach in Classification.
[6] K. Taunk, S. De, S. Verma and A. Swetapadma, "A Brief Review of Nearest Neighbor Algorithm for Learning and Classification," 2019 International Conference on Intelligent Computing and Control Systems (ICCS), 2019, pp. 1255-1260, doi: 10.1109/ICCS45141.2019.9065747.
[7] Google tests: https://github.com/google/googletest
[8] Peter N Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, In Fourth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics Philadelphia, volume 93, pages 311–321, 1993.
[9] Priority Queue using Linked List: https://www.geeksforgeeks.org/priority-queue-using-linked-list/