Θεόφιλος Σπύρου ΑΕΜ: 8583

ΠΑΡΑΛΛΗΛΟΠΟΙΗΣΗ ΑΛΓΟΡΙΘΜΟΥ KNN

Παράλληλα και διανεμημένα συστήματα – 7ο Εξάμηνο

Ανάλυση προβλήματος

Ο αλγόριθμος kNN έχει σαν στόχο την εύρεση των k κοντινότερων γειτόνων κάθε στοιχείου του συνόλου δεδομένων του. Για να το πετύχει αυτό αρχικά υπολογίζει για κάθε ένα από τα σημεία την απόστασή του με όλα τα άλλα και αποθηκεύει τις k μικρότερες από αυτές καθώς και σε αντίστοιχο πίνακα τους δείκτες των στοιχείων για τα οποία έχουν υπολογιστεί οι συγκεκριμένες αποστάσεις. Αφού ολοκληρωθεί η παραπάνω διαδικασία πρέπει να αποφανθούμε για την κλάση στην οποία ανήκει το κάθε σημείο. Αντιστοιχίζουμε λοιπόν τους γείτονες τού κάθε σημείου με μία κλάση σύμφωνα με το αρχείο των labels που μας έχει δοθεί. Έπειτα, ορίζουμε την κλάση ενός σημείου ανάλογα με την πλειοψηφία των κλάσεων των γειτόνων του. Στην τελική φάση μπορούμε να δούμε για πόσα στοιχεία οι κλάσεις που υπολογίσαμε συμφωνούν με τις σωστές από το αρχικό αρχείο, ώστε να βρούμε το ποσοστό επιτυχίας του αλγορίθμου. Με άλλα λόγια, μπορούμε να πούμε ότι ο kNN χωρίζεται σε τρία απλά βήματα: 1) Εύρεση k κοντινότερων γειτόνων κάθε σημείου, 2) αντιστοίχιση κάθε γείτονα στην ετικέτα του και 3) υπολογισμός ετικέτας κάθε σημείου και έλεγχος αποτελεσμάτων.

Πρώτες σκέψεις για παραλληλοποίηση

Δεδομένης της φύσης του αλγορίθμου και του ότι όλα τα στοιχεία είναι ανεξάρτητα από τα υπόλοιπα, μπορούμε να χωρίσουμε το σύνολο των δεδομένων μας ανάλογα με τις διεργασίες που έχουμε διαθέσιμες και κάθε μία από αυτές να αναλάβει ένα κομμάτι του συνολικού προβλήματος. Έτσι, αφού γίνει ο διαχωρισμός, κάθε διεργασία εκτελεί τον kNN για τα σημεία που της αντιστοιχούν με τον εαυτό τους και έπειτα μεταξύ αυτών και των δεδομένων από κάθε άλλη διεργασία, ανανεώνοντας κάθε φορά τους κοντινότερους γείτονες για τα δικά της στοιχεία. Μόλις τελειώσουν όλοι οι kNN, με παρόμοιο τρόπο γίνεται η αντιστοίχιση των κλάσεων, δηλαδή πρώτα για όσους γείτονες ανήκουν στο τρέχον block δεδομένων και στη συνέχεια διαδοχικά για αυτούς από όλα τα άλλα blocks των υπόλοιπων διεργασιών. Οι διεργασίες λοιπόν επικοινωνούν σε τοπολογία δακτυλίου για να στείλουν και να λάβουν σημεία.

Παραδοχές & Σημειώσεις

Καθώς η ανάγνωση δεδομένων από αρχείο mat είναι πολύπλοκη και απαιτεί βιβλιοθήκες που ενδεχομένως να μην είναι διαθέσιμες σε κάποια συστήματα, τα δεδομένα των δύο αρχείων της άσκησης αποθηκεύτηκαν σε μορφή απλού κειμένου txt · ένα για κάθε μεταβλητή MATLAB στην πρώτη σειρά των οποίων αναγράφονται οι διαστάσεις

Code Link: https://github.com/TheoSpyrou/Parallila/tree/master/Ergasia%202

του πίνακα που περιέχουν. Ακόμη τα paths των αρχείων αυτών εισάγονται σαν σταθερές στο πρόγραμμα, οπότε για να υπολογίσουμε τους k γείτονες για άλλα σημεία πρέπει να ξαναχτιστεί το πρόγραμμα με τα σωστά paths των νέων αρχείων. Ακόμη, επειδή προφανώς για κάθε σημείο το κοντινότερό του είναι το ίδιο το σημείο, η συνάρτηση knnSearch (που λειτουργεί με παρόμοιο τρόπο όπως η αντίστοιχη του MATLAB) πρέπει να τρέξει για k+1 γείτονες, ώστε μετά να αφαιρέσουμε τον πρώτο και να μείνουν οι k. Τέλος, τα αποτελέσματα του αλγορίθμου είναι επαληθευμένα με το script που δίνεται μαζί με τα δεδομένα της εργασίας (ποσοστό επιτυχίας για mnist_train.mat 92.63 %).

Παραλληλοποίηση με MPI

- **Blocking υλοποίηση:**

Για την blocking υλοποίηση του παράλληλου αλγορίθμου χρησιμοποιήθηκε η συνάρτηση `MPI_Sendrecv`. Με τη συγκεκριμένη συνάρτηση η ροή του προγράμματος μπλοκάρει έως ότου σταλθούν όλα τα δεδομένα του send buffer και ληφθούν όλα τα ζητούμενα δεδομένα στο receive buffer με tag από το source. Με τον τρόπο αυτό, αποφεύγονται πιθανά deadlocks που θα μπορούσαν να προκύψουν λόγω της ακανόνιστης σειράς αποστολής μεταξύ των διεργασιών. Επίσης με τον τρόπο αυτό η ροή μπλοκάρεται μόνο μία φορά και όχι δύο (μία για send και μία για receive).

Ξεκινώντας, το πρόγραμμα αρχικοποιεί όλες τις απαραίτητες παραμέτρους για τη σωστή λειτουργία του MPI. Έπειτα ξεκινάει η διαδικασία όπως περιγράφθηκε παραπάνω για κάθε task. Αφού ολοκληρώσουν όλες οι διεργασίες, χρησιμοποιείται η συνάρτηση MPI_Reduce με όρισμα MPI_SUM για τον υπολογισμό του ολικού πλήθους των σημείων που η κλάση τους ορίστηκε σωστά. Τέλος, σαν χρόνος του αλγορίθμου θεωρείται ο μέγιστος χρόνος που χρειάστηκε για την εκτέλεση του knnSearch από τις διεργασίες και προκύπτει μέσω της συνάρτησης MPI_Reduce, αλλά με όρισμα MPI_MAX. Δεν υπολογίζεται δηλαδή ο χρόνος για την ανταλλαγή των δεδομένων, ο χρόνος για τον υπολογισμό των ετικετών και αυτός για την επαλήθευση των αποτελεσμάτων.

- **Non-Blocking υλοποίηση:**

Σε αυτήν την περίπτωση η ροή του προγράμματος δεν μπλοκάρεται κάθε φορά που μία διεργασία στέλνει ή λαμβάνει δεδομένα. Για να επιτευχθεί αυτό χρησιμοποιούνται οι συναρτήσεις MPI_Isend και MPI_Irecv, οι οποίες ξεκινούν την αποστολή και τη λήψη αντίστοιχα των μηνυμάτων ασύγχρονα από το υπόλοιπο πρόγραμμα. Η πρώτη καλείται νωρίτερα από τη δεύτερη και στο ενδιάμεσο των δύο αυτών κλήσεων εκτελείται μέρος του αλγορίθμου που δε χρειάζεται τα συγκεκριμένα δεδομένα, ώστε να «κρυφτεί» η καθυστέρηση της αποστολής (hide network latency).

Ο αλγόριθμος λειτουργεί με τον ίδιο τρόπο με πριν, μόνο που αυτή τη φορά η τοπολογία δακτυλίου δεν πραγματοποιείται απαραίτητα με τη σειρά των διεργασιών, αλλά μπορεί να ανακατευθεί λόγω της ακανόνιστης εκτέλεσης των διεργασιών.

Αξιοσημείωτο εδώ είναι το σημείο όπου αναγκαστικά πρέπει να περιμένουμε να τελειώσει κάποια λήψη δεδομένων προτού ξεκινήσουμε την επεξεργασία τους (MPI_Waitany). Αυτό διότι πρακτικά η non-blocking υλοποίηση δεν είναι και τόσο non-blocking, μιας και πρέπει να ληφθεί ολόκληρο το buffer για να προχωρήσουμε! Μία λύση θα ήταν ο διαχωρισμός των προς αποστολή στοιχείων σε μικρότερα κομμάτια.

Ακόμα μία διαφορά σε σχέση με τον blocking αλγόριθμο είναι η διαχείριση της μνήμης, όπου υποχρεωτικά δημιουργούμε NumTasks-1 buffers για τη λήψη των blocks, διότι κάθε κλήση της MPI_Recv αποθηκεύει σε ξεχωριστό buffer. Επομένως, μπορεί να γλυτώσαμε σημαντικά στο χρόνο εκτέλεσης (ειδικά με το κρύψιμο του network latency), αλλά χάσαμε αρκετά σε μνήμη.

Μετρήσεις χρόνων εκτέλεσης (HellasGrid)

Ακολουθεί ενδεικτικό batch file όπου τρέχουμε τον αλγόριθμο σε 1 node με 4 πυρήνες:

```
#!/bin/bash
#PBS -N knn_mpi
#PBS -q pdlab
#PBS -j oe
#PBS -l walltime=1:00:00
#PBS -l nodes=1:ppn=4

cd $PBS_O_WORKDIR

module load gcc
module load openmpi

mpicc knn_mpi.c -o knn_mpi -O3 -lm
echo "Tasks #: 1x4" > results_1x4.txt
mpirun knn_mpi 30 -np 4 >> results_1x4.txt
```

Όμοια έτρεξαν στο cluster του HellasGrid scripts με συνδυασμούς: 1x2, 1x4, 1x8, 2x1, 2x2, 2x4. Για μεγαλύτερο πλήθος διεργασιών δυστυχώς δεν υπήρχαν διαθέσιμα slots τη στιγμή που τα έτρεξα. Τα διαγράμματα των χρόνων εκτέλεσης παρουσιάζονται στα σχήματα 1 και 2 στο τέλος της αναφοράς.

Παρατηρήσεις και συμπεράσματα

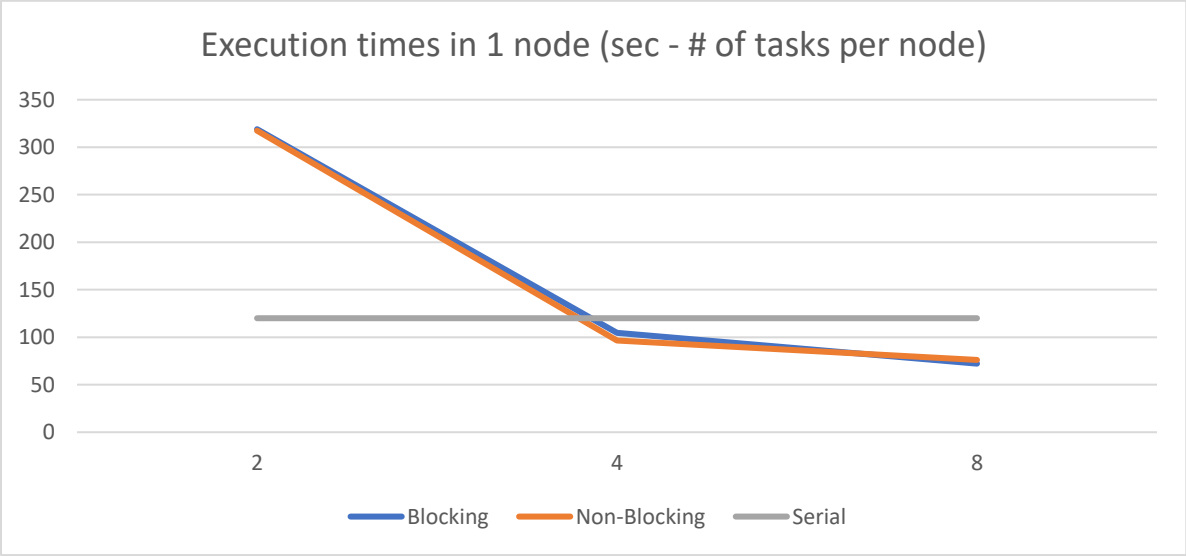
Οι παρακάτω μετρήσεις αφορούν το αρχείο mnist_train.mat το οποίο περιέχει ένα dataset με 10.000 στοιχεία 784 διαστάσεων. Παρατηρούμε πως καθώς αυξάνει το πλήθος των διεργασιών ο απαιτούμενος χρόνος μειώνεται όπως περιμέναμε. Για ένα πλήθος tasks και πάνω ο παράλληλος αλγόριθμος ξεπερνά τον σειριακό, ενώ σε αυτές που ο παράλληλος είναι πιο αργός ευθύνεται κυρίως η υψηλή χρήση του cluster τη στιγμή των μετρήσεων. Αξιοσημείωτο είναι ακόμη το γεγονός ότι όταν

χρησιμοποιήθηκαν 2 από 4 tasks που ζητήθηκαν, παρουσιάσθηκαν πολύ καλύτερα αποτελέσματα απ' ότι στην περίπτωση που ζητάμε μόνο 2 tasks.

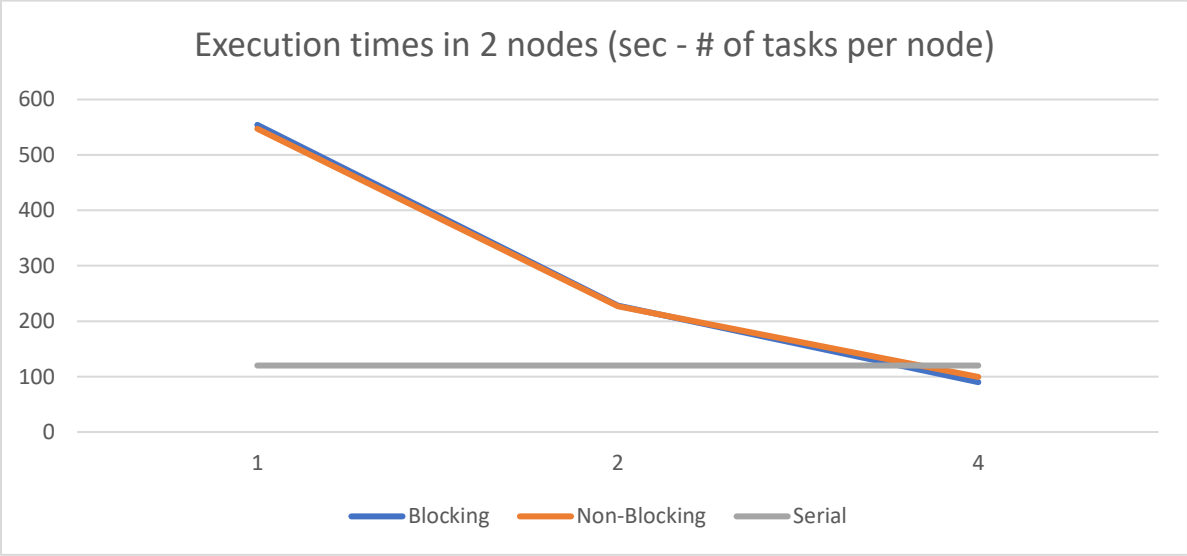
Για ένα πιο «έγκυρο» αποτέλεσμα (λόγω της υπερφόρτωσης του cluster τη δεδομένη στιγμή που έγιναν οι μετρήσεις) παραθέτω τις μετρήσεις από το σύστημά μου για 4 πυρήνες:

Serial	Blocking	Non-Blocking
140 secs	45 secs	44 secs

Είναι λογικό, τέλος, οι blocking και non-blocking αλγόριθμοι να έχουν παραπλήσιους χρόνους μιας και μετράμε μόνο τον χρόνο που κάνει ο kNN χωρίς την αποστολή και λήψη των δεδομένων.



Σχήμα 1. *Διάγραμμα χρόνων για έναν υπολογιστικό κόμβο*



Σχήμα 2. *Διάγραμμα χρόνων για δύο υπολογιστικούς κόμβους*

Code Link: https://github.com/TheoSpyrou/Parallila/tree/master/Ergasia%202