

Παράλληλα & Διανεμημένα Συστήματα

2^η Εργασία

Ραφαήλ Μπουλογεώργος 9186

rafampou@ece.auth.gr

Κυριακή 1/12/2019

LINK GITHUB

Ο κώδικας στο ζητούμενο code.tat.gz αρχείο έχει ανεβεί στο github στο link

<https://github.com/rafampou/mpi/raw/master/code.tar.gz>

Η εργασία μαζί με την αναφορά και τον tester έχει ανέβει στο github στο link

<https://github.com/rafampou/mpi>

Η τελική αναφορά θα βρίσκεται στο 2^ο [link](#) με όνομα PDS_9186_ex2_report.pdf.

ΕΙΣΑΓΩΓΗ

Σκοπός της εργασίας είναι η δημιουργία αλγορίθμου εύρεσης των k κοντινότερων σημείων από κάθε σημείο x που ανήκουν στον χώρο X , με την χρήση της γλώσσας C.

Ο αλγόριθμος θα γίνεται αρχικά σειριακά για όλα τα σημεία και θα πραγματοποιείτε από μία διεργασία και έπειτα ο τμηματικά και παράλληλα με την χρήση MPI.

Η χρήση των MPI (Message Passing Interface) επιτρέπει την ανταλλαγή μηνυμάτων και επεξεργασία δεδομένων από παράλληλες διεργασίες που δεν εκτελούνται απαραίτητα στον ίδιο υπολογιστή. Ο σκοπός της χρήσης τους είναι η χρήση μνήμης διαφορετικών αρχιτεκτονικών για την εύρεση των knn. Η χρήση των MPI επιτρέπει την παράλληλοποίηση του προγράμματος και την επικοινωνία υπολογιστών ακόμα και σε υπολογιστές με τελείως διαφορετική αρχιτεκτονική.

ΣΥΝΤΟΜΗ ΕΠΕΞΗΓΗΣΗ ΚΩΔΙΚΑ

Σειριακή υλοποίηση

Στο αρχείο *src/knnring_sequential.c* βρίσκεται ο κώδικας για το πρώτο μέρος την εργασίας (V0).

Η ζητούμενη συνάρτηση με τα δοσμένα ορίσματα είναι και η βασική συνάρτηση εύρεσης των k κοντινότερων γειτόνων (knn). Η συνάρτηση αυτή εκτελεί των υπολογισμών αποστάσεων σύμφωνα με τον παρακάτω κώδικα Matlab $D = \text{sqrt}(\text{sum}(X^2, 2) - 2 * X * Y' + \text{sum}(Y^2 2))'$;

Για τον υπολογισμό του πρώτου και του δεύτερου αθροίσματος χρησιμοποιείτε SumX και SumY αντίστοιχα. Ενώ για τον πολλαπλασιασμό πινάκων χρησιμοποιείτε η βιβλιοθήκη cblash.

Τα αποτελέσματα πριν μπουν στην δομή knnresult knn αποθηκεύονται και ταξινομούνται στον πίνακα D (nxm) και επιλέγονται τα k από τα n σημεία.

Για την ταξινόμηση χρησιμοποιήθηκε όπως και στην πρώτη εργασία εξωτερική select sort και έπειτα qsort για τα k πρώτα στοιχεία.

Όλοι οι πίνακες είναι σε Column Major και ισχύει $D_{n \times m}[i][j] = D[i + j * n]$

Synchronous

Για την υλοποίηση στο αρχείο *src/knnring_mpi_v1.c* με την χρήση p MPI διεργασιών χρησιμοποιήσα την συνάρτηση *distrAllkNN* με τα δοθέντα ορίσματα και 2 buffers *yRecv* και *ySent* για αποστολή και λήψη πακέτων. Σε μια for-loop εκτελείτε ο αλγόριθμος από το V0 για τα στοιχεία που λάβαμε και έπειτα με *merge-sort* ταξινομούνται τελικό πίνακα *myKnn*.

Σημαντική είναι η χρήση του *offset* για τα *ids* των στοιχείων που λαμβάνουμε. Rank ο αριθμός της διεργασίας, *numtasks* ο συνολικός αριθμός διεργασιών και *n* ο αριθμός των επαναλήψεων από την αρχική τιμή. Να τονιστεί ότι τα *ids* ξεκινάνε από το *task1* και τελευταίο το *task0* σύμφωνα με τον *tester_mpi.c*

$$idPolarization = (rank + numtasks - l - 2) \% numtasks * n;$$

Asynchronous

Για την 2^η υλοποίηση στο αρχείο *src/knnring_mpi.c* ο κώδικας δεν διαφοροποιείτε ουσιαστικά. Προσθέτω έναν ακόμα buffer *resvBuff* για να λαμβάνω τα νέα δεδομένα ενώ χρησιμοποιώ τα προηγούμενα. Ενώ περιμένω να ολοκληρωθούν η αποστολή και η λήψη δεδομένων όταν χρειαστούν νέα δεδομένα. Επομένως ο χρόνος αποστολής και λήψης είναι ο χρόνος που διαρκεί η εντολή *MPI_waitall*. Ο υπόλοιπος χρόνος αντιστοιχεί με τον χρόνο εκτέλεσης των εντολών και δεν προσμετράτε.

Ο χρόνος εκτέλεσης στα *mpi* είναι ο μέγιστος χρόνος από τον χρόνο εκτέλεσης των *tasks*

ΑΠΟΤΕΛΕΣΜΑΤΑ ΜΕΤΡΗΣΕΙΣ

Τοπική εκτέλεση

Τα παρακάτω αποτελέσματα έγιναν σε *vm* με τα εξής χαρακτηριστικά.

Επεξεργαστής: Intel® Core i7-7500 @ 2.70GHz, 2 Processors , 4 Logical Processors

Ram: 5.6GB με 2GB Swap σε HDD

	Sequential	Synchronous		Asynchronous	
		Job Time	Iterate Time	Job Time	Iterate Time
d=37 , k=13					
n=1423 m=762	0,122	2,414	0,605	1,401	0,016
n=2000 m=1800	0,343	4,873	0,256	6,311	0,056
n=5000 m=4500 -		58,770	0,987	34,879	0,008
d=200 , k=20					
n=1423 m=762	0,423	4,652	0,046	4,980	0,006

Συμπεράσματα (Τοπική εκτέλεση)

Από τα παραπάνω αποτελέσματα βλέπουμε ότι την βελτίωση στην ασύγχρονη εκτέλεση να γίνονται πιο διακριτά σε μεγάλες τιμές του *n* καθώς τότε η διαδικασία επικοινωνίας μεταξύ των MPI διεργασιών είναι ανάλογη με τον χρόνο εκτέλεσης των *knn* σε επιμέρους διεργασία.

Επιπλέον να τονίσουμε ότι οι χρόνοι στο *sequential* αφορούν για των υπολογισμό των *knn* για ένα *query* με *m* στοιχεία μέσα σε ένα σύνολο *n* corpus. Σε αντίθεση με το MPI όπου κάθε διεργασία αναζητά για ένα *query* *n* στοιχείο όλα τα KNN των *n*(αριθμός των MPI)*.

Όμως για πολύ μεγάλες τιμές των *n,m* δεν είναι δυνατόν να τα διαχειριστεί το σύστημα και τα *mpi* είναι μια πολύ καλή λύση

Εκτέλεση στην συστοιχία

	Sequential	Synchronous		Asynchronous	
		Job Time	Iterate Time	Job Time	Iterate Time
d=80 , k=80					
n=100 m=100	0,00360	0,03270	0,00071	0,04560	0,00014
n=500 m=500	0,07651	0,63220	0,00428	0,57440	0,00030
n=1000 m=1000	-	-	-	-	-
d=10 , k=10					
n=100 m=100	0,00096	0,03730	0,00019	0,04690	0,00001
n=500 m=500	0,00303	0,22840	0,00065	0,20700	0,00002
n=1000 m=1000	0,11810	0,84657	0,00109	0,81458	0,00024
d10 , k=80					
n=100 m=100	3,62125	29,92490	1,19854	29,76141	0,00051

Συμπεράσματα (Συστοιχία)

Οι υψηλές επιδόσεις των συστημάτων στους υπολογιστές της συστοιχίας είναι φανερό πως ανταποκρίνονται στις απαιτήσεις προγραμμάτων με τέτοιο μέγεθος δεδομένων, γεγονός που η παραλληλοποίηση με MPI δεν είναι καλή λύση.

ΣΥΜΠΕΡΑΣΜΑΤΑ

Όπως φαίνεται από τις μετρήσεις η χρήση των mpi σε τοπικό επίπεδο δεν είναι ιδιαίτερα αποδοτική για παράλληλο προγραμματισμό καθώς η επικοινωνία των διεργασιών είναι χρονοβόρα, κάτι που επιβεβαιώνεται και από τις δοκιμές στην συστοιχία.

Εξάλλου η δημιουργία των MPI ξεκίνησε για την εκτέλεση μεγάλων προγραμμάτων των οποίων τα δεδομένα είναι μοιρασμένα σε μνήμες πολλών διαφορετικών συστημάτων.

Η ασύγχρονη αποστολή και λήψη επιταχύνει ιδιαίτερα το πρόγραμμα με έντονη την διαφορά στο χρόνο να εντοπίζεται για μεγάλες τιμές του n όπου ο χρόνος υπολογισμού των αποστάσεων είναι ανάλογος του χρόνου αποστολής και λήψης. Αποτελεί ξεκάθαρα μια αποτελεσματική χρήση των MPI