Εργασία 1
Παράλληλα και Διανεμημένα Συστήματα

Καρελής Παναγιώτης (9099) | Μιχάλαινας Εμμανουήλ (9070)
3/11/2019

Εισαγωγή

Σκοπός της εργασίας αυτής είναι η συγγραφή κώδικα στη γλώσσα *C* για τη δημιουργία ενός *vantage-point tree*, δοσμένου ενός συνόλου *n* σημείων που ανήκουν στον *d*-διάστατο χώρο, χρησιμοποιώντας τεχνικές παράλληλου προγραμματισμού (*pthreads, cilk, openmp*).

Στον αλγόριθμο για τη δημιουργία του *vantage-point tree* υπάρχουν δύο μέρη που μπορούν να παραλληλοποιηθούν:

- ο υπολογισμός των αποστάσεων παράλληλα
- ο υπολογισμός του inner-vptree παράλληλα με το outer-vptree

Στην εργασία μας, λοιπόν, υλοποιήσαμε μια σειριακή έκδοση για τη δημιουργία του δέντρου και τρεις εκδόσεις με παραλληλισμό των παραπάνω σημείων οι οποίες εκμεταλλεύονται περισσότερα του ενός *threads*. Έπειτα κάναμε μετρήσεις για διαφορετικές εισόδους n και d και μετρήσαμε τους μέσους χρόνους εκτέλεσης της κάθε έκδοσης. Τα πειράματα, όπως είναι αναμενόμενο, έδειξαν ότι για "επαρκώς μικρές" υπολογιστικές διαδικασίες η παράλληλη υλοποίηση υστερεί έναντι της σειριακής, επομένως χρειάστηκε να ορίσουμε ένα κατώφλι/*threshold* ώστε να περιορίσουμε τον αριθμό των ενεργών *threads* και ταυτόχρονα να προτιμάται η σειριακή έκδοση αντί της παράλληλης για μικρές εισόδους. Μετά από αυτές τις βελτιστοποιήσεις, επαναλάβαμε τα πειράματα.

Παρακάτω περιγράφεται η λειτουργικότητα του κώδικα καθώς και οι μετρήσεις που λάβαμε. Ολόκληρος ο κώδικας του *project*μαζί με το πρόγραμμα των *benchmarks*είναι διαθέσιμο στην ιστοσελίδα https://github.com/karelispanagiotis/PDS_Exercise_1.

Επεξήγηση της Λειτουργικότητας του Κώδικα

Για τους σκοπούς της εργασίας και μόνο, σε όλες τις υλοποιήσεις μας (σειριακή και παράλληλες) αποφασίσαμε να χρησιμοποιήσουμε *global variables* με τα δεδομένα του προβλήματος. Το κίνητρο πίσω από αυτή την απόφαση ήταν ότι το *argument passing* σε συναρτήσεις θα γινόταν πολύ πιο εύκολο, γιατί οι συναρτήσεις θα έχουν λιγότερα ορίσματα, αναλογιζόμενοι και το γεγονός ότι για τη δημιουργία ενός *pthread* επιτρέπεται μόνο η χρήση ενός ορίσματος.

Επιπλέον, φροντίσαμε οι υλοποιήσεις που χρησιμοποιούν παράλληλο προγραμματισμό να είναι απολύτως συμβατές με επεξεργαστές που υποστηρίζουν διαφορετικό αριθμό *threads*. Αυτό γίνεται με την αλλαγή της σταθερής *MAX_THREADS*, που βρίσκεται στο πάνω μέρος των αρχείων.

src/vptree_sequential.c

Ο κώδικας του αρχείου αυτού υλοποιεί την σειριακή δημιουργία του *vptree*. Ορίσαμε τον βοηθητικό πίνακα *idArr* και μέσω της

```
void recursiveBuildTree(vptree* node, int start, int end)
```

χτίζεται αναδρομικά το δέντρο. Η συνάρτηση αυτή θεωρεί ως *vp*το τελευταίο σημείο που της δίνεται με id `node->idx = idArr[end]`, έπειτα υπολογίζει τις αποστάσεις των σημείων με *id* από `idArr[start]` έως `idArr[end-1]` προς το *vp*. Μετά την κλήση της *quickSelect()* τα *id* των σημείων που απέχουν μικρότερη απόσταση από τη διάμεσο βρίσκονται στις θέσεις *start* έως *(end-start)/2* του πίνακα *idArr*. Επομένως πλέον καλείται η recursiveBuildTree() δύο φορές, μια για τα σημεία που απέχουν λιγότερο από τη διάμεσο και μια για αυτά που απέχουν μεγαλύτερη.

src/vptree_pthreads.c

Σε αυτή την έκδοση ο κώδικας παραμένει ο ίδιος με πριν πέραν των σημείων όπου υπολογίζονται οι αποστάσεις από το *vp* και όταν καλείται η *recursiveBuildTree()* για τον υπολογισμό του *inner* υπό-δέντρου. Ο παραλληλισμός των παραπάνω σημείων γίνεται εφόσον υπάρχουν διαθέσιμα *threads* και το μέγεθος της δουλειάς που απαιτείται να εκτελέσει ένα *thread*είναι επαρκώς μεγάλο. Αν δεν πληρούνται τα κριτήρια αυτά, τότε η ροή του προγράμματος είναι σειριακή. Για τον παραλληλισμό:

- του υπολογισμού των αποστάσεων, οι θέσεις του πίνακα *distArray* χωρίζονται σε *k* μέρη, όπου *k* είναι ο αριθμός των διαθέσιμων *threads* εκείνη τη στιγμή. Κάθε *thread*αναλαμβάνει το δικό του μερίδιο του πίνακα. Η ροή του προγράμματος σταματά μέχρις ότου τα *threads* να τελειώσουν τη δουλειά τους και να ενωθούν με αυτόν που τις κάλεσε, με την *pthread_join()*.
- του *inner*υπο-δέντρου με το *outer* υπό-δεντρο, αν υπάρχει διαθέσιμο *thread* εκείνη τη στιγμή και τα στοιχεία που απομένουν να οργανωθούν ικανοποιούν ένα κατώφλι που έχουμε ορίσει, τότε το *inner* υπό δεντρο υπολογίζεται σε ξεχωριστό *thread*. Εδώ, πάλι για τον συγχρονισμό των *threads* χρησιμοποιούμε την *pthread_join()*.

Να σημειωθεί ότι στη πρώτη έκδοση των *pthreads*, δεν υπήρχε έλεγχος των ενεργών *threads*, επομένως το πρόγραμμα μας άνοιγε συνεχώς *threads* που περίμεναν να εξυπηρετηθούν. Για τον έλεγχο των ενεργών *threads* ορίσαμε μια *global variable* η οποία ανανεώνεται κάθε φορά που ανοίγουν ή κλείνουν *threads*. Η ανανέωση της χρησιμοποιεί *pthread_mutex* για την αποφυγή του *data racing*.

Για να δώσουμε περισσότερα του ενός ορίσματα στις συναρτήσεις που τρέχουν σε threads, δημιουργήσαμε *structs*.

src/vptree_cilk.c

Εδώ η δομή του κώδικα είναι ίδια με του *pthreads*, και η οργάνωση των *threads* γίνεται ακριβώς με τον ίδιο τρόπο. Αξίζει να σημειωθεί ότι για τον υπολογισμό των αποστάσεων δοκιμάσαμε να χρησιμοποιήσουμε το *cilk_for*όμως ήταν αρκετά πιο αργό από το να χωρίσουμε τη δουλειά σε *threads*, σειρές *47:64*.Εδώ, ο περιορισμός των ενεργών *threads*γίνεται από την ίδια τη βιβλιοθήκη του *cilk*.

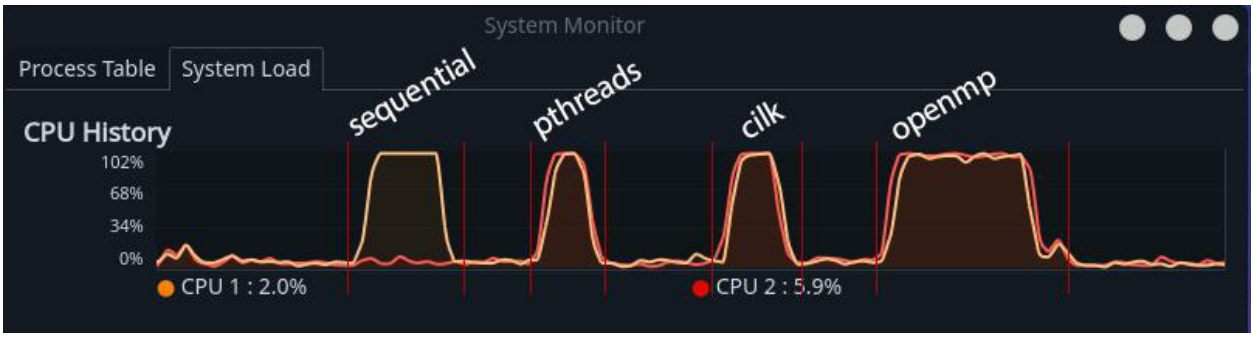
src/vptree_openmp.c

Σε αυτή την υλοποίηση, σε αντίθεση με την *cilk*, για τον υπολογισμό των αποστάσεων χρησιμοποιήσαμε την αυτόματη μετατροπή του *for*, ενώ για τον παράλληλο υπολογισμό του *inner* υπό-δέντρου η υλοποίηση είναι ίδια με πριν. Δυστυχώς, σε αυτή την έκδοση δεν καταφέραμε να πάρουμε χρόνους συγκρίσιμους με τις προηγούμενες υλοποιήσεις, όπως θα φανεί παρακάτω.

Benchmarks

Για να ελέγξουμε τους χρόνους εκτέλεσης των υλοποιήσεων μας, χρησιμοποιήσαμε υπολογιστή με επεξεργαστή AMD Athlon 2, x2 250, 3GHz με 2CPUs και συνολικά 2 threads.

Από το πρόγραμμα *KSysGuard* είδαμε ότι κατά την εκτέλεση ενός test, η sequential έκδοση απασχολούσε έναν πυρήνα, ενώ οι υπόλοιπες απασχολούσαν και τους δύο.



Παραπάνω φαίνεται η χρήση του κάθε πυρήνα συναρτήσει του χρόνου. Η κάθε υλοποίηση επεξεργάζεται είσοδο ίσων διαστάσεων. Επίσης φαίνεται πως οι εκδόσεις *pthreads* και *cilk* παίρνουν λιγότερο χρόνο από την *sequential*, ενώ αντίθετα η *openmp* όχι.

Είσοδος NxD	***sequential***	***pthreads***	***cilk***	***openMP***
10000 x 5	0.004176	0.003735	0.007472	0.015249
10000 x 50	0.015957	0.014899	0.016008	0.032797
10000 x 500	0.104645	0.065298	0.070474	0.105657
100000 x 5	0.074820	0.045205	0.052897	0.267774
100000 x 50	0.238934	0.139755	0.146374	0.380506
100000 x 500	1.639685	0.848958	0.856130	1.335573
1000000 x 5	0.964336	0.558368	0.569859	1.910934
1000000 x 50	4.173672	2.279214	2.743446	3.732805
10000000 x 5	19.23449	10.748540	11.644528	24.155395

Πίνακας 1: Οι μέσοι χρόνοι σε δευτερόλεπτα που μετρήθηκαν για διάφορα μεγέθη εισόδου, για τις 4 υλοποιήσεις. Για τα pthreads και το cilk, οι επιταχύνσεις βρίσκονται πολύ κοντά στο 2, που είναι και το μέγιστο θεωρητικό όριο για το hardware που έτρεξε τις δοκιμές.