Παράλληλα καί Διανεμημένα Συστήματα

# Finding the Strongly Connected Components of a directed graph in a shared memory multiprocessor

***Τσιράκης Ορέστης, ΑΕΜ: 9995***
Github: https://github.com/Orestistsira/auth-parallel-and-distributed-systems-ex1

*Σύντομη περιγραφή του προβλήματος:*

Στήν πρώτη εργασία μας ζητήθηκε να υλοποιήσουμε έναν αλγόριθμο ο οποίος θα βρίσκει όλα τα Strongly Connected Components ενός κατευθυνόμενου γράφου, δηλαδή όλους τους μέγιστους υπο-γράφους για τους οποίους υπάρχει μονοπάτι που συνδέει κάθε κορυφή με όλες τις υπόλοιπες κορυφές του υπο-γράφου.

*Δομή Γράφου:* (Έστω n ο αριθμός τών κορυφών και m ο αριθμός τών ακμών).

Στο αρχείο **“graph.h”** υπάρχει η δομή του struct που επιλέξαμε να μετατρέψουμε τον γράφο ώστε να είναι εύκολο να διαχειριστούμε τις ακμές του καί να μπορούμε γρήγορα να δούμε πρός τα που οδηγούν τα μονοπάτια κάθε κορυφής.

- Πίνακας “*vertices*”: Περιέχει τα ids όλων τον κορυφών του γράφου όπως μας δίνεται από το αρχείο mtx. Ο πίνακας αυτός είναι ταξινομημένος (μεγέθους n). Αν κάποια κορυφή αφαιρεθεί τότε το id παίρνει την τιμή -1.
- Πίνακας “*end*”: Περιέχει τήν κορυφή τέλους κάθε ακμής που υπάρχει στον γράφο (μεγέθους m).
- Πίνακας “*startAll*”: Περιέχει την αρχική κορυφή κάθε ακμής του γράφου και είναι ταξινομημένος (μεγέθους m). Ο συνδιασμός του πίνακα end με του startAll μας δίνει όλες τις ακμές του γράφου.
- Πίνακας “*start*”: Περιέχει όλες τις κορυφές από τις οποίες ξεκινάει μία ακμή. Ο πίνακας είναι ταξινομημένος και κάθε κορυφή περιέχεται μόνο μία φορά.
- Πίνακας “*startPointer*”: Μας δείχνει σε ποιό index του πίνακα “end” ξεκινάνε οι ακμές που έχουν ως αρχή τη κορυφή του πίνακα “start”. Με τους πίνακες “start” και “startPointer” μπορούμε εύκολα να βρίσκουμε που καταλήγει η ακμή της κάθε κορυφής που λειτουργεί ως αρχή.
- Πίνακας “*vertexPosInStart*”: Συνδέει τον πίνακα “vertices” με τον πίνακα “start”. Μας δείχνει σε ποιό index του πίνακα “start” βρίσκεται κάθε κορυφή του γράφου, αν η κορυφή δεν αποτελεί αρχή μιας ακμής, εκείνη η θέση παίρνει τη τιμή -1 (μεγέθους n).
- Πίνακες “*inDegree*” και “*outDegree*”: Μας δείχνουν τον βαθμό της κάθε κορυφής (μεγέθους n).
- Πίανκας “*sccIdOfVertex*”: Μας δείχνει σε ποιό SCC περιέχεται η κάθε κορυφή και λειτουργεί σαν έξοδος του αλγορίθμου (μεγέθους n).

*0. Δημιουργία του SCC σειριακού αλγορίθμου σε C:*

Η συνάρτηση **“SequentialColorSCC”** καλείται απο τη main οτάν έχουμε επιλέξει τη σειριακή εκτέλεση. Αρχικοποιούμε έναν πίνακα “vertexColor” μεγέθους n, ο οποίος κρατάει το χρώμα κάθε κορυφής ως έναν αριθμό, στην αρχή ίσο με το id της κάθε κορυφής. Στη συνέχεια υλοποιείται ο παραπάνω αλγόριθμος.

Το non-recursive trimming πραγματοποιείται απο τη συνάρτηση **“trimGraph”** και καλείται πρίν από κάθε επανάληψη του αλγορίθμου και όχι μόνο στην αρχή, καθώς αφού διαγραφούν κάποιες

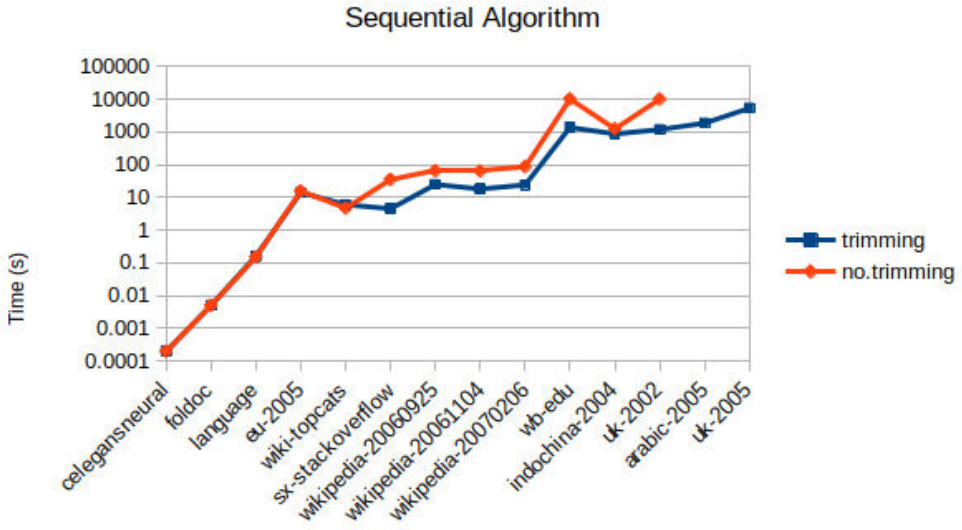
κορυφές επηρεάζουν τον βαθμό και άλλων κορυφών του γράφου. Παρατηρήθηκε ότι το trimming σε κάθε επανάληψη μας δίνει αρκετά μεγάλη επιτάχυνση στούς μεγάλους γράφους (n > 1.000.000) καθώς βρίσκει πολλά trivial SCCs. Δοκιμάστηκε και recursive trimming αλλα διαπιστώθηκε πως καθυστερούσε αρκετά τον αλγόριθμο χωρίς να μας δίνει πολλά περισσότερα trivial SCCs. Η συνάρτηση έχει πολυπλοκότητα O(n+m).

Η συνάρτηση **“spreadColor”** προωθεί κάθε φορά το μικρότερο σε τιμή χρώμα στους γείτονες κάθε κορυφής μέχρι να μήν μπορεί να γίνει κάποια επιπλέον αλλαγή.

Η συνάρτηση **“findUniqueColors”** βρίσκει όλα τα διακριτά χρώματα που έχουν απομείνει στον πίνακα “vertexColor” και τα τοποθετεί σε έναν πίνακα “uniqueColors”. Ένας τρόπος θα ήταν να χρησιμοποιήσουμε δυο εμφωλευμένα for loops, ώστε για όλα τα χρώματα να ελέγχουμε εάν έχουν μπεί ήδη στον πίνακα “uniqueColors”, και αν όχι να τα προσθέτουμε σε αυτόν με συνολική πολυπλοκότητα $O(n^2)$. Για να μειώσουμε τη χρονική πολυπλοκότητα σε O(n*logn) μπορούμε αρχικά να ταξινομήσουμε τον πίνακα “vertexColor” με τη βοήθεια της merge sort με πολυπλοκότητα O(n*logn) και έπειτα να προσπελάσουμε αυτόν τον πίνακα με πολυπλοκότητα O(n), κερδίζοντας έτσι αρκετό χρόνο.

Η συνάρτηση **“AccessUniqueColors”** βρίσκει τις κορυφές που αποτελούν ενα Strongly Connected Component για κάθε ένα από τα διακριτά χρώματα που βρήκαμε παραπάνω. Καλούμε τη συνάρτηση “bfs”, η οποία πραγματοποιεί ένα Breadth First Search στον υπο-γράφο με τις κορυφές που έχουν το αντίστοιχο χρώμα, ξεκινώντας από τη κορυφή που έχει ώς id αυτό το χρώμα μέχρι να ξαναεπιστρέψουμε στην ίδια κορυφή.

Αποτελέσματα:




Παρατηρούμε ότι το trimming μειώνει κατά πολύ τον συνολικό χρόνο του αλγορίθμου στους μεγαλύτερους γράφους και μας γλιτώνει αρκετό χρόνο και υπολογισμούς. Βλέπουμε πως στον γράφο <wiki-topcats> το trimming δεν μας ευνοεί, αλλά αυτό συμβαίνει καθώς δεν υπάρχουν trivial SCCs και ο συνολικός αριθμός των SCCs είναι ίσος με 1.

*1. Παραλληλοποίηση του αλγορίθμου σε πολυπύρηνους επεξεργαστές:*

Γιά την παράλληλη υλοποίηση έγινε προσπάθεια να παραλληλοποιηθούν αρκετά κομμάτια του αλγορίθμου.

Ο υπολογισμός των βαθμών κάθε κορυφής γίνεται παράλληλα, καθώς είναι ανεξάρτητοι από τις υπόλοιπες κορυφές. Έπειτα το trimming γίνεται ξεχωριστά γιά κάθε κορυφή ελέγχοντας αν έχει κάποιο μηδενικό βαθμό, χρησιμοποιώντας mutex για να αυξήσουμε τον μετρητή των SCCs στον οποίο έχουν πρόσβαση όλα τα threads.

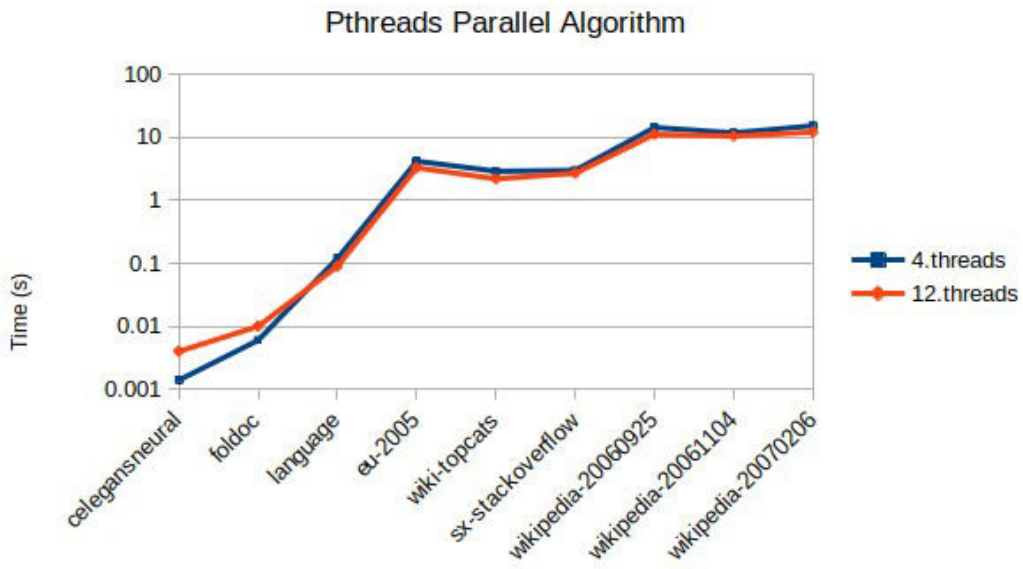
Η αρχικοποίηση των χρωμάτων και η προώθηση τους γίνεται επίσης ξεχωριστά για κάθε κορυφή και χωρίς να υπάρχουν data races μεταξύ των threads, αξιοποιώντας στο έπακρο τη παραλληλοποίηση, αφού δεν μας επηρέαζει το ποιό thread θα κάνει πρώτο την αλλαγή χρώματος.

Η εύρεση τών διακριτών χρωμάτων που απομένουν μετά τη προώθηση αποφασίστηκε να μείνει σειριακή, παρόλο που η merge sort επιλέχθηκε για την δυνατότητα της να παραλληλοποιηθεί. Λόγω της καλύτερης χρονικής πολυπλοκότητας και της γρήγορης εκτέλεσης της συνάρτησης δεν παίρναμε την επιθυμητή αύξηση της ταχύτητας.

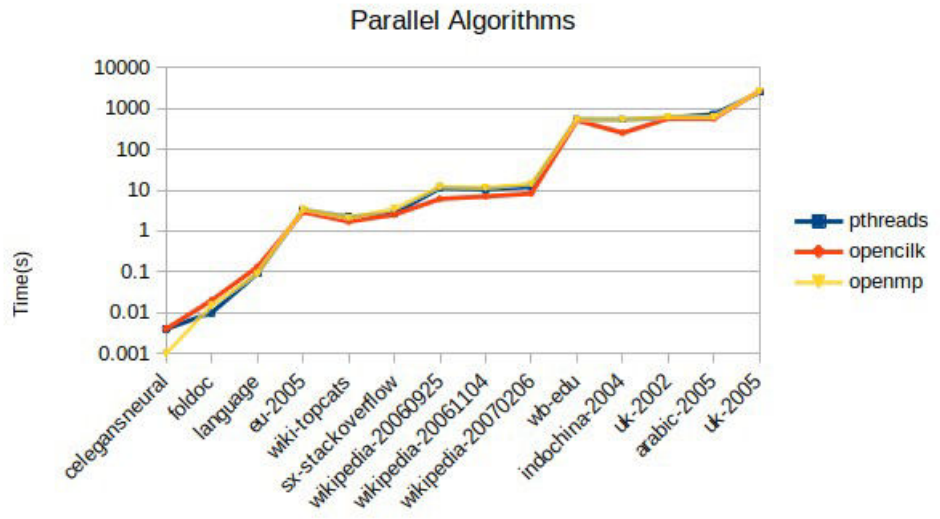
Η εύρεση των SCCs από τη συνάρτηση “bfs” γίνεται και αυτή παράλληλα, ξεχωριστά για κάθε διακριτό χρώμα χρησιμοποιώντας και πάλι mutex για να αυξήσουμε τον μετρητή των SCCs στον οποίο έχουν πρόσβαση όλα τα threads.

*2. Ορθότητα και απόδοση παράλληλου αλγορίθμου:*

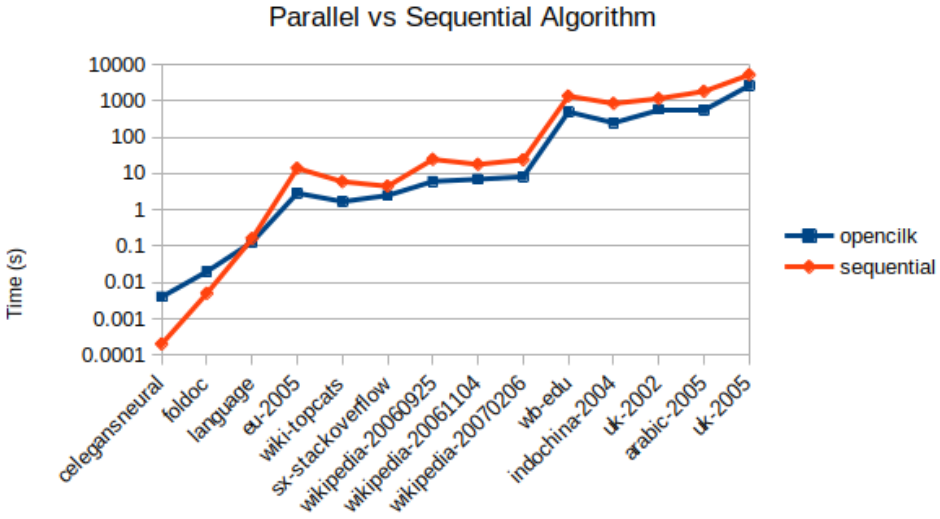
Αποτελέσματα:




Δοκιμάστηκαν διαφορετικοί αριθμοί νημάτων καθώς τρέχουμε των αλγόριθμο σε επεξεργαστή με 12 threads και παρατηρούμε πως για n > 100.000 η εκτέλεση με 12 threads γίνεται αισθητά πιό γρήγορη από αυτήν με 4 threads, οπότε για μεγαλύτερους γράφους επιλέχθηκαν τα 12 νήματα.




Παρατηρούμε πως από τον γράφο <eu-2005> και μετά η παράλληλη υλοποίηση σε openCilk είναι αισθητά πιό γρήγορη από τις αντίστοιχες σε openMP και Pthreads, ενώ σε γράφους με n < 500.000 δέν έχει τόσο καλή επίδοση (χρησιμοποιείται λογαριθμική κλίμακα στο γράφημα).




Βλέπουμε πως με τη παραλληλοποίηση του αλγορίθμου για n > 500.000 παίρνουμε μια αύξηση στη ταχύτητα της τάξης του 65-70%,  εκτός από τον γράφο <sx-stackoverflow> στον οποίο δεν παρατηρούμε τόσο μεγάλη επιτάγχυνση. Συνεπώς θα προτείναμε μία λύση η οποία θα ξεκινάει με τον παράλληλο αλγόριθμο και όταν ο αριθμός τον κορυφών που μένουν μειωθεί στις 500.000 θα συνεχίζει με τον σειριακό, ώστε να έχουμε τη μέγιστη απόδοση.

Ορθότητα: Ο αλγόριθμος μας δίνει σωστά αποτελέσματα για όλους τους παραπάνω γράφους.

**Σημείωση:** Ο γράφος <twitter7> δεν υπάρχει στα αποτελέσματα καθώς χρειαζόταν μεγάλο χώρο στην RAM και σταματούσε η εκτέλεση του. Όλοι οι υπόλοιποι γράφοι μπορούν να τρέξουν με 16GB RAM. Οι δοκιμές έγιναν με cpu AMD Ryzen 5 3600 6-Core Processor (2 threads per core).