ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

"ΚΕΝΤΡΙΚΟΠΟΙΗΜΕΝΟΣ ΚΑΙ ΠΑΡΑΛΛΗΛΟΣ ΑΛΓΟΡΙΘΜΟΣ ΓΙΑ ΤΡΙΓΩΝΙΣΜΟ ΓΡΑΦΗΜΑΤΩΝ"

ΠΑΝΤΕΛΙΔΟΥ ΚΥΡΙΑΚΗ-ΝΕΚΤΑΡΙΑ
ΑΕΜ: 2395

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ:

ΠΑΠΑΔΟΠΟΥΛΟΣ ΑΠΟΣΤΟΛΟΣ, ΕΠΙΚΟΥΡΟΣ ΚΑΘΗΓΗΤΗΣ

ΘΕΣΣΑΛΟΝΙΚΗ, ΣΕΠΤΕΜΒΡΙΟΣ 2017

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΑΤΑΛΟΓΟΣ ΣΧΗΜΑΤΩΝ

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

ΚΕΦΑΛΑΙΟ 1:

ΠΕΡΙΛΗΨΗ - ΕΙΣΑΓΩΓΗ

ΠΕΡΙΛΗΨΗ

Αντικείμενο της παρούσας εργασίας είναι η μέτρηση του αριθμού των τριγώνων σε γράφημα. Το πρόβλημα του υπολογισμού των τριγώνων σε ένα γράφημα παρουσιάζει εξαιρετικά μεγάλο ενδιαφέρον στην μελέτη κοινωνικών δικτύων (Facebook, Twitter κλπ.), στον διαχωρισμό ανεπιθύμητων μηνυμάτων ηλεκτρονικού ταχυδρομείου και σε άλλους τομείς της πληροφορικής. Στην εργασία μελετώνται δύο αλγόριθμοι για την καταμέτρηση τριγώνων σε πολύ μεγάλα δίκτυα και γίνεται σύγκριση της χρονικής τους απόδοσης. Ένας ακολουθιακός, ο NodeIteratorN και ένας παράλληλος, ο Patric.

ABSTRACT

The main concept of this work is counting the number of triangles in a graph. Counting triangles is attracting increasing interest in studying social networks (Facebook, Twitter etc.), in the creation of successful spam filters and many more. In this work, two counting algorithms are examined for their time performance in massive networks. The first one is sequential and is called NodeIteratorN and the second, Patric, is parallel.

ΕΙΣΑΓΩΓΗ

Ένα γράφημα είναι μια αφηρημένη αναπαράσταση ενός συνόλου στοιχείων, όπου μερικά ζευγάρια στοιχείων συνδέονται μεταξύ τους με δεσμούς. Τα διασυνδεδεμένα στοιχεία αναπαριστώνται με μαθηματικές έννοιες οι οποίες ονομάζονται κορυφές ενώ οι δεσμοί που συνδέουν τα ζευγάρια των κορυφών ονομάζονται ακμές. Συνήθως, ένα γράφημα απεικονίζεται σε διαγραμματική μορφή ως ένα σύνολο κουκκίδων για τις κορυφές, ενωμένα μεταξύ τους με γραμμές ή καμπύλες για τις ακμές. Στην πιο κοινή έννοια του όρου, ένα γράφημα είναι ένα διατεταγμένο ζεύγος $G = (V, E)$ αποτελούμενο από ένα σύνολο V των κορυφών ή κόμβων μαζί με το σύνολο E από ακμές ή γραμμές, οι οποίες είναι υποσύνολα δύο στοιχείων V (δηλαδή, μια ακμή σχετίζεται με δύο κορυφές και η σχέση απεικονίζεται ως μη ταξινομημένο ζεύγος των κορυφών σε σχέση με τη συγκεκριμένη ακμή). Το γράφημα μπορεί να είναι κατευθυνόμενο ή μη-κατευθυνόμενο.

Ένα κατευθυνόμενο γράφημα ή διγράφημα είναι ένα διατεταγμένο ζεύγος $D = (V, A)$ όπου V, είναι ένα σύνολο του οποίου τα στοιχεία λέγονται κορυφές ή κόμβοι και A, είναι μια σειρά από διατεταγμένα ζεύγη κορυφών, τα οποία ονομάζονται τόξα, διατεταγμένες ακμές ή βέλη. Ένα τόξο $a = (x, y)$ θεωρείται ότι κατευθύνεται από το x στο y; y ονομάζεται η αρχή και x το τέλος του τόξου; το y λέγεται ότι είναι άμεσος διάδοχος του x, και το x λέγεται ότι είναι ο άμεσος προκάτοχος του y. Αν ένα μονοπάτι οδηγεί από το x στο y, τότε το y λέγεται ότι είναι διάδοχος του x και προσβάσιμο από το x, και το x λέγεται ότι είναι ο προκάτοχος του y. Το τόξο (y, x) ονομάζεται το τόξο (x, y) ανεστραμμένο.

Ένα μη-κατευθυνόμενο γράφημα είναι εκείνο στο οποίο οι ακμές δεν έχουν προσανατολισμό. Η ακμή (α, β) είναι ταυτόσημη με την άκρη (β, α), δηλαδή, δεν υπάρχουν διατεταγμένα ζεύγη, αλλά σύνολα $\{u, v\}$. Εμείς θα ασχοληθούμε μόνο με μη-κατευθυνόμενα γραφήματα.

Ένα τρίγωνο σε ένα μη-κατευθυνόμενο γράφημα είναι ένα σύνολο τριών κορυφών έτσι ώστε κάθε πιθανή ακμή μεταξύ τους να είναι παρούσα στο γράφημα. Τα προβλήματα απόφασης εάν ένα δεδομένο γράφημα περιέχει ένα τρίγωνο, μέτρησης του αριθμού των τριγώνων στο γράφημα, και καταγραφής όλων αυτών ονομάζονται εύρεσης, μέτρησης και καταγραφής αντίστοιχα.

Η μέτρηση του αριθμού των τριγώνων σε ένα γράφημα είναι ένα θεμελιώδες πρόβλημα με διάφορες εφαρμογές. Στο παρελθόν, το πρόβλημα της καταμέτρησης και της καταγραφής των τριγώνων ενός γραφήματος έχει μελετηθεί εντατικά από θεωρητική άποψη. Πρόσφατα, η καταμέτρηση τριγώνων έχει γίνει επίσης ένα ευρέως χρησιμοποιούμενο εργαλείο στην ανάλυση δικτύων. Λόγω του πολύ μεγάλου μεγέθους δικτύων όπως το Διαδίκτυο, ο Παγκόσμιος Ιστός ή τα κοινωνικά δίκτυα, η αποτελεσματικότητα των αλγορίθμων για την καταμέτρηση και καταγραφή τριγώνων

είναι ένα σημαντικό ζήτημα. Η κύρια πρόθεση αυτής της εργασίας είναι να αξιολογηθεί η πρακτικότητα της καταμέτρησης τριγώνων σε πολύ μεγάλους γράφους με διάφορες κατανομές βαθμών. Μελετούμε έναν γνωστό αλγόριθμο που αποδίδει πολύ καλά και καθιστά εφικτή την καταμέτρηση τριγώνων σε τεράστια δίκτυα.

Αυτός είναι ο παράλληλος αλγόριθμος που ονομάζεται Patric. Αυτό που κάνει είναι να λαμβάνει ένα τεράστιο δίκτυο, να το διαχωρίζει σε κατάλληλα σημεία και να αναθέτει κάθε τμήμα σε έναν επεξεργαστή. Στο κάθε τμήμα εφαρμόζεται ο αλγόριθμος NodeIteratorN, ο οποίος είναι αλγόριθμος μέτρησης και επιστρέφει τον αριθμό των τριγώνων που περιέχει ένα γράφημα. Τέλος, αθροίζεται ο αριθμός των τριγώνων όλων των τμημάτων και έτσι προκύπτει ο αριθμός τριγώνων του μεγάλου γραφήματος.

Η εργασία δομείται σε κεφάλαια ως εξής:

Στο Κεφάλαιο 2 παρουσιάζεται ο τρόπος εκτέλεσης του αλγορίθμου NodeIteratorN για την καταμέτρηση τριγώνων σε ένα τμήμα του μεγάλου γράφου.

Στο Κεφάλαιο 3 παρουσιάζεται ο αλγόριθμος Patric και επεξηγείται με τη βοήθεια σχηματικών αναπαραστάσεων και διαγραμμάτων.

Στο Κεφάλαιο 4 παρουσιάζονται μετρήσεις και χρονικές αποδόσεις ανάλογα με διάφορες παραμέτρους εκτέλεσης του Patric.

Στο Κεφάλαιο 5 γίνεται μία σύνοψη της εργασίας και επισημαίνονται τα θέματα για περαιτέρω μελέτη.

ΚΕΦΑΛΑΙΟ 2:
ΚΕΝΤΡΙΚΟΠΟΙΗΜΕΝΟΣ ΑΛΓΟΡΙΘΜΟΣ

ΚΕΝΤΡΙΚΟΠΟΙΗΜΕΝΟΣ ΑΛΓΟΡΙΘΜΟΣ

2.1 ΘΕΩΡΗΤΙΚΗ ΠΕΡΙΓΡΑΦΗ ΑΛΓΟΡΙΘΜΟΥ

Ένας απλός αλλά αποτελεσματικός αλγόριθμος για την καταμέτρηση των τριγώνων είναι: για κάθε κόμβο v ∈ V, βρες τον αριθμό των ακμών μεταξύ των γειτόνων του, δηλ. τον αριθμό ζευγών γειτόνων που συμπληρώνουν ένα τρίγωνο με την κορυφή v. Σε αυτή τη μέθοδο, το κάθε τρίγωνο (u, v, w) υπολογίζεται έξι φορές - έξι μεταθέσεις των u, v και w. Υπάρχουν πολλοί αλγόριθμοι οι οποίοι παρέχουν σημαντική βελτίωση σε σχέση με την παραπάνω μέθοδο. Ένας από τους πιο σύγχρονους αλγορίθμους είναι γνωστός ως NodeIterator ++.

```
1: T ← 0        {T stores the count of triangles}
2: for v ∈ V do
3:     for u ∈ N_v and v ≺ u do
4:         for w ∈ N_v and u ≺ w do
5:             if (u, w) ∈ E then
6:                 T ← T + 1
```

Σχήμα 1: Ψευδοκώδικας του αλγορίθμου NodeIterator++, όπου ≺ διάταξη των κόμβων με βάση τον βαθμό τους.

Αυτός ο αλγόριθμος χρησιμοποιεί μια συνολική διάταξη των κόμβων, για να αποφύγει τις διπλές μετρήσεις του ίδιου τριγώνου. Οποιαδήποτε αυθαίρετη διάταξη των κόμβων, π.χ. η διάταξη των κόμβων με βάση το ID τους, διασφαλίζει ότι κάθε τρίγωνο μετράται ακριβώς μία φορά - υπολογίζει μόνο μία από τις έξι πιθανές μεταθέσεις. Ωστόσο, ο αλγόριθμος NodeIterator ++ ενσωματώνει μια ενδιαφέρουσα διάταξη κόμβων με βάση τους βαθμούς τους, όπως ορίζεται παρακάτω:

$$u \prec v \Leftrightarrow d_u < d_v$$

$$\acute{\eta} \quad (d_u = d_v \text{ and } u < v)$$

Αυτή η ταξινόμηση βάσει βαθμού μπορεί να βελτιώσει το χρόνο εκτέλεσης. Το $\hat{d}_v$ είναι ο αριθμός των γειτόνων u του v, έτσι ώστε v ≺ u. Ονομάζουμε $\hat{d}_v$ τον πραγματικό βαθμό του v. Υποθέτοντας ότι οι γείτονες του v (N_v), για όλους τους v, είναι ταξινομημένοι και ότι χρησιμοποιείται μια δυαδική αναζήτηση για τον έλεγχο

(u, w) ∈ E, μπορεί να επιτευχθεί ο χρόνος λειτουργίας $O(\sum_v(\hat{d}_v d_v + d_v^2 \, log \, d_{max}))$, όπου $d_{max} = max_v \, d_v$.

Αυτός ο χρόνος λειτουργίας ελαχιστοποιείται όταν οι τιμές $\hat{d}_v$ των κόμβων είναι όσο το δυνατόν πλησιέστερες μεταξύ τους. Αν και για οποιαδήποτε διάταξη των κόμβων, το $\sum_v \hat{d}_v = m$ είναι αμετάβλητο.

Παρατηρούμε επίσης ότι η διάταξη των κόμβων με βάση το βαθμό συμβάλλει στη διατήρηση της ισορροπίας του φόρτου εργασίας μεταξύ των επεξεργαστών σε κάποιο βαθμό, στον παράλληλο αλγόριθμο. Χρησιμοποιούμε αυτή την διάταξη βάσει βαθμού στον παράλληλο αλγόριθμό μας, τον Patric.

Μια απλή τροποποίηση του NodeIterator ++ έχει ως εξής:

```
 1: {Preprocessing: Step 2-6}
 2: for each edge (u, v) do
 3:    if u ≺ v, store v in N_u
 4:    else store u in N_v
 5: for v ∈ V do
 6:    sort N_v in ascending order
 7: T ← 0        {T is the count of triangles}
 8: for v ∈ V do
 9:    for u ∈ N_v do
10:       S ← N_v ∩ N_u
11:       T ← T + |S|
```

Σχήμα 2: Ψευδοκώδικας του αλγορίθμου NodeIteratorN, τροποποίηση του NodeIterator++.

Κάνουμε σύγκριση u ≺ v για κάθε ακμή (u, v) ∈ E σε ένα βήμα επεξεργασίας, αντί να το κάνουμε κατά την καταμέτρηση των τριγώνων. Αυτό το βήμα προεπεξεργασίας μειώνει τον συνολικό αριθμό των συγκρίσεων σε O (m). Για κάθε κόμβο v, το σύνολο γειτόνων N_v διατηρείται στη μνήμη. Ωστόσο, για κάθε ακμή (v, u), το u αποθηκεύεται στο σύνολο N_v αν και μόνο αν v ≺ u. Ο τροποποιημένος αλγόριθμος ονομάζεται NodeIteratorN. Όλα τα τρίγωνα που περιέχουν τον κόμβο v και οποιοδήποτε u ∈ N_v μπορούν να βρεθούν μέσω του συνόλου της τομής $N_u \cap N_v$. Η ορθότητα του NodeIteratorN αποδεικνύεται παρακάτω.

Θεώρημα 1: Ο αλγόριθμος NodeIteratorN μετρά κάθε τρίγωνο στο γράφημα G μία και μόνο μία φορά.

Απόδειξη:

Θεωρούμε ένα τρίγωνο (x_1, x_2, x_3) στο G και χωρίς την απώλεια της γενικότητας, υποθέτουμε ότι $x_1 \prec x_2 \prec x_3$. Με την κατασκευή του N_x στο βήμα προεπεξεργασίας, έχουμε x_2, $x_3 \in N_{x1}$ και $x_3 \in N_{x2}$. Όταν ξεκινούν οι βρόχοι επανάληψης στις γραμμές 8-9 με $v = x_1$ και $u = x_2$, ο κόμβος x_3 εμφανίζεται στο S (γραμμές 10-11) και το τρίγωνο (x_1, x_2, x_3) μετράται μία φορά. Αλλά αυτό το τρίγωνο δεν μπορεί να μετρηθεί για άλλες τιμές των v και u (στις γραμμές 8-9) επειδή $x_1 \notin N_{x2}$ και x_1, $x_2 \notin N_{x3}$.

Στον NodeIteratorN, $| N_v | = \hat{d}_v$ είναι ο πραγματικός βαθμός του v. Όταν ταξινομούνται τα N_v και N_u, μπορεί να υπολογιστεί το $N_u \cap N_v$ σε χρόνο O ($\hat{d}_u + \hat{d}_v$). Έπειτα έχουμε $O(\sum_{v \in V} d_v \hat{d}_v)$ πολυπλοκότητα για το NodeIteratorN όπως θα δούμε παρακάτω στο Θεώρημα 2, σε αντίθεση με την πολυπλοκότητα $O(\sum_v (\hat{d}_v d_v + d_v^2 \log d_{max}))$ για τον NodeIterator ++.

Θεώρημα 2: Η πολυπλοκότητα του αλγορίθμου NodeIteratorN είναι $O(\sum_{v \in V} d_v \hat{d}_v)$.

Απόδειξη:

Ο χρόνος για την κατασκευή του N_v για όλους τους v είναι $O(\sum_v d_v) = O(m)$, και η ταξινόμηση των N_v απαιτεί $O(\sum_v \hat{d}_v \log \hat{d}_v)$ χρόνο. Τώρα, ο υπολογισμός της τομής $N_v \cap N_u$ παίρνει $O(\hat{d}_u + \hat{d}_v)$ χρόνο. Έτσι, η χρονική πολυπλοκότητα του NodeIteratorN είναι:

$$O(m) + O(\sum_{v \in V} \hat{d}_v \log \hat{d}_v) + O(\sum_{v \in V} \sum_{u \in N_v} (\hat{d}_u + \hat{d}_v)) =$$

$$O(\sum_{v \in V} \hat{d}_v \log \hat{d}_v) + O(\sum_{(v,u) \in E} (\hat{d}_u + \hat{d}_v)) =$$

$$O(\sum_{v \in V} \hat{d}_v \log \hat{d}_v) + O(\sum_{v \in V} d_v \hat{d}_v) =$$

$$O(\sum_{v \in V} d_v \hat{d}_v).$$

Το προτελευταίο βήμα προκύπτει από το γεγονός ότι για το κάθε v ∈ V, ο όρος $\hat{d}_v$ εμφανίζεται d_v φορές σε αυτή την έκφραση.

Παρατηρούμε ότι μπορούμε επίσης να χρησιμοποιήσουμε το σύνολο της τομής στον NodeIterator ++ αντικαθιστώντας τις γραμμές 4-6 του NodeIteraTor ++ στο Σχήμα 1 με τις ακόλουθες τρεις γραμμές:

```
1: S ← N_v ∩ N_u
2: for w ∈ S and u ≺ w do
3:     T ← T + 1
```

Ωστόσο, με αυτή τη λειτουργία τομής, ο χρόνος του NodeIterator ++ είναι $O(\sum_v(d_v^2))$, αφού είναι $|N_v| = d_v$ στον NodeIterator ++, και ο υπολογισμός $N_v \cap N_u$ παίρνει $O(d_u + d_v)$ χρόνο. Επιπλέον, η απαίτηση μνήμης για τον NodeIteratorN είναι η μισή από αυτή για τον NodeIterator ++. Ο NodeIteratorN αποθηκεύει $\sum_v \hat{d}_v = m$ στοιχεία σε όλο το N_v και ο NodeIterator ++ αποθηκεύει $\sum_v d_v = 2m$ στοιχεία.

Συγκρίνουμε επίσης πειραματικά την απόδοση του NodeIteratorN και NodeIterator++ χρησιμοποιώντας και πραγματικά και κατασκευασμένα δίκτυα. Ο NodeIteratorN είναι σημαντικά ταχύτερος από τον NodeIterator ++ για αυτά τα δίκτυα όπως φαίνεται στον Πίνακα 1.

Networks	Runtime (sec.)		Triangles
	NodeIterator++	NodeIteratorN	
Email-Enron	0.14	0.07	0.7M
web-BerkStan	3.5	1.4	64.7M
LiveJournal	106	42	285.7M
Miami	46.35	32.3	332M
PA(25M, 50)	690	360	1.3M
Gnp(500K, 20)	1.81	0.6	1308

Πίνακας 1: Χρόνοι εκτέλεσης των δύο αλγορίθμων.

2.2 ΑΝΑΠΤΥΞΗ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ JAVA

Υλοποιούμε τον αλγόριθμο NodeIteratorN, εφόσον είναι καλύτερος σε χρονική απόδοση, όπως είδαμε παραπάνω. Δουλεύουμε πάνω σε μη κατευθυνόμενους γράφους.

Πρώτα απ΄ όλα, χρησιμοποιούμε την εξωτερική βιβλιοθήκη JGraphT για να μας βοηθήσει στην υλοποίηση του αλγορίθμου, αφού περιέχει ήδη έτοιμες βασικές συναρτήσεις που θα χρειαστούμε στη συνέχεια.

Έχουμε κατασκευάσει τέσσερις συναρτήσεις, οι οποίες καλούνται με τη σειρά στη main συνάρτηση του προγράμματος. Πρώτη καλείται η συνάρτηση ***createStringGraph*** για τη δημιουργία του γραφήματος.

private static Graph<String, DefaultEdge> createStringGraph()

Η συνάρτηση δεν δέχεται ορίσματα. Επιστρέφει ένα γράφημα τύπου Graph<String, DefaultEdge>. Στο κύριο μέρος της, η συνάρτηση διαβάζει το αρχείο εισόδου που είναι της παρακάτω μορφής:

0 1

0 2

1 0

1 2

1 3

1 6

Δείχνει δηλαδή ποιοι κόμβοι ενώνονται μεταξύ τους. Στη συνέχεια, διαβάζει κάθε γραμμή, με split παίρνει τον κάθε κόμβο και τον αναθέτει σε μεταβλητές (v1 και v2). Ελέγχει αν η ακμή (v1,v2) υπάρχει ήδη στο γράφημα με τη συνάρτηση containsEdge. Αν δεν υπάρχει, προσθέτει τους κόμβους με τη συνάρτηση addVertex και την ακμή με την addEdge. Οι συναρτήσεις που αναφέρθηκαν είναι υλοποιημένες στη βιβλιοθήκη JGraphT.

Έπειτα, από τη main καλείται η συνάρτηση ***findDegree***.

private static ArrayList findDegree(Graph<String, DefaultEdge> g)

Δέχεται σαν όρισμα ένα γράφημα τύπου Graph<String, DefaultEdge> και επιστρέφει μία ArrayList.

Στο κύριο μέρος, η συνάρτηση δημιουργεί ένα set με όλους τους κόμβους του γραφήματος με τη συνάρτηση vertexSet της JGraphT. Στη συνέχεια, με τη βοήθεια λίστας, συγκρίνει και ταξινομεί τους κόμβους σε αύξουσα σειρά. Έπειτα, για κάθε κόμβο βρίσκει πόσες ακμές εξέρχονται/εισέρχονται από αυτόν με τη συνάρτηση edgesOf. Αυτός θα είναι και ο βαθμός του κόμβου που προστίθεται σε μία ArrayList. Στο τέλος αυτής της επανάληψης λοιπόν, έχουμε μία ArrayList, όπου κάθε index αντιστοιχεί σε έναν κόμβο (δηλ. ο index 0 αντιστοιχεί στον κόμβο 0, ο index 1 στον κόμβο 1, κοκ.) και κάθε στοιχείο στο βαθμό του αντίστοιχου κόμβου. Αυτή είναι και η ArrayList που επιστρέφει η συνάρτηση.

Πίσω στη main, δημιουργούμε ένα set με όλες τις ακμές του γραφήματος με τη συνάρτηση edgesSet. Από δω και κάτω, ακολουθούμε τα βήματα του ψευδοκώδικα του αλγορίθμου NodeIteratorN, όπως αυτά φαίνονται στο Σχήμα 2 παραπάνω.

Βήματα 2-4:

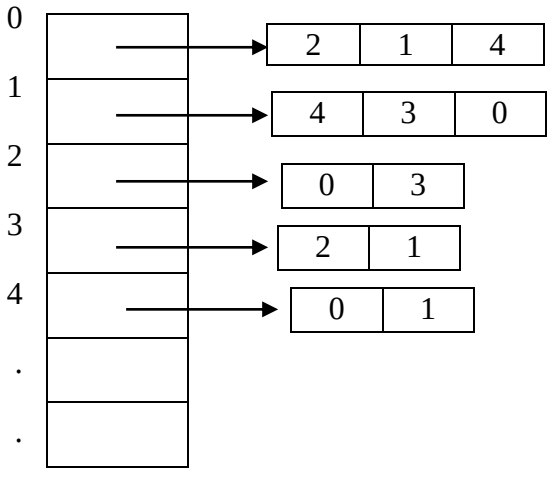
Για κάθε ακμή, παίρνουμε με split τους δύο κόμβους και τους αποθηκεύουμε στις μεταβλητές u και v. Έπειτα, παίρνουμε τον βαθμό του κάθε κόμβου από την ArrayList με τους βαθμούς που έχει επιστρέψει η συνάρτηση findDegree και τους αποθηκεύουμε στις μεταβλητές du και dv αντίστοιχα. Μετά γίνεται ο έλεγχος du < dv για να ελέγξουμε αν ο κόμβος u έχει μικρότερο βαθμό από τον v. Αν ισχύει αυτό, προσθέτουμε τον κόμβο v στους γείτονες του u, με τη βοήθεια της συνάρτησης ***addToInnerArrayList***. Αν όχι, προσθέτουμε τον κόμβο u στους γείτονες του v και πάλι με τη ίδια συνάρτηση.

private static ArrayList<ArrayList<String>> addToInnerArrayList (ArrayList<ArrayList<String>> N, String index, String element)

Η συνάρτηση δέχεται μια nested ArrayList και δύο String, ένα για τον index και ένα για το element. Επιστρέφει και πάλι μια nested ArrayList.

Στο κύριο μέρος, ελέγχει αν ο index είναι μεγαλύτερος από το μέγεθος του μεγάλου ArrayList N. Αν ναι, προσθέτει ακόμα μία θέση στο μεγάλο ArrayList. Τέλος, προσθέτει το element στο εσωτερικό ArraList του index.

Έτσι έχουμε ένα μεγάλο ArrayList που κάθε index του αντιστοιχεί σε έναν κόμβο. Κάθε element είναι ένα ArrayList που περιέχει κόμβους, οι οποίοι είναι οι γείτονες του index-κόμβου. Έχει δηλαδή την παρακάτω μορφή:



Βήματα 5-6:

Πίσω στη main, αφού έχει επιστραφεί η ArrayList N από την συνάρτηση addToInnerArrayList, κάνουμε sort τις φωλιασμένες λίστες για κάθε κόμβο.

Βήματα 7-11:

Τώρα πάμε ουσιαστικά να μετρήσουμε τα τρίγωνα, καλώντας τη συνάρτηση ***countTriangles***.

private static int countTriangles (ArrayList<ArrayList<String>> N)

Δέχεται σαν όρισμα την μεγάλη ArrayList N. Επιστρέφει έναν int, που είναι ο αριθμός των τριγώνων.

Στο σώμα της, η συνάρτηση αρχικοποιεί έναν μετρητή T. Έπειτα, για κάθε κόμβο v, για κάθε κόμβο u που ανήκει στους γείτονες του v (δηλαδή στη φωλιασμένη λίστα του v), παίρνει την τομή S των γειτόνων του v και των γειτόνων του u (Nv ∩ Nu). Προσθέτει στον μετρητή T το μέγεθος της τομής S. Τελικά, το T περιέχει τον αριθμό των τριγώνων του γραφήματος.

ΚΕΦΑΛΑΙΟ 3:

ΠΑΡΑΛΛΗΛΟΣ ΑΛΓΟΡΙΘΜΟΣ

ΠΑΡΑΛΛΗΛΟΣ ΑΛΓΟΡΙΘΜΟΣ

3.1 ΠΕΡΙΓΡΑΦΗ ΑΛΓΟΡΙΘΜΟΥ

Σε αυτό το σημείο, παρουσιάζουμε τον παράλληλο αλγόριθμο PATRIC για τη μέτρηση τριγώνων σε πολύ μεγάλα δίκτυα.

Υποθέτουμε ότι το δίκτυο είναι τεράστιο και δεν χωράει στην τοπική μνήμη ενός μόνο υπολογιστικού κόμβου. Τοπικά ο κάθε επεξεργαστής αποθηκεύει μόνο ένα μέρος του δικτύου στη μνήμη του. Έστω ότι P είναι ο αριθμός των επεξεργαστών που χρησιμοποιούνται στον υπολογισμό. Το δίκτυο είναι χωρισμένο σε P τμήματα και κάθε διαμέρισμα G_i (V_i, E_i) αποδίδεται σε έναν επεξεργαστή. Ο επεξεργαστής i εκτελεί τον υπολογισμό στο δικό του διαμέρισμα, G_i. Τα δεδομένα δικτύου δίνονται ως είσοδος σε ένα μόνο δίσκο. Κάθε επεξεργαστής, παράλληλα, διαβάζει το δικό του μέρος του δίκτυο (τα απαραίτητα δεδομένα για την κατασκευή του δικού του τμήματος G_i) στην τοπική του μνήμη. Τα κύρια βήματα του PATRIC δίνονται στο Σχήμα 3.

1: Each processor i, in parallel, executes the following:(lines 2-4)
2: $G_i(V_i, E_i) \leftarrow$ ComputePartition(G, i)
3: $T_i \leftarrow$ CountTriangles(G_i, i)
4: Barrier
5: Find $T = \sum_i T_i$ {processor 0 computes T}
6: return T

Σχήμα 3: Βήματα του αλγορίθμου PATRIC.

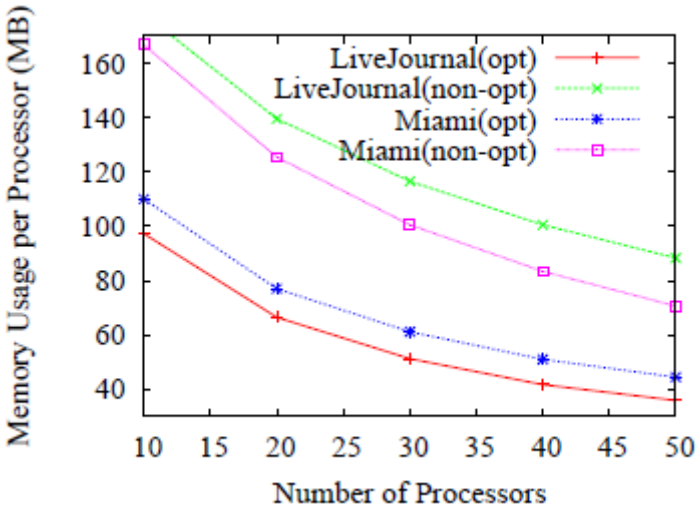
3.2 ΔΙΑΜΕΡΙΣΗ ΔΙΚΤΥΟΥ

Ο περιορισμός της μνήμης δημιουργεί δυσκολία όσον αφορά το σημείο που πρέπει να χωριστεί το γράφημα και τον τρόπο, ώστε η απαιτούμενη μνήμη για την αποθήκευση ενός διαμερίσματος να ελαχιστοποιείται και συγχρόνως το διαμέρισμα να περιέχει επαρκείς πληροφορίες για την ελαχιστοποίηση της επικοινωνίας μεταξύ των επεξεργαστών. Για το input γράφημα G (V, E), ο επεξεργαστή i εργάζεται στο G_i (V_i, E_i), το οποίο είναι υπογράφημα του G. Το υπογράφημα G_i κατασκευάζεται ως εξής:

Πρώτον, το σύνολο των κόμβων V είναι χωρισμένο σε P ξεχωριστά υποσύνολα V_0^c, $V_1^c, \ldots, V_{p-1}^c$, τέτοια ώστε για κάθε j και k, $V_j^c \cap V_k^c = \emptyset$ και $\bigcup_k V_k^c = V$. Δεύτερον, το σύνολο V_i είναι κατασκευασμένο περιέχοντας όλους τους κόμβους στο V_i^c και $\bigcup_{v \in V_i^c} N_v$. Το σύνολο ακμών $E_i \subset E$ είναι το σύνολο των ακμών (u, v) ώστε u, v ∈ V_i και (u, v) ∈ E.

Κάθε επεξεργαστής i είναι υπεύθυνος για την καταμέτρηση των τριγώνων που προσπίπτουν στους κόμβους του V_i^c. Ονομάζουμε κάθε κόμβο v ∈ V_i^c *core node* του επεξεργαστή i. Κάθε v ∈ V είναι core node σε ακριβώς ένα διαμέρισμα. Ο τρόπος με τον οποίο κατανέμονται οι κόμβοι του V μεταξύ των core συνόλων V_i^c για όλους τους επεξεργαστές i, επηρεάζει σημαντικά την εξισορρόπηση του φόρτου και τις επιδόσεις του αλγορίθμου.

Τώρα, ο επεξεργαστής i αποθηκεύει το σύνολο γειτόνων N_v όλων των v ∈ V_i. Παρατηρούμε ότι για έναν κόμβο w ∈ (V_i - V_i^c), το σύνολο γειτόνων N_w μπορεί να περιέχει ορισμένους κόμβους που δεν βρίσκονται στο V_i. Τέτοιοι γείτονες του w, που δεν βρίσκονται στο V_i, μπορούν να απομακρυνθούν με ασφάλεια από το N_w και ο αριθμός των τριγώνων που προσπίπτουν σε όλα τα v ∈ V_i^c μπορεί ακόμα να υπολογιστεί σωστά. Αλλά, η παρουσία αυτών των κόμβων στο N_w δεν επηρεάζει την ορθότητα του αλγορίθμου. Ωστόσο, δεν αποθηκεύουμε τέτοιους κόμβους στο N_w για να βελτιστοποιήσουμε τη χρήση μνήμης. Το Σχήμα 4 δείχνει τις διαφορές στη χρήση της μνήμης με και χωρίς αυτή τη βελτιστοποίηση για δύο δίκτυα: Miami και LiveJournal. Όπως δείχνουν τα πειραματικά αποτελέσματα, αυτή η βελτιστοποίηση εξοικονομεί περίπου το 50% του χώρου μνήμης. Το Σχήμα 4 δείχνει επίσης τη δυνατότητα επέκτασης της μνήμης του PATRIC: όσο περισσότεροι επεξεργαστές χρησιμοποιούνται, κάθε επεξεργαστής καταναλώνει λιγότερο χώρο μνήμης.



Σχήμα 4: Χρήση μνήμης με και χωρίς τη βελτιστοποίηση της αποθήκευσης δεδομένων.

3.3 ΜΕΤΡΗΣΗ ΤΡΙΓΩΝΩΝ

Μόλις κάθε επεξεργαστής i έχει το διαμέρισμά του G_i (V_i, E_i), χρησιμοποιείται ο αλγόριθμος *NodeIteratorN* που παρουσιάστηκε στο Κεφάλαιο 2, για να μετρήσει τα τρίγωνα στα G_i για κάθε core node v ∈ V_i^c. Τα σύνολα γειτόνων N_w για τους κόμβους w ∈ (V_i - V_i^c) βοηθούν μόνο στην εύρεση των ακμών μεταξύ των γειτόνων των core nodes. Για να μπορέσουμε να χρησιμοποιήσουμε μια αποτελεσματική λειτουργία τομής, τα σύνολα N_v, για όλα τα v ∈ V_i, ταξινομούνται. Ο κώδικας που εκτελείται από τον επεξεργαστή i δίνεται στο Σχήμα 5.

```
1: for v ∈ V_i do
2:     sort N_v in ascending order
3: T ← 0
4: for v ∈ V_i^c do
5:     for u ∈ N_v do
6:         S ← N_v ∩ N_u
7:         T ← T + |S|
8: return T
```

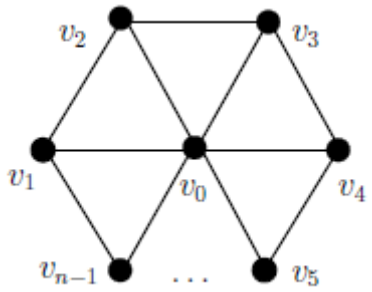
Σχήμα 5: Ο αλγόριθμος που εκτελείται από τον επεξεργαστή i για την μέτρηση των τριγώνων στο G_i (V_i, E_i).

Μόλις όλοι οι επεξεργαστές ολοκληρώσουν τα βήματα μέτρησης, οι μετρήσεις από όλους τους επεξεργαστές συγκεντρώνονται σε μία μόνο μέτρηση, η οποία παίρνει O(logP) χρόνο. Η διάταξη των κόμβων, η κατασκευή του N_v και τα ξεχωριστά διαμερίσματα V_i^c, βεβαιώνουν ότι κάθε τρίγωνο στο δίκτυο εμφανίζεται ακριβώς σε ένα διαμέρισμα G_i. Έτσι, η ορθότητα του διαδοχικού αλγορίθμου *NodeIteratorN*, που παρουσιάζεται στο Κεφάλαιο 2, εξασφαλίζει ότι κάθε τρίγωνο μετράται ακριβώς μία φορά.

3.4 ΕΞΙΣΟΡΡΟΠΗΣΗ ΦΟΡΤΟΥ ΣΤΟΝ PATRIC

Ένας παράλληλος αλγόριθμος ολοκληρώνεται όταν όλοι οι επεξεργαστές ολοκληρώσουν τις εργασίες τους. Έτσι, για να μειωθεί ο χρόνος εκτέλεσης ενός παράλληλου αλγορίθμου, είναι επιθυμητό κανένας επεξεργαστής να μην παραμένει αδρανής και όλοι να ολοκληρώσουν τις εκτελέσεις τους σχεδόν ταυτόχρονα. Επιπλέον, για την διαχείριση ενός πολύ μεγάλου δικτύου, είναι επίσης επιθυμητό όλα τα διαμερίσματα G_i (V_i, E_i) να απαιτούν σχεδόν το ίδιο μέγεθος χώρου μνήμης.

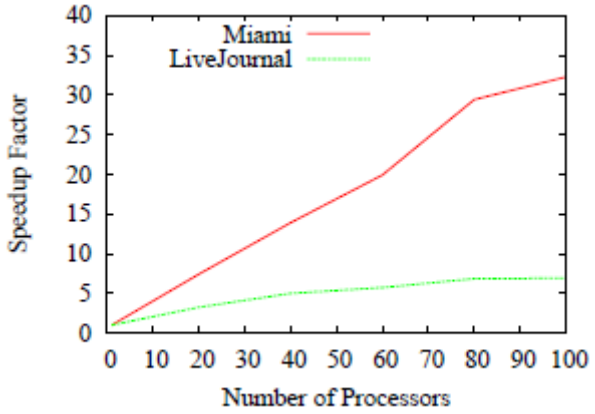
Στο Κεφάλαιο 2, συζητήσαμε πώς η ταξινόμηση με βάση το βαθμό των κόμβων μπορεί να μειώσει το χρόνο εκτέλεσης του διαδοχικού αλγορίθμου και συνεπώς να μειώσει τον χρόνο εκτέλεσης του τοπικού υπολογισμού σε κάθε επεξεργαστή i. Παρατηρούμε ότι αυτή η ταξινόμηση με βάση το βαθμό παρέχει επίσης εξισορρόπηση φόρτου σε κάποιο βαθμό, τόσο από την άποψη του χρόνου εκτέλεσης όσο και του χώρου, χωρίς επιπλέον κόστος. Παρατηρήστε το δίκτυο παραδειγμάτων που φαίνεται στο Σχήμα 6. Με μια αυθαίρετη διάταξη των κόμβων, το $|N_{v0}|$ μπορεί να είναι όσο το n-1, και ένας επεξεργαστής που περιέχει το v_0 ως core node είναι υπεύθυνος για την καταμέτρηση όλων των τριγώνων που συμβαίνουν στο v_0. Τότε, ο χρόνος εκτέλεσης του παράλληλου αλγορίθμου μπορεί ουσιαστικά να είναι ίδιος με αυτόν ενός διαδοχικού αλγορίθμου. Με την ταξινόμηση βάσει βαθμού, έχουμε $|N_{v0}| = 0$ και $|N_{vi}| \leq 3$ για όλα τα i. Τώρα αν οι core nodes είναι μοιρασμένοι εξίσου μεταξύ των επεξεργαστών, τόσο ο χώρος όσο και ο χρόνος υπολογισμού είναι σχεδόν εξισορροπημένοι.



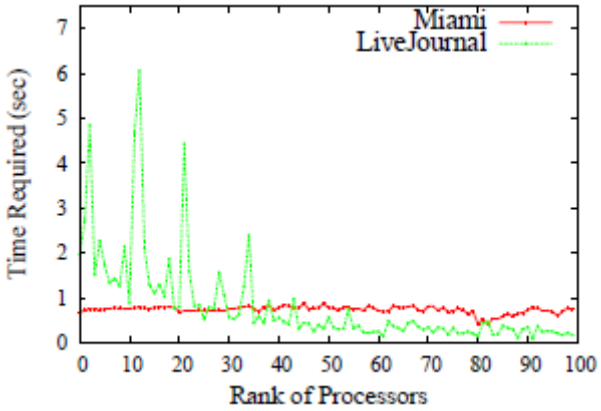
Σχήμα 6: Ένα δίκτυο με κατακόρυφη κατανομή βαθμού:
$d_{v0} = n\text{-}1$, $d_{vi\neq0} = 3$.

Αν και η ταξινόμηση βάσει βαθμού συμβάλλει στην άμβλυνση του αποτελέσματος της κατακόρυφης κατανομής βαθμού και του φόρτου εξισορρόπησης, σε μερικά πιο σύνθετα δίκτυα αποκαλύπτεται ότι η διανομή των core nodes εξίσου μεταξύ των επεξεργαστών δεν κάνει το φορτίο ισορροπημένο σε πολλές περιπτώσεις. Το Σχήμα 7 δείχνει την επιτάχυνση του παράλληλου αλγορίθμου με ίσο αριθμό core nodes να έχει εκχωρηθεί σε κάθε επεξεργαστή. Ο συντελεστής επιτάχυνσης (speedup factor) λόγω παραλληλισμού ορίζεται ως t_s / t_p, όπου t_s και t_p είναι οι χρόνοι υπολογισμού που απαιτούνται από έναν διαδοχικό και έναν παράλληλο αλγόριθμο, αντίστοιχα. Όπως φαίνεται στο Σχήμα 7, το δίκτυο LiveJournal έχει κακή επιτάχυνση, ενώ το δίκτυο Miami παρουσιάζει μια σχετικά καλύτερη επιτάχυνση. Αυτή η κακή επιτάχυνση για το δίκτυο LiveJournal είναι συνέπεια ενός εξαιρετικά μη ισορροπημένου υπολογισμού φόρτου στους επεξεργαστές, όπως φαίνεται στο Σχήμα 8. Αν και οι περισσότεροι από τους επεξεργαστές ολοκληρώνουν τα καθήκοντά τους σε λιγότερο από ένα δευτερόλεπτο, πολύ λίγοι από αυτούς χρειάζονται ασυνήθιστα μεγαλύτερο

χρόνο που οδηγεί σε κακή επιτάχυνση. Σε αντίθεση με το δίκτυο Miami, το δίκτυο LiveJournal έχει πολύ διαστρεβλωμένη κατανομή.



Σχήμα 7: Επιτάχυνση με ίσο αριθμό core nodes σε όλους τους επεξεργαστές.



Σχήμα 8: Υπολογιστικός φόρτος κάθε επεξεργαστή ξεχωριστά (ίσος αριθμός core nodes).

Στην επόμενη ενότητα, παρουσιάζουμε διάφορα συστήματα εξισορρόπησης φόρτου, που βελτιώνουν σημαντικά την απόδοση του αλγορίθμου μας.

Προτεινόμενα Συστήματα Εξισορρόπησης Φόρτου

Τα συστήματα εξισορρόπησης φόρτου που προτείνουμε απαιτούν κάποια προεπεξεργασία πριν εκτελέσουν τα κύρια βήματα για την καταμέτρηση των

τριγώνων. Έτσι, ο παράλληλος αλγόριθμος PATRIC λειτουργεί σε δύο φάσεις, όπως φαίνεται παρακάτω.

1. *Υπολογισμός ισορροπημένου φόρτου*: Αυτή η φάση υπολογίζει τα διαμερίσματα V_i^c, ώστε τα υπολογιστικά φορτία να είναι καλά ισορροπημένα.

2. *Μέτρηση τριγώνων*: Αυτή η φάση μετράει τα τρίγωνα ακολουθώντας τους αλγορίθμους των Σχημάτων 3 και 5.

Το υπολογιστικό κόστος για τη φάση 1 αναφέρεται ως *κόστος φόρτου εξισορρόπησης*, για τη φάση 2 ως *κόστος καταμέτρησης*, και το συνολικό κόστος για αυτές τις δύο φάσεις ως *συνολικό υπολογιστικό κόστος*. Προκειμένου να είναι δυνατή η ομοιόμορφη κατανομή φόρτου μεταξύ των επεξεργαστών, χρειαζόμαστε μια εκτίμηση του φορτίου υπολογισμού για τον υπολογισμό των τριγώνων. Για το σκοπό αυτό, ορίζουμε μια *συνάρτηση κόστους* f: V → ℝ, έτσι ώστε το f(v) να είναι το υπολογιστικό κόστος για την καταμέτρηση των προσπιπτόντων τριγώνων στον κόμβο v (γραμμές 4-7 στο Σχήμα 5). Στη συνέχεια, το συνολικό κόστος που προκύπτει για τον επεξεργαστή i δίνεται από τον τύπο $\sum_{\mathrm{v} \in V_i^c} f(v)$. Για να επιτευχθεί μια καλή εξισορρόπηση φόρτου, το $\sum_{\mathrm{v} \in V_i^c} f(v)$ θα πρέπει να είναι σχεδόν ίσο για όλα τα i. Έτσι, ο υπολογισμός του ισορροπημένου φόρτου αποτελείται από τα ακόλουθα:

1. **Υπολογισμός της f**: Υπολογίζουμε την f (v) για κάθε v ∈ V.

2. **Υπολογισμός διαμερισμάτων**: Προσδιορίζουμε τα P ξεχωριστά διαμερίσματα V_i^c έτσι ώστε να ισχύει

$$\sum_{\mathrm{v} \in V_i^c} f(v) \approx \frac{1}{P} \sum_{\mathrm{v} \in \mathrm{V}} f(v)$$

Ο παραπάνω υπολογισμός πρέπει επίσης να γίνει παράλληλα. Διαφορετικά, παίρνει τουλάχιστον Ω(n) χρόνο, ο οποίος μπορεί να εξαλείψει εντελώς το όφελος από την εξισορρόπηση του φόρτου ή ακόμα και να έχει αρνητικές επιπτώσεις στην απόδοση. Ο παραλληλισμός των παραπάνω υπολογισμών, ιδίως το βήμα υπολογισμού των διαμερισμάτων, είναι ένα μη τετριμμένο πρόβλημα. Στη συνέχεια, περιγράφουμε τον παράλληλο αλγόριθμο για την εκτέλεση του παραπάνω υπολογισμού.

Υπολογισμός της f:

Μπορεί να μην είναι δυνατό να υπολογιστεί με ακρίβεια η τιμή της f (v) πριν από την πραγματική μέτρηση των τριγώνων. Ευτυχώς, το Θεώρημα 2 (που είδαμε στο Κεφάλαιο 2) παρέχει μια μαθηματική διατύπωση του κόστους καταμέτρησης από την άποψη του αριθμού των κορυφών, των ακμών, του αρχικού βαθμό d και του πραγματικού βαθμού $\hat{d}$. Οδηγούμενοι από το Θεώρημα 2, έχουμε καταλήξει σε αρκετές συναρτήσεις προσέγγισης κόστους f(v), που παρατίθενται στον Πίνακα 2.

Κάθε συνάρτηση αντιστοιχεί σε ένα σύστημα εξισορρόπησης φόρτου. Η δεξιά στήλη του πίνακα περιέχει σύντομες σημειογραφίες που χρησιμοποιούνται για τον προσδιορισμό του κάθε συστήματος.

Node Function	Identifying Notation
$f(v) = 1$	$\mathbb{N}$
$f(v) = d_v$	$\mathbb{D}$
$f(v) = \hat{d}_v$	$\mathbb{DH}$
$f(v) = d_v \hat{d}_v$	$\mathbb{DDH}$
$f(v) = \hat{d}_v^{\,2}$	$\mathbb{DH}^2$
$f(v) = \sum_{u \in N_v} (\hat{d}_v + \hat{d}_u)$	$\mathbb{DPD}$

Πίνακας 2: Συναρτήσεις κόστους f(v) για συστήματα εξισορρόπησης φόρτου.

Το input δίκτυο δίνεται σε ένα αρχείο σε μορφή λίστας γειτνίασης: λίστα γειτνίασης του πρώτου κόμβου ακολουθούμενη από εκείνη του δεύτερο κόμβο και ούτω καθεξής. Το αρχείο input θεωρείται διαιρεμένο κατά μέγεθος (αριθμός bytes) σε P κομμάτια. Αρχικά, ο επεξεργαστής i διαβάζει το i-οστό τμήμα από το αρχείο παράλληλα. Έτσι ο επεξεργαστής i περιέχει τις λίστες γειτνίασης N_v για τους κόμβους v που βρίσκονται το i-οστό κομμάτι. Εάν το σύνορο ενός κομματιού πέσει στη μέση μιας λίστας γειτνίασης, το όριο μετατοπίζεται έτσι ώστε όλη η λίστα γειτνίασης να είναι μόνο σε ένα κομμάτι. (Σημειώστε ότι οι κόμβοι στο το i-οστό κομμάτι δεν αποτελούν απαραίτητα τους core nodes V_i^c για τον επεξεργαστή i. Αυτά τα κομμάτια χρησιμοποιούνται μόνο για τον υπολογισμό του ισορροπημένου φόρτου και την εύρεση των πραγματικών διαμερισμάτων V_i^c.) Έτσι, ο βαθμός d_v του κάθε κόμβου v στο i-οστό κομμάτι είναι γνωστός στον επεξεργαστή i. Κάθε επεξεργαστής i υπολογίζει το f(v) για όλους τους κόμβους v στο i-οστό κομμάτι παράλληλα. Ο υπολογισμός του f(v) για διαφορετικά συστήματα δίνεται παρακάτω.

• f(v) = 1: Αυτή η συνάρτηση δεν απαιτεί υπολογισμό και εκχωρεί ίσο αριθμό από core nodes σε κάθε επεξεργαστή.

• f(v) = d_v: Αυτή η συνάρτηση επίσης δεν απαιτεί κανέναν υπολογισμός, καθώς το d_v είναι ήδη γνωστό στον επεξεργαστή i. Αυτό το σύστημα εκχωρεί ίσο αριθμό ακμών σε κάθε επεξεργαστή.

• f(v) = $\hat{d}_v$: Ο επεξεργαστής i χρειάζεται τους βαθμούς των γειτόνων του v για να υπολογίσει το $\hat{d}_v$. Ορισμένοι κόμβοι u ∈ N_v μπορεί να βρίσκονται σε άλλους επεξεργαστές. Ο επεξεργαστής i επικοινωνεί με εκείνους τους επεξεργαστές για να βρει τους d_u.

• f(v) = d_v $\hat{d}_v$: Ο υπολογισμός του d_v $\hat{d}_v$ απαιτεί τον υπολογισμό των d_v και $\hat{d}_v$, τα οποία έχουν συζητηθεί παραπάνω.

• f(v) = $\hat{d}_v^2$: Απαιτεί τον υπολογισμό του $\hat{d}_v$, ο οποίος γίνεται όπως περιγράφεται παραπάνω.

• f(v) = $\sum_{u \in N_v} (\hat{d}v + \hat{d}u)$: Αυτή η συνάρτηση δίνει την καλύτερο εκτίμηση του κόστους καταμέτρησης. Ωστόσο, ο υπολογισμός αυτής της συνάρτησης απαιτεί δύο επίπεδα επικοινωνίας, όπως περιγράφεται παρακάτω.

i. Υπολογισμός $\hat{d}_v$: συζητήθηκε παραπάνω.

ii. Υπολογισμός $\hat{d}_u$ για τους κόμβους u ∈ N_v: αφού υπολογιστεί το $\hat{d}_v$ για όλα τα v στο i-οστό κομμάτι από όλους τους επεξεργαστές i, αυτός ο επεξεργαστής επικοινωνεί με τους επεξεργαστές j για να πάρει το $\hat{d}_u$, όπου το u βρίσκεται στο j-οστό κομμάτι.

Υπολογισμός διαμερισμάτων:

Δεδομένου ότι κάθε επεξεργαστής i γνωρίζει την f(v) για όλα τα v στο i-οστό κομμάτι, όπως περιγράφεται παραπάνω, στόχος μας είναι να χωρίσουμε το V σε P ξεχωριστά υποσύνολα V_i^c τέτοια ώστε $\sum_{v \in V_i^c} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v)$. Έστω ότι οι κόμβοι του V είναι αριθμημένοι ως 0, 1, 2,. . . , n - 1 με αυτή τη σειρά, πρώτα υπολογίζεται το συσσωρευτικό άθροισμα g(v) = $\sum_{k=0}^{v} f(k)$ για κάθε v ∈ V χρησιμοποιώντας έναν παράλληλο αλγόριθμο που συνοψίζεται παρακάτω:

i. Έστω οι κόμβοι στο i-οστό κομμάτι είναι οι n_i, n_i + 1,. . . , n_{i+1} - 1. Κάθε επεξεργαστής i βρίσκει το τοπικό άθροισμα S_i = $\sum_{v=ni}^{ni+1-1} f(v)$.

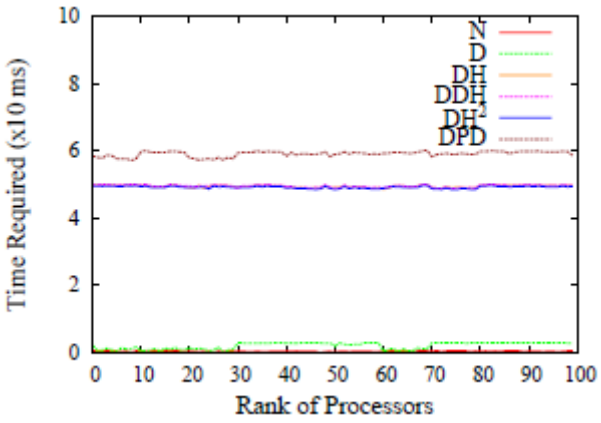
ii. Οι επεξεργαστές υπολογίζουν τα συσσωρευτικά τοπικά αθροίσματα R_i ως εξής: ο επεξεργαστής 0 θέτει το $R_0 = S_0$ και στέλνει το R_0 στον επεξεργαστή 1. Κάθε άλλος επεξεργαστής i περιμένει μέχρι να λάβει το R_{i-1} από τον επεξεργαστή i-1. Μόλις ληφθεί το R_{i-1}, υπολογίζει το $R_i = R_{i-1} + S_i$ και στέλνει το R_i στον επεξεργαστή i+1.

iii. Κάθε επεξεργαστής i υπολογίζει το συσσωρευτικό άθροισμα g(v) ως εξής: $g(n_i) = R_{i-1} + f(n_i)$ και $g(v) = g(v-1) + f(v)$ για $n_i < v < n_{i+1}$.

Στη συνέχεια, θα δείξουμε πώς τα διαμερίσματα V_i^c μπορούν να υπολογιστούν από τα συσσωρευτικά αθροίσματα g(v) για όλα τα v ∈ V. Παρατηρήστε ότι R_{P-1} = $\sum_{v \in V} f(v)$. Ο επεξεργαστής (P - 1) υπολογίζει το α = $\frac{1}{P} \sum_{v \in V} f(v) = \frac{1}{P} R_{P-1}$ και το μεταδίδει σε όλους τους άλλους επεξεργαστές. Κάθε επεξεργαστής i βρίσκει τους

οριακούς κόμβους x_j σε κάθε κομμάτι του: ο κόμβος x_j είναι ο j-οστός οριακός κόμβος αν και μόνο αν $g(x_j - 1) < j\alpha \leq g(x_j)$. Ο επεξεργαστής i μπορεί να βρει τους οριακούς κόμβους στο i-οστό κομμάτι με απλή σάρωση του συσσωρευτικού αθροίσματος g(v) για τους κόμβους σε αυτό το κομμάτι. Παρατηρήστε ότι ορισμένα κομμάτια μπορεί να έχουν πολλούς οριακούς κόμβους και μερικά να μην έχουν κανένα. Για κάθε οριακό κόμβο x_j που βρίσκεται στο i-οστό κομμάτι, ο επεξεργαστής i στέλνει μηνύματα που περιέχουν τα x_j-1 και x_j στον επεξεργαστή j-1 και j, αντίστοιχα. Κάθε επεξεργαστής j λαμβάνει ακριβώς δύο μηνύματα που περιέχουν τα x_j και x_{j+1}-1. Στη συνέχεια, το διαμέρισμα V_j^c είναι το σύνολο των κόμβων $\{x_j, x_j+1, \ldots, x_{j+1}-1\}$.

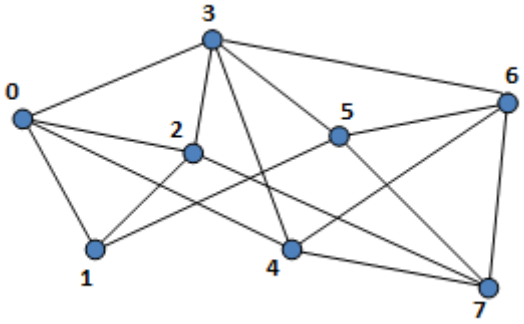
Δεδομένου ότι το σύστημα DPD απαιτεί δύο επίπεδα επικοινωνίας για τον υπολογισμό της f(v), έχει το μεγαλύτερο κόστος εξισορρόπησης φόρτου μεταξύ όλων των συστημάτων. Ο υπολογισμός της f(v) για το DPD απαιτεί $O(\frac{m}{P} + P \log P)$ χρόνο. Ο υπολογισμός των διαμερισμάτων έχει πολυπλοκότητα χρόνου εκτέλεσης $O(\frac{m}{P} + P)$. Ως εκ τούτου, το κόστος εξισορρόπησης φόρτου της DPD δίνεται από τον τύπο $O(\frac{m}{P} + P \log P)$. Το Σχήμα 9 δείχνει ένα πειραματικό αποτέλεσμα του κόστους εξισορρόπησης φόρτου για διαφορετικά συστήματα, στο δίκτυο LiveJournal. Το σύστημα N έχει το χαμηλότερο κόστος και το DPD το υψηλότερο. Τα συστήματα DH, DH^2 και DDH έχουν αρκετά παρόμοιο κόστος εξισορρόπησης φόρτου.



Σχήμα 9: Το κόστος εξισορρόπησης φόρτου για το δίκτυο LiveJournal με διαφορετικά συστήματα.

Παράδειγμα διάσπασης δικτύου:

Έστω ότι έχουμε το παρακάτω γράφημα και θέλουμε να το χωρίσουμε σε 4 διαμερίσματα:



Η adjacency list του γραφήματος είναι η εξής:

0 1

0 2

0 3

0 4

1 2

1 5

2 3

2 7

3 4

3 5

3 6

4 6

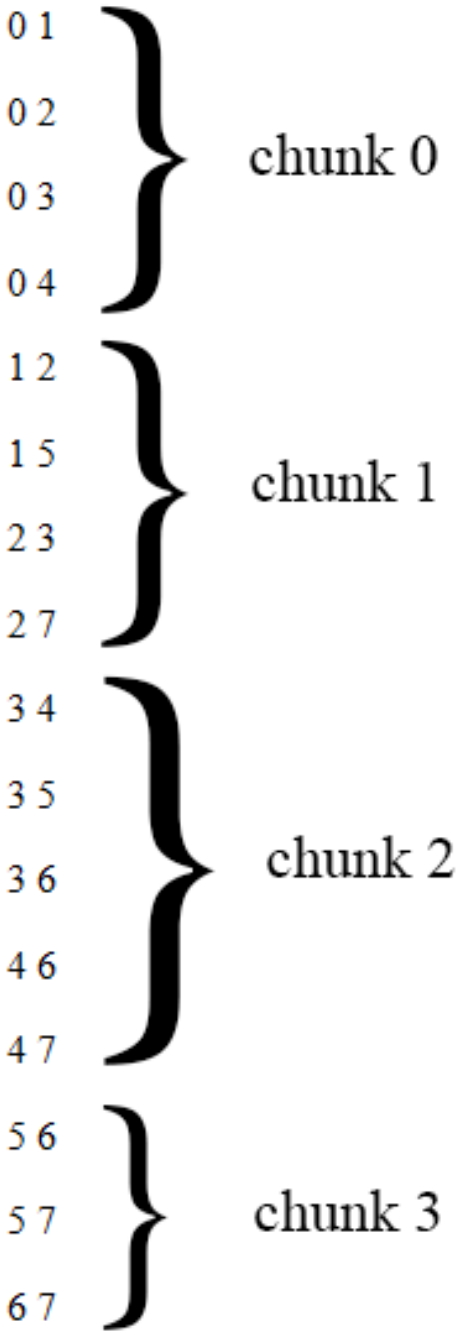
4 7

5 6

5 7

6 7

Αυτή η λίστα πρέπει να χωριστεί σε 4 τμήματα, στα κατάλληλα σημεία, με βάση το μέγεθος του αρχείου. Στη μορφή που βρίσκεται, τα 16 ζευγάρια πρέπει να χωριστούν έτσι ώστε κάθε τμήμα να περιέχει 4. Ξεκινώντας από την αρχή, το chunk 0 παίρνει τα ζευγάρια 0 1, 0 2, 0 3, 0 4. Το chunk 1 παίρνει τα 1 2, 1 5, 2 3, 2 7. Το chunk 2 παίρνει τα 3 4, 3 5, 3 6, 4 6 και φτάνει στο όριό του. Όμως οι γείτονες του 4 κόβονται στη μέση, οπότε το όριο του chunk 2 πρέπει να κάνει ολίσθηση προς τα κάτω μέχρι να διαβαστούν όλοι οι γείτονες του 4. Έτσι, διαβάζει και το ζεύγος 4 7. Έπειτα, το chunk 3 συνεχίζει με τους κόμβους που βρίσκονται μετά το 4 και παίρνει τα ζεύγη 5 6, 5 7, 6 7. Τα σημεία διάσπασης της λίστας και τα νέα τμήματα φαίνονται παρακάτω:



3.5 ΑΝΑΠΤΥΞΗ ΑΛΓΟΡΙΘΜΟΥ ΣΕ ΚΩΔΙΚΑ C++

Ο κώδικας αναπτύχθηκε σε C++ στην πλατφόρμα Visual Studio. Έγινε χρήση της βιβλιοθήκης MPI για την επίτευξη του παράλληλου προγραμματισμού.

Συναρτήσεις:

*int *partition_file (string filename, const char *delimiter, int chunks)*

Καλείται όταν έχουμε παραπάνω από έναν επεξεργαστή. Χρησιμοποιείται για το διαχωρισμό του input file σε chunks, των οποίων το πλήθος στέλνεται ως παράμετρος στη συνάρτηση. Στη συνέχεια υπολογίζεται το chunk size, το οποίο ισούται με file_size / chunks. Επίσης chunk_size = split_boundary (το σημείο διαχωρισμού). Μετά διαβάζουμε κάθε γραμμή του αρχείου και παίρνουμε κάθε ακμή με τη βοήθεια της συνάρτησης ***parse_line***. Αν έχουμε φτάσει σε σημείο που πρέπει να γίνει διαχωρισμός, ελέγχουμε αν είμαστε στη μέση μίας λίστας γειτνίασης. Αν δεν είμαστε, τότε δημιουργείται ένα νέο αρχείο για το επόμενο chunk. Αλλιώς, κρατάμε ανοιχτό το προηγούμενο αρχείο για να συνεχίσουμε να γράφουμε σε εκείνο. Στο τέλος των ελέγχων, γράφουμε τη γραμμή στο κατάλληλο αρχείο που έχουμε ανοιχτό εκείνη τη στιγμή. Η συνάρτηση επιστρέφει έναν integer πίνακα με τα όρια στα οποία πρέπει να χωριστεί το input file. (βλ. κώδικα στο Παράρτημα, Συνάρτηση 1).

Έπειτα, στη main, κάθε επεξεργαστής αναλαμβάνει και ένα chunk και καλείται η συνάρτηση parse_file για το καθένα από αυτά.

*map<int, Node> parse_file (string filename, const char *delimiter, int n_proc)*

Διαβάζει μία μία τις γραμμές του αρχείου και επιστρέφει τους κόμβους που περιέχει σε ένα map. (βλ. κώδικα στο Παράρτημα, Συνάρτηση 2).

Πίσω στη main, για κάθε κόμβο του συνόλου Vi, για κάθε γείτονα του κόμβου, καλείται η συνάρτηση ***intersectionCount***. Αν έχουμε ένα επεξεργαστή, δεν χρειάζεται να γίνει κάποιος έλεγχος για το αν ο γείτονας του κόμβου βρίσκεται στο Vi, γιατί έτσι κι αλλιώς έχουμε μόνο ένα αρχείο, άρα Vi = 1. Αν όμως έχουμε 2 ή παραπάνω επεξεργαστές, πρέπει να ελέγξουμε αν ο γείτονας είναι core node στο Vi. Αν ναι, καλείται η ***intersectionCount***. Αν όχι, τότε στέλνουμε τον τρέχων κόμβο στον επεξεργαστή (στο Vi) όπου ο γείτονας του κόμβου είναι core node.

int intersectionCount (vector<int> a, vector<int> b)

Αυτή η συνάρτηση αποτελείται από τα κυριότερα βήματα του αλγορίθμου NodeIteratorN για την καταμέτρηση τριγώνων σε κάθε chunk. Παίρνει ως ορίσματα δύο κόμβους a και b (οι οποίοι είναι γείτονες) και επιστρέφει τον αριθμό των γειτόνων που είναι κοινοί και για τους δύο κόμβους. (Αν δύο κόμβοι είναι γείτονες και ένας τρίτος κόμβος είναι γείτονας και με τον πρώτο και με τον δεύτερο, τότε σχηματίζεται ένα τρίγωνο). (βλ. κώδικα στο Παράρτημα, Συνάρτηση 3).

Τέλος, στη main περιμένουμε μέχρι όλοι οι επεξεργαστές να τελειώσουν με τις εργασίες τους και να φτάσουν στο MPI_Barrier. Έπειτα, με τη συνάρτηση MPI_Reduce, αθροίζονται όλα τα counts των τριγώνων σε μία μεταβλητή, global_sum.

ΚΕΦΑΛΑΙΟ 4:

ΠΕΙΡΑΜΑΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ

ΠΕΙΡΑΜΑΤΙΚΑ ΑΠΟΤΕΛΕΣΜΑΤΑ

Τα πειράματα διεξήχθησαν σε υπολογιστή με επεξεργαστή Intel Core i5-5200U 2.20 GHz και μνήμη RAM 8 GB.

4.1 ΠΡΟΕΤΟΙΜΑΣΙΑ ΑΡΧΕΙΟΥ ΕΙΣΟΔΟΥ

Όπως έχει στηθεί ο κώδικας σε C++, δέχεται αρχεία εισόδου των οποίων οι κόμβοι είναι ταξινομημένοι και κάθε ακμή υπάρχει μόνο μία φορά (π.χ. υπάρχει μόνο η ακμή 0-1, όχι και η 1-0). Για αυτό το λόγο, αναπτύχθηκε ένα πρόγραμμα σε Java. Το πρόγραμμα διαβάζει το input file line by line και ελέγχει για κάθε γραμμή αν ο πρώτος κόμβος είναι μικρότερος από τον δεύτερο. Με αυτό τον τρόπο, "κόβονται" οι τυχόν διπλές ακμές. Αν ισχύει η συνθήκη, οι κόμβοι αποθηκεύονται σε ένα HashMap. Στο τέλος, το HashMap ταξινομείται και γράφεται σε ένα αρχείο, το οποίο πλέον αποτελεί το input file για τον παράλληλο αλγόριθμο.

4.2 ΠΡΑΓΜΑΤΙΚΑ ΣΥΝΟΛΑ ΔΕΔΟΜΕΝΩΝ

Τρέξαμε τον αλγόριθμο PATRIC πάνω σε πραγματικά σύνολα δεδομένων και συγκεκριμένα πάνω σε γραφήματα που έχουν δημιουργηθεί από διάφορες διαδικτυακές σελίδες (π.χ. σελίδες κοινωνικής δικτύωσης). Επιλέχθηκαν δίκτυα με διαφορετικό αριθμό κόμβων και ακμών, κάποια μικρότερα και κάποια μεγαλύτερα και πιο πυκνά σε έκταση.

Δίκτυα που χρησιμοποιήθηκαν:

- **ego-Facebook**: Αυτό το σύνολο δεδομένων αποτελείται από «κύκλους» (ή «λίστες φίλων») από το Facebook. Τα δεδομένα συλλέχθηκαν από τους συμμετέχοντες στην έρευνα, χρησιμοποιώντας την εφαρμογή του Facebook. Το σύνολο δεδομένων περιλαμβάνει χαρακτηριστικά κόμβων (προφίλ), κύκλους και ego networks.

- **ca-GrQc**: Το δίκτυο συνεργασίας Arxiv GR-QC (General Relativity and Quantum Cosmology) προέρχεται από την ιστοσελίδα arXiv και καλύπτει τις επιστημονικές συνεργασίες μεταξύ συγγραφέων στην κατηγορία General Relativity και Quantum Cosmology. Εάν ένας συγγραφέας i συνέγραψε ένα paper με έναν συγγραφέα j, το γράφημα περιέχει μια μη κατευθυνόμενη ακμή από το i στο j. Αν το paper έχει συγγραφεί από k συγγραφείς, αυτό δημιουργεί

ένα εντελώς συνδεδεμένο (υπο)γράφημα στους k κόμβους. Τα τρία δίκτυα που ακολουθούν προέρχονται επίσης από το e-print arXiv και το καθένα καλύπτει τις επιστημονικές συνεργασίες μεταξύ συγγραφέων σε μία διαφορετική κατηγορία.

- **ca-HepTh**: Το δίκτυο συνεργασίας Arxiv HEP-TH (High Energy Physics - Theory) καλύπτει την κατηγορία High Energy Physics - Theory.

- **ca-HepPh**: Το δίκτυο Arxiv HEP-TH (High Energy Physics - Theory) την κατηγορία High Energy Physics - Phenomenology.

- **ca-AstroPh**: Τέλος, το δίκτυο Arxiv ASTRO-PH (Astro Physics) καλύπτει την κατηγορία Astro Physics.

- **email-Enron**: Το δίκτυο επικοινωνίας email-Enron καλύπτει την επικοινωνία μέσω ηλεκτρονικού ταχυδρομείου μέσα σε ένα σύνολο δεδομένων περίπου μισού εκατομμυρίου emails. Αυτά τα δεδομένα δημοσιοποιήθηκαν αρχικά στο διαδίκτυο από την Federal Energy Regulatory Commission κατά τη διάρκεια της έρευνάς της. Οι κόμβοι του δικτύου είναι διευθύνσεις ηλεκτρονικού ταχυδρομείου και αν μια διεύθυνση έστειλε τουλάχιστον ένα μήνυμα στη διεύθυνση j, το γράφημα περιέχει μια μη κατευθυνόμενη ακμή από το i στο j.

- **soc-Slashdot0922**: Το Slashdot είναι μία ιστοσελίδα σχετική με τεχνολογικά νέα και με μία συγκεκριμένη κοινότητα χρηστών. Το 2002 το Slashdot παρουσίασε το Slashdot Zoo που είναι ένα feature το οποίο επιτρέπει στους χρήστες να επισημάνουν τον άλλον σαν φίλο ή εχθρό. Το δίκτυο περιέχει συνδέσεις φίλων/εχθρών μεταξύ των χρηστών του Slashdot. Δημιουργήθηκε τον Φεβρουάριο του 2009.

- **com-Amazon**: Το δίκτυο δημιουργήθηκε με βάση την ιστοσελίδα Amazon. Αν ένα προϊόν i αγοράζεται συχνά μαζί με το προϊόν j, το γράφημα περιέχει μια μη κατευθυνόμενη ακμή από το i στο j.

- **com-LiveJournal**: Το LiveJournal είναι ένα μέσο κοινωνικής δικτύωσης που επιτρέπει στα μέλη του να διατηρούν περιοδικά, ατομικά και ομαδικά ιστολόγια και επίσης να δηλώνουν ποια άλλα μέλη είναι φίλοι τους.

	Αριθμός κόμβων	Αριθμός Ακμών
ego-Facebook	4,039	88,234
ca-GrQc	5,242	14,496
ca-HepTh	9,877	25,998
ca-HepPh	12,008	118,521
ca-AstroPh	18,772	198,110
email-Enron	36,692	183,831
soc-Slashdot0922	77,360	905,468
com-Amazon	334,863	925,872
com-LiveJournal	3,997,962	34,681,189

Πίνακας 3: Αριθμοί κόμβων και ακμών στα σύνολα δεδομένων μας.

4.3 Αποτελέσματα ταχύτητας

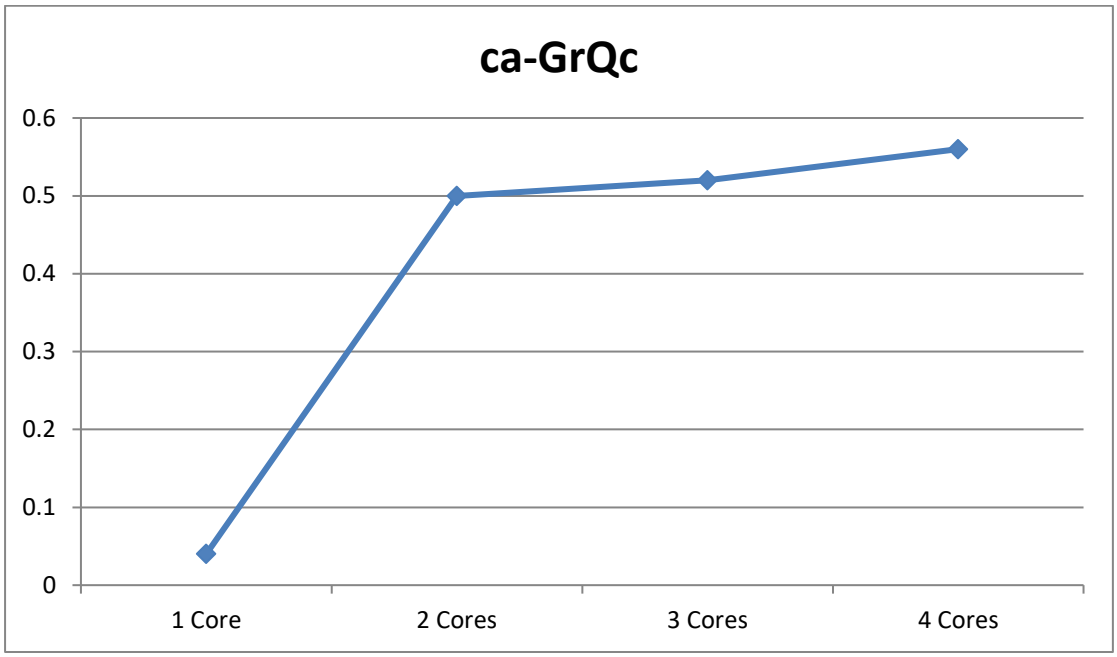
Έχοντας τα παραπάνω δίκτυα, τρέχουμε τον αλγόριθμο PATRIC με διαφορετικό αριθμό επεξεργαστών κάθε φορά και συγκρίνουμε το χρόνο ολοκλήρωσης. Η υπολογιστική ισχύς που διαθέτουμε είναι 4 επεξεργαστές. Τρέχοντας τον αλγόριθμο με έναν επεξεργαστή, εκτελείται ουσιαστικά απευθείας ο αλγόριθμος NodeIteratorN. Ο χρόνος για κάθε περίπτωση που φαίνεται στον παρακάτω πίνακα δίνεται σε λεπτά.

	1 Core (NodeIteratorN)	2 Cores	3 Cores	4 Cores	Αριθμός Τριγώνων
ego-Facebook	4.55	3.67	3.65	3.77	1,612,010
ca-GrQc	0.04	0.50	0.52	0.56	48,260
ca-HepTh	0.05	1.72	1.93	1.91	28,339
ca-HepPh	13.66	12.67	11.41	11.33	3,358,499
ca-AstroPh	16.02	15.91	13.86	11.78	1,351,441
email-Enron	23.12	22.64	21.32	19.79	727,044
soc-Slashdot0922	67.43	65.07	57.13	51.88	602,592
com-Amazon	289.76	239.55	226.04	211.71	667,129
com-LiveJournal	1526.74	1338.19	1215.63	1178.50	177,820,130

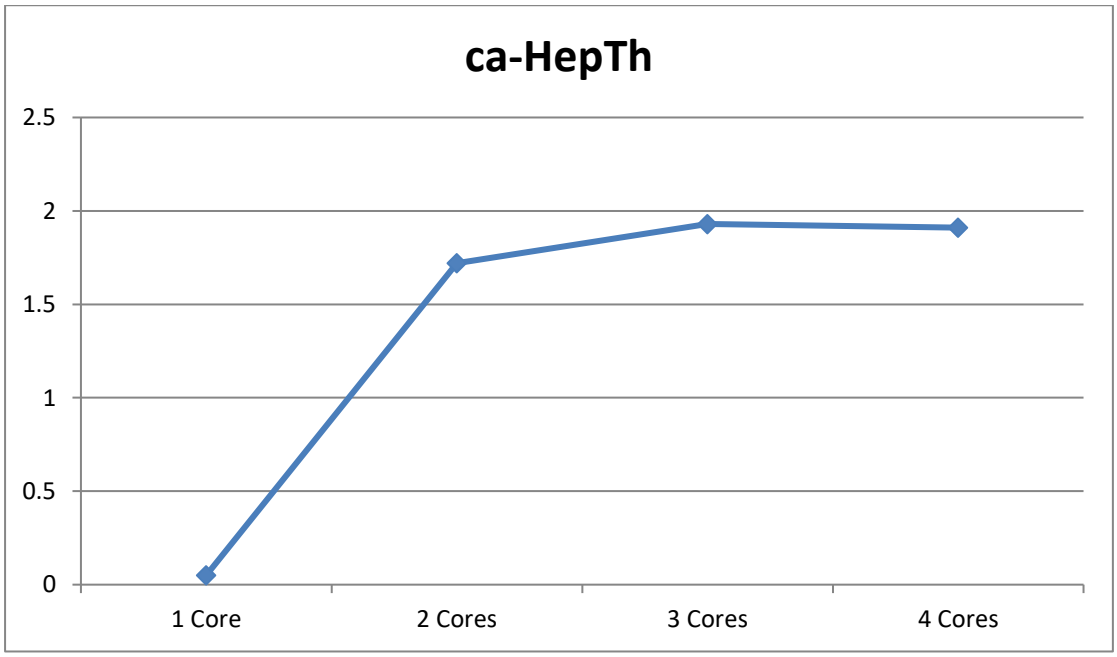
Πίνακας 4: Χρόνος εκτέλεσης (σε λεπτά) για διαφορετικό αριθμό επεξεργαστών.

Παρακάτω παρατηρούμε σε γραφική παράσταση πώς συμπεριφέρεται ο αλγόριθμος σε δίκτυα διαφορετικού μεγέθους. Στον άξονα x αναφέρεται ο αριθμός των κόμβων και στον άξονα y ο χρόνος σε λεπτά.

Σε δύο μικρά δίκτυα, το ca-GrQc (5,242 κόμβοι) και το ca-HepTh (9,877 κόμβοι) παρατηρούμε ότι με έναν επεξεργαστή η μέτρηση των τριγώνων γίνεται πολύ γρήγορα. Αυτό συμβαίνει γιατί ο αριθμός των τριγώνων είναι πολύ μικρός, 48,260 και 28,339 αντίστοιχα. Έτσι, στους 2, 3 και 4 πυρήνες, όπου απαιτείται χρόνος για τη διάσπαση του γράφου, υπάρχει και χρονική καθυστέρηση που δεν βελτιώνει το συνολικό χρόνο εκτέλεσης.

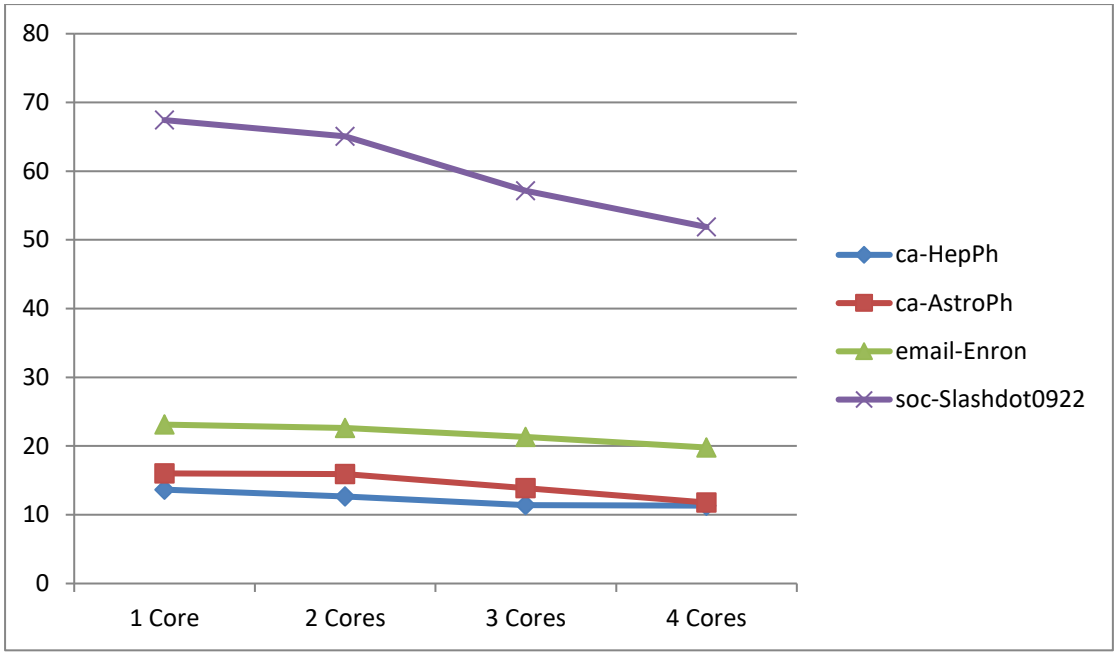


Σχήμα 10: Γραφική αναπαράσταση εκτέλεσης του PATRIC στο δίκτυο ca-GrQc.



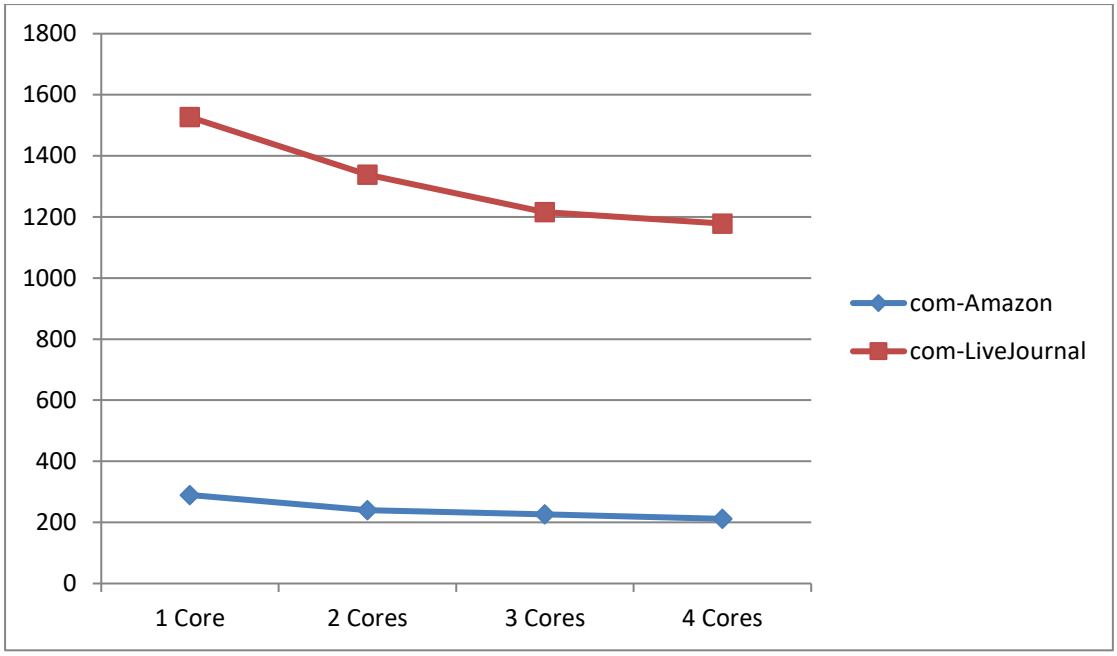
Σχήμα 11: Γραφική αναπαράσταση εκτέλεσης του PATRIC στο δίκτυο ca-HepTh.

Στο παρακάτω γράφημα έχουμε 4 καμπύλες για καθένα από 4 δίκτυα μεσαίου μεγέθους. Παρατηρούμε σε όλα ότι ο χρόνος μειώνεται καθώς αυξάνεται ο αριθμός των επεξεργαστών. Ειδικότερα, φαίνεται ότι στο μεγαλύτερο από τα 4 δίκτυα, ο χρόνος εκτέλεσης μειώνεται πιο ραγδαία.



Σχήμα 12: Γραφική αναπαράσταση εκτέλεσης του PATRIC σε δίκτυα μεσαίου μεγέθους.

Παρακάτω παραθέτουμε μία γραφική παράσταση για δύο μεγάλα δίκτυα. Το com-Amazon με 334,863 κόμβους και το com-LiveJournal με 3,997,962. Παρατηρούμε ότι στο πρώτο ο χρόνος εκτέλεσης μειώνεται σταδιακά καθώς αυξάνεται ο αριθμός των πυρήνων, αλλά ότι στο δεύτερο μειώνεται πολύ πιο ραγδαία.



Σχήμα 13: Γραφική αναπαράσταση εκτέλεσης του PATRIC στα δίκτυο com-Amazon και com-LiveJournal.

ΚΕΦΑΛΑΙΟ 5:

ΣΥΜΠΕΡΑΣΜΑΤΑ

ΣΥΜΠΕΡΑΣΜΑΤΑ

Η εργασία αυτή είχε ως στόχο να παρουσιάσει αποδοτικούς τρόπους καταμέτρησης τριγώνων που βρίσκονται σε γραφήματα. Το ενδιαφέρον μας εστιάστηκε κυρίως σε γραφήματα μεγάλου μεγέθους, που αποτελούνται από χιλιάδες κόμβους και ακμές.

Εξετάσαμε δύο αλγορίθμους για την μέτρηση των τριγώνων. Ο πρώτος ήταν ο κεντρικοποιημένος NODE ITERATOR N και ο δεύτερος ήταν ο παράλληλος PATRIC. Ο NodeIteratorN υπολόγιζε ακολουθιακά τον αριθμό των τριγώνων σε ολόκληρο το γράφημα. Από την άλλη, ο Patric εκτελούνταν παράλληλα και χώριζε το γράφημα σε n κομμάτια που ορίζονταν πριν από κάθε εκτέλεση. Κάθε επεξεργαστή αναλάμβανε ένα κομμάτι και πάνω σε αυτό εκτελούσε τον αλγόριθμο NodeIteratorN για την μέτρηση των τριγώνων. Όταν τελείωναν όλοι οι επεξεργαστές, αθροιζόταν ο αριθμός των τριγώνων από όλους και προέκυπτε το συνολικό count.

Το πρόβλημα που έπρεπε να μελετήσουμε ήταν αν μειώνεται ο χρόνος εκτέλεσης με τον αλγόριθμο Patric σε σχέση με τον NodeIteratorN. Και αν συμβαίνει αυτό, κατά πόσο μειώνεται και τι σχέση έχει με τον αριθμό των επεξεργαστών. Το τελικό συμπέρασμα είναι ότι σχεδόν σε όλες τις περιπτώσεις, ο χρόνος εκτέλεσης όντως μειώνεται, και ειδικά σε κάποια γραφήματα αρκετά σημαντικά. Επίσης, όσο περισσότερους επεξεργαστές χρησιμοποιούμε, τόσο μειώνεται ο χρόνος. Υπήρξε μόνο μία εξαίρεση, όπου ο NodeIteratorN ήταν πιο γρήγορος από τον Patric. Αυτή η περίπτωση ήταν όταν είχαμε μικρά γραφήματα με πολύ μικρό αριθμό τριγώνων, όπου ο χρόνος που απαιτούνταν για τη διάσπαση του δικτύου σε υπογραφήματα λειτουργούσε ανασταλτικά.

Τα ανοιχτά ζητήματα που θα μπορούσαν να διερευνηθούν από μία μελλοντική επέκταση της παρούσας εργασίας είναι αρκετά και πολύ ενδιαφέροντα. Κατ' αρχάς θα μπορούσε να μελετηθεί και ο χρόνος που χρειάζεται για τον διαχωρισμό του γραφήματος, πόσο αυτός επιβαρύνει τον συνολικό χρόνο εκτέλεσης και αν παίζει πιο σημαντικό ρόλο από τον χρόνο καταμέτρησης των τριγώνων σε κάθε υπογράφημα.

ΠΑΡΑΡΤΗΜΑ

ΥΛΟΠΟΙΗΣΗ ΚΩΔΙΚΑ ΣΕ C++

1. Συνάρτηση partition_file

```
int *partition_file(string filename, const char *delimiter, int chunks) {

	string line;
	ifstream file(filename);
	long file_size = get_file_size(filename);
	long chunk_size = file_size / chunks;
	long split_boundary = chunk_size;
	bool needsSplit = false;
	int *boundaries = new int[chunks];
	Edge last, curr_node;
	int current = 0;

	ofstream temp_file;
	temp_file.open(CHUNK_PREFIX + to_string(current) + ".txt");
	cout << "-splitting chunk " << current << "..." << endl;
	if (file.is_open()) {
		while (getline(file, line)) {
			if (line.at(0) != '#') {
				curr_node = parse_line(line, delimiter);

				if (needsSplit) {
					if (last.a != curr_node.a) {
						temp_file.close();
						boundaries[current] = last.a;
						current++;
						needsSplit = false;
						temp_file.open(CHUNK_PREFIX +
							to_string(current) + ".txt");
						cout << "-splitting chunk " << current
						<< "..." << endl;
					}
				}
				else {
					if (file.tellg() > split_boundary) {
						last = parse_line(line, delimiter);
						needsSplit = true;
						split_boundary += chunk_size;
					}
				}
				temp_file << line << endl;
			}
		}
		file.close();
		temp_file.close();
		boundaries[current] = curr_node.a;
	}
	else {
		cout << "Unable to open file" << endl;
	}
	return boundaries;
}
```

2. Συνάρτηση parse_file

```
map<int, Node> parse_file(string filename, const char *delimiter, int n_proc) {
	string line;

	ifstream file(filename);
	map<int, Node> nodes;
	file.seekg(0, file.beg);

	if (file.is_open()) {
		while (getline(file, line) && line.at(0) != '#') {
			if (line.at(0) != '#') {
				Edge edge = parse_line(line, delimiter);
				nodes[edge.a].id = edge.a;
				nodes[edge.a].neighbors.push_back(edge.b);
			}
		}
		file.close();
		if (n_proc > 1)
			remove(filename.c_str());
	}
	else {
		cout << "Unable to open file" << endl;
	}

	return nodes;
}
```

3. Συνάρτηση intersectionCount

```
int intersectionCount(vector<int> a, vector<int> b) {
	int sum = 0;
	for (int i = 0; i < a.size(); ++i) {
		for (int j = 0; j < b.size(); ++j) {
			if (a[i] == b[j]) sum++;
		}
	}
	return sum;
}
```

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Thomas Schank, Dorothea Wagner. Finding, Counting and Listing all Triangles in Large Graphs, an Experimental Study. 2005.

[2] Matthieu Latapy. Practical algorithms for triangle computations in very large (sparse (power-law)) graphs. 2007.

[3] Shaikh Arifuzzaman, Maleq Khan, Madhan Marathe. PATRIC: A Parallel Algorithm for Counting Triangles in Massive Netwroks. 2013.

[4] Madhav Jha, C. Seshadhri, Ali Pinar. Counting Triangles in Real-World Graph Streams. 2013.

[5] Paul Ezhilchelvan, Alexander Romanovsky. Concurrency in Dependable Computing. 2013.

[6] Teresa W. Haynes, Stephen T. Hedetniemi, Peter J. Slater. Fundamentals of Domination in Graphs. 1998.

[7] Rolf S. Rees. Graphs, Matrices and Designs. 1992.

[8] Oystein Ore. Graphs and their uses. 1990.

[9] Chen Ding, John Criswell, Peng Wu. Languages and Compilers for Parallel Computing. 2016.

[10] William Gropp, Ewing Lusk, Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. 1999.

[11] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, Marc Snir. MPI - The complete reference. 1998.