ΕΠΙΛΕΓΜΕΝΕΣ

ΠΤΥΧΙΑΚΕΣ & ΔΙΠΛΩΜΑΤΙΚΕΣ ΕΡΓΑΣΙΕΣ

1837 2017 ΧΡΟΝΙΑ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Τόμος 15

ΑΘΗΝΑ
2018

ΕΠΙΛΕΓΜΕΝΕΣ ΠΤΥΧΙΑΚΕΣ & ΔΙΠΛΩΜΑΤΙΚΕΣ ΕΡΓΑΣΙΕΣ

Εκδίδεται μία φορά το χρόνο από το:

Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών,
Πανεπιστημιούπολη, 15784 Αθήνα

Επιμέλεια έκδοσης:
Επιτροπή Ερευνητικών και Αναπτυξιακών Δραστηριοτήτων
Θ. Θεοχάρης (υπεύθυνος έκδοσης), Καθηγητής, Τμήμα Πληροφορικής και Τηλεπικοινωνιών
Η. Μανωλάκος, Καθηγητής, Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Γραφιστική επιμέλεια - Επιμέλεια κειμένων:
Ε. Φλωριάς, Τμήμα Πληροφορικής και Τηλεπικοινωνιών

ISSN 1792-8826

Περιεχόμενα

Πρόλογος 5

Υλοποίηση Γενικευμένου Πολλαπλασιασμού Γράφου σε Κάρτα Γραφικών 9
Ευάγγελος Ν. Νικολόπουλος

An SDN QoE Monitoring Framework for VoIP and video applications 23
Maria-Evgenia I. Xezonaki

Algorithms, Hashing, Parallel processing, and the Nearest Neighbor problem in High Dimensions 35
Georgios C. Samaras

Graphity: Out-of-core graph multiplications 49
Stamatis Christoforidis

Πρόλογος

Ο τόμος αυτός περιλαμβάνει περιλήψεις επιλεγμένων διπλωματικών και πτυχιακών εργασιών που εκπονήθηκαν στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών κατά το διάστημα **01/01/2017 - 31/12/2017**. Πρόκειται για τον **15ο τόμο** στη σειρά αυτή. Στόχος του θεσμού είναι η ενθάρρυνση της δημιουργικής προσπάθειας και η προβολή των πρωτότυπων εργασιών των φοιτητών του Τμήματος.

Η έκδοση αυτή είναι ψηφιακή και έχει δικό της ISSN. Αναρτάται στην επίσημη ιστοσελίδα του Τμήματος και έτσι, εκτός από τη μείωση της δαπάνης κατά την τρέχουσα περίοδο οικονομικής κρίσης, έχει και μεγαλύτερη προσβασιμότητα. Για το στόχο αυτό, σημαντική ήταν η συμβολή του κ. Ευάγγελου Φλωριά που επιμελήθηκε φέτος την ψηφιακή έκδοση και πέτυχε μια ελκυστική ποιότητα παρουσίασης, ενώ βελτίωσε και την ομοιογένεια των κειμένων.

Η στάθμη των επιλεγμένων εργασιών είναι υψηλή και κάποιες από αυτές έχουν είτε δημοσιευθεί είτε υποβληθεί για δημοσίευση.

Θα θέλαμε να ευχαριστήσουμε τους φοιτητές για το χρόνο που αφιέρωσαν για να παρουσιάσουν τη δουλειά τους στα πλαίσια αυτού του θεσμού και να τους συγχαρούμε για την ποιότητα των εργασιών τους. Ελπίζουμε η διαδικασία αυτή να προσέφερε και στους ίδιους μια εμπειρία που θα τους βοηθήσει στη συνέχεια των σπουδών τους ή της επαγγελματικής τους σταδιοδρομίας.

Η Επιτροπή Ερευνητικών και Αναπτυξιακών Δραστηριοτήτων

Θ. Θεοχάρης (υπεύθυνος έκδοσης), Η. Μανωλάκος

Αθήνα, Ιούνιος 2018

ΔΙΠΛΩΜΑΤΙΚΕΣ ΕΡΓΑΣΙΕΣ

Υλοποίηση Γενικευμένου Πολλαπλασιασμού Γράφου σε Κάρτα Γραφικών

Ευάγγελος Ν. Νικολόπουλος (vgnikolop@di.uoa.gr)

Περίληψη

Η αλματώδης τεχνολογική εξέλιξη των καρτών γραφικών τα τελευταία χρόνια, η σχέση κόστους προς επεξεργαστική ισχύ καθώς και η εισαγωγή μοντέλων προγραμματισμού γενικού σκοπού σε αυτές, τις καθιστούν ελκυστική επιλογή για πληθώρα εφαρμογών. Οι κάρτες γραφικών μας προσφέρουν μαζική παραλληλία και συνεπώς μεγάλη επεξεργαστική ισχύ. Σε αυτή την εργασία ασχολούμαστε με επεξεργασία γράφων σε κάρτες γραφικών. Συγκεκριμένα υλοποιούμε το μοντέλο του γενικευμένου πολλαπλασιασμού γράφων με το οποίο μπορούμε να εκφράσουμε ενδιαφέροντα προβλήματα σε δεδομένα γράφων όπως είναι η εύρεση των φίλων των φίλων κάθε χρήστη ενός κοινωνικού δικτύου. Δείχνουμε πως μπορούμε να εκμεταλλευτούμε τη μαζική παραλληλία, ανάλογα με τις ιδιότητες κάθε κόμβου του γράφου, προκειμένου να ελαχιστοποιήσουμε τον χρόνο εκτέλεσης. Παρουσιάζουμε μια νέα υβριδική προσέγγιση ταξινόμησης μικρών πινάκων σε κάρτες γραφικών. Στη συνέχεια μέσα από ένα σύνολο από πειράματα με γράφους με διαφορετικές ιδιότητες δείχνουμε ότι η υλοποίηση σε κάρτα γραφικών είναι γρηγορότερη στις περιπτώσεις από μια σε επεξεργαστή γενικού σκοπού και με χαμηλότερο κόστος υλικού. Τέλος παρουσιάζουμε εν συντομία σχετικές εργασίες σε κάρτες γραφικών τόσο σε επεξεργασία γράφων όσο και κλασικών τελεστών βάσεων.

Λέξεις-Κλειδιά: Πολλαπλασιασμός Γράφων, Κάρτα Γραφικών, Βελτιστοποίηση, Μαζική Παραλληλία

Επιβλέπων

Ιωάννης Ιωαννίδης, Καθηγητής (τμήμα Πληροφορικής και Τηλ/νιων ΕΚΠΑ)

1. ΕΙΣΑΓΩΓΗ

Οι γράφοι αποτελούνται από κόμβους και ακμές που τους συνδέουν, τόσο οι κόμβοι όσο και οι ακμές μπορεί να έχουν κάποια τιμή, οι ακμές μπορεί να είναι κατευθυνόμενες ή όχι. Αποτελούν τη βασική μοντελοποίηση δεδομένων για προβλήματα που εκτείνονται σε ένα ευρύ σύνολο από περιοχές όπως κοινωνικά και οδικά δίκτυα, το Διαδίκτυο, τη βιολογία κ.ά. Μετά από επεξεργασία τους μπορούμε να εξάγουμε χρήσιμα συμπεράσματα όπως τη συντομότερη διαδρομή μεταξύ δύο σημείων στο χάρτη ή τη σχέση μεταξύ δύο ιστοσελίδων στο Διαδίκτυο. Μας ενδιαφέρει λοιπόν να μπορούμε να επεξεργαστούμε αποδοτικά και γρήγορα έναν γράφο. Από τη φύση τους οι γράφοι συνήθως έχουν κακή τοπικότητα (locality) και ακανόνιστη μορφή, συχνά υπάρχουν λίγοι πολύ δημοφιλείς κόμβοι και πολλοί λιγότερο δημοφιλείς. Αυτά είναι χαρακτηριστικά που δυσκολεύουν την αποδοτική εκτέλεση αλγόριθμων σε δεδομένα γράφων και για τον λόγο αυτό απαιτούν ειδική μεταχείριση.

Με το μοντέλο του γενικευμένου πολλαπλασιασμού γράφων (ΓΠΓ), που θα μας απασχολήσει στη παρούσα εργασία, μπορούμε να μοντελοποιήσουμε ενδιαφέροντα προβλήματα όπως, εύρεση των φίλων των φίλων κάθε χρήστη ενός κοινωνικού δικτύου, μετρικές ομοιότητας και άλλα. Προκειμένου να επιταχύνουμε την εκτέλεση του ΓΠΓ θα χρησιμοποιήσουμε εξειδικευμένο υλικό και συγκεκριμένα κάρτες γραφικών. Οι συνεισφορές της εργασίας είναι (α) ο καθορισμός και η υλοποίηση ενός πλάνου εκτέλεσης ΓΠΓ με αποδοτική αξιοποίηση των πόρων της κάρτας γραφικών (β) παρουσίαση μιας νέας προσέγγισης ταξινόμησης μικρών πινάκων σε κάρτα γραφικών που κρίθηκε απαραίτητο να αναπτύξουμε και (γ) ένα σύνολο από πειράματα που αξιολογούν τις προηγούμενες συνεισφορές.

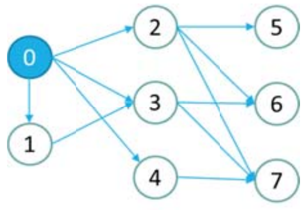
2. ΓΕΝΙΚΕΥΜΕΝΟΣ ΠΟΛΛΑΠΛΑΣΙΑΣΜΟΣ ΓΡΑΦΩΝ

Έστω ότι αναπαριστούμε έναν γράφο G με χρήση πίνακα γειτνίασης (adjacency matrix). Ο πίνακας γειτνίασης είναι ένας πίνακας 2 διαστάσεων, στο κελί (i, j) αντιστοιχίζουμε τις ιδιότητες της ακμής του γράφου από τον κόμβο i στον κόμβο j (i, j). Ορίζουμε το μοντέλο των Γενικευμένων Πολλαπλασιασμών Γράφων (ΓΠΓ) γενικεύοντας τη πράξη του κλασσικού πολλαπλασιασμού πινάκων G·G με αντικατάσταση της πράξης του πολλαπλασιασμού και της πρόσθεσης με τελεστές συνένωσης (concatenation – CON) και σύνθεσης (aggregation – AGG). Ο τελεστής ΓΠΓ $\circ$ ορίζεται ως: $G \circ G = G^2$, όπου $\mathrm{G}_{\mathrm{ij}}^2 = \mathrm{AGG}_{\mathrm{k}=0}^{\mathrm{N}-1}\left(\mathrm{G}_{\mathrm{ik}}\ \mathrm{CON}\ \mathrm{G}_{\mathrm{kj}}\right)$.

Πράξη Συνένωσης. Η πράξη της συνένωσης, πραγματοποιείται με μια διαδικασία ένωσης (join), ενώνει κάθε εισερχόμενη ακμή ενός κόμβου c με κάθε εξερχόμενη ακμή του δημιουργώντας μια νέα ακμή. Έτσι, αν (s, c) μια εισερχόμενη και (c, t) μια εξερχόμενη ακμή του κόμβου c η πράξη της ένωσης παράγει μια νέα ακμή (s, t). Κατά τη πράξη της ένωσης μπορούμε να εφαρμόσουμε έναν τελεστή ένωσης (“con” operator) οριζόμενο από τον χρήστη (User Defined Function – UDF). Ο τελεστής θα παράγει τη τιμή (label) της καινούργιας ακμής με βάση τις αρχικές ακμές των ακμών που ενώθηκαν, μπορεί για παράδειγμα να είναι το μέγιστο, το ελάχιστο, ο μέσος όρος, το άθροισμα κοκ.

Πράξη Σύνθεσης. Η πράξη της σύνθεσης, πραγματοποιείτε με μια διαδικασία ομαδοποίησης (group by), ομαδοποιεί τις ακμές με κοινό κόμβο αφετηρίας και προορισμού. Κατά τη ομαδοποίηση, εφαρμόζεται ο οριζόμενος από τον χρήστη τελεστής σύνθεσης (“agg” operator) που καθορίζει την τιμή της ακμής που προκύπτει μετά την ομαδοποίηση. Ο τελεστής μπορεί να είναι, όπως και στην πράξη της συνένωσης, το μέγιστο, το ελάχιστο, ο μέσος όρος, το άθροισμα κοκ.

Παράδειγμα Προβλήματος ΓΠΓ. Θεωρούμε κατευθυνόμενο τον γράφο κοινωνικού δικτύου (Σχήμα 1) με τους κόμβους να αναπαριστούν χρήστες και τις ακμές να αναπαριστούν σχέσεις «ακολουθεί». Θέλουμε να απαντήσουμε στο ερώτημα «Ποιους χρήστες ακολουθούν οι ακόλουθοι κάθε χρήστη και μέσω πόσων διαφορετικών μονοπατιών;» Δηλαδή για τον χρήστη 0 η απάντηση είναι ότι: Ακολουθεί τον χρήστη 3 μέσω ενός μονοπατιού, τον χρήστη 5 μέσω ενός μονοπατιού, τον χρήστη 6 μέσω 2 μονοπατιών (μέσω των χρηστών 2 και 3) και τον χρήστη 7 μέσω 3 μονοπατιών (μέσω των χρηστών 2, 3 και 4). Με χρήση των τελεστών «τίποτα» (nil) για con και «μέτρηση» (count) για agg μπορούμε να απαντήσουμε τη συγκεκριμένη ερώτηση με ένα σύστημα που υλοποιεί ΓΠΓ. Επίσης μπορούν να υπολογιστούν ερωτήματα όπως εύρεσης συντομότερου μονοπατιού από όλους τους κόμβους προς όλους τους άλλους του γράφου (Multiple Source Shortest Path – MSSP) με ακολουθία από πολλαπλασιασμούς G∘G∘…∘G.

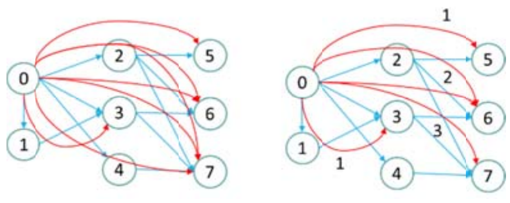


Σχήμα 1. Παράδειγμα γράφου.

3. ΕΠΕΞΕΡΓΑΣΙΑ ΓΡΑΦΟΥ

Θα παρουσιάσουμε συνοπτικά, μέσα από ένα παράδειγμα, το πλάνο εκτέλεσης ΓΠΓ σε επεξεργαστή γενικού σκοπού CPU. Με βάση αυτό θα συνεχίσουμε με την GPU υλοποίηση.

Επιθυμητό Αποτέλεσμα. Για τον γράφο στο Σχήμα 1 το αποτέλεσμα της πρώτης φάσης, δηλαδή του join, είναι: [0: {5, 6, 7, 6, 7, 7, 3}, 1: {6, 7}, 2, 3, 4, 5, 6, 7: {}]. Δηλαδή, ο κόμβος 0 έχει εξερχόμενες ακμές προς τους κόμβους 5, 6, 7, 6, 7, 7 και 3, ο κόμβος 1 προς τους 6 και 7 ενώ οι υπόλοιποι κόμβοι δεν έχουν εξερχόμενες ακμές μετά τη πράξη της ένωσης (άρα δεν υπάρχουν ακμές με 2 βήματα για αυτούς τους κόμβους). Στο Σχήμα 2 (Αριστερά) φαίνονται οι εξερχόμενες ακμές που προκύπτουν για τον κόμβο 0. Στη δεύτερη φάση, τη φάση του group by, προκύπτουν τα βάρη των ακμών: [0: {3:1, 5:1, 6:2, 7:3}, 1: {6:1, 7:1}, 2, 3, 4, 5, 6, 7: {}]. Δηλαδή, η ακμή (0, 5) έχει βάρος 1, η ακμή (0, 6) έχει βάρος 2 κοκ. Στο Σχήμα 2 (δεξιά) φαίνονται οι εξερχόμενες ακμές με τα βάρη που προκύπτουν για τον κόμβο 0. Τελικά το επιθυμητό αποτέλεσμα είναι: για κάθε κόμβο αφετηρίας, ένας πίνακας με ζεύγη αριθμών. Ο πρώτος αριθμός υποδηλώνει τον κόμβο προορισμού και ο δεύτερος το βάρος της ακμής.



Σχήμα 2. Οι ακμές που προκύπτουν για τον κόμβο 0 μετά τη πράξη της ένωσης (αριστερά) και μετά τη πράξη της ομαδοποίησης, με τα βάρη τους (δεξιά).

Βασικό Πλάνο Εκτέλεσης. Στη παράγραφο αυτή θα περιγράψουμε τη διαδικασία υπολογισμού του ΓΠΓ με σειριακό τρόπο. Σε αυτή τη διαδικασία θα βασιστούμε στη συνέχεια για να περιγράψουμε τον αντίστοιχο αλγόριθμο για τη κάρτα γραφικών. Ας θεωρήσουμε και πάλι τον γράφο στο Σχήμα 1. Θέλουμε να βρούμε τους χρήστες που ακολουθούν, οι χρήστες που ακολουθεί ο χρήστης 0 και πόσα διαφορετικά μονοπάτια οδηγούν εκεί. Προκειμένου να επιταχυνθεί η διαδικασία θα κάνουμε τη πράξη της ομαδοποίησης ταυτόχρονα με τη πράξη της ένωσης με σταδιακό τρόπο (incrementally). Ορίζουμε έναν πίνακα ομαδοποίησης μεγέθους όσο και το πλήθος των κόμβων του γράφου και τον αρχικοποιούμε με 0. Εξερευνούμε κάθε εξερχόμενη ακμή του κόμβου 0 του γράφου μας και τις εξερχόμενες ακμές των κόμβων που καταλήγουν οι πρώτες. Έτσι αρχικά επισκεπτόμαστε τον κόμβο 1 μέσω της ακμής (0, 1) και από εκεί τον κόμβο 3 μέσω της ακμής (1, 3). Ο κόμβος 3 είναι ένας κόμβος που ανήκει στο αποτέλεσμα μας αφού τον συναντήσαμε στο δεύτερο βήμα, θα ενημερώσουμε τη θέση 3 του πίνακα εκτελώντας τη πράξη της ομαδοποίησης (μέτρηση) και άρα θα αυξήσουμε κατά 1 τη τιμή του κελιού. Συνεχίζουμε την ίδια διαδικασία για τις υπόλοιπες εξερχόμενες ακμές του κόμβου 0. Από την ακμή (0, 2) θα επισκεφτούμε τους κόμβους 5, 6 και 7 κ.ο.κ. Τελικά ο πίνακας ομαδοποίησης θα φτάσει στη μορφή που βλέπουμε στο δεξί μέρος του Πίνακα 1. Στη φάση αυτή έχει υπολογιστεί το αποτέλεσμα της ομαδοποίησης και αρκεί μια προσπέλαση του πίνακα για να προκύψει το τελικό αποτέλεσμα. Τελικά προκύπτει το αποτέλεσμα όπως στον Πίνακα 1.

Ένας τετριμμένος τρόπος παραλληλοποίησης του παραπάνω πλάνου είναι να θεωρήσουμε πολλούς αρχικούς ενεργούς κόμβους και να αναθέσουμε 1 κόμβο σε 1 νήμα, αυτό σημαίνει ότι θα χρειαστούμε πολλούς πίνακες ομαδοποίησης, έναν για κάθε νήμα. Τέλος να αναφέρουμε ότι στη συγκεκριμένη προσέγγιση ένα πρόβλημα απόδοσης που συναντάμε είναι η προσπέλαση του πίνακα ομαδοποίησης όταν αυτός είναι αραιός. Θα προτιμούσαμε να μην προσπελάσουμε καθόλου τις κενές θέσεις. Υπάρχουν κατάλληλες τεχνικές οι οποίες ξεφεύγουν από το πλαίσιο της συγκεκριμένης εργασιας αλλά χρησιμοποιούνται στα τελικά πειράματα.

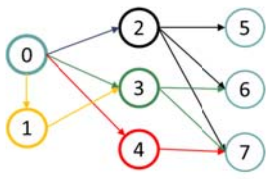
0	1	2	3	4	5	6	7
0	0	0	1	0	1	2	3

Πίνακας 1: Τελική κατάσταση πίνακα ομαδοποίησης.

3.1 Υλοποίηση GPU

Θα περιγράψουμε τη διαδικασία υπολογισμού του ΓΠΓ στη κάρτα γραφικών. Θεωρούμε ότι ο γράφος χωράει να αποθηκευτεί ολόκληρος με τη CSR μορφή εντός της μνήμης της κάρτας γραφικών. Ο περιορισμός αυτός μας απαγορεύει να επεξεργαστούμε πολύ μεγάλους γράφους. Όπως αναφέραμε στο κεφάλαιο 3 οι κάρτες γραφικών είναι επεξεργαστές που προσφέρουν μαζική παραλληλία, για να τις αξιοποιήσει η εφαρμογή μας θα πρέπει να εκμεταλλεύεται χιλιάδες νήματα ταυτόχρονα. Η βασική προσέγγιση που κάναμε στη προηγούμενη παράγραφο απαιτεί O(N*T) μνήμη όμως, επειδή στη GPU το πλήθος των νημάτων (T) είναι πολύ μεγάλο και η μνήμη περιορισμένη, η λύση αυτή δεν είναι εφικτή. Στη προσέγγιση που κάνουμε στη συνέχεια μεταφέρουμε τη παραλληλία εντός του ενεργού κόμβου. Έτσι έχουμε έναν ενεργό κόμβο κάθε στιγμή στη κάρτα γραφικών και κατ’ επέκταση έναν πίνακα ομαδοποίησης και πολλά νήματα που τον ενημερώνουν και παράγουν το τελικό αποτέλεσμα.

Πλάνο Εκτέλεσης σε GPU. Θα αναθέσουμε σε κάθε block από νήματα μια εξερχόμενη ακμή του κόμβου αφετηρίας και σε κάθε νήμα από μια εξερχόμενη ακμή του ενδιάμεσου κόμβου. Ενώ όλα μαζί τα νήματα θα ενημερώνουν τον πίνακα ομαδοποίησης. Θεωρούμε και πάλι τον γράφο στο Σχήμα 1 αποθηκευμένο στη μνήμη της κάρτας με τη CSR μορφή. Θεωρούμε ενεργό κόμβο τον 0. Ας θεωρήσουμε ότι έχουμε 4 blocks με 4 νήματα το καθένα. Σε πρώτη φάση όλα τα νήματα κάθε block θα διαβάσουν το σημείο από το οποίο ξεκινούν οι εξερχόμενες ακμές του κόμβου 0 στον πίνακα των ακμών (Edges) και θα διαβάσει από μία εξερχόμενη ακμή του κόμβου κάθε block. Οπότε το block 0 διαβάζει την ακμή (0, 1), το block 1 την ακμή (0, 2), το block 2 την ακμή (0, 3) και το block 3 την ακμή (0, 4).

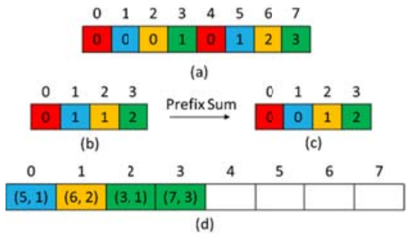


Σχήμα 3. Ανάθεση blocks σε εξερχόμενες ακμές, κάθε χρώμα συμβολίζει ένα διαφορετικό block από νήματα

Στη συνέχεια κάθε block θα επιστρέψει στον πίνακα Offsets προκειμένου να αναγνωρίσει τις εξερχόμενες ακμές του κόμβου που έχει καταλήξει. Έτσι το block 1 που έχει αναλάβει την ακμή (0, 2) θα διαβάσει τη θέση 2 και τη θέση 3 του πίνακα offsets από το CSR για να βρει ότι οι εξερχόμενες ακμές του κόμβου 2 ξεκινούν στη θέση 5 του πίνακα Edges και είναι 3. Στη συνέχεια θα μείνουν ενεργά τα 3 από τα 4 νήματα του block 1 αφού ο κόμβος 2 έχει 3 εξερχόμενες ακμές, αυτά τα 3 νήματα θα διαβάσουν τις ακμές από τον αντίστοιχο πίνακα και θα ανακαλύψουν ότι καταλήγουν στου κόμβους 5, 6 και 7. Την ίδια διαδικασία ακολουθούν και τα υπόλοιπα blocks για διαφορετικούς κόμβους. Στο Σχήμα 3 βλέπουμε τις περιοχές του γράφου που έχει αναλάβει κάθε block. Κάθε χρώμα συμβολίζει και ένα διαφορετικό block, το block 1 είναι το μαύρο χρώμα. Το block 3 (κόκκινο) διαβάζει την εξερχόμενη ακμή (0, 4), βλέπει ότι ο κόμβος 4 έχει μια εξερχόμενη ακμή, αφήνει 1 νήμα ενεργό το οποίο θα την εξερχόμενη ακμή (0, 7). Φυσικά αν τα blocks είναι λιγότερα από τις εξερχόμενες ακμές του αρχικού κόμβου ή τα νήματα λιγότερα από τις εξερχόμενες ακμές του ενδιάμεσου κόμβου τότε απλά συνεχίζουν επαναληπτικά τη παραπάνω διαδικασία.

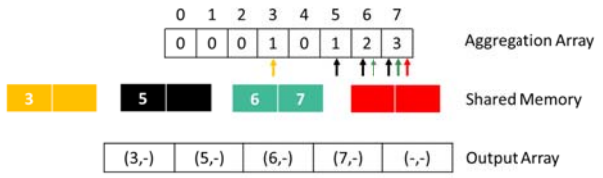
Όταν ένα νήμα ανακαλύπτει έναν κόμβο προορισμού προχωράει στην διαδικασία ενημέρωσης του πίνακα ομαδοποίησης. Γυρνώντας στο παράδειγμά μας όλα τα νήματα έχουν διαβάσει τους κόμβους προορισμού και μεταφέρονται στην αντίστοιχη θέση του πίνακα ομαδοποίησης, το νήμα του κίτρινου block πάει στη θέση 3, τα νήματα του μαύρου στις θέσεις 5, 6 και 7 κοκ. Σε αυτές τις θέσεις θα κάνουν μια πράξη ατομικής πρόσθεσης με το 1 ώστε να υπολογιστεί το αποτέλεσμα. Η συνάρτηση ατομικής πρόσθεσης (atomicAdd) υποστηρίζεται από τη CUDA.

Σε αυτό το σημείο έχουμε ενημερώσει πλήρως τον πίνακα ομαδοποίησης πρέπει όμως να μεταφέρουμε το αποτέλεσμα στον πίνακα εξόδου με την επιθυμητή μορφή. Μια προσέγγιση είναι να προσπελάσουμε, παράλληλα, 2 φορές τον πίνακα για να το πετύχουμε αυτό. Στο Σχήμα 4 κάθε χρώμα αντιστοιχεί σε ένα νήμα, ο πίνακας (a) είναι ο πίνακας ομαδοποίησης, στον πίνακα (b) κάθε νήμα αθροίζει το πλήθος των μη μηδενικών τιμών του (a) που του αντιστοιχούν, το μπλε και το κίτρινο νήμα έχουν από 1 αποτέλεσμα ενώ το πράσινο 2. Στη συνέχεια υπολογίζεται το Prefix Sum, του πίνακα (b) στον πίνακα (c). Ο πίνακας (c) έχει τώρα τις τιμές που δείχνουν ποιες θέσεις του πίνακα εξόδου μπορεί να χρησιμοποιήσει κάθε νήμα. Η διαδικασία αυτή γίνεται για να μπορούν όλα τα νήματα να μεταφέρουν ταυτόχρονα το αποτέλεσμα χωρίς να χρειάζεται κάποιος συγχρονισμός μεταξύ τους. Στη συνέχεια τα νήματα διασχίζουν και πάλι τον πίνακα ομαδοποίησης, αυτή τη φορά μεταφέρουν τα αποτελέσματα στον πίνακα εξόδου (d). Η παραπάνω διαδικασία εφαρμόζεται γενικά σε αντίστοιχα προβλήματα σε κάρτες γραφικών [6]. Στη δική μας περίπτωση δεν δουλεύει καλά στην πολύ συνηθισμένη περίπτωση όπου ο πίνακας ομαδοποίησης είναι σχετικά αραιός καθώς πρέπει να τον διατρέξουμε ολόκληρο και να προσπεράσουμε πολλά μηδενικά στοιχεία που δεν συνεισφέρουν στο αποτέλεσμα.



Σχήμα 4. Διαδικασία μεταφοράς του αποτελέσματος από τον πίνακα ομαδοποίησης (a) στον πίνακα εξόδου (d).

Προτείνουμε μια διαφορετική προσέγγιση που εκμεταλλεύεται το γεγονός ότι η πράξη της ατομικής πρόσθεσης επιστρέφει τη προηγούμενη τιμή που είχε το αντίστοιχο κελί. Έτσι όταν κάποιο νήμα δει ότι αλλάζει τη τιμή του κελιού από μηδέν σε ένα αποθηκεύει στη κοινή μνήμη (shared memory) τη θέση του κελιού που άλλαξε, για να το κάνει αυτό αρκεί ένας δείκτης για κάθε block που είναι κοινός μεταξύ των νημάτων ενός block και δείχνει στην επόμενη διαθέσιμη θέση για εγγραφή αποτελέσματος. Αυτός ο δείχτης ενημερώνεται με ατομική πρόσθεση από κάθε νήμα που θέλει να γράψει ένα αποτέλεσμα στη κοινή μνήμη. Μόλις γεμίσει η κοινή μνήμη όλα τα νήματα του block συνεργάζονται ώστε να μεταφερθεί το αποτέλεσμα στον πίνακα εξόδου. Και εκεί υπάρχει ένας δείκτης που ενημερώνεται με ατομική πρόσθεση, αυτός όμως είναι κοινός για όλα τα blocks, άρα βρίσκεται στη κύρια μνήμη και είναι πιο αργό να τον προσπελάσουμε και να τον ενημερώσουμε, γι' αυτό ελαχιστοποιούμε της ενημερώσεις του μέσω της χρήσης της κοινής μνήμης. Με αυτόν τον τρόπο μεταφέρουμε τους κόμβους οι οποίοι έχουν τουλάχιστον ένα αποτέλεσμα στον πίνακα εξόδου. Τους μεταφέρουμε όμως πριν ολοκληρωθεί η διαδικασία εξερεύνησης του γράφου άρα δεν έχουμε και τη τελική τιμή τους. Αρκεί όμως μια απλή προσπέλαση στον πίνακα εξόδου στο τέλος, όπου διαβάζουμε τους κόμβους προορισμού που έχουν γραφτεί εκεί (και άρα έχουν αποτέλεσμα) και παίρνουμε τη τελική τιμή τους από τον πίνακα ομαδοποίησης (τον οποίο το μηδενίζουμε για να εκτελεστεί η επόμενη επανάληψη).



Σχήμα 5. Μεταφορά αποτελέσματος με χρήση της κοινής μνήμης.

Στο Σχήμα 5 βλέπουμε τη διαδικασία που περιγράψαμε για το παράδειγμά μας. Ο πίνακας ομαδοποίησης ενημερώνεται από τα νήματα και κάθε νήμα που αλλάζει τη τιμή του κελιού από μηδέν σε ένα γράφει τη διεύθυνση του κελιού που άλλαξε στη κοινή μνήμη του block του. Οπότε το μοναδικό νήμα που ανήκει στο κίτρινο block άλλαξε τη τιμή του κελίου 3 από μηδέν σε ένα και έγραψε στη κοινή μνήμη του κίτρινου block το αναγνωριστικό του κόμβου (3). Για το κελί 6 φαίνεται ότι συναγωνίστηκαν τα νήματα από το μαύρο και το πράσινο block και απ' ότι φαίνεται κατάφερε το νήμα του πράσινου block να αλλάξει τη τιμή από μηδέν σε ένα άρα αυτό θα γράψει στη κοινή του μνήμη το αναγνωριστικό του κόμβου (6). Στη συνέχεια (ή κατά τη διάρκεια της εκτέλεσης σε περίπτωση που γεμίσει η κοινή μνήμη κάποιου block) οι κόμβοι 3, 5, 6 και 7 μεταφέρονται στον πίνακα εξόδου και μένει μια απλή προσπέλαση σε αυτόν, μόλις ολοκληρωθεί η διαδικασία ενημέρωσης του πίνακα ομαδοποίησης, ώστε να πάρουμε τις τελικές τιμές από αυτόν και να τις μεταφέρουμε στον πίνακα εξόδου.

Ακραίες περιπτώσεις υλοποίησης GPU. Η τεχνική μου μόλις περιγράψαμε, όπως θα δούμε και στα πειράματα, είναι αρκετά καλή για τις περισσότερες περιπτώσεις αλλά υστερεί λίγο όταν τα αποτελέσματα είναι πολλά, όταν δηλαδή ο πίνακας ομαδοποίησης είναι πυκνός. Τότε ίσως αξίζει να εκτελέσουμε το πλάνο που περιγράψαμε στην αρχή της παραγράφου. Σημαντικότερο όμως μειονέκτημα έχει όταν τα αποτελέσματα είναι εξαιρετικά λίγα. Αυτό πρακτικά συμβαίνει όταν ο κόμβος αφετηρίας και οι κόμβοι με τους οποίους συνδέεται έχουν λίγες εξερχόμενες ακμές.

Για τους κόμβους με λίγα αποτελέσματα η λύση που προτείνουμε είναι αρκετά διαφορετική από τη προηγούμενη, καταρχήν δεν θα χρησιμοποιήσουμε πίνακα ομαδοποίησης μεγέθους όσο και οι κόμβοι του γράφου αλλά θα έχει μέγεθος όσο το πάνω όριο των αποτελεσμάτων που περιμένουμε (δηλαδή μικρό). Συνεπώς μπορούμε να εκτελέσουμε πολλούς ενεργούς κόμβους ταυτόχρονα, συγκεκριμένα θα αναθέσουμε έναν ενεργό κόμβο σε κάθε block και θα έχουμε ενεργούς κόμβους ίσους με το πλήθος των blocks μας. Το πρώτο μέρος του πλάνου εκτέλεσης είναι ίδιο με εκείνο των υπόλοιπων κόμβων με τη διαφορά ότι το αναλαμβάνει ένα μόνο block και δεν ενημερώνει των πίνακα ομαδοποίησης υπολογίζοντας το αποτέλεσμα σταδιακά. Αντίθετα, αποθηκεύει κάθε κόμβο που συναντάει ανεξάρτητα του αν είναι πρώτη φορά που τον συναντά ή όχι. Έτσι για τον γράφο από το Σχήμα 1 και με ενεργό κόμβο το 0 πάλι, προκύπτει ο πίνακας ομαδοποίησης: [3, 5, 6, 7, 6, 7, 7]. Παρατηρούμε ότι ο κόμβος 6 υπάρχει 2 φορές καθώς τον έχουμε επισκεφτεί μέσω του κόμβου 2 και 3, ο κόμβος 7 υπάρχει 3 φορές μέσω των κόμβων 2, 3 και 4 ενώ οι κόμβοι 3 και 5 από μία φορά. Προκειμένου να υπολογίσουμε το αποτέλεσμα αρκεί να ταξινομήσουμε τον πίνακα και να τον προσπελάσουμε μια φορά κρατώντας έναν αθροιστή για το πόσες φορές συναντάμε τον ίδιο αριθμό. Μια κλασική τεχνική υπολογισμού του Group By σε βάσεις δεδομένων.

Ταξινόμηση Μικρών Πινάκων στη GPU. Οι ιδιαιτερότητες του προβλήματος μας είναι ότι οι πίνακες πρέπει να ταξινομηθούν εντός του block που έχουν παραχθεί και, όπως θα δούμε στα πειράματα, είναι μικροί μεν αλλά όχι αρκετά μικροί για να χωράνε στη κρυφή μνήμη ενός block όπως υποθέτουν στο [7]. Μια καθιερωμένη τεχνική για ταξινόμηση πολύ μικρών πινάκων σε κάρτες γραφικών είναι τα δίκτυα ταξινόμησης όπως το bitonic sorting network [8]. Το bitonic sort μας δίνει σταθερό αριθμό συγκρίσεων, συγκεκριμένες συγκρίσεις που πρέπει να γίνουν σε κάθε βήμα (πχ. Στο βήμα 1 το στοιχείο 0 με το 1, το 2 με το 3 κοκ..), έχει χαμηλή πολυπλοκότητα $O(n * log^2(n))$ και μπορεί να υλοποιηθεί παράλληλα και αποδοτικά. Αν έχουμε ικανό αριθμό από νήματα, δηλαδή όσοι και οι αριθμοί που θέλουμε να ταξινομήσουμε η παράλληλη πολυπλοκότητα (ή τα παράλληλα βήματα που θα εκτελεστούν) είναι $O(log^2(n))$.

Τα μειονεκτήματα που συναντάμε είναι δύο. Καταρχήν, το πλήθος των αριθμών που θέλουμε να ταξινομήσουμε πρέπει να είναι δύναμη του 2, δηλαδή της μορφής $n = 2^{\kappa}$. Διαφορετικά πρέπει να προσθέσουμε κάποιο padding. Χρειάζονται αποκλειστικά και μόνο για να γίνουν οι συγκρίσεις. Αυτό σε έναν πίνακα με 2100 στοιχεία σημαίνει ότι θα πρέπει να προσθέσουμε $4096 - 2100 = 1996$ στοιχεία padding, σχεδόν να διπλασιάσουμε την είσοδο μας δηλαδή. Το δεύτερο πρόβλημα είναι ότι το bitonic network δουλεύει πολύ αποδοτικά όταν όλα τα στοιχεία βρίσκονται στη κοινή μνήμη του block, αν δεν χωράνε όμως, στα τελευταία βήματα πρέπει να γίνουν συγκρίσεις μεταξύ στοιχείων που απέχουν πολύ στη μνήμη αυτό οδηγεί σε μη αποδοτική προσπέλαση της μνήμης. Για να ξεπεράσουμε τα παραπάνω προβλήματα προτείνουμε μια υβριδική προσέγγιση όπως και στο [6]. Θα χωρίσουμε τον πίνακα που θέλουμε να ταξινομήσουμε σε μικρότερα μέρη που να χωράνε στη κοινή μνήμη και να έχουν πλήθος στοιχείων την μορφής $n = 2^{\kappa}$ έτσι αρκεί να προσθέσουμε padding μόνο για το τελευταίο μέρος το οποίο θα έχει όμως πολύ μικρότερο μέγεθος. Θα ταξινομήσουμε αυτά τα μέρη με τη μέθοδο του bitonic network όπως και στο [6]. Στη συνέχεια αρκεί να συγχωνεύσουμε τους ταξινομημένους πίνακες. Θα το κάνουμε με δεντρικό τρόπο ανα 2 παράλληλα με χρήση του μονοπατιού συγχώνευσης [9] [10].

4. ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ

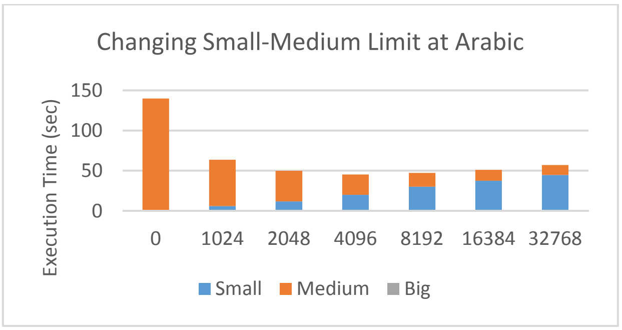
Στο κεφάλαιο αυτό θα αξιολογήσουμε πειραματικά την υλοποίηση του ΓΠΓ σε GPU. Αξίζει να σημειωθεί ότι η CPU υλοποίηση έχει επίσης αρκετές βελτιώσεις με κυριότερη ότι ούτε σε εκείνη προσπελαύνουμε όλο τον πίνακα ομαδοποίησης. Ο Πίνακας 2 παρουσιάζει τους γράφους που χρησιμοποιήθηκαν για τη πειραματική αξιολόγηση. Μας ενδιαφέρει τόσο το μέγεθός τους όσο και το πλήθος των αποτελεσμάτων που παράγουν. Οι Arabic και GSH αν αποθηκευτούν ως CSR καταλαμβάνουν περίπου 2.5GB ο καθένας ενώ τα αποτελέσματα του GSH μετά τη πράξη της ομαδοποίησης περίπου 4TB. Οι γράφοι αυτοί αποτελούν ένα αντιπροσωπευτικό δείγμα με διαφορετικά χαρακτηριστικά έτσι ώστε να αξιολογήσουμε την υλοποίησή μας.

	Live Journal	Hollywood	Arabic	GSH
#Nodes	$4.8 * 10^6$	$2.1 * 10^6$	$22.7 * 10^6$	$30.8 * 10^6$
#Edges	$69 * 10^6$	$229 * 10^6$	$639 * 10^6$	$602 * 10^6$
#Join Results	$5.9 * 10^9$	$242 * 10^9$	$145 * 10^9$	$1.6 * 10^{12}$

Πίνακας 2. Οι Γράφοι που χρησιμοποιήθηκαν.

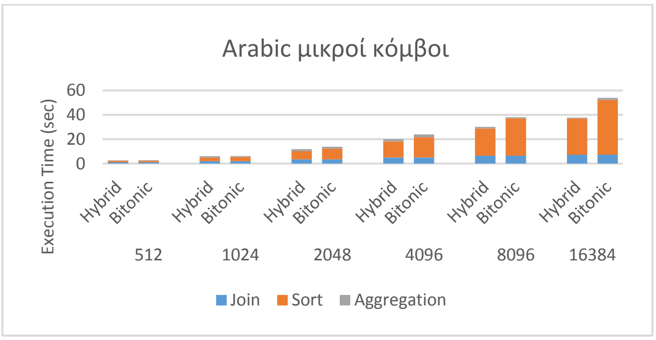
Για τα πειράματα σε CPU είχαμε στη διάθεση μας ένας υπολογιστή με Intel Xeon E5-2630v4 (2.2-3.1GHz 10core/20threads) και 128GB RAM. Η υλοποίηση σε GPU έγινε σε μια GTX1060 6GB της Nvidia ενώ στη διάθεση μας για τη τελική σύγκριση είχαμε και μια GTX1070 8GB.

Επίδραση Ορίου Μικρών Κόμβων. Στο Σχήμα 7 βλέπουμε στον οριζόντιο άξονα τη τιμή του ορίου μεταξύ μικρών και μεγάλων κόμβων σε πλήθος αποτελεσμάτων. Στον κάθετο άξονα τον χρόνο εκτέλεσης, οι στήλες δείχνουν τον συνολικό χρόνο εκτέλεσης καθώς και το ποσοστό που δαπανάται σε μικρούς και μεσαίους κόμβους. Όπως είναι αναμενόμενο όσο αυξάνουμε το όριο τόσο μεγαλύτερο ποσοστό του χρόνου καταλαμβάνουν οι μικροί κόμβοι σε σχέση με τους μεγάλους ενώ βλέπουμε ότι όταν το όριο είναι στο 4096 έχουμε το καλύτερο χρόνο. Η χρήση των μικρών κόμβων κρίνεται απαραίτητη.



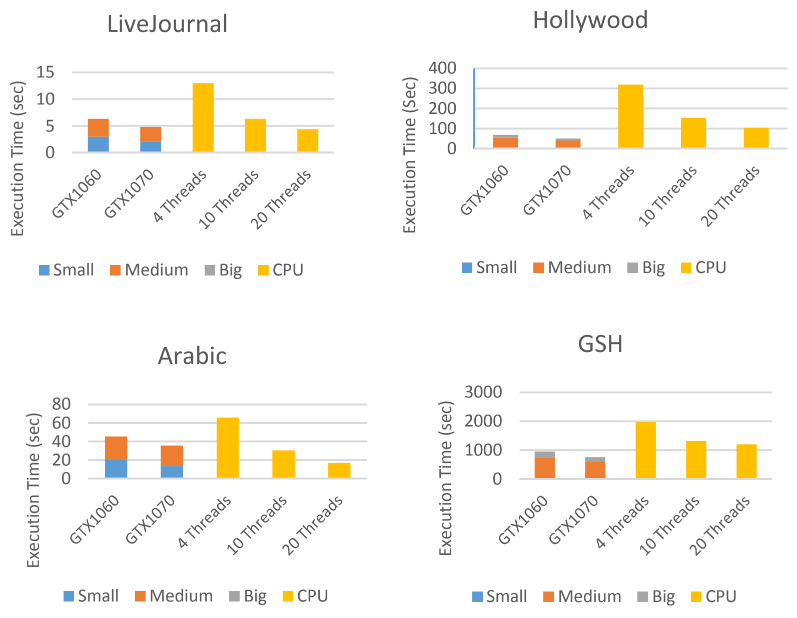
Σχήμα 7. Επίδραση ορίου μικρών και μεσαίων κόμβων στον γράφο Arabic.

Αξιολόγηση Υβριδικής Ταξινόμησης. Στο Σχήμα 8 βλέπουμε το συνολικό χρόνο εκτέλεσης για τους μικρούς κόμβους με χρήση bitonic και υβριδικής ταξινόμησης. Η κοινή μνήμη έχει ρυθμιστεί έτσι ώστε να χωράει 1024 στοιχεία συνεπώς για όριο εως 1024 και στην υβριδική λύση εκτελείτε μόνο bitonic sort στη κοινή μνήμη. Παρατηρούμε ότι εκεί και πέρα η υβριδική ταξινόμηση είναι γρηγορότερη. Αν παρατηρήσουμε μόνο τον χρόνο της ταξινόμησης η διαφορά μεταξύ των ταξινομήσεων για πίνακες μέχρι 4096 στοιχεία είναι 30%!



Σχήμα 8. Σύγκριση υβριδικής ταξινόμησης με bitonic sort.

Σύγκριση με υλοποίηση σε CPU. Τέλος, συγκρίνουμε την υλοποίηση σε GPU με εκείνη σε CPU με διαφορετικό αριθμό από νήματα για να δείξουμε πως κλιμακώνει η υλοποίηση σε CPU. Με κίτρινο χρώμα βλέπουμε τον χρόνο εκτέλεσης σε CPU ενώ για τη GPU έχουμε χωρίσει με μπλε τον χρόνο για τους μικρούς κόμβους, με κόκκινο για τους μεσαίους και με γκρι για τους μεγάλους. Παρατηρούμε ότι στους γράφους με λίγα αποτελέσματα, η CPU έχει παρόμοια (LiveJournal) ή και λίγο καλύτερα (Arabic) αποτελέσματα από τις κάρτες γραφικών. Στους γράφους με πολλά αποτελέσματα, Hollywood και GSH οι GPUs, κερδίζουν τη CPU. Ακόμα και η πιο αδύναμη GTX1060 είναι γρηγορότερη από τον επεξεργαστή με χρήση όλων των νημάτων.



Σχήμα 9. Σύγκριση CPU-GPU με χρήση των LiveJournal, Hollywood, Arabic, GSH.

5. ΣΥΜΠΕΡΑΣΜΑΤΑ

Σε αυτή την εργασία μελετήσαμε το πρόβλημα των Γενικευμένων Πολλαπλασιασμών Γράφων (ΓΠΓ) σε κάρτες γραφικών. Παρουσιάσαμε τις ιδιαιτερότητες του προγραμματισμού στο προγραμματιστικό περιβάλλον CUDA της Nvidia, είδαμε το πλάνο εκτέλεσης του ΓΠΓ με παραλληλία εντός του ενεργού κόμβου καθώς και μια υβριδική ταξινόμηση που συνδυάζει bitonic sort και το μονοπάτι συγχώνευσης για παράλληλη ταξινόμηση πολλών μεσαίου μεγέθους πίνακες σε κάρτες γραφικών. Όσον αφορά τον ΓΠΓ είδαμε ότι η κάρτα γραφικών και η υλοποίηση με παραλληλία εντός του ενεργού κόμβου είναι πολύ καλή προσέγγιση όταν ο γράφος έχει κόμβους με πολλά αποτελέσματα. Αντίθετα σε κόμβους με λίγα αποτελέσματα η προσέγγιση με τη ταξινόμηση βοηθάει μεν αρκετά όμως δεν είναι γρηγορότερη από μια υλοποίηση σε CPU αυτό είναι αναμενόμενο καθώς η προσέγγιση με ταξινόμηση έχει μεγαλύτερη πολυπλοκότητα. Παρ' όλα αυτά οι χρόνοι εκτέλεσης αν λάβουμε υπόψιν και το κόστος του υλικού είναι αρκετά καλοί. Μια υβριδική λύση με αξιοποίηση τόσο της CPU όσο και της GPU θα ήταν ακόμα καλύτερη. Οι ιδέες τόσο του μονοπατιού ταξινόμησης όσο και του bitonic sort ήταν ήδη γνωστές. Είναι πρώτη φορά όμως, εξ' όσων γνωρίζουμε, που χρησιμοποιείται ο συνδυασμός των δύο μεθόδων. Είδαμε ότι έχει πολύ καλά αποτελέσματα για ταξινόμηση μικρών πινάκων που δεν χωράνε στη κοινή μνήμη.

6. ΑΝΑΦΟΡΕΣ

[1] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in Proceedings of the

2010 International Conference on Management of Data, pp 135-146, New York, NY, USA, 2010.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, , November 1999.

[3] "CUDA," [Online]. Available: https://en.wikipedia.org/wiki/CUDA.

[4] "Programming Guide :: CUDA Toolkit Documentation.," [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction.

[5] "Race condition," [Online]. Available: https://en.wikipedia.org/wiki/Race_condition.

[6] Bingsheng He , Ke Yang , Rui Fang , Mian Lu , Naga Govindaraju , Qiong Luo , Pedro Sander, "Relational joins on graphics processors," in International Conference on Management of Data, SIGMOD, Vancouver, Canada, 2008.

[7] Muaaz Awan and Fahad Saeed, "GPU-ArraySort: A parallel, in-place algorithm for sorting large number of arrays.," in 45th International Conference on Parallel Processing Workshops (ICPPW), 2016.

[8] "Bitonic sorter," [Online]. Available: https://en.wikipedia.org/wiki/Bitonic_sorter.

[9] Odeh, S., Green, O., Mwassi, Z., Shmueli, O., & Birk, Y., "Merge path-parallel merging made simple," in Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International (pp. 1611-1618). IEEE..

[10] Green, O., McColl, R., & Bader, D. A., "GPU merge path: a GPU merging algorithm," in Proceedings of the 26th ACM international conference on Supercomputing (pp. 331-340). ACM..

[11] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., & Muthukrishnan, S. , "One trillion edges: Graph processing at facebook-scale.," in Proceedings of the VLDB Endowment, 8(12), 1804-1815., 2015.

[12] "Apache Giraph," [Online]. Available: http://giraph.apache.org/.

[13] Gharaibeh, T. Reza, E. Santos-Neto, L. B. Costa, S. Sal-linen, and M. Ripeanu, "Efficient large-scale graph processing on hybrid cpu and gpu systems," arXiv:1312.3018, 2014.

[14] Kim, M. S., An, K., Park, H., Seo, H., & Kim, J., "GTS: A fast and scalable graph processing method based on streaming topology to GPUs.," in In Proceedings of the 2016 International Conference on Management of Data (pp. 447-461). ACM., 2016.

An SDN QoE Monitoring Framework for VoIP and video applications

Maria-Evgenia I. Xezonaki (mxezonaki@di.uoa.gr, me.xezonaki@gmail.com)

ABSTRACT

This diploma thesis aims at presenting SDN technology, reviewing existing research in the field of QoE on SDN networks and then developing an SDN application that monitors and preserves the QoE for VoIP and video applications. More specifically, the developed SDN QoE Monitoring Framework (SQMF) periodically monitors various network parameters on the VoIP/video packets transmission path, based on which it calculates the QoE. If it is found that the result is less than a predefined threshold, the framework changes the transmission path, and thus the QoE recovers.

Keywords: Software-Defined Networks, Quality of Experience, VoIP, video, Monitoring, SDN Controller, OpenDaylight, OpenFlow, Mininet.

Advisors

Lazaros Merakos, Professor (University of Athens)

1. INTRODUCTION

For many years, the Information and Communication Technologies (ICTs) have been representing highly profitable business areas with continuous developments of technologies, devices and services in order to serve all types of users [1]. Undoubtedly, the communications and computer networking sector is one of the most crucial elements in the global ICT strategy, underpinning many other industries. It is one of the fastest growing and most dynamic sectors worldwide, allowing for the interconnection between either individual persons or institutions, companies, businesses, industries and in general every kind of functional departments worldwide [2]. Lately, a drive for changing the conventional networking architecture and moving towards new networking paradigms is beginning to show. This trend can be explained by the limitations imposed by the current networking state, which is characterized by static architecture and complex devices, as well as to the emerging needs of next generation networks. To this end, there is an emerging need for an alternative networking approach to effectively face these challenges.

2. SOFTWARE-DEFINED NETWORKING

Software Defined Networking (SDN) is an emerging networking paradigm that promises to address the challenges occurring from the current networking state and the predictions for its future evolution. SDN breaks the vertical integration by separating the network's control logic (the control plane) from the underlying routers and switches that forward the traffic (the data plane), using an open standard protocol for the communication between them [3]. OpenFlow (OF) is the most-well known such protocol, used for this thesis. With the separation of the control and data planes, the function of control element no longer executes in the switches but rather in a logically centralized controller with a global view of the entire network [4]. The most well-known open source SDN Controller is OpenDaylight (ODL), which supports OF and is also used for this thesis. The network switches become simple forwarding devices routing the traffic according to rules set to them by the control plane. SDN's high-level architecture is illustrated below. It consists of:

- The infrastructure layer, which contains the network elements e.g. switches, routers
- The control layer, which contains the centralized entity with control on network elements
- The applications layer, which contains the applications with control on resources.

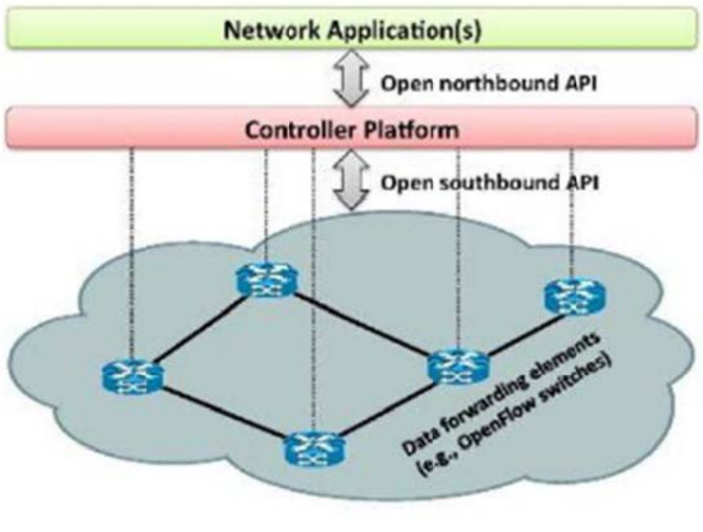


Figure 1 : The SDN high-level architecture

The OF protocol enables the control plane to define forwarding rules for the network devices of the data plane. Each OF switch has one or more flow tables, and each flow tables contains flow rules. A flow rule is a packet handling instruction and contains:

● Match fields : Fields to match against packets

● Priority : Field to match the precedence of the flow rule

● Counters : increased by one when a packet is matched

● Instructions : modification of the action set or pipeline processing

● Cookie : Data value handled and selected by the controller

An incoming packet matches a flow rule, if the values in the packet match fields match those specified in the flow rule. An example flow rule is given below:

$$\texttt{cookie=0x2b000000000000f6, table=0, priority=2, }\underbrace{\texttt{udp,in\_port=3}}_{\textbf{Match fields}}\texttt{, }\overbrace{\texttt{actions=output:2}}^{\textbf{Instructions}}$$

The lookup process starts in the first table and ends either with a match in one of the tables of the pipeline or with a miss (when no rule is found for that packet). A packet matches a flow table entry if the values in the packet match fields, used for the lookup, match those specified in the flow entry. Each packet is matched against the table and only the highest priority entry that matches the packet must be selected. In case of a successful match, the action(s) specified in the rule are executed. If there is no matching rule in the flow tables, the packet is either dropped or an OpenFlow message containing the packet header is sent to the controller for processing. The controller calculates the action the network element should take with regard to the packet and communicates it. Furthermore, the controller can specify a flow rule and send it to the network element(s). This way, all following packets of the flow are treated the same way by the network, and the controller does not need to be involved any longer. The controller can also introduce new flow rules or modify existing ones without being triggered by an incoming packet.

Moreover, every flow table must support a table-miss flow entry to process table misses. This flow entry defines how to handle packets that are not matched against other flow entries in the flow table. As a result, such packets may be sent to the controller, be dropped or be directed to a subsequent table. If such a table-miss entry does not exist, by default, packets unmatched by flow entries are discarded [5].

3. QUALITY OF EXPERIENCE

As in all networks, the importance of taking care of user satisfaction with service provisioning in SDNs has been realized. Content providers are immensely interested in ensuring a high degree of satisfaction for their end-users. Understanding and measuring quality of communication services and underlying networks from an end-user perspective has attracted increased attention over the course of the last decade.

Networks try to support the requirements based on Quality of Service (QoS) parameters, such as throughput, latency and jitter. However, the performance of a specific application cannot be determined by simply relying on QoS metrics. A growing awareness of the scientific community that technology-centric QoS concept is not powerful enough to cover every relevant performance aspect of a given application or service has been witnessed [6]. Network level metrics traditionally used by network administrators are not adequate to indicate how satisfied a user is with his experience. In addition, research shows that there is not always a direct or deterministic correlation of the impact of the network-level metrics to the users' satisfaction.

Thus, the evaluation of network applications should be based on user-centric metrics that provide a better indication of the satisfaction of the end-users and define the Quality of Experience (QoE) [7], [8]. QoS measurement is most of the time not related to a customer, but to the media or network itself. On the contrary, QoE takes into consideration the end-to-end connection and applications that are currently running over that network connection and how multimedia elements are satisfying or meeting the end user's requirements.

In order to have an exact and unified measure of the QoE in various applications, several QoE models were developed. Below are presented two QoE models, one for VoIP and one for video applications, which are used in the context of this thesis.

a) The ITU-T G.107 E-model for VoIP

Based on this model, the QoE for VoIP applicatIons is given by the following relationship:

$$MOS = \begin{cases} 1 & if\ R < 0 \\ 1 + 0.035R + R(R-60)(100-R)7 * 10^{-6} & if\ 0 \leq R \leq 100 \\ 0 & if\ R > 100 \end{cases}$$

where:

$$R = 94.2 - 0.024d - 0.11(d - 177.3)H(d - 177.3) - 11 - 40\ln(1 + 10p)$$

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

and *d* stands for delay, *p* for packet loss [9].

b) The ITU-T G.1070 E-model for video

Based on this model, the QoE for video applications is given by the following relationship:

$$Vq = 1 + Icoding * Itransmission$$

where:

$$Icoding = \ Iofr * \ e^{-\frac{(\ln(Frv) - \ln(Ofr))^2}{2DFrv^2}}$$

$$Itransmission = \mathrm{e}^{-\frac{\mathrm{Pplv}}{\mathrm{Dpplv}}}$$

All the factors can be estimated if the following are known:

- *Fr_v* (video frame rate)
- *Br_v* (video bit rate)
- *Ppl_v* (video packet loss rate) [10].
-

4. THE SDN QoE MONITORING FRAMEWORK

This thesis develops the SDN QoE Monitoring Framework (SQMF), an SDN framework implemented in order to overcome situations where the QoE can face degradation in an SDN network, and preserve the QoE in VoIP and video applications. This is achieved by using an SDN Controller and implementing extra functionality on top of it in order to change the traffic's transmission path to an alternative one, when QoE falls below a specified threshold.

The SDN Controller used for the current thesis is version Boron SR1 of ODL. The project was developed in Ubuntu 14.04 OS using Java 8 JDK, Maven 3.3.9 and IntelliJ IDE and extended the SDN Controller functionality by implementing an extra SDN module, named *sqmf* . The network topology used for validation and experiments was created using Mininet network emulator and is depicted below.

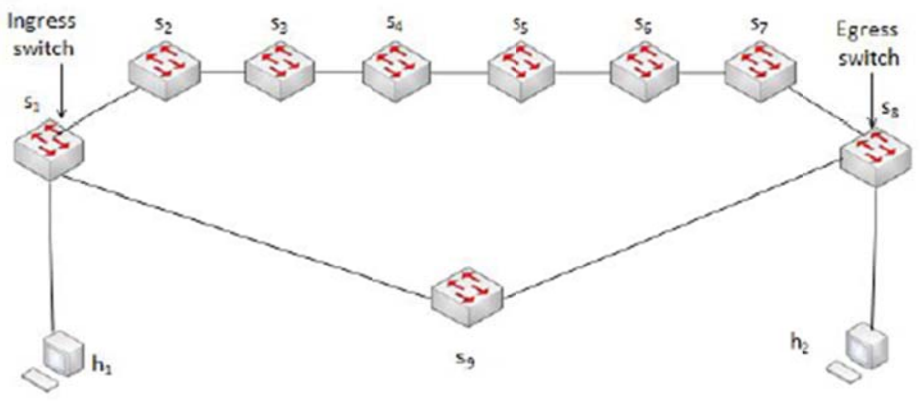


Figure 2 : The topology used for SQMF

The approach used in order to ensure that the QoE remains in satisfactory levels is the *periodical link monitoring and QoE estimation* based on their statistics. In particular, the application computes the shortest path between the source and destination hosts, which will be the main transmission path, as well as the second shortest path (if exists) which will assist as a failover path. Then, rules are inserted to forward the traffic to the main path. Afterwards, the QoE monitoring process starts; the SDN controller periodically collects statistics from the switches (different statistics for each application type) and uses them to compute the QoE level. If the estimated value is lower than a specified threshold, then appropriate rules are inserted to redirect traffic to the failover path.

The collected statistics according to the streamed application type are:

- For VoIP applications, the delay and packet loss are necessary, in order to be able to use the G.107 E-model.
    - Each time the SDN controller needs to measure the **delay**, it creates a packet with a specific source MAC address – 00:00:00:00:00:09 in particular – and sends it on behalf of each switch of the path (except for the egress switch) to the output interface, so that the next switch of the path receives it. Each switch (except for the ingress switch, as it has no previous switch to receive a packet from) is configured with an appropriate flow rule to forward to the Controller any packet with the specific MAC address. The difference between the time that a switch receives a packet and the time that the previous switch had sent the packet is the delay of a particular link. The addition of all the path links' delays results in the path delay. Each switch is configured with a rule of the following format:

      ```
      priority=1000,dl_src=00:00:00:00:00:09 actions=CONTROLLER:65535
      ```

    - In order to compute the **packet loss** rate, and given that VoIP traffic generates UDP packets, the SDN controller periodically monitors the number of UDP packets sent from the sender (h_1) and the number of UDP packets received by the receiver (h_2) and computes their difference divided by the number of sent packets. To achieve packet loss monitoring, the ingress and the egress switches are configured appropriately so as to forward to the Controller – apart from the predefined output interface to the next node - any UDP packet they receive (the ingress receives UDP packets from the sender host and the egress from the previous path node). In its turn, the Controller counts the path's total incoming and outgoing UDP packets and is able to determine the packet loss. The ingress and the egress switch are configured with rules of the following format:

```
priority=1000,udp,in_port=x actions=CONTROLLER:65535,output:y
```

For example, the appropriate rules for s_1 and s_8 of Figure 2 : The topology used for SQMF are:

```
s1 : priority=1000,udp,in_port=1 actions=CONTROLLER:65535,output:3
s8 : priority=1000,udp,in_port=3 actions=CONTROLLER:65535,output:2
```

- For video applications, the bit rate, frame rate and packet loss are necessary, in order to use the G.1070 E-model.
    - In order to compute the **bit rate** , the command

      ```
      ffmpeg –i [VIDEO_PATH] –hide_banner
      ```

      is executed through the Java code and the output is parsed until the bit rate value is accessed.
    - In order to compute the **frame rate** , the command

      ```
      ffmpeg –i [VIDEO_PATH] –hide_banner
      ```

      is executed through the Java code and the output is parsed until the frame rate value is accessed.
    - The **packet loss** is computed in the same way as for VoIP applications.

For both approaches, the implemented delay and packet loss computation methods are based on the OpenNetMon framework, proposed in [11].

5. SQMF EVALUATION

The current subchapter conducts a proof-of-concept evaluation of SQMF through graphical representations that compare the network's behavior with and without the SQMF functionality in the event of a link failure, i.e. comparing the default forwarding with the QoE-based forwarding when a link failure occurs. The experiments were conducted both on VoIP and video traffic cases.

A. In order to evaluate the SQMF functionality for VoIP applications, **VoIP traffic was generated from h1 to h 2 for 105 seconds** . The traffic was generated using a traffic generator tool named D-ITG. The parameters used for the current experiment are shown in the table below.

Table 1: Parameters used for VoIP traffic generation

Packets per second	50
VoIP codec	G.729.2
Traffic generation duration	105 sec

As a first case, the QoE monitoring – based forwarding functionality did not take place and therefore the packets kept being forwarded to the main path as initially configured when a link failure occurred. This caused total packet loss after the link failure. Then, the same experiment was carried out using the SQMF functionality. In this case, the controller efficiently changed the transmission path when the link failure occurred, as it detected QoE levels below the predefined threshold, by configuring the packets to be forwarded to the failover path, so the overall packet losses were much lower.

By illustrating the results of the two cases, with and without QoE monitoring, in the same graphical representation, it is obvious that QoE monitoring-based forwarding performs much better than the default forwarding to a configured main path. The figure below shows that QoE monitoring-based forwarding achieves much lower packet losses when a link failure occurs, managing to recover immediately after a small period of packet loss detection, in contrast to the default case where all the packets are lost after the link failure.

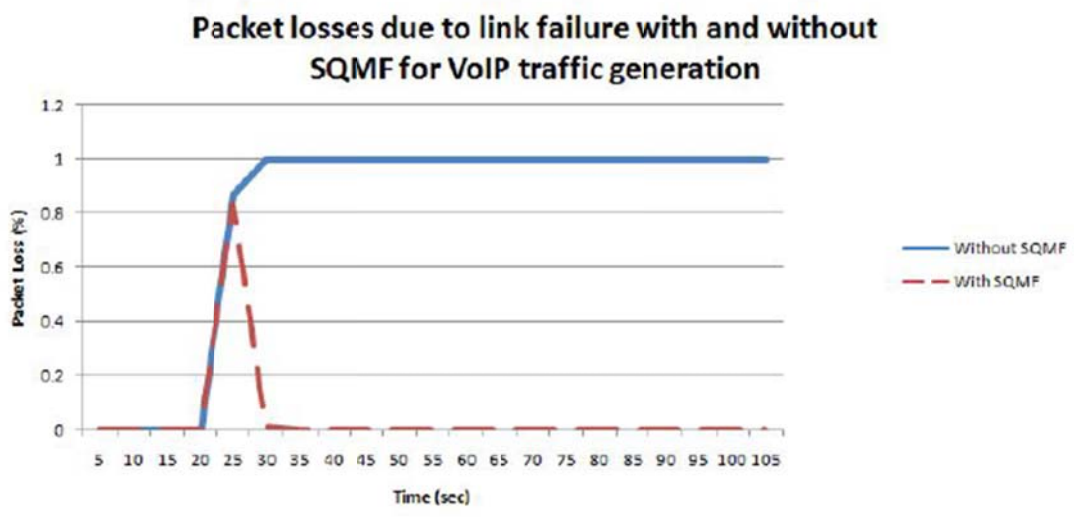


Figure 3: Packet losses with and without SQM for VoIP traffic

Therefore, the QoE Monitoring-based forwarding preserves the total QoE and keeps it at high levels even after the link failure, whereas in the default case the QoE faces a permanent degradation after the failure, as depicted in the figure below.

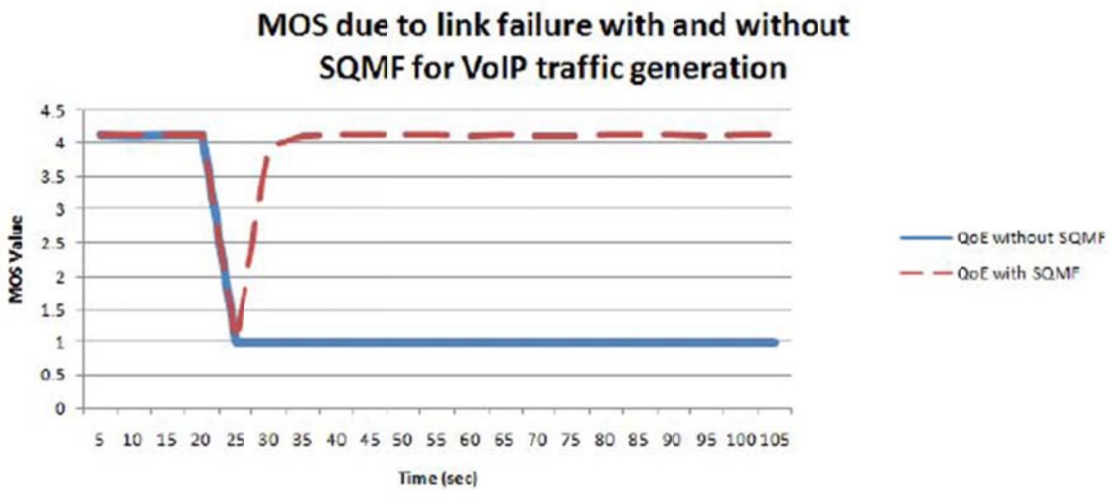
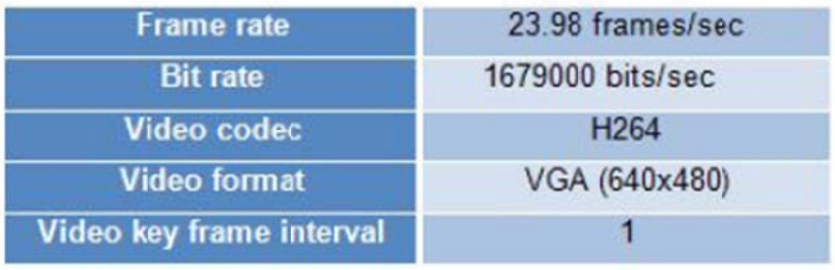


Figure 4: QoE with and without SQMF for VoIP traffic

B. In order to evaluate the SQMF functionality for video applications, **video traffic was generated from h1 to h 2 for 105 seconds.** The traffic was generated using VLC player. The video parameters used for the current experiment are depicted in the table below.

Table 2: Parameters used for VoIP traffic generation



Frame rate	23.98 frames/sec
Bit rate	1679000 bits/sec
Video codec	H264
Video format	VGA (640x480)
Video key frame interval	1

As a first case, the SQMF functionality did not take place and therefore the packets kept being forwarded to the main path as initially configured when a link failure occurred. This caused total packet loss after the link failure. Then, the same experiment was carried out using the SQMF functionality. In this case, the controller monitored periodically the QoE and when the QoE was detected to be lower than the threshold, due to the link failure, it efficiently changed the transmission path by configuring the packets to be forwarded to the backup path, so the packet losses were much lower and of smaller duration.

By illustrating the results of the two cases, with and without QoE monitoring, in the same graphical representation, it is obvious that QoE monitoring-based forwarding performs much better for video applications than the default forwarding to a configured main path.

The figure below shows that QoE monitoring-based forwarding achieves much lower packet losses when a link failure occurs, managing to recover immediately after a small period of packet loss detection, in contrast to the default case where all the packets are lost after the link failure.

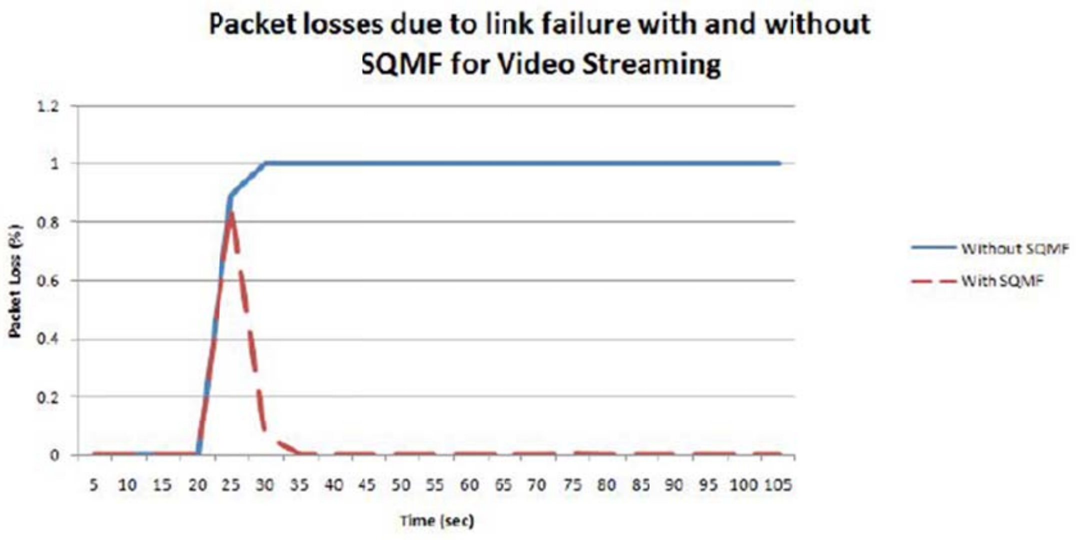


Figure 5: Packet losses with and without SQM for video traffic

Therefore, the QoE monitoring-based forwarding preserves the total QoE and keeps it at high levels even after the link failure, whereas in the default case the QoE faces a permanent degradation after the failure, as depicted in the figure below.

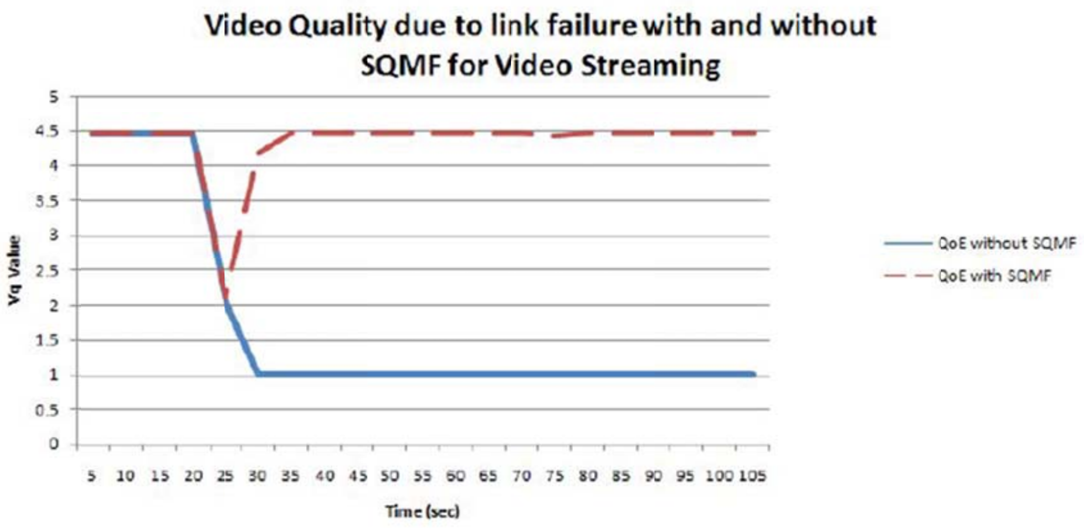


Figure 6: QoE with and without SQM for video traffic

6. REFERENCES

[1] L. Sørensen, K. Skouby, "Visions and research directions for the Wireless World", July 2009, pp. 5-9.

[2] C. Wang et al., "Cellular Architecture and Key Technologies for 5G Wireless Communication Networks", IEEE Communications Magazine, February 2014.

[3] S. Ramakrishnan and X. Zhu, "An SDN Based Approach To Measuring And Optimizing ABR Video Quality Of Experience", Cisco Systems, 2014.

[4] W. Hsu et al., "The Implementation of a QoS/QoE Mapping and Adjusting Application in software-defined networks", 2 nd International Conference on Intelligent Green Building and Smart Grid (IGBSG), 2016.

[5] Σ. Γ. Μαστοράκης, "Μέθοδοι εξουσιοδότησης για δέσμευση πόρων σε Ευφυή – Προγραμματιζόμενα - Δίκτυα (Software-Defined-Networks)", May 2014.

[6] R. Schatz et al., "From Packets to People: Quality of Experience as a New Measurement Challenge" , Lecture Notes in Computer Science, Springer, pp. 219-263.

[7] A. Farshad et al., "Leveraging SDN to Provide an In-network QoE Measurement Framework", IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), May 2015.

[8] M. Jarschel et al., "SDN-based Application-Aware Networking on the Example of YouTube Video Streaming", 2 nd European Workshop on Software Defined Networks (EWSDN), September 2013.

[9] R. G. Cole and J. H. Rosenbluth, "Voice over IP Performance Monitoring", ACM SIGCOMM Computer Communication Review, Volume 31, Issue 2, pp. 9 – 24, April 2001.

[10] INTERNATIONAL TELECOMMUNICATIONS UNION, TELECOMMUNICATIONS STANDARDIZATION SECTOR, "Opinion model for video-telephony applications", **https://www.itu.int/rec/T-REC-G.1070** , July 2012.

[11] N. L. M. van Adrichem et al., "OpenNetMon: Network monitoring in OpenFlow Software-Defined Networks", IEEE Network Operations and Management Symposium (NOMS), May 2014.

Algorithms, Hashing, Parallel processing, and the Nearest Neighbor problem in High Dimensions

Georgios C. Samaras (gsamaras@di.uoa.gr)

ABSTRACT

The *c*-approximate Near Neighbor decision problem in high-dimensional spaces has been mainly addressed by Locality Sensitive Hashing (LSH). While, in practice, it is important to ensure linear space usage, most previous work in this regime focuses on the case that *c* exceeds 1 by a constant term. We present a simple data structure using linear space and sublinear query time for any *c*>1: given an LSH family of functions for some metric space, we randomly project points to vertices of the Hamming cube in dimension ≤log*n*, where *n* is the number of input points. The search algorithm projects the query, then examines points which are assigned to nearby vertices on the cube. We report on several experiments with synthetic and real datasets in dimension ≤1000 and , and compare against the state-of-the-art LSH-based library FALCONN: our algorithm and implementation are significantly simpler, with comparable performance in terms of query time and storage for one of the LSH families used by FALCONN, whereas our code is orders of magnitude faster than brute force.

Keywords: Near Neighbor, high dimension, linear storage, sublinear query, random projection, implementation, machine learning

Advisors
Ioannis Z. Emiris, Professor, Dimitrios Gounopoulos, Professor, Panagiotis Stamatopoulos, Assistant Professor

1. INTRODUCTION

The problem of Approximate Nearest Neighbor search is fundamental in machine learning: one has to preprocess a dataset so as to answer proximity queries efficiently, for a given query object. We focus on Euclidean or other metric spaces, when the dimension is high. Typically one supposes dimension d≫logn, where n denotes the number of input data; in fact, dimension is an input parameter, so we need to address the curse of dimensionality. Due to known reductions, e.g. HIM12, one may focus on designing an efficient data structure for the Approximate Near Neighbor (ANN) problem instead of directly solving the Approximate Nearest Neighbor problem. The former is a decision problem, whose output may contain a witness point (as in Definition 1), whereas the latter is an optimization question. The $(1+ε,r)$-ANN problem, where $c=1+ε$, is defined as follows.

Definition 1 (Approximate Near Neighbor problem) Let be a metric space. Given $P⊆M$, and reals $r>0$ and $ε>0$, build a data structure s.t. for any query $q∈M$, there is an algorithm performing as follows:

- if s.t. , then return any point $p'∈P$ s.t. ,
- if $∀p∈P$, , then report “no”.

An important approach for such problems today is Locality Sensitive Hashing (LSH). It has been designed precisely for problems in general dimension. The LSH method is based on the idea of using hash functions enhanced with the property that it is more probable to map nearby points to the same bucket.

Definition 2 Let reals and . We call a family F of hash functions -sensitive for a metric space M if, for any x, $y∈M$, and h distributed uniformly in F, it holds:

- $d_M(x,y) \le r_1 \Rightarrow \Pr[h(x) = h(y)] \ge p_1$
- $d_M(x,y) \ge r_2 \Rightarrow \Pr[h(x) = h(y)] \le p_2$.

Let us survey previous work, focusing on methods whose complexity has polynomial, often even linear, dependence on the dimension. LSH was introduced in IM98; HIM12 and yields data structures with query time and space . Since then, the optimal value of the LSH exponent, $ϱ<1$, has been extensively studied for several interesting metrics, such as and . In a series of papers IM98; DI04; MNP07; AI08; OWZ11, it has been established that the optimal value for the Euclidean metric is , and that for the Hamming distance is $ϱ=1/c±o(1)$.

In contrast to the definition above, which concerns data-independent LSH, quite recently the focus has been shifted to data-dependent LSH. In the latter case, the algorithms exploit the fact that every dataset has some structure and consequently this approach yields better bounds for the LSH exponent. Specifically, in AR15 they showed that $ϱ=1/(2c−1)$ for the distance and for the distance.

The data-dependent algorithms, though better in theory, are quite challenging in practice. In AILRS15, they present an efficient implementation of one part of AR15. Another attempt towards practicality for a data-dependent algorithm was recently made in ARN17, where they presented a new approach based on LSH forests. Typically though, data-independent algorithms, such as the one proposed in this work, yield better results in practice than data-dependent algorithms.

For practical applications, an equally important parameter is memory usage. Most of the previous work in the (near) linear space regime focuses on the case that c is greater than 1 by a constant term. When c approaches 1, these methods become trivial in the sense that query time becomes linear in n. One such LSH-based approach Pan06 offers query time proportional to , which is sublinear in n only for large enough c>1. Improvements on practical aspects of the above result leaded to the novel multi-probe scheme for LSH Ch07. Two noteworthy exceptions are the recently accepted papers ALRW17 and Christ17, where they achieve near-linear space and sublinear query time, even for .

Another line of work that achieves linear space and sublinear query time for the Approximate Nearest Neighbor problem is based on random projections to drastically lower-dimensional spaces, where one can simply search using tree-based methods, such as BBD-trees AEP15; AEP16. This method relies on a projection extending the type of projections typically based on the Johnson-Lindenstrauss lemma. The new projection only ensures that an approximate nearest neighbor in the original space can be found among the preimages of k approximate nearest neighbors in the projection. From the practical perspective, similar ideas have been employed in SWQZL14. Random projections which preserve the relative order of points have been also considered in LM16.

In this paper, we specify a random projection from any space endowed with an LSH-able metric, to the vertices of the Hamming hypercube of dimension logn, , where n is the number of input points. Random projections which map points from a high dimensional Hamming space to lower dimensional Hamming space have been already used in the ANN context KOR00. The projected space contains strings which serve as keys for buckets containing the input points. The query algorithm simply projects the query point, then examines points which are assigned to the same or nearby vertices on the Hamming cube.

Our strategy resembles the multi-probe approach in the sense that after locating the query, we start searching in "nearby" buckets. However, in our case, nearby buckets are simply the ones which are close to each other with respect to the Hamming distance of their keys, whereas in the multi-probe case, the nearby buckets are the ones with high probability of success for a given query. Computing which are the right buckets to explore in the multi-probe approach is potentially harder than in our case.

The random projection in our algorithm relies on the existence of an LSH family for the input metric. We study standard LSH families for and , for which we achieve query time where , $\varepsilon \in (0,1]$. The constants appearing in δ vary with the LSH family, but it holds that $\delta > 0$ for any $\varepsilon > 0$. The space and preprocessing time are both linear for constant probability of success, which is important in practical applications.

We illustrate our approach with an open-source implementation, and report on a series of experiments with n up to and d up to 1000, where results are very encouraging. It is evident that our algorithm is 8.5–80 times faster than brute force search, depending on the difficulty of the dataset. Moreover, we handle a real dataset of images represented in 960 dimensions with a query time of less than 128 msec on average. We test our implementation with synthetic and real image datasets, and we compare against FALCONN, an LSH-based library AILRS15, for which we use a "multi-probe" setting of parameters. For SIFT, MNIST and GIST image datasets, we achieve comparable performance in terms of memory usage and query time for one of the two LSH families used by FALCONN. Our software outperforms FALCONN when it uses the other LSH family.

The rest of the paper is structured as follows. The next section states our main algorithmic and complexity results for the (c,r)-ANN problem in the Euclidean and Manhattan metrics. In Section 3, we discuss our implementation, and in Section 4, we present our experimental results. We conclude with open questions.

2. DTA STRUCTURES

This section introduces our main data structure, and the corresponding algorithmic tools. We start our presentation with an idea applicable to any metric admitting an LSH-based construction, aka LSH-able metric. Then, we study some classical LSH families which are also simple to implement.

The algorithmic idea is to apply a random projection from any LSH-able metric to the Hamming hypercube. Given an LSH family of functions for some metric space, we uniformly select d' hash functions, where d' is specified later. The nonempty buckets defined by any hash function are randomly mapped to {0,1}, with equal probability for each bit. Thus, points are projected to the Hamming cube of dimension d'. Thus, we obtain binary strings serving as keys for buckets containing the input points. The query algorithm projects a given point, and tests points assigned to the same or nearby vertices on the hypercube. To achieve the desired complexities, it suffices to choose d'=logn. In Algorithms 1 and 2, we present the preprocessing and query algorithms.

The main lemma below describes the general ANN data structure whose complexity and performance depends on the LSH family that we assume is available. The proof details the data structure construction.

Lemma 3 (Main) Given a -sensitive hash family F for some metric and input dataset P⊆M, there exists a data structure for the (c,r)-ANN problem with space O(dn), time preprocessing O(dn), and query time , where

$$\delta = \delta(p_1, p_2) = \frac{(p_1 - p_2)^2}{(1 - p_2)} \cdot \frac{\lg e}{4}$$

where e denotes the basis of the natural logarithm, and H(·) is the binary entropy function. The bounds hold assuming that computing and computing the hash function cost O(d). Given some query q∈M, the building process succeeds with constant probability.

Discussion on parameters.

We set the dimension d'=logn (which denotes the binary logarithm), since it minimizes the expected number of candidates under the linear space restriction. We note that it is possible to set d'<logn and still have sublinear query time. This choice of d' is interesting in practical applications since it improves space requirement. The number of candidate points is set to for the purposes of Lemma 3 and under worst case assumptions on the input. In practice, this should be a user-defined parameter and hence it is denoted by the parameter StopSearch in Algorithm 2.

Algorithm 1 Dolphinn: Preprocessing (data structure)

```
input Metric $(\mathcal{M}, d_{\mathcal{M}})$, radius $r > 0$, approximation factor $c > 1$, LSH family $F = F(c, r)$, data set $P \subset \mathcal{M}$, parameter $d'$.
  Initialize empty hashtable $T$.
  for $i = 1$ to $d'$ do
    Sample $h_i \in F$ u.a.r.
    for each $x \in h_i(P)$ do
      Flip a fair coin and assign the result to $f_i(x)$.
    end for
  end for
  For all $p \in P$, $f(p) = (f_1(h_1(p)), \dots, f_{d'}(h_{d'}(p)))$.
  For all $p \in P$, add $p$ to the bucket of $T$ with key $f(p)$.
```

Algorithm 2 Dolphinn: Query Algorithm

```
input Metric $(\mathcal{M}, d_{\mathcal{M}})$, LSH family $F$, data set $P \subset \mathcal{M}$, parameter $d'$, integer StopSearch, query $q$. (We assume that this algorithm has access to the ANN data structure created in Algorithm 1)
output Point $p \in P$ or "no"
  for $i = 1$ to $d'$ do
    if $f_i(h_i(q))$ is not defined in Algorithm 1 then
      Flip a fair coin and assign the result to $f_i(h_i(q))$.
    else
      Compute $f_i(h_i(q))$.
    end if
  end for
  i=0
  for each $x$ in $f(P)$ s.t. $\|x - f(q)\|_1 \leq 0.5 \cdot d' \cdot (1 - p_1)$ do
    for each point $p$ inside the bucket with key $x$ do
      if $d_{\mathcal{M}}(p, q) \leq c \cdot r$ then
        return $p$.
      end if
      $i \leftarrow i + 1$
      if $i > StopSearch$ then
        return "no".
      end if
    end for
  end for
  return "no"
```

2.1 The L2 case

In this subsection, we consider the (c,r)-ANN problem when the dataset consists of n points , the query is , and the distance is the Euclidean metric.

We may assume, without loss of generality, that r=1, since we can uniformly scale . We shall consider two LSH families, for which we obtain slightly different results. The first is based on projecting points to random lines, and it is the algorithm used in our implementation, see Section 3. The second family relies on reducing the Euclidean problem to points on the sphere, and then partitioning the sphere with random hyperplanes.

2.1.1 Project on random lines

Let p, q two points in and η the distance between them. Let w>0 be a real parameter, and let t be a random number distributed uniformly in the interval [0,w]. In DI04, they present the following LSH family. For , consider the random function

$h(p) = \lfloor \frac{\langle p,v \rangle + t}{w} \rfloor$, p, v $\in R^d$, (1)

where v is a vector randomly distributed with the d-dimensional normal distribution. This function describes the projection on a random line, where the parameter t represents the random shift and the parameter w the discretization of the line. For this LSH family, the probability of collision is

$$\alpha(\eta, w) = \int_{t=0}^{w} \frac{2}{\sqrt{2\pi}\eta} \exp(-\frac{t^2}{2\eta^2})(1-\frac{t}{w})dt.$$

Lemma 4 Given a set of n points , there exists a data structure for the (c,r)-ANN problem under the Euclidean metric, requiring space O(dn), time preprocessing O(dn), and query time , where

$$\delta \geq 0.03(c-1)^2.$$

Given some query point , the building process succeeds with constant probability.

2.1.2 Hyperplane LSH

This section reduces the Euclidean ANN to an instance of ANN for which the points lie on a unit sphere. The latter admits an LSH scheme based on partitioning the space by randomly selected halfspaces.

In Euclidean space , let us assume that the dimension is d=O(logn·loglogn), since one can project points à la Johnson-Lindenstrauss DG02, and preserve pairwise distances up to multiplicative factors of 1±o(1). Then, we partition using a randomly shifted grid, with cell edge of length . Any two points for which lie in the same cell with constant probability. Let us focus on the set of points lying inside one cell. This set of points has diameter bounded by . Now, a reduction of Val15, reduces the problem to an instance of ANN for which all points lie on a unit sphere , and the search radius is roughly . These steps have been also used in ALRW17, as a data-independent reduction to the spherical instance.

Let us now consider the LSH family introduced in Cha02. Given n unit vectors, we define, for each, hash function h(q)=sign⟨q,v⟩, where v is a random unit vector. Obviously, , where Θ(p,q) denotes the angle formed by the vectors . Instead of directly using the family of Cha02, we employ its amplified version, obtained by concatenating k≈1/r' functions h(·), each chosen independently and uniformly at random from the underlying family. The amplified function g(·) shall be fully defined in the proof below. This procedure leads to the following.

Lemma 5 Given a set of n points $P \subseteq R^d$, there exists a data structure for the (c,r)-ANN problem under the Euclidean metric, requiring space O(dn), time preprocessing O(dn), and query time $O(dn^{1-\delta} + n^{0.91})$, where

$\delta \geq 0.05 \cdot (\frac{c-1}{c})^2$.

Given some query , the building process succeeds with constant probability.

The data structure of Lemma 5 provides slightly better query time than that of Lemma 4, when c is small enough.

2.2 The L1 case

In this section, we study the (c,r)-ANN problem under the metric. The dataset consists again of n points and the query point is.

For this case, let us consider the following LSH family, introduced in AI06. A point p is hashed as follows:

$$h(p) = (\lfloor \frac{p_1+t_1}{w} \rfloor, \lfloor \frac{p_2+t_2}{w} \rfloor, \dots, \lfloor \frac{p_d+t_d}{w} \rfloor),$$

where is a point in P, w=αr, and the are drawn uniformly at random from [0,…,w). Buckets correspond to cells of a randomly shifted grid.

Now, in order to obtain a better lower bound, we employ an amplified hash function, defined by concatenation of k=α functions h(·) chosen uniformly at random from the above family.

Lemma 6 Given a set of n points $P \subseteq R^d$, there exists a data structure for the (c,r)-ANN problem under the metric, requiring space O(dn), time preprocessing O(dn), and query time $O(dn^{1-\delta} + n^{0.91})$, where

$\delta \geq 0.05 \cdot (\frac{c-1}{c})^2$.

Given some query point , the building process succeeds with constant probability.

3. IMPLEMENTATION

This section discusses our C++ library, named Dolphinn which is available online . The project is open source, under the BSD 2-clause license. The code has been compiled with g++ 4.9 compiler, with the O3 optimization flag enabled. Important implementation issues are discussed here, focusing on efficiency.

Our implementation is based on the algorithm from Subsection 2.1.1 and supports similarity search under the Euclidean metric. We denote by F the LSH family introduced above (see also Data et al., 2004), based on projection on random lines, see Subsection 2.1.1. The data structure to which the points are mapped is called Hypercube.

Parameters.

Our implementation provides several parameters to allow the user to fully customize the data structure and search algorithm, such as:

d' The dimension of the Hypercube to which the points are going to be mapped. The larger this parameter, the faster the query time, since it leads to a finer mapping of the points. However, this speedup comes with increased memory consumption and build time.

StopSearch Maximum number of points to be checked during search phase. The greater this parameter, the more accurate results will be produced, at the cost of an increasing query time.

r It is provided by the user as input. The algorithm checks whether a point lies within a radius r from the query point.

μ, σ,w Specify the hash function h from LSH family F, by Equation 1. In particular, μ,σ specify the random vector under the Normal Distribution , which is multiplied by the point vector, whereas w is the “window" size of the line partition.

Configuration.

Despite the simple parameter set, we employ default values for all parameters (except r), which are shown to make Dolphinn run efficiently and accurately for several datasets. Moreover, the whole configuration is designed for an one-threaded application. However, it would be interesting to modify it and take advantage of the huge parallel potential of Dolphinn.

Hypercube.

The Hypercube data structure contains all the necessary information, so that it can efficiently hash a query on arrival, by using two hash tables, that indicate the statistical choices made upon build and the index of every associated original point. Moreover, another hashtable is constructed, which maps the vertices of the Hypercube to their assigned original points. This hierarchy provides space and time efficiency, and is very natural to code, because of the simplicity of the algorithm.

Miscellaneous.

Parallel processing is able to give a boost in Dolphinn’s performance, both in the preprocessing and the search phase, while we managed to minimize thread communication and balance every thread’s workload. Efficient distance computation is important, since in high dimensions this operation is very time consuming. Bit manipulation and caching were also very important, since hashing was faster by employing certain relative design choices. Selecting between recursive or iterative schemes for certain tasks provided additional speedup for Dolphinn.

4. EXPERIMENTAL RESULTS

This section presents our experimental results and comparisons on a number of synthetic and real datasets. We also analyze our findings.

All experiments are conducted on a processor at 3 GHz×4 with 8 GB. We compare with the state-of-the-art LSH-based FALCONN library, and the straightforward brute force approach. We examine build and search times, as well as memory consumption. FALCONN implements two LSH families in 4,748 C++ lines of code, while Dolphinn implements one LSH family in 716 lines. In particular, FALCONN implements the Hyperplane LSH family Charikar 2002 and the Cross-Polytope LSH family Andoni et al., 2015, which are both designed for cosine similarity search. However, this is known to be sufficient in the Euclidean distance case, assuming that the dataset is isotropic. Notice that such assumptions are not required in our method.

Datasets.

We use 5 datasets of varying dimensionality and cardinality. To test special topologies, two sets, Klein bottle and Sphere, which are synthetic. We generate points on a Klein bottle and on a sphere embedded in , then add to each coordinate zero-mean Gaussian noise of standard deviation 0.05 and 0.1 respectively. In both cases, queries are nearly equidistant to all points, implying high miss rates. In particular, queries are constructed as follows: pick a point from the point set uniformly at random and, following a Normal Distribution N(0,1), add a small random quantity to every coordinate. A query is chosen to lie within the fixed radius 1 with a probability of 50%.

The other three datasets, MNIST, SIFT and GIST are presented in JeDS11, and are very common in computer vision, image processing, and machine learning. SIFT is a 128-dimensional vector that describes a local image patch by histograms of local gradient orientations. MNIST contains vectors of 784 dimensions, that are 28×28 image patches of handwritten digits. There is a set of 60k vectors, plus an additional set of 10k vectors that we use as queries. GIST is a 960-dimensional vector that describes globally an entire image. SIFT and GIST datasets each contain one million vectors and an additional set for queries, of cardinality for SIFT and 1000 for GIST. Small SIFT is also examined, with vectors and 100 queries.

For the synthetic datasets, we solve the ANN problem with a fixed radius of 1 and we compare to brute force, since FALCONN does not provide a method for the ANN problem as this is defined in Definition 1. For the image datasets, we find all near neighbors within a fixed radius of 1, by modifying our algorithm in order not to stop when it finds a point that lies within the given radius, but to continue until it reports all points inside the fixed radius, or reach the threshold of points to be checked. This objective is also supported by FALCONN.

Moreover, we tune the number of probes for multiprobe LSH used by FALCONN. For a fair comparison, both implementations are configured in a way that yields the same accuracy and memory consumption.

Preprocessing.

The preprocessing time of Dolphinn has a linear dependence in n and d, as expected, which is shown in Tables 1 and 2. Moreover, Dolphinn is observed to be competitive with FALCONN when employing the Cross-Polytope family, but slower than FALCONN for the Hyperplane LSH family. Table 3 shows representative experiments. We use different values for the number of hash-bits, i.e. the number of bits per key (in our case d'). Note that any normalization and/or centering of the point set, which is a requirement for FALCONN in order to make the pointset more isotropic, is not taken into account when counting runtimes.

Let us explain this performance. The main issue behind the larger building times is the use of two hash tables which simulate the two random functions, one which maps points to keys in and one which maps keys in to keys in . Using an LSH family which directly maps points to would require only one hashtable, but such an LSH family does not always exist. Nevertheless, it is probable that the two random functions may be implemented more efficiently.

Search and Memory.

We conduct experiments on our synthetic datasets while keeping n or d fixed, in order to illustrate how our algorithm's complexity scales in practice. The Sphere dataset is easier than the Klein bottle, which explains the reduced accuracy, as well as the dramatic decrease of speedup (w.r.t. brute force), since more points are likely to lie within a fixed radius of 1, than in the Klein bottle. In general, our algorithm is significantly faster than brute force as expected (especially since the latter is not optimized), and scales well, namely sublinearly in n and linearly in d, as shown in Tables 1 and 2.

Moreover, we report query times between FALCONN and Dolphinn on the image datasets; small SIFT, SIFT, MNIST and GIST, for equal accuracy and memory consumption. Dolphinn outperforms the cross-polytope LSH implementation of FALCONN and it has comparable performance with the Hyperplane LSH implementation, as shown in Figure 1.

Table 1: Sphere dataset: build, Dolphinn search, and brute force, when varying one of n,d.

n	d	build (sec)	search (µsec)	brute f. (sec)
10^5	128	0.053	62.37	0.006
10^5	256	0.092	152.3	0.012
10^5	512	0.168	257.1	0.025

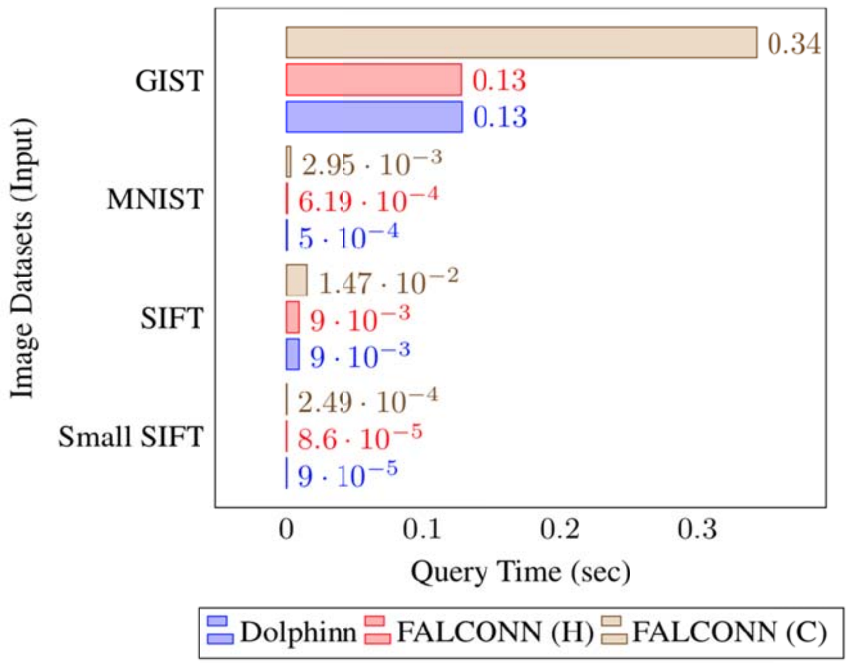


Figure 1. Query time (sec) for 3 implementations: `Dolphinn`, `FALCONN` for Hyperplane LSH family, `FALCONN` for Cross-polytope LSH family.

n	d	build (sec)	search (μsec)	brute f. (sec)
10^5	800	0.255	374.1	0.039
10^5	1024	0.321	499.6	0.050
10^2	512	0.0002	1.001	7.5E-05
10^3	512	0.0016	4.924	0.0004
10^4	512	0.0169	47.72	0.0049
10^5	512	0.1683	477.0	0.0499
10^6	512	1.6800	2529	0.2492

Table 2: Klein bottle dataset: build, Dolphinn search, and brute force when varying one of *n*,*d*.

n	*d*	build (sec)	search (sec)	brute f. (s)
10^5	128	0.053	0.0009	0.0061
10^5	256	0.091	0.0029	0.0147
10^5	512	0.168	0.0031	0.0254
10^5	800	0.259	0.0056	0.0425
10^5	1024	0.321	0.0061	0.0513
10^3	512	0.0003	1E-05	7E-05
10^4	512	0.0016	4E-05	0.0004
10^5	512	0.0169	0.0004	0.0049
10^6	512	0.1679	0.0051	0.0501
10^3	512	1.6816	0.0252	0.2497

Table 3: Build time (sec) for 3 representative datasets; *d'* is the Hypercube dimension in `Dolphinn`, or the number of hashbits in `FALCONN`. We report on: `FALCONN` with the cross-polytope LSH family (`F(c)`), `FALCONN` with the Hyperplane family (`F(h)`), `Dolphinn` (`D`).

	small SIFT		SIFT		GIST
d'	4	16	8	16	8
`Fc`	0.01	0.02	1.33	2.62	8.44
`Fh`	0.00	0.00	0.44	0.72	2.47
`D`	0.02	0.05	1.49	3.33	7.98

5. CONCLUSION

We have designed a conceptually simple method for a fast and compact approach to Near Neighbor queries, and have tested it experimentally. This offers a competitive approach to Approximate Nearest Neighbor search.

Our method is optimal in space, with sublinear query time for any constant approximation factor c>1. The algorithm randomly projects points to the Hamming hypercube. The query algorithm simply projects the query point, then examines points which are assigned to the same or nearby vertices on the hypercube. We have analyzed the query time for the Euclidean and Manhattan metrics.

We have focused only on data-independent methods for ANN, while data-dependent methods achieve better guarantees in theory. Hence, designing a practical data-dependent variant of our method will be a challenging step. Moreover, since our implementation easily extends to other LSH families, it would be interesting to implement and conduct experiments for other metrics.

6. REFERENCES

1. Anagnostopoulos, E., Emiris, I.Z., and Psarros, I. Ran- domized embeddings with slack, and high-dimensional approximate nearest neighbor. *CoRR*, abs/1412.1683, 2014–2016.
2. Anagnostopoulos, E., Emiris, I. Z., and Psarros, I. Low- quality dimension reduction and high-dimensional approximate nearest neighbor. In *Proc. 31st International Symp. on Computational Geometry (SoCG)*, pp. 436– 450, 2015. doi: 10.4230/LIPIcs.SOCG.2015.
3. Andoni, A. and Indyk, P. Efficient algorithms for substring near neighbor problem. In *Proc. 17th Annual ACM- SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1203–1212, 2006.
4. Andoni, A. and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.
5. Andoni, A. and Razenshteyn, I. Optimal data-dependent hashing for approximate near neighbors. In *Proc. 47th ACM Symp. Theory of Computing*, STOC'15, 2015.
6. Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I.P., and Schmidt, L. Practical and optimal LSH for angular dis- tance. In *Advances Neural Information Proc. Systems 28: Annual Conf. Neural Information Processing Sys- tems*, pp. 1225–1233, 2015.
7. Charikar, M. Similarity estimation techniques from round- ing algorithms. In *Proc. 34th Annual ACM Sympo- sium on Theory of Computing, 2002, Montre´al, Que´bec, Canada*, pp. 380–388, 2002.
8. Dasgupta, S. and Gupta, A. An elementary proof of a the- orem of Johnson and Lindenstrauss. *Random Struct. Al- gorithms*, 22(1):60–65, 2003.
9. Har-Peled, S., Indyk, P., and Motwani, R. Approximate nearest neighbor: Towards removing the curse of dimen- sionality. *Theory of Computing*, 8(1):321–350, 2012.
10. Indyk, P. and Motwani, R. Approximate nearest neigh- bors: Towards removing the curse of dimensionality. In *Proc. 30th Annual ACM Symp. Theory of Computing*, STOC'98.
11. Jegou, H., Douze, M., and Schmid, C. Product quantiza- tion for nearest neighbor search. *IEEE Trans. Pattern Analysis & Machine Intell.*, 33(1):117–128, 2011.
12. Panigrahy, R. Entropy based nearest neighbor search in high dimensions. In *Proc. 17th Annual ACM-SIAM Symp. Discrete Algorithms*, SODA'06.
13. Sun, Y., Wang, W., Qin, J., Zhang, Y., and Lin, X. Srs: Solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proc. VLDB Endow.*, 8(1):1–12, September 2014.
14. Valiant, G. Finding correlations in subquadratic time, with applications to learning parities and the closest pair prob- lem. *J. ACM*, 62(2):13, 2015.

Graphity: Out-of-core graph multiplications

Stamatis Christoforidis {stachris@di.uoa.gr}

ABSTRACT

In this work, we deal with generalized graph multiplications (GGM). The model of generalized graph multiplications can express interesting problems. In particular, we present an out-of-core technique that can perform multiple iterations consecutively while managing graphs whose original and / or generated data may not fit into the main memory of a computational node. This technique performs essentially actions on edge data and generates new knowledge through the creation of new edges in the original graph. Then we present Graphity, a prototype system that implements this basic technique and discuss on the challenges we faced during its implementation. Furthermore, we model various problems in known systems based on the Think-Like-a-Vertex model, as well as on relational database systems and compare them experimentally with Graphity. The experimental results show a significant increase in performance compared to existing approaches, while also highlighting the suitability of our approach especially at edge problems.

Keywords: Graph multiplication, Data storage, Query optimization

Advisors

Yannis Ioannidis, Professor, Manos Karvounis, PhD candidate

1. INTRODUCTION

Graphs are a very interesting structure in the IT field. Graphs can represent data from different branches, such as social, biological networks, road networks etc. However, this versatile ability to represent different kind of data means that graphs require special treatment depending on their characteristics and attributes. For example, metrics such as the number of in/out degree, the diameter, node centrality, etc., affect directly the performance of graph processing systems. In addition, graphs are by nature the structure that in general have no traversal locality, while scaling across multiple machines introduces many issues of synchronization issues, graph partitioning, and effective paralleling. In recent years, the Think-like-a-vertex (TLAV) model presented in Pregel [1] have emerged over different architectures, such as distributed systems, shared memory systems, disc based systems, and GPUs.

In the Generalized Graph Multiplications (GGMs) model, many interesting problems can be modeled like friend-of-friend discovery, similarity metrics, common neighbors, etc. In this model, such problems can be expressed in a natural way, but they can also be efficiently calculated with appropriate design principles.

The main contributions of this research are the following: (a) the definition of an execution plan that executes Generalized Graphs Multiplications, while supporting multiple consecutive multiplications; (b) an out-of-core system implementation that applies the above techniques and support graphs, where the initial graph data or the results don’t fit necessarily into the main memory and the disk is used as secondary storage medium; (c) a set of experiments verifying the suitability of this approach, compared to systems that implement the same problem in the TLAV model and relational database systems.

The rest of the text is structured as follows: Chapter 2 defines the basic properties and operations of the GGM model. In Chapter 3 we analyze the proposed plan for executing out-of-core GGMs. In Chapter 4 we compare Graphity experimentally with other existing systems. Then, in Chapter 5, we describe related work around models and systems that can model GGM problems and finally, in Chapter 6, we conclude the survey.

2. GENERALIZED GRAPH MULTIPLICATIONS

2.1 Definition

In Generalized Graph Multiplication (GGM) model, given a graph G, we generalize its adjacency matrix in such a way that each element of (i,j) (i-line, j-column) is mapped to the properties of the edge (i,j). In addition, the GGM substitutes the operators of multiplication and addition during the execution of multiplication, replacing the latter with the CON and AGG operators by the user. The following definition defines GGM mathematically ∘ as:

$$G \circ G = G^2, where\ G^2_{ij} = AGG^{N-1}_{k=0}(G_{ik}\ CON\ G_{kj})$$

2.2 Basic operations

In GGM, there are two basic operations which performed in multiplication, the join operation and the groupby operation.

Join operation. The join operation joins the graph edges which connect graph nodes that are two hops away in graph, in order to create a new one. During the join operation, a UDF operator called "con" (Figure 1, left) is applied. This operator determines the label of the new edge that will be produced taking into account the labels of the starting edges involved in the join operation. Any edge produced during the join operation is called join result.

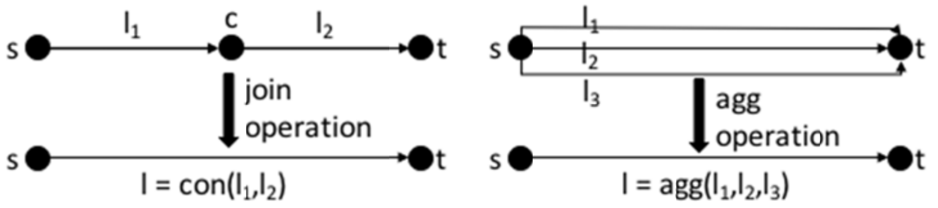


Fig. 7. Example of join operation where the edges (s,c,l_1) and (c,t,l_2) produce the new edge (s,t,con(l_1,l_2)) and agg operation where the edges {(s,t,l_1), (s,t,l_2), (s,t,l_3)} produce the new edge (s,t,agg(l_1,l_2,l_3))

Groupby operation. The groupby operation takes as input edges with the same ends (s,t) produced by the join operation and groups them into a single edge. During grouping, the UDF operator called "agg" is applied and determines the value of the grouped edge based on the labels of the edges involved in the groupby operation (Figure 1, right). Any groupby result is referred as groupby or agg result.

Figure 2 shows an example of a graph where the join and groupby operations are applied.

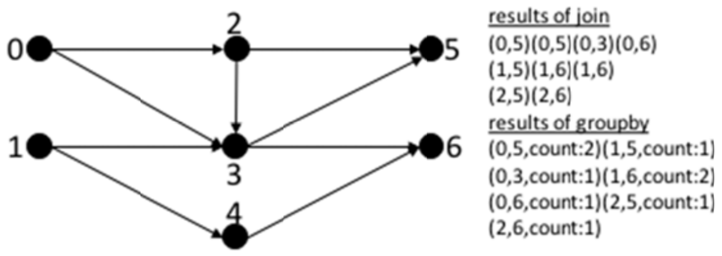


Fig. 8. Example graph and results of join and groupby operations; the groupby operation uses count as agg UDF

2.3 GGM problems

Below are presented some realistic problems that are expressed as GGMs.

$\mathrm{Friend-of-Friend}: \mathrm{G} \circ \mathrm{G}, \mathrm{where}\ \{*, +\} \to \{\mathrm{nil}, \mathrm{count}\}$. The purpose of the problem is given as a node k, the identification of "friend of friend", that is, the set of nodes that are 2 hops in the graph. In the above definition, changing the operator "+" to the operator "nil" implies that no UDF operator is applied during the con operation.

$\mathrm{Common\ neighbors}: \mathrm{G} \circ \mathrm{G}^{\mathrm{T}}, \mathrm{where}\ \{*, +\} \to \{\mathrm{nil}, \mathrm{count}\}$. The above expression counts the common neighbors for each pair of nodes in the graph.

$\mathrm{Trusts\ in\ 3\ hops}: \mathrm{G} \circ \mathrm{G} \circ \mathrm{G}, \mathrm{where}\ \{*, +\} \to \{\mathrm{nil}, \mathrm{count}\}$. With the above expression, we calculate the number of nodes that each node trusts in the graph within three hops away.

3. EXECUTION PLAN

In this chapter, we will describe the input data format and then we will analyze an execution plan that supports the execution of one or more consecutive GGMs as described in Algorithm 1. The basic stages of the execution plan are four: 1) load the next source partition, 2) stream all target partitions, 3) estimate the results for each source-target partition pair and 4) store the new results. The above procedure is repeated until all source partitions are processed. Next, we will discuss the main advantages of this execution plan.

```
Algorithm 1: GGM execution plan

For each source partition s
    Load partition s at memory
    Create inverted-index on partition s’ data
    For each target partition t
        Load asynchronously partition t at memory
        For each node $t_i$ in target partition t
        Execute join and groupby operation
        Store results at disk
```

3.1 Graph Preprocessing

First of all, let’s consider as input a directed graph that is represented as an adjacency list with the following form:

$$[N_0|edge_{count_0}|(e_0, v_0), (e_1, v_1), \dots (e_k, v_k)],$$
$$[N_1|edge_{count_1}|(e_0, v_0), (e_1, v_1), \dots (e_m, v_m)], \dots$$

where N_i: a graph node, $edge_{count_i}$ the out degree of node N_i , e_k, v_k: the edge’s target node, followed by the edge’s label. The label of an edge can have arbitrary size, but is constant for all edges in the graph during an execution. Then, we store the graph twice in the disk, one where the edges are sorted at the source node (s field), and another one where the edges are sorted at the target node (t field).

3.2 Run Execution Plan

Once the initial graph is sorted, it begins the in-memory graph loading process. First of all, there are two buffers; the first one is used to load segments of the graph sorted at edges’ source nodes, these segments are called source partitions. The second buffer, is used to load segments of the graph sorted at edges’ target nodes respectively. The latest segments are called target partitions.

Load source partition. The buffer that handles source partitions works as follows: Uses a small-sized buffer to read edges from the file asynchronously.

Fetching edges is interrupted when either of the following conditions ceases to apply. a) Total PageSize data have been loaded, and therefore there is no space available in the buffer. b) Data corresponding to k nodes have been loaded. Moreover, the maximum number of distinct nodes that can participate in a source partition should be determined before execution.

As a source partition is loaded, an inverted index is built at the target field of the edge. Figure 3 shows the index creation process. The index is a N-sized array (since the other edge end, can be any node of the graph) and each cell i indicates the position in the source partition buffer where the edges of node with id i begin. In order to create the inverted index, the edges of the current source partition must be scanned twice. During first scan, we count for each node in the target field how many edges in the source field are expected to come across. The second time, the edges are placed in the appropriate position in the source partition buffer, based on the counts calculated at the 1st scan. At this point, an index is created in the target field of edges for the current source partition, which will be used then in the join operation.

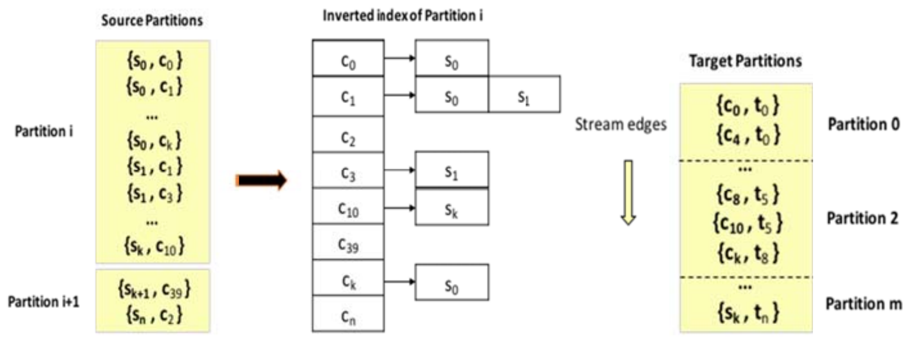


Fig. 9. Inverted index creation at source partition (left); target partition streaming (right)

Load target partition. Loading target partitions is much simpler. In particular, edges are loaded asynchronously as much as the target partition buffer's capacity and then the buffer is ready for use.

At this point, it is worth noting that when loading edges in either source or target partition, any node's edges must be included in the same partition. For that reason, any source partition must have size at least equal with the maximum out-degree of the graph and any target partition must have size at least equal with the maximum in-degree of the graph respectively.

Compute results. As the nodes t of a target partition are loaded, they are assigned to groups to be processed by the available threads. Each thread calculates the results produced by the node that executes the join and after the groupby operation. The produced results of each node t are stored locally in an array called groupby array.

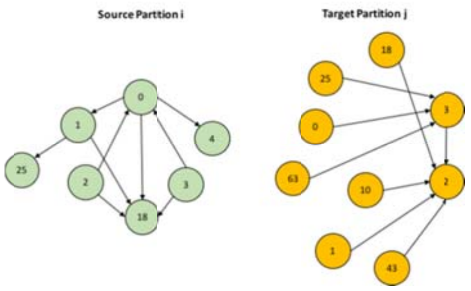


Fig. 10. Example of source and target partitions on which will be applied the execution plan

At this point, we will outline in a small example the procedure, in order to estimate the final results. First, suppose that we are processing the edges of the source and target partitions as shown at Figure 4. The inverted index of source partition i that it will be created appears at Figure 5; the target nodes with id 2 and 3 respectively are also assigned to a different executor. Then each node will execute the join operation.

Join Operation. The edges of the executing node are joined with those of the current source partition using the inverted index. While joining the edges, the operator "con" is applied to the edge values. The produced edge is stored locally at the thread. Figure 5 shows the results of the join operation that each executor produces locally.

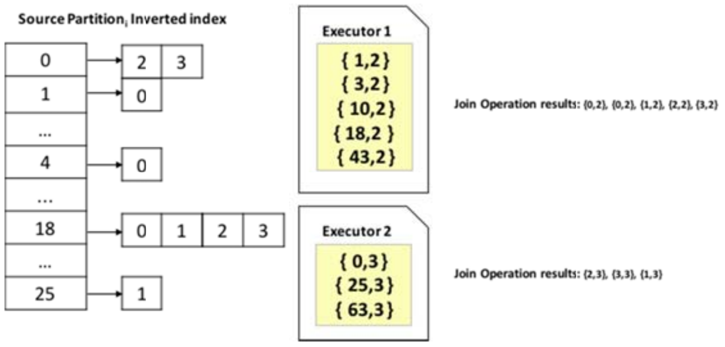

Source Partition, Inverted index
0
1
...
4
...
18
...
25
2 3
0
0
0 1 2 3
1
Executor 1
{ 1,2}
{ 3,2}
{ 10,2}
{ 18,2 }
{ 43,2}
Executor 2
{ 0,3}
{ 25,3}
{ 63,3}
Join Operation results: {0,2}, {0,2}, {1,2}, {2,2}, {3,2}
Join Operation results: {2,3}, {3,3}, {1,3}

Fig. 11. Estimating join results locally at executors

Groupby Operation. It is crucial for the efficient execution of operations that groupby operation to be executed rapidly. Groupby operation, needs two K-sized arrays, where K is the maximum number of nodes that we process in a single source partition. The 1st array is used as an index, while the 2nd (called groupby array) is used to store the new results.

For example, let node t produce a new edge (s,v), where s is the edge node destination and v the value of the new edge. Firstly, we access the cell with index s at the index array and the latest returns the location of the edge data (s, v) at the groupby array. With this technique, each join result with just two random accesses, one in each array performs the groupby operation. This technique is much faster than using other structures such as hashmaps, etc., especially when the set of data to be aggregated is quite large. In addition, all produced results of a specific node are placed side by side in the groupby array, which helps to efficiently transfer them latter from main-memory to disk. Figure 6 illustrates the structures used by the executors for the groupby operation, as well as the status of groupby arrays that have stored the final results for target nodes 2 and 3. Groupby arrays results are now ready for transport to the disk.

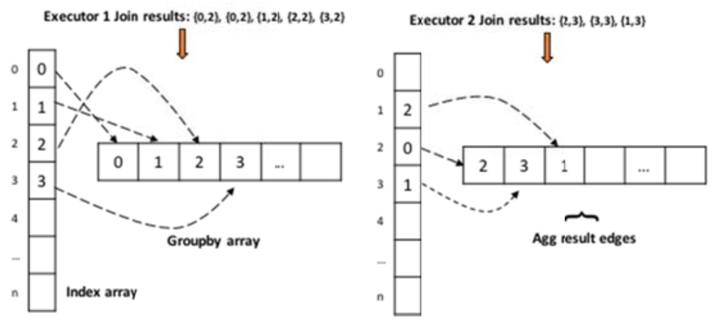

Executor 1 Join results: (0,2), (0,2), (1,2), (2,2), (3,2)
Executor 2 Join results: (2,3), (3,3), (1,3)
Groupby array
Index array
Agg result edges

Fig. 12. Executing groupby locally at executors

At the moment, an edge is produced by the join operation, it is grouped with the existing edge with the same ends (if already existed) by applying the operator "agg" and thereby composing the new value of edge with ends (si,tj). Otherwise, it is placed in a new cell in the groupby array.

3.3 Execution plan advantages

This execution plan is suitable for performing out-of-core GGMs for three main reasons.

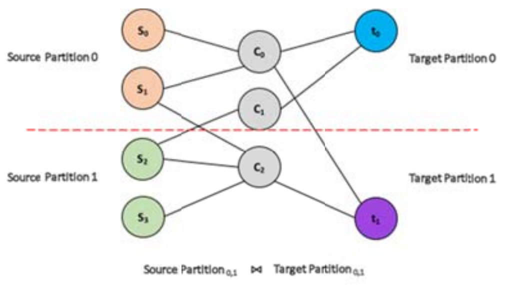


Fig. 13. Divide the initial graph at two source and target partitions

Produced results are final. As we have already mentioned, the source partitions are sorted into the source edge field, while the target partitions are sorted into the target edge field, respectively. Therefore, when joining a source partition with a target partition, the final results that will result for an edge (s,t) will eventually be the final, since it is impossible for any next partitions join, to produce some new edge (s,t), since the initial partitions are sorted.

Execution plan supports natively multiple iterations. The generated data of each output partition, which are produced during the GGM have the following two properties. a) All edges are grouped on target field, b) all source fields of the results edges are a subset of all nodes s, originated from the initial source partition.

With respect to the first property, a thread processes all the edges of target node, so it is possible to write all edges having a common end t in adjacent positions in the output partition. This is important, because in each subsequent iteration after the first one, since the edges are already grouped in the target field, the inverted index at the source partition is essentially ready, and only one pass is needed to map the target nodes in the index array.

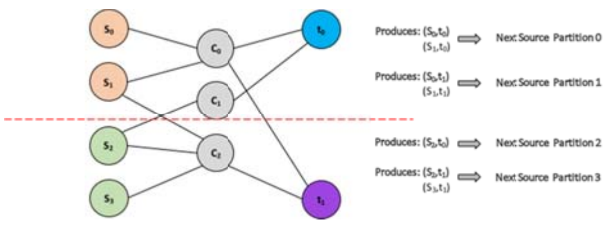


Fig. 14. The estimated results are the source partitions in the next iteration

The division of the initial problem into sub-problems may improve performance. In graphs with a large number of nodes, we have noticed that there are many cache misses in the CPU during the groupby operation. This is because when the number of nodes is large enough, random accessing in both to index and the groupby array accesses a wide range of memory addresses that are not cached and eventually cache misses are created. For that reason, processing smaller source partitions from the total number of nodes and thus limiting the range of arrays during the groupby operation generally improves the performance of the cache. Of course, splitting into smaller source partitions means increasing the passes in the graph to estimate all the results.

Identifying the point where the improvement in the performance of the groupby operation in relation to the overheads added by the multiple passes over the graph ultimately achieves better performance when performing one or more GGMs is a complex process and is an interesting research direction.

To reinforce this argument, we cite the result of an experiment below.

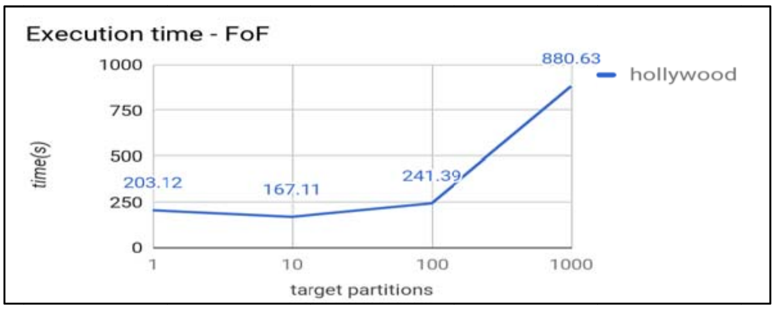


Fig. 15. Chart of the effect at runtime of dividing the problem into smaller

In this experiment, we ran the friend-of-friend problem and compared the total run time of one graph multiplication by dividing the original problem into 1,10,100 and 1,000 sub-problems, practically limiting the maximum number of nodes participating in a source partition. In addition, operations related to disk reads are simulated by in memory calls to show the "net" gain from the groupby operation. As can be seen in Figure 9, for a small number of passes in the graph, better performance is achieved. However, when the number of passes begins to rise considerably, then the extra cost of the overheads added by the additional graph passes is far greater than the profit due to fewer caches misses in the groupby operation.

4. SYSTEM EVALUATION

In this chapter, we will experimentally analyze the performance of Graphity. More specifically, we compare the proposed execution plan that implements Graphity, compared to other TLAV model systems and a relational database in the friend-of-friend problem. The following table (Table 1) shows the graphs characteristics used in the experiments. as the number of join and aggregate results produced at 2hops.

Table 3. Graph Datasets

Graph	Nodes	Edges	Join/Agg results count 2 hops
amazon	$7 * 10^5$	$5 * 10^6$	$4 * 10^7 / 2 * 10^7$
roadNet-CA	$1.9 * 10^6$	$5 * 10^6$	$10^7/10^7$
in	$1.4 * 10^6$	10^7	$10^9 / 2 * 10^8$
frwiki	$1.3 * 10^6$	$3 * 10^7$	$5 * 10^9 / 3 * 10^9$
hollywood	$2 * 10^6$	$2 * 10^8$	$2 * 10^{11} / 4 * 10^7$

The experiments conducted on a machine that has the following specifications:

- Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-62-generic)
- 2 x 10 cores - Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
- 144 GB RAM - 2133MHZ, DDR4
- 2TB - Hard Disk - RAID 1

4.1 Comparing to TLAV systems

Initially, we decided to compare the proposed execution plan for GGMs with the TLAV model to the friend-of-friend problem, described in Section 2.3. We chose the open source systems Graphchi [9] and X-Stream [10], which are two good representatives that execute TLAV model problems, while they run on a single-node like Graphity and support out-of-core processing. Graphchi uses a traditional vertex centric approach and uses an innovative technique to reduce random access to disk called "Parallel Sliding Windows". X-Stream instead uses an edge-centric approach, which is implemented using streams called "Streaming Partitions" and can scale efficiently.

Both in Graphchi and X-Stream we used the provided application programming interface (API) to implement the friend-of-friend problem. Regarding the modeling of the problem, we considered that there is a defined set of active nodes, for which we count the number of friends of their friends. Once the computations for this set of nodes are over, then the next set of nodes is scheduled and this process is repeated until the results of all nodes are estimated. In fact, the whole computation is performed in multiple batch processing groups.

In order to execute the computation, each node of the graph must reserve memory proportional to the number of the nodes b participating in the computation. Therefore, if we wanted to calculate all the results in a single batch, then each node would store information for all other nodes and eventually the total required memory would be $O(N^2)$. Since this memory size is too large, computations must be performed on smaller groups of nodes in order fit to the available memory.

4.2 Comparing to RDBMS systems

In this case, we will calculate the same problem compared to a relational database. Generally, relational databases are general purpose and therefore do not have optimized data structures for graph operations. However, the fact that we execute join and group by operations on independent sets (set operations) is a classic relation model use-case. We chose SQLite[1] as the representative, which is widely used and has competitive execution times compared to other open source databases. The query we executed in SQLite is the following:

```
insert into results
select g1.edge_from, count(distinct g2.edge_to)
from graph g1, graph g2
where g1.edge_to = g2.edge_from
group by g1.edge_from;
```

The experiments were conducted on a Docker virtual machine with total 8GB of RAM, 8 physical cores on the same processor and Ubuntu 16.04 LTS operating system. We have run all the experiments within the virtual machine so that the operating system behaves like it has as available memory, only the portion of it

[1] SQLite, https://www.sqlite.org

that we have defined, so that it manages the system pages correctly, and also determine which specific cores will be used.

The following diagram (Figure 10) shows the execution of the friend-of-friend problem. Graphity is far faster than other systems, which is mainly due to the suitability of its execution plan, as well as to the efficient implementation of the two basic operations of the join and groupby. As far as X-Stream and Graphchi are concerned, they do not perform well, because they process only a few thousand nodes in a batch and thus, they have to perform too many iterations to estimate the results of all the nodes. In addition, each individual batch, doesn't have any great workload particularly, and these systems cannot scale efficiently in most cases.

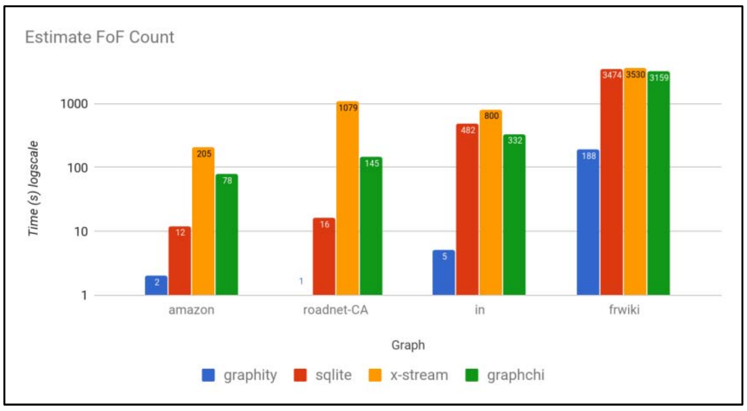


Fig. 16. Execution time fof problem.

5. RELATED WORK

The Think-like-a-vertex model is the most widely used model for large scale graph processing and is used to solve known graph problems. Pregel [1] is a procedural expression of the Think-like-a-vertex model and introduces an application programming interface (API) for application development through a messaging mechanism. Pregel as a computational model has been applied to different architectures such as distributed, in-memory systems, GPUs, and so on. A typical example of using the Pregel model is Pegasus [2], which is a graph processing engine, implemented in the framework for processing large-scale MapReduce data [3]. Pegasus follows the principles of the MapReduce model and expresses a set of graph algorithms as a generalized form of sparse matrix-vector multiplication. This representation is ideal for solving problems like PageRank [4], propagation [5] (label propagation), but has a very low performance in graph traversal algorithms and especially in performing multiple multiplications of different vectors with a matrix.

In our case, we are looking at systems that support out-of-core graph processing. In the category of such systems are disk-based systems such as

Graphchi [9], X-Stream [10] and TurboGraph [11]. The above systems apply techniques that focus on reducing random access to the disk, so that they can support very large graphs on a node.

Systems that follow the TLAV model, at mathematical level, practically execute a matrix-vector multiplication with generalized operators. In our case, the GGM problems we focus on are a matrix-matrix multiplication as mathematical representation. Related research that studies matrix multiplication problems is the friend-of-friend [7] that runs Facebook in a distributed way using the Giraph system [13], but supporting only one multiplication. Yet another effort is to find the shortest distances for multiple nodes (Multiple-source Shortest Distance) implemented in the Galois in-memory system [8].

6. CONCLUSIONS

In work, we study the problem of Generalized Graph Multiplications (GGMs). More specifically, we describe an execution plan that allows the out-of-core execution of subsequent GGMs and analyze the design decisions and choices required to efficiently implement it.

We then describe Graphity, a prototype system that implements that execution plan. Then, we performed a set of experiments to evaluate our design choices at the friend-of-friend problem. The results of the latest experiments showed the competitive execution times achieved by Graphity in an environment with limited memory compared to other approaches that shows the superiority of our approach in GGM problems.

7. REFERENCES

[12] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, Pregel: a system for large-scale graph processing, *International Conference on Management of data (SIGMOD), ACM*, New York, NY, USA, 2010, pp. 135-146.

[13] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C, PEGASUS: A peta-scale graph mining system implementation and observations, *IEEE International Conference on Data Mining*, 2009.

[14] DEAN, J., AND GHEMAWAT, S, MapReduce: Simplified data processing on large clusters, *Conference on Symposium on Operating Systems Design & Implementation*, vol. 6, 2004.

[15] BRIN, S., AND PAGE, L., The anatomy of a large-scale hypertextual web search engine, *International Conference on World Wide Web 7*, 1998.

[16] ZHU, X., AND GHAHRAMANI, Z., Learning from labeled and unlabeled data with label propagation, Tech. rep., Carnegie Mellon University, 2002.

[17] Aidan B. G. Chalk, Pedro Gonnet, Matthieu Schaller, Using Task-Based Parallelism Directly on the GPU for Automated Asynchronous Data Transfer, *PARCO*, 2015, pp. 683-696.

[18] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, One trillion edges: graph processing at Facebook-scale, *VLDB Endow. 8,12,* Aug. 2015, pp. 1804-1815.

[19] D. Nguyen, A. Lenharth, and K. Pingali, A lightweight infrastructure for graph analytics, *ACM Symposium on Operating Systems Principles*, SOSP, 2013, pp. 456-471.

[20] Aapo Kyrola, Guy E. Blelloch, Carlos Guestrin, GraphChi, Large-Scale Graph Computation on Just a PC, *USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2012, pp. 31-46.

[21] Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel, X-Stream: edge-centric graph processing using streaming partitions, *ACM Symposium on Operating Systems Principles*, SOSP, 2013, pp. 472-488.

[22] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, Hwanjo Yu, TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC, ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD, 2013, pp. 77-85.

[23] A. E. Feldmann, Fast Balanced Partitioning is Hard, Even on Grids and Trees, Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science, Slovakia, 2010, pp. 372-382.

[24] Apache Giraph, http://giraph.apache.org/

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών,

Πανεπιστημιούπολη, 15784 Αθήνα

ISSN 1792-8826