Παράλληλα και Διανεμημένα Συστήματα

Εργασία 1

1) Εισαγωγικά

Σε αυτήν την εργασία κληθήκαμε να υπολογίσουμε τα τρίγωνα που σχηματίζει ένας undirected unweighted γράφος. Για την εύρεση αυτών των τριγώνων, χρειάζεται να βρούμε τα nodes τα οποία σχηματίζουν 3 edges. Καταλαβαίνοντας το πρόβλημα λίγο πιο βαθιά με τη βοήθεια του octave, βγάζουμε το συμπέρασμα ότι στην πιο ακραία περίπτωση, που είναι η εκάστοτε κορυφή μπορεί να συμμετέχει σε max $\sum_{c3=0}^{n-2} c3$, όπου n : το μήκος του πίνακα, ο οποίος στην περίπτωσή μας είναι nxn επειδή ο γράφος μας είναι undirected unweighted. Το παραπάνω ισχύει μόνο εάν δε λάβουμε υπόψιν μας την πιθανότητα του θεωρήματος Erdős-Rényi. Στα πρώτα δύο versions του κώδικά μας, κάναμε την παραγωγή των πινάκων μας με μία rand, αφού αλλάζαμε φυσικά το seed για να μην μας τους ίδιους "τυχαίους" πίνακες σε κάθε δοκιμή και φυσικά κάναμε τον πίνακα συμμετρικό. Να σημειώσουμε ότι οι αρχικοί μας κώδικες ήταν πολύ basic, χωρίς συναρτήσεις και χωρίς structs. Στο κοντινό μέλλον στο github θέλουμε να φτιάξουμε πιο σωστά τους κώδικες στη λογική των V3_seq, V4_seq και V4_pthread με struct.

Όσον αφορά την παραλληλοποίηση, είναι προφανές πως βελτιώσεις στους χρόνους είδαμε σε μεγάλα δεδομένα, αφού σε μικρά δεδομένα έχω για λόγους overhead χειρότερα αποτελέσματα. Παρατηρήσαμε επίσης, ότι όσο πιο αραιός είναι ο πίνακας αναλογικά με τις διαστάσεις του και με τις μονάδες, τόσο πιο πολύ χρόνο παίρνει η εκτέλεση παράλληλα. Αντίθετα, όσο πιο dense είναι ένας πίνακας, τόσο πιο έντονα φαινόταν η διαφορά μεταξύ sequential και παράλληλων code. Εξάλλου ισχύει ότι :

$$\text{Speedup}(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

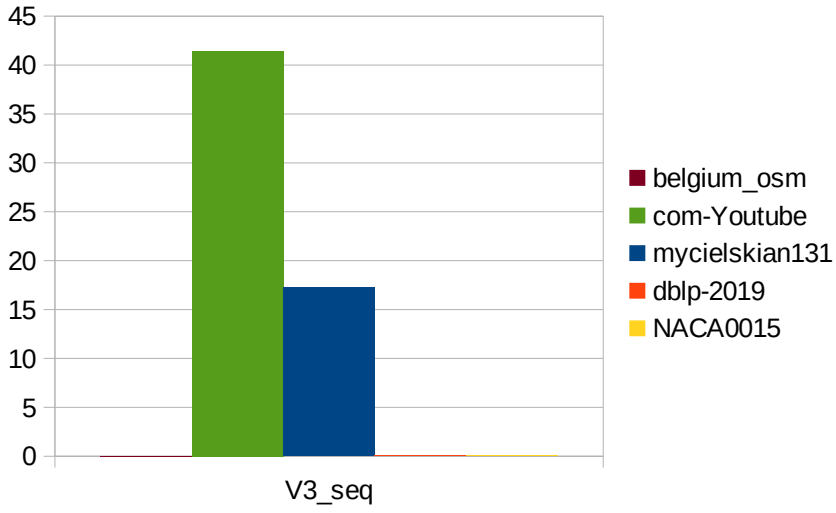
Serial part of job = 1 (100%) - Parallel part

Parallel part is divided up by N workers

2) Version 3

Σε αυτό το version έπρεπε να υπολογίσουμε τα τρίγωνα στους γράφους εκεμεταλλευόμενοι την μορφή CSC. Η πρώτη μας σκέψη ήταν ότι ένα C αρχείο δεν θα μπορούσε να διαβάσει ένα .mtx αρχείο, οπότε στην αρχή υλοποιήσαμε ένα πρόγραμμα, το οποίο θα μετέτρεπε απλά ένα .mtx αρχείο σε .txt . Αφού αντιληφθήκαμε ότι κάτι τέτοιο ήταν αχρείαστο, επικεντρωθήκαμε στο να αλλάξουμε λίγο την συνάρτηση coo2csc για να μπορούμε να μετατρέπουμε τους πίνακες του Matrix Market σε CSC. Η μορφή COO μας δείχνει πρακτικά τις συντεταγμένες των μονάδων που υπάρχουν στον πίνακα, δηλαδή τα nodes. Από την άλλη η μορφή CSC αποδείχθηκε αρκετά χρήσιμη, αφού η μία γραμμή μας δείχνει ανά διάδες ποιο column βλέπω κάθε φορά, αλλά και η διαφορά της κάθε δυάδας μου δείχνει το πόσες μονάδες έχω σε αυτό το column(είναι κάτι σαν dense vector). Από την άλλη, η άλλη γραμμή, χωρισμένη ανά την εκάστοτε διαφορά της δυάδας του csc_col, μας δείχνει τα rows που βρίσκονται οι μονάδες στον πίνακα. Έτσι έχουμε μια πλήρη αναπαράσταση του γράφου μας, προσπαθώντας να βρούμε τα τρίγωνα. Για να ακολουθήσουμε κορυφές, υλοποιήσαμε την εξής λογική : Όπως περιγράψαμε και παραπάνω βρίσκω τον κάθε γείτονα της αρχικής μου κορυφής και χωρίζοντας το csc_row σε μήκη κάθε φορά ίσα με τη διαφορά της κάθε δυάδας. Θα πηγαίνω στον γείτονα λοιπόν της κάθε κορυφής και θα κοιτάω τους δικούς του γείτονες. Συγκρίνοντας έτσι τους γείτονες της πρώτης και της δεύτερης κορυφής, αν βρίσκω κοινό γείτονα, υπάρχει τρίγωνο.

i) Sequential Implementation

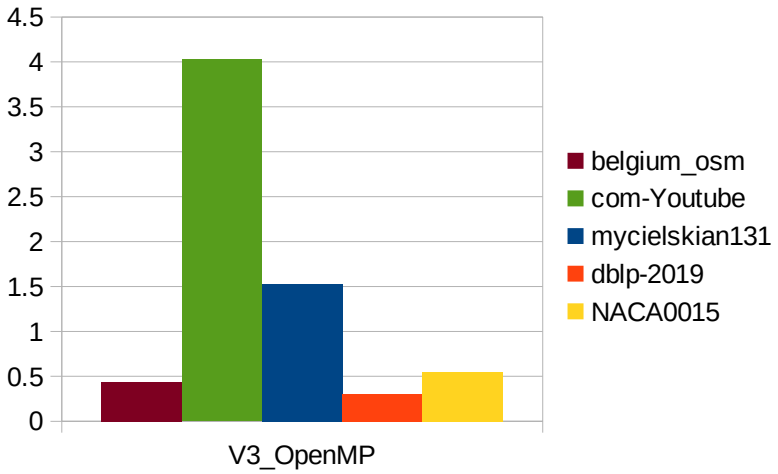


Η σειριακή υλοποίηση του κώδικα V3 μας βγήκε αρκετά γρήγορη, μάλιστα πιο γρήγορη απ'όσο περιμέναμε. Παρακάτω φαίνονται και τα διαγράμματα με τους χρόνους για τους γράφους :

V3_seq (seconds)	
belgium_osm	0.0110600
com-Youtube	41.4204540
mycielskian131	17.2697510
dblp-2019	0.0523550
NACA0015	0.0994210

ii) OpenMP Implementation

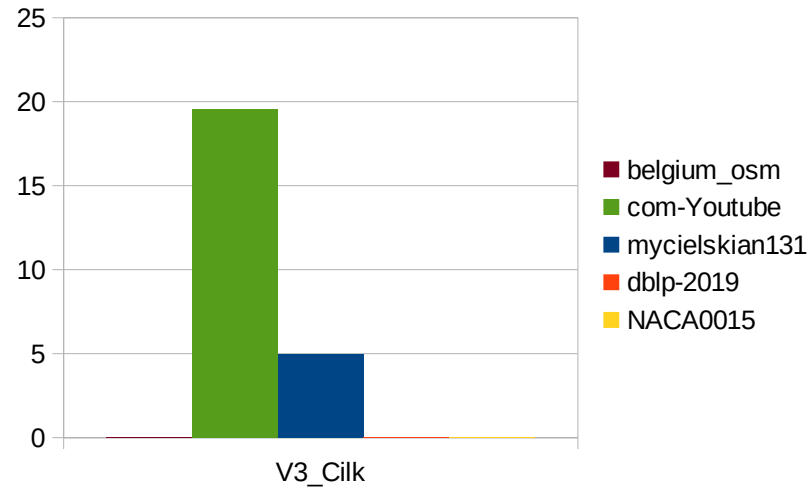
Η υλοποίηση της OpenMP ήθελε λίγο ψάξιμο αναφορικά με το πώς θα τρέξει παράλληλα ο κώδικας. Αφού πειραματιστήκαμε με τις διάφορες λογικές dynamic,static,guided scheduling, καταλήξαμε πως για dynamic scheduling παίρναμε τα καλύτερα αποτελέσματα. Αυτό που μας φάνηκε περίεργο είναι ότι χωρίς το optimization flag -O3 στον gcc δεν παίρναμε καλά αποτελέσματα γενικά για την openmp. Παρακάτω φαίνονται τα αποτελέσματα για τις δοκιμές που κάναμε για 24 threads. Παρατηρούμε ότι στους γράφους που το sequential είχε κακό χρόνο, πήρε πολύ λιγότερο για την υλοποίηση με OpenMP.



V3_OMP(seconds)	
belgium_osm	0.4333380
com-Youtube	4.0267800
mycielskian131	1.5265240
dblp-2019	0.3040090
NACA0015	0.5513820

iii) Cilk Implementation

Η υλοποίηση της cilk μας φάνηκε αρκετά απλή. Θα μπορούσαμε ίσως να βελτιώσουμε τον κώδικα σε κάποια σημεία κυρίως αναφορικά με τον αριθμό των threads. Παρακάτω βλέπουμε τα αποτελέσματα για τις δοκιμές που κάναμε.



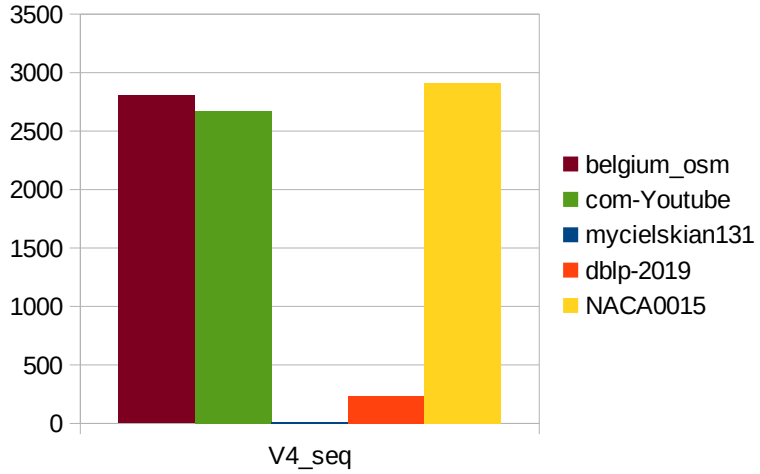
V3_Cilk(seconds)	
belgium_osm	0.0021080
com-Youtube	19.5541170
mycielskian131	4.9673900
dblp-2019	0.0195690
NACA0015	0.0198830

3) Version 4

Σε αυτό το version δυσκολευτήκαμε πολύ, από την άποψη ότι μας φάνηκε πιο αργή απ'ότι η V3, πράγμα που μας έκανε να πιστεύουμε ότι έχουμε κάνει κάπου λάθος. Αφού ελέγξαμε τον κώδικα και στο elearning, δοκιμάσαμε την υλοποίησή μας. Η υλοποίησή μας βασίστηκε στο V3, αλλά και στην εκφώνηση. Κάποιος αφελής, θα μπορούσε στην αρχή να ορίσει έναν 2x2 πίνακα με 1M στοιχεία σε κάθε στήλη , δηλαδή να θέλει να χρησιμοποίησει ~4 TB μνήμης για να υλοποιήσει το ζητούμενο. Αυτό ίσως ήταν και το πιο δύσκολο σημείο του V4, το πώς θα κάνω τις πράξεις των γινομένων και του Hadamard, χωρίς να τις κάνω στην πραγματικότητα, αλλά να εκμεταλλευτώ την μορφή CSC. Έτσι, λοιπόν χρισιμοποίησαμε μια λογική μεταβλητή πρώτα και μετά υλοποιήσαμε την πράξη του γινομένου πάνω στον csc, αλλά για λόγους ευκολίας επιλέξαμε να μην κρατάμε κάποιον αθροιστή με τον οποίο να καταλήγαμε στο ζητούμενο.

i) Sequential Implementation

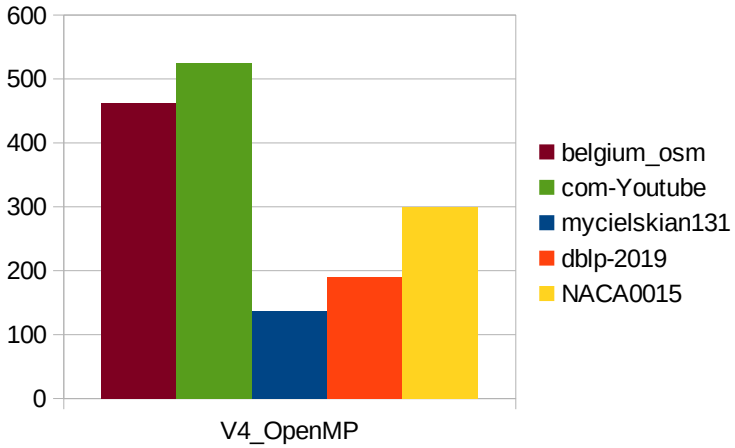
Η σειριακή υλοποίηση του κώδικα V4 μας φάνηκε πολύ λιγότερο αποδοτική από αυτήν του V3, είτε σε αρκετά αραιούς πίνακες είτε σε πιο πυκνούς (αραιούς) πίνακες.



V4_seq (seconds)	
belgium_osm	2,804.3032370
com-Youtube	2,673.2005760
mycielskian131	14.7877980
dblp-2019	230.8129610
NACA0015	2,913.6474150

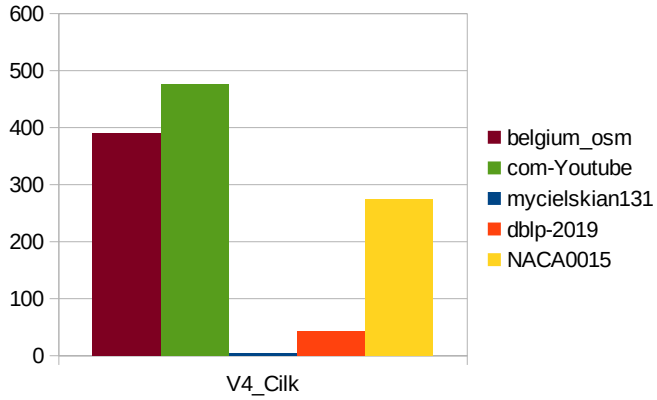
ii) OpenMP Implementation

Στηριχτήκαμε στο V3 και υλοιποιήσαμε μια παρόμοια λογική. Παρακάτω φαίνονται τα αποτελέσματα με 24 threads. Παρατηρούμε ότι ο χρόνος μειώθηκε στους περισσότερους graphs εκτός από τον mycielskian131.



V4_OMP(seconds)	
belgium_osm	380.7548630
com-Youtube	427.2751310
mycielskian131	137.6542360
dblp-2019	190.4215360
NACA0015	300.4587620

iii) Cilk Implementation

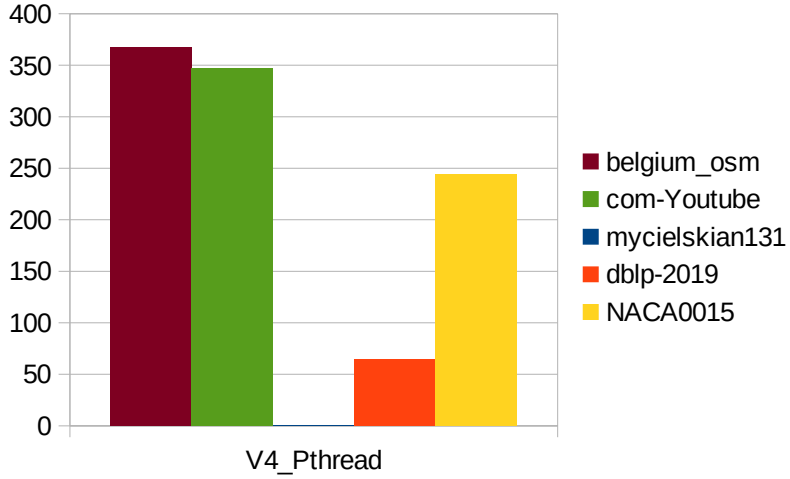


Πάλι στηριχτήκαμε στο V3. Παρακάτω φαίνονται τα αποτελέσματα. Και εδώ παρατηρούμε πως η Cilk έκανε σε όλα καλύτερους χρόνους από το sequential. Παρ'όλαυτα σε κάποια tests το OpenMP τα πήγε καλύτερα. Η cilk τα πήγε πολύ καλά στους πιο dense αραιούς πίνακες όπως ο mycielskian131 και ο dblp-2019.

V4_Cilk(seconds)	
belgium_osm	389.4512800
com-Youtube	476.325416
mycielskian131	3.7868260
dblp-2019	42.2679500
NACA0015	274.2563410

iv) Pthreads Implementation

Ίσως το πιο ενδιαφέρον κομμάτι της εργασίας. Σε αυτήν την υλοποίηση, η αλήθεια είναι ότι θα μπορούσαμε να εμείνουμε λίγο παραπάνω σε κάποια πράγματα, όπως για παράδειγμα την δημιουργία struct και μιας πιο σωστής λογικής του κώδικα (χρήση λιγότερων global μεταβλητών). Επίσης επιλέξαμε αντί για mutex να χρησιμοποιήσουμε μια άλλη λογική πιο Hardware-Based και να κάνουμε join στο τέλος. Όπως, λοιπόν θα μπορούσαμε να κόβουμε κάθε κομμάτι του πίνακα σε Buffer_Size για να μπορέσουμε να προγραμματίσουμε ένα FPGA με μια τέτοια λογική (δηλαδή να μεταφέρει κομμάτι κομμάτι τον πίνακα στην global memory ενός τέτοιου συστήματος χρησιμοποιώντας AXI ή PCI-e), έτσι και εδώ σκεφτήκαμε να κόψουμε τον πίνακα σε increments ανάλογα με το πόσα threads δίνονται στην είσοδο. Με τις απαραίτητες if, διασφαλίσαμε ότι δε θα χάνεται κάποια επανάληψη, καθώς και ότι κάθε thread θα κάνει διαφορετικές επαναλήψεις από το άλλο, αλλά παράλληλα. Έτσι και δεν υπάρχει ο κίνδυνος να πάρω λάθος αποτελέσματα. Επιλέξαμε επίσης να χρησιμοποιήσουμε την usleep(), ούτως ώστε να δώσουμε και ένα μικρό προβάδισμα σε κάθε thread που δημιουργείται, αφαιρώντας το στο τέλος για τον σωστό υπολογισμό του χρόνου.



	V4_pthread
belgium_osm	367.7496380
com-Youtube	347.365241
mycielskian131	0.0136120
dblp-2019	64.2354180
NACA0015	244.2096530

Ίσως να μην πήραμε και τα καλύτερα αποτελέσματα, μας εντυπωσίασε όμως η τεράστια διαφορά στο mycielskian131.

4) Conclusion

Σε γενικές γραμμές, η εργασία μας φάνηκε αρκετά απαιτητική, αλλά και αρκετά χρήσιμη για την εξοικείωσή μας με διάφορες έννοιες του παράλληλου προγραμματισμού. Ήθελε αρκετή σκέψη και λογική για να καταλήξουμε σε σωστά συμπεράσματα αναφορικά με τους αλγόριθμους και τα ζητούμενα. Μπορεί η υλοποίηση να μην ήταν η καλύτερη δυνατή, παρ'όλα αυτά καταλάβαμε αρκετά πράγματα. Το πιο ενδιαφέρον και πιο κατανοητό κομμάτι μας φάνηκε το pthreads. Αν και το OpenMP και το Cilk είναι APIs που "μας λύνουν τα χέρια" σε πολλά πράγματα αναφορικά με τον παράλληλο προγραμματισμό, τα Pthreads έκαναν πολύ πιο κατανοητή τη λογική του.

Ηλιάδης-Αποστολίδης Δημοσθένης 8811 https://github.com/iliadisd/PDS
Γεωργόπουλος Γιώργος 9356 https://github.com/ggeorgop99