ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμημα ΗΜΜΥ. Τομεας Ηλεκτρονικης

Παραλληλα και Διανεμημενα Συστηματα

Εργασία 1: vertexwise triangle counting

5 Δεκεμβρίου 2020

Συντάκτης:
Χρηστος Χουτουριδης ΑΕΜ:8997
cchoutou@ece.auth.gr

Διδάσκων:
Πιτσιανης Νικολαος
nikos.pitsianis@eng.auth.gr

1. Εισαγωγη

Μια δεξιότητα ενός μηχανικού λογισμικού, που πάντα αποτελεί πρόκληση, είναι ο παράλληλος προγραμματισμός. Με αυτόν θα ασχοληθούμε στην παρούσα εργασία. Για το σκοπό αυτό θα υλοποιήσουμε ένα πρόγραμμα που θα εντοπίζει τις ακμές που δημιουργούν τρίγωνα σε ένα μη κατευθυντικό γράφο. Και αν αυτό είναι το τι, ο πραγματικός μας σκοπός είναι να καταφέρουμε να παραλληλοποιήσουμε την διαδικασία. Να βρούμε δηλαδή τα κομμάτια που μπορούν να εκτελεστούν ταυτόχρονα και που δεν επηρεάζουν το ένα το άλλο.

Στην παρούσα εργασία υλοποιούμε δύο αλγόριθμους, έναν που αναζητά μέσα στον πίνακα όλους τους δυνατούς συνδυασμούς και έναν που κάνει χρήση λίγης γραμμικής άλγεβρας. Όπως θα δούμε παρακάτω ο δεύτερος μας δίνει καλύτερες δυνατότητες παραλληλοποίησης. Για να ελέγξουμε την ορθότητα και να πάρουμε μετρήσεις τα εκτελέσιμα που παράξαμε τα τρέξαμε στη υπολογιστική συστοιχία του ΑΠΘ.

2. Παραδοτεα

Τα παραδοτέα της εργασίας αποτελούνται από:

- Την παρούσα αναφορά.
- Το σύνδεσμο με το αποθετήριο που περιέχει τον κώδικα για την παραγωγή των εκτελέσιμων, της αναφοράς και τις μετρήσεις από τη συστοιχία του πανεπιστημίου.

3. Υλοποιηση

Πριν ξεκινήσουμε να αναλύουμε τον παραλληλισμό και τις μετρήσεις καλό θα ήταν να αναφερθούμε στην υλοποίηση. Για την παρούσα εργασία χρησιμοποιήσαμε τη γλώσσα C++ και βοηθητικά έγινε χρήση του bash και της matlab. Στην εργασία γίνεται χρήση αραιών πινάκων με το format **CSC**, καθώς βολεύει πολύ για πολλαπλασιασμούς.

3.1. SpMat - SpMatCol - SpMatRow

Πρώτα έπρεπε να δημιουργήσουμε τις κατάλληλες αφαιρέσεις ώστε ο κώδικάς μας να είναι πιο ευανάγνωστος και καλύτερα δομημένος. Για τον πίνακα δημιουργήσαμε το template **SpMat**. Πρόκειται για μια δομή που περιέχει εργαλεία για ανάγνωση-εγγραφή στοιχείων, γραμμών, στηλών κτλ, δίνοντάς μας την ψευδαίσθηση ότι κινούμαστε σε πλήρη πίνακα. Αντίθετα όμως εσωτερικά γίνεται εκτενής χρήση του format CSC. Το template μας δίνει τη δυνατότητα επιλογής βελτιστοποίησης, όταν ο πίνακας είναι συμμετρικός, κάτι που συμβαίνει και στην περίπτωσή μας, ώστε όταν ζητάμε να προσπελάσουμε μια γραμμή, στην ουσία να προσπελνούμε την αντίστοιχη συμμετρική στήλη, καθώς αυτό σε ένα πίνακα CSC είναι βελτιστοποιημένο.

Οι δομές **SpMatCol** και **SpMatRow** αποτελούν κάτι που στην C++ ονομάζουμε views[1] και αποτελούν non-owning objects που δείχνουν στον πίνακα SpMat. Επίσης ταυτόχρονα αποτελούν ένα είδος iterator που μας επιτρέπει να συμπεριφερόμαστε στη στήλη(ή στη γραμμή) σαν να ήταν range. Αντί όμως να χρησιμοποιούμε για δείκτες pointers, χρησιμοποιούμε ακεραίους, καθώς έτσι μπορούμε να αναφερόμαστε στην αντίστοιχη θέση ενός άλλου πίνακα πολύ εύκολα. Έτσι μπορούμε για παράδειγμα να γράψουμε:

```
for (int i=0 ; i<A.size() ; ++i)
  for (auto j = A.getRow(i); j.index() != j.end() ; ++j)
    C(i, j.index()) = A.getRow(i)*A.getCol(j.index());
```

και να πάρουμε στον **SpMat C** το $A \odot (A \cdot A)$ και μάλιστα αφού ο A είναι συμμετρικός ο πολλαπλασιασμός να γίνει με CSC στήλες μόνο. Στην υλοποίησή μας εδώ κάνουμε μια ακόμα βελτιστοποίηση .

[1] όπως πχ το: string view

4. Βελτιστοποιηση για το συνολικο αριθμο τριγωνων

Η υλοποίηση του 2ου αλγόριθμου(V4) βασίζεται στις ιδιότητες των συμμετρικών πινάκων και δημιουργεί ένα διάνυσμα $\vec{c_3}$ του οποίου το κάθε στοιχείο είναι ίσο με τον αριθμό των τριγώνων στους οποίους συμμετέχει ο κάθε κόμβος. Το διάνυσμα δίνεται από την σχέση $\vec{c_3} = \frac{1}{2} A \odot (A \cdot A) \cdot \vec{1}$. Γνωρίζοντας όμως ότι ο πίνακας γειτονίασης A είναι συμμετρικός, κάποιος μπαίνει στη σκέψη ότι όλη η πληροφορία για τον αριθμό τριγώνων θα μπορούσε να βρίσκεται στο μισό του πίνακα. Πράγματι, όπως αποδεικνύεται και στην παράγραφο Α΄.1, για τον κάτω τριγωνικό ενός συμμετρικού γράφου $L = (l_{ij}) \neq 0 \quad \forall i > j$ έχουμε $\vec{c_1} = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} l_{ij} \cdot l_{ki} \cdot l_{kj}$, όπου το $\vec{c_1}$ είναι ένα διάνυσμα που σε κάθε θέση έχει τον αριθμό των *μή διατεταγμένων τριγώνων* του κάθε κόμβου.

Αυτό μας δίνει την δυνατότητα να παρακάμψουμε τη δημιουργία του συμμετρικού πίνακα και να δουλέψουμε με τον κάτω τριγωνικό, όπως ακριβώς μας έρχεται από το MatrixMarket format. Αν και αυτό μπορεί να εφαρμοστεί στον αλγόριθμο της έκδοσης V3, αυτό δεν μπορεί να γίνει στην έκδοση V4, στην οποία πρέπει να κάνουμε ολόκληρο τον πολλαπλασιασμό για το διάνυσμα $\vec{c_3}$. Αν όμως επιθυμούσαμε μόνο το συνολικό αριθμό τριγώνων τότε θα μπορούσαμε και εδώ να κάνουμε χρήση της βελτιστοποίησης. Τα αποτελέσματα αυτά είναι σημειωμένα στον πίνακα 5 ως *sum*.

5. Παραλληλισμος

Η στρατηγική μας για τον παραλληλισμό, τόσο για την έκδοση V3, όσο και για την V4, αρχικά ήταν να χωρίσουμε όλες τις πράξεις που μπορούν να γίνουν ταυτόχρονα. Αυτό οδηγεί σε πολύ μεγάλο αριθμό πιθανών ταυτόχρονων νημάτων τα οποία:

- Στην περίπτωση του V3: Για κάθε νήμα, αφού έχουμε σε O(1) τις δύο πρώτες ακμές του κάθε τριγώνου, κάνουμε αναζήτηση την τρίτη ακμή, πίσω στην στήλη αναφοράς.
- Στην περίπτωση του V4: Για κάθε νήμα, αφού έχουμε σε O(1) τη κάθε μή μηδενική θέση του πίνακα, υπολογίζουμε το εσωτερικό γινόμενο της αντίστοιχης γραμμής και στήλης.

Αυτή η προσέγγιση δυστυχώς έχει ένα μεγάλο μειονέκτημα, κάτι που επιβεβαιώσαμε και πειραματικά [2]. Κάθε χρονική στιγμή πολλά νήματα προσπαθούν να αναγνώσουν από τη μνήμη, μία ένα αντικείμενο από τη στήλη αναφοράς και μία από τη στήλη κάποιας άλλης ακμής. Επίσης προσπαθούν να γράψουν στην θέση του διανύσματος εξόδου στη θέση που αντιστοιχεί στη στήλη αναφοράς. Αυτό εκτός από το μεγάλο race, αυξάνει δραματικά τις αποτυχίες της cache, με αποτέλεσμα να χάνεται οποιαδήποτε επιτάχυνση μπορούσαμε να αποκτήσουμε.

Για να υπερκεράσουμε αυτό το πρόβλημα αποφασίσαμε να **"δώσουμε" σε κάθε νήμα μία και μόνο στήλη**. Αυτό μειώνει τον αριθμό των νημάτων πάρα πολύ και σχεδόν εξαλείφει τα κομμάτια που πρέπει να προστατευτούν για race. Σε αυτή την προσέγγιση το κάθε νήμα εκτελεί ότι περιγράψαμε παραπάνω, μόνο που τώρα το εκτελεί *για κάθε μη μηδενικό αντικείμενο της κάθε στήλης*. Με αυτό τον τρόπο εκμεταλλευόμαστε την cache του κάθε πυρήνα και τα άλματα στη μνήμη γίνονται από μια(ίδια στήλη ανά πυρήνα), σε μια διαφορετική κάθε φορά. Το μειονέκτημα σε αυτή την περίπτωση προκύπτει από τη σειρά με την οποία δημιουργούνται τα νήματα. Σε ένα πίνακα που τα μη μηδενικά στοιχεία είναι ομοιόμορφα κατανεμημένα τα νήματα χρειάζονται περίπου τον ίδιο χρόνο για να εκτελεστούν. Αν όμως κάποιες στήλες είναι πολύ *"βαρύτερες"* από κάποιες άλλες τότε τα νήματα σε αυτές καθυστερούν περισσότερο και μαζί τους καθυστερούν όλη τη διαδικασία.

Σε αυτό το σημείο έρχονται να βοηθήσουν τα εργαλεία διαχείρισης του χρονοπρογραμματισμού(scheduling) των νημάτων. Στην περίπτωση της cilk δεν έχουμε κάποια πρόσβαση. Από την άλλη όμως η cilk έχει ένα αρκετά ευέλικτο μοντέλο που στηρίζεται στο work stealing[3] με αποτέλεσμα να διαχειρίζεται αρκετά ομαλά τις ανομοιομορφίες.

Στην openMP από την άλλη έχουμε τη δυνατότητα να ελέγξουμε το scheduling, κάτι που για την υλοποίησή μας, μπορεί να γίνει με την παράμετρο "--dynamic" από τη γραμμή εντολών. Έτσι για τους πίνακες που είναι αρκετά ανομοιόμορφοι μπορούμε να επιλέξουμε δυναμικό, ενώ για τους ποιο ομοιόμορφους στατικό. Με το δυναμικό το αποτέλεσμα είναι τα νήματα που εκτελούνται κάθε χρονική στιγμή να έρχονται από "τυχαίες" στήλες του πίνακα. Αυτή η τυχαιότητα έχει ως αποτέλεσμα να εξαλείφει τις ανομοιομορφίες επιβαρύνοντας λιγάκι την cache. Συνολικά όμως σε ένα ανομοιόμορφο πίνακα αξίζει τον κόπο. Για παράδειγμα στον πίνακα com_Youtube στην έκδοση V4, για 8 πυρήνες με στατικό scheduling είχαμε 2780 msec ενώ με δυναμικό 1598 msec.

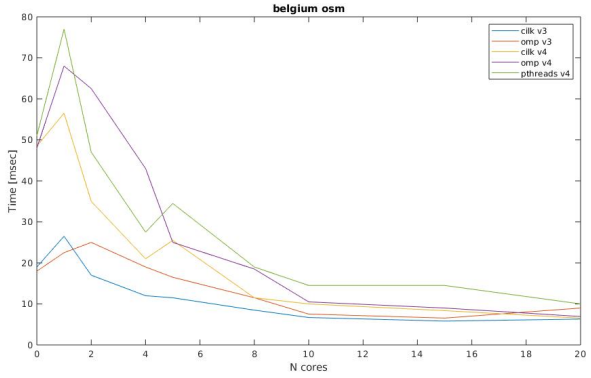
Τέλος στην περίπτωση των pthreads, δεν έχουμε κανένα αυτοματισμό έτοιμο. Παρόλα αυτά προσπαθήσαμε να κάνουμε κάτι αντίστοιχο της υλοποίησης του openMP για το static και dynamic scheduling. Αντί να προσπελάσουμε τον πίνακα ανά στήλη με τη σειρά, τον προσπελνάμε με βάση τις τιμές μια ακολουθίας. Όταν θέλουμε static scheduling την ακολουθία αυτή τη έχουμε σε αύξουσα σειρά, όταν όμως θέλουμε "dynamic", τότε την ανακατεύουμε με τυχαίο τρόπο. Έτσι πάλι έχουμε το κέρδος της εξάλειψης της ανομοιομορφίας όταν αυτό είναι απαραίτητο.

Παρακάτω παραθέτουμε ένα δείγμα από τις μετρήσεις που έγιναν στη συστοιχία. Σε αυτές η πρώτη μέτρηση (0 cores) αφορά τη σειριακή υλοποίηση και η δεύτερη (1 core) ένα και μόνο νήμα. Από τα γραφήματα μπορούμε να δούμε και την αύξηση του χρόνου στο ένα νήμα, καθώς και την αργή πτώση των χρόνων στα λίγα νήματα. Αυτά οφείλονται στον έξτρα φόρτο που προσθέτει στο πρόγραμμα μας η διαχείριση των νημάτων.

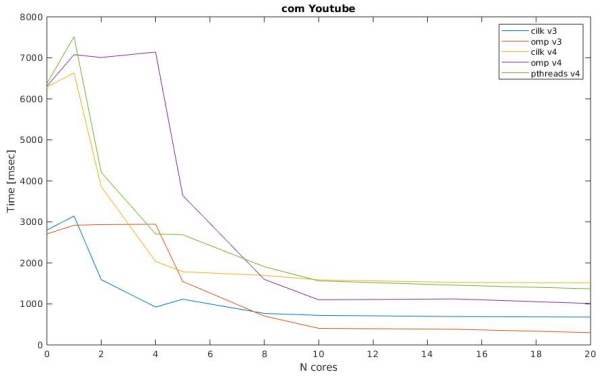
[2] Χρησιμοποιήσαμε τα valgrind και gprof

[3] `https://en.wikipedia.org/wiki/Work_stealing`

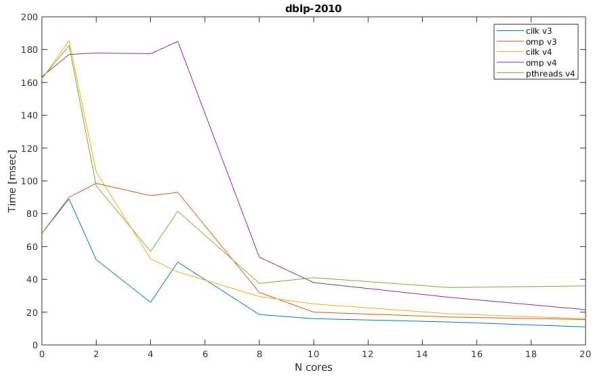
Algo/Matrix	belgium_osm	com_Youtube	dblp-2010	mycielskian13	NACA0015
Serial v3	19.5 [ms]	2705.0 [ms]	68.0 [ms]	1056.0 [ms]	175.5 [ms]
OpenMP v3 (x8)	11.5 [ms]	704.0 [ms]	32.0 [ms]	276.0 [ms]	102.0 [ms]
OpenMP v3 (x20)	8.9 [ms]	301.0 [ms]	15.5 [ms]	76.0 [ms]	34.5 [ms]
Cilk v3 (x8)	8.5 [ms]	768.0 [ms]	18.5 [ms]	395.0 [ms]	28.0 [ms]
Cilk v3 (x20)	6.3 [ms]	678.5 [ms]	11.0 [ms]	171.5 [ms]	16.5 [ms]
Serial v4	48.5 [ms]	6289 [ms]	162.0 [ms]	1743 [ms]	633.5 [ms]
OpenMP v4 (x8)	18.5 [ms]	1598 [ms]	53.5 [ms]	517.0 [ms]	235.0 [ms]
OpenMP v4 (x20)	6932 [us]	1010 [ms]	21.5 [ms]	128.5 [ms]	62.5 [ms]
Cilk v4 (x8)	11.5 [ms]	1695 [ms]	29.5 [ms]	303.0 [ms]	123.5 [ms]
Cilk v4 (x20)	6521 [us]	1516 [ms]	16.0 [ms]	137.5 [ms]	51.0 [ms]
pthreads v4 (x8)	19.0 [ms]	1907 [ms]	37.5 [ms]	303.5 [ms]	128.5 [ms]
pthreads v4 (x20)	10.0 [ms]	1369 [ms]	36.0 [ms]	154.0 [ms]	56.0 [ms]
Serial v4 (sum)	28.3 [ms]	2943 [ms]	36.5 [ms]	561.5 [ms]	135.2 [ms]
OpenMP v4 (sum)	3931 [us]	883 [ms]	7761 [us]	43.0 [ms]	23.9 [ms]
Cilk v4 (sum)	3872 [us]	1288 [ms]	5710 [us]	95.0 [ms]	13.2 [ms]
pthreads v4 (sum)	8424 [us]	980.6 [ms]	9370 [us]	89.9 [ms]	25.9 [ms]



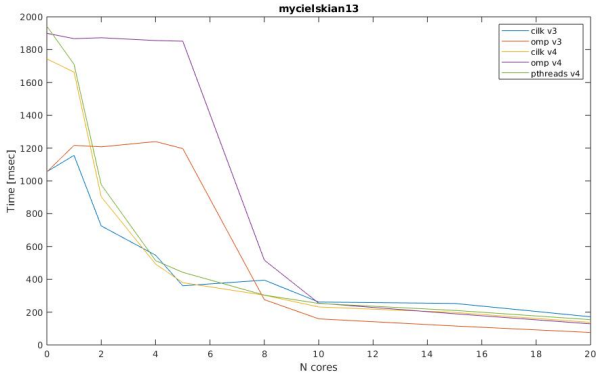
(α′) *Πίνακας belgium_osm*



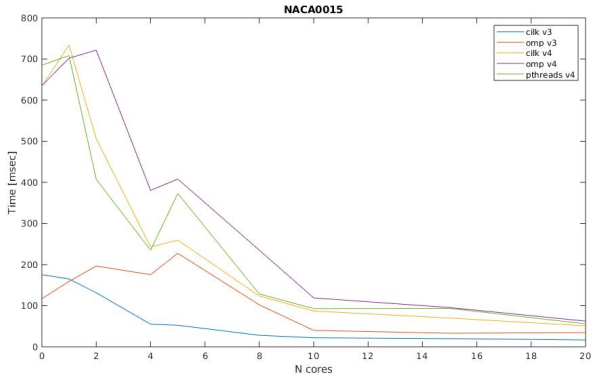
(β′) *Πίνακας com_Youtube*



(γ′) *Πίνακας dblp-2010*



(δ′) *Πίνακας mycielskian13*



(ε′) *Πίνακας NACA0015*

Α′. Παραρτημα

Α′.1. Μέτρηση τριγώνων με κάτω τριγωνικό πίνακα

Έστω **L** ο κάτω τριγωνικός πίνακας ενός συμμετρικού γράφου **G**, όπου $L = (l_{ij}) \neq 0 \quad \forall i > j$. Τότε:

$$(L^T \cdot L)_{(i,j)} = \sum_{k=0}^{N-1} l_{ik}^T \cdot l_{kj} = \sum_{k=0}^{N-1} l_{ki} \cdot l_{kj} \quad \forall k > i, k > j \tag{1}$$

Έτσι από την (1) προκύπτει:

$$(L^T \cdot L)_{(i,j)} = \sum_{k=0}^{N-1} l_{ki} \cdot l_{kj} \Rightarrow \ (L \odot (L^T \cdot L))_{(i,j)} = l_{ij} \cdot \sum_{k=0}^{N-1} l_{ki} \cdot l_{kj} = \sum_{k=0}^{N-1} l_{ij} \cdot l_{ki} \cdot l_{kj} \tag{2}$$

$$\Rightarrow C_{(i,j)} = \sum_{k=0}^{N-1} l_{ij} \cdot l_{ki} \cdot l_{kj} \quad \forall k > i > j \tag{3}$$

Όπου φαίνεται πως ο **C** είναι ένας κάτω τριγωνικός πίνακας με μη μηδενικά στοιχεία μόνο στις θέσεις (i,j) για τις οποίες υπάρχουν στον **G** τα τρίγωνα (i,j,k) για κάθε i,j,k τ.ω: $k > i > j$. Δηλαδή **μόνο μία φορά**. Και τελικά:

$$\vec{c_1} = (L \odot (L^T \cdot L)) \cdot \vec{1} \Rightarrow \vec{c_1}_{(i)} = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} l_{ij} \cdot l_{ki} \cdot l_{kj} \quad \forall k > i > j \tag{4}$$

Απ′ όπου φαίνεται πως το $\vec{c_1}$ είναι διάνυσμα που στη κάθε γραμμή έχει το άθροισμα των τριγώνων μετρημένο μόνο σε μία ακμή(την $l_{ix}, \forall x > i$ του αρχικού γράφου), όχι και στις τρεις. Το άθροισμα αυτού του διανύσματος, προφανώς, ισοδυναμεί με το συνολικό αριθμό των ***μη διατεταγμένων τριγώνων*** του γράφου.