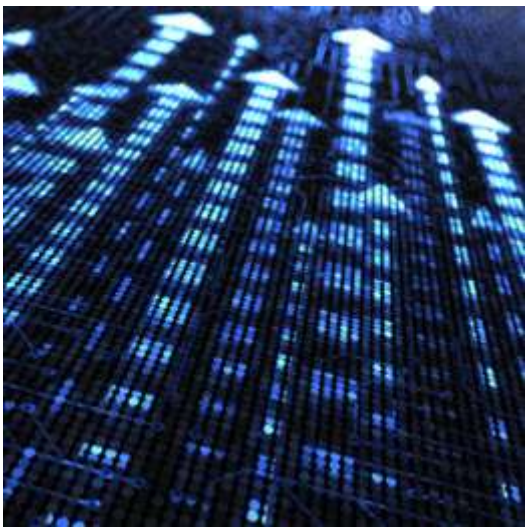
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάθημα: Παράλληλη Επεξεργασία

Αναφορά στην Εργασία Εξαμήνου



Εργασία των Φοιτητών:	**Καββαδίας Παναγιώτης**	**1054350**
	Κάλλιστρος Ανδρέας	**1054351**
	Τριανταφυλλόπουλος Παναγιώτης	**1054367**
	Χατζόπουλος Θωμάς	**1054288**
Υπεύθυνοι Καθηγητές:	**Ευστράτιος Γαλλόπουλος**	
	Ιωάννης Βενέτης	

ΠΑΤΡΑ | 20 ΜΑΪΟΥ 2019

Περιεχόμενα Εργασίας

0. Εισαγωγή 2
1. Στατική ανάθεση φόρτου εργασίας σε βρόχο for. 2
2. Δυναμική ανάθεση φόρτου εργασίας σε βρόχο for 3
3. Παραλληλοποίηση με χρήση έργων (tasks) 1. 3
4. Παραλληλοποίηση με χρήση έργων (tasks) 2. 4
5. Διαγράμματα χρόνου εκτέλεσης. 4

0. Εισαγωγή

Στα πλαίσια αυτής της εργασίας ασχοληθήκαμε με το πρόβλημα της απλούστευσης τεθλασμένων γραμμών (polylines ή «πολυγραμμών»), το οποίο αποτελεί απαραίτητη επεξεργασία στην χαρτογραφία, αλλά η χρήση του μπορεί να γενικευθεί και σε άλλα πεδία. Πιο συγκεκριμένα, πάνω στον αλγόριθμο Ramer–Douglas–Peucker, ο οποίος τυπικά εξαλείφει λεπτομέρειες που δεν είναι απαραίτητες πάνω σε έναν χάρτη, παραλληλοποιήσαμε κάποια μέρη του, ώστε να βελτιωθεί η συνολική του απόδοση. Όπως ήταν απαραίτητο, δοκιμάσαμε στατική και δυναμική ανάθεση φόρτου εργασίας σε βρόχο for, αλλά και παραλληλοποίηση με χρήση έργων (tasks). Για την λήψη των απαιτούμενων μετρήσεων δημιουργήσαμε bash script το οποίο περιλαμβάνεται στα παραδοτέα αρχεία.

1. Στατική ανάθεση φόρτου εργασίας σε βρόχο for

Σε αυτό το ερώτημα κληθήκαμε να παραλληλοποιήσουμε τον αλγόριθμο Ramer–Douglas–Peucker ως προς τις πολλαπλές πολυγραμμές που περιέχονται στην λίστα AllPolylines και πρέπει να επεξεργαστούν. Η πρώτη λογική παραλληλοποίησης που ακολουθήσαμε είναι κάθε νήμα να αναλάβει με στατικό τρόπο ανάθεσης να κάνει απλοποίηση για ένα συγκεκριμένο αριθμό από συνεχόμενες πολυγραμμές, ο οποίο εξαρτάται από το πλήθος των threads που χρησιμοποιούμε, στην λίστα AllPolylines. Στην συνέχεια, κάθε νήμα εκτελεί σειριακά τον αλγόριθμο Ramer–Douglas–Peucker για κάθε πολυγραμμή που ανέλαβε να επεξεργαστεί.

(αρχείο κώδικα με όνομα: “Static.cpp”)

Έτσι, για να παραλληλοποιήσουμε τον block κώδικα που κάνει τους υπολογισμούς (for loop) χρησιμοποιήσαμε την οδηγία **omp parallel firstprivate(simplified_polyline, polyline)**. Ο όρος firstprivate απαιτείται, ώστε το κάθε νήμα να έχει την δική του έκδοση των μεταβλητών simplified_polyline και polyline οι οποίες έχουν αρχικοποιηθεί πριν την παράλληλη περιοχή.

Δημιουργήσαμε μια μεταβλητή **chunk_size,** η οποία παίρνει ως τιμή το πηλίκο του μεγέθους του vector AllPolylines με τον αριθμό των threads, τον οποίο παίρνουμε χρησιμοποιώντας την συνάρτηση της βιβλιοθήκης Run time της OpenMP omp_get_num_threads(). Για να μετρήσουμε τον χρόνο που απαιτεί κάθε thread, ορίζουμε μια μεταβλητή **wtime** χρησιμοποιώντας την συνάρτηση omp_get_wtime() της ίδιας βιβλιοθήκης.

Για να επιτύχουμε στατικό τρόπο ανάθεσης στο βρόχο for χρησιμοποιήσαμε την οδηγία omp for **schedule(static, chunk_size) nowait**. Η παράμετρος nowait επιτρέπει σε κάθε thread να εκτελείται χωρίς να χρειάζεται να περιμένει στο φράγμα (barrier) της παράλληλης περιοχής. Πριν την αποθήκευση των απλοποιημένων πολυγραμμών στο vector SimplifiedAllPolylines ορίζουμε κρίσιμη περιοχή μέσω της οδηγίας **omp critical,** διότι, αν περισσότερα από ένα νήματα προσπαθήσουν να εισάγουν ταυτόχρονα στοιχεία στο vector, θα δημιουργηθούν συνθήκες ανταγωνισμού. Τέλος ενημερώνουμε την μεταβλητή **wtime** με τη διαφορά του τελικού από τον αρχικό χρόνο (omp_get_wtime()) ώστε να μετρήσουμε τον χρόνο εκτέλεσης κάθε νήματος.

Από τις μετρήσεις που πήραμε (βρίσκονται στην ενότητα 5 της αναφοράς) παρατηρούμε ότι κάθε φορά ένα νήμα χρειάζεται σημαντικά περισσότερο χρόνο από τα υπόλοιπα. Αυτό συμβαίνει διότι το συγκεκριμένο νήμα εκτελεί, εκτός από τους υπολογισμούς που εκτελούν και τα υπόλοιπα, και τις εργασίες εισόδου-εξόδου (I/O operations). Αυτό πραγματοποιείται λόγω του **στατικού τρόπου ανάθεσης,** καθώς ο scheduler της OpenMP προαποφασίζει τα τμήματα που θα εκτελέσει κάθε νήμα, χωρίς να λαμβάνει υπόψιν του άλλες διεργασίες που πιθανόν εκτελεί το νήμα.

2. Δυναμική ανάθεση φόρτου εργασίας σε βρόχο for

Σε αυτό το ερώτημα, στην ίδια λογική με το προηγούμενο, κληθήκαμε να παραλληλοποιήσουμε τον αλγόριθμο, ώστε κάθε νήμα να αναλάβει με δυναμικό τρόπο ανάθεσης να κάνει απλοποίηση για ένα συγκεκριμένο αριθμό από συνεχόμενες πολυγραμμές, ο οποίο εξαρτάται από το πλήθος των threads που χρησιμοποιούμε στην λίστα AllPolylines. Στην συνέχεια, κάθε νήμα εκτελεί σειριακά τον αλγόριθμο Ramer–Douglas–Peucker για κάθε πολυγραμμή που ανέλαβε να επεξεργαστεί.

(αρχείο κώδικα με όνομα: "Dynamic.cpp")

Για να επιτύχουμε δυναμικό τρόπο ανάθεσης, χρησιμοποιήσαμε τον κώδικα του προηγούμενου ερωτήματος μετατρέποντας omp for schedule(static, chunk_size) nowait σε **omp for schedule(dynamic, 1)**. Η παράμετρος που πρέπει να καθοριστεί πειραματικά είναι το chunk_size που εμείς έχουμε ορίσει ως 1. Δώσαμε τιμή 1 στο chunk_size, καθότι μετά από μετρήσεις με διάφορες τιμές και δοκιμάζοντας την εκτέλεση με διαφορετικό αριθμό threads, παρατηρήσαμε ότι με chunk_size=1 επιτυγχάνεται (συνολικά) η μέγιστη απόδοση. Στο δυναμικό τρόπο ανάθεσης κάθε νήμα εκτελεί ένα μέρος (chunk) των επαναλήψεων της for και, αφού ολοκληρώσει το έργο του, ζητάει από το scheduler άλλο ένα chunk ίδιου μεγέθους. Επομένως το μέγεθος των chunk επηρεάζει σημαντικά την απόδοση.

Χρονομετρώντας τον χρόνο εκτέλεσης κάθε νήματος χωριστά παρατηρούμε ότι είναι περίπου ίσοι, σε αντίθεση με το πρώτο ερώτημα. Αυτό συμβαίνει επειδή στο δυναμικό τρόπο ανάθεσης κάθε νήμα ζητάει νέα chunks αφού τελειώσει με την εργασία που κάνει. Δηλαδή, η ανάθεση των chunks σε κάθε νήμα δεν είναι προαποφασισμένη. Συνεπώς, συγκριτικά με τον στατικό τρόπο ανάθεσης, το νήμα που κάνει τις I/O operations αναλαμβάνει λιγότερα chunks.

3. Παραλληλοποίηση με χρήση έργων (tasks) 1

Με βάση τον αλγόριθμο, όταν η μεγαλύτερη απόσταση ενός σημείου από την ευθεία που ορίζεται από το πρώτο και τελευταίο σημείο της πολυγραμμής είναι μεγαλύτερη από μια συγκεκριμένη τιμή, τότε δημιουργούνται δύο νέες, μικρότερες πολυγραμμές και επαναλαμβάνεται αναδρομικά ο αλγόριθμος για κάθε μια από τις νέες πολυγραμμές. Σε αυτό το ερώτημα κληθήκαμε να υλοποιήσουμε μια παράλληλη έκδοση του αλγορίθμου, ώστε όταν μια πολυγραμμή διασπαστεί σε δύο μικρότερες, τότε για την πρώτη να δημιουργείται ένα νέο έργο (task), ενώ την δεύτερη θα την επεξεργάζεται το τρέχων έργο (task).

(αρχείο κώδικα με όνομα: "Task1.cpp")

Για την υλοποίηση της παραλληλοποίησης με χρήση έργων (**Tasks**) ορίσαμε ως παράλληλη περιοχή το for loop της main. Χρησιμοποιήσαμε τις οδηγίες omp parallel και omp single nowait, η οποία απαιτείται, ώστε μόνο ένα νήμα να αναλάβει την δημιουργία των **Tasks**. Στο σημείο της μεθόδου void RamerDouglasPeucker, που η πολυγραμμή διασπάται σε δύο μικρότερες, για την πρώτη αναδρομική κλήση χρησιμοποιούμε την οδηγία omp task shared(recResults1) για να δημιουργήσουμε ένα νέο έργο. Τον vector recResults1 τον ορίζουμε shared, καθώς δεν υπάρχουν συνθήκες ανταγωνισμού και κάθε νήμα χρειάζεται να έχει το ίδιο αντίγραφο του vector. Μετά την ολοκλήρωση και των δύο αναδρομικών κλήσεων χρησιμοποιούμε την οδηγία omp taskwait, γιατί πριν την δημιουργία της λίστας αποτελεσμάτων είναι απαραίτητο να έχει ολοκληρωθεί το task.

4. Παραλληλοποίηση με χρήση έργων (tasks) 2

Σε αυτό το ερώτημα, ακολουθώντας την λογική του προηγούμενου, κληθήκαμε να υλοποιήσουμε μια παράλληλη έκδοση του αλγορίθμου, ώστε όταν μια πολυγραμμή διασπαστεί σε δύο μικρότερες, τότε να δημιουργείται ένα νέο έργο (task) και για τις δύο νέες πολυγραμμές.

(αρχείο κώδικα με όνομα: "Task2.cpp")

Βασιστήκαμε στον κώδικα του προηγούμενου ερωτήματος: για να δημιουργήσουμε νέα έργα για κάθε μια από τις αναδρομικές κλήσεις, στον κώδικα του προηγούμενου ερωτήματος, πριν την δεύτερη αναδρομική κλήση έχουμε προσθέσει την οδηγία omp task shared(recResults2) με διαμοιραζόμενη μεταβλητή τον vector recResults2 για τους ίδιους λόγους με παραπάνω.

Παρατηρούμε ότι ο χρόνος εκτέλεσης είναι σημαντικά μεγαλύτερος από το προηγούμενο ερώτημα πλησιάζοντας τον χρόνο της σειριακής εκτέλεσης. Αυτό συμβαίνει επειδή ο αλγόριθμος είναι αναδρομικός, και συνεπώς, από κάθε task μπορούν αναδρομικά να προκύψουν νέα task. Για να συνεχιστεί η εκτέλεση του προγράμματος μετά την εντολή taskwait θα πρέπει να ολοκληρωθούν και τα δύο tasks. Όμως, για να ολοκληρωθεί κάθε task πρέπει να έχουν ολοκληρωθεί και τα παιδία του. Στο προηγούμενο ερώτημα δημιουργούνταν νέα task μόνο για την πρώτη αναδρομή, προσέγγιση που έδωσε καλύτερους χρόνους εκτέλεσης. Για να συμβαίνει αυτό το φαινόμενο, ο αλγόριθμος θα μπορούσε να μετατραπεί σε μη ανα-δρομικό. Αν αυτό δεν είναι δυνατό, θα μπορούσαμε πριν την δημιουργία του task να έχουμε μια εκτίμηση του βάθους της αναδρομής συγκρίνοντας πόσο μεγαλύτερη είναι η μέγιστη απόσταση από το epsilon και αν το βάθος είναι μεγάλο, να μην δημιουργείται νέο task για την επεξεργασία του δεύτερου μέρους της πολυγραμμής.

5. Διαγράμματα χρόνου εκτέλεσης

Προκειμένου να ελέγξουμε τις παραπάνω παραλληλοποιήσεις δημιουργήσαμε ένα αρχείο script (όνομα αρχείου: "script.sh") με το οποίο εκτελούμε τα παραπάνω αρχεία για διαφορετικές παραμέτρους (-O0 / -O3) και διαφορετικό πλήθος threads (1/2/4/8).

Παρακάτω παραθέτουμε δύο πίνακες, οι οποία περιέχουν τα χρονικά αποτελέσματα της εκτέλεσης του αλγορίθμου (σειριακή, με στατική και δυναμική ανάθεση φόρτου εργασίας σε βρόχο for και με παραλληλοποίηση με χρήση έργων (tasks)), χρησιμοποιώντας 1, 2, 4 ή 8 threads κάθε φορά. Ο πρώτος περιέχει τα αποτελέσματα όταν χρησιμοποιήσαμε την παράμετρο -O0 και ο δεύτερος όταν χρησιμοποιήσαμε την παράμετρο -O3. Μονάδα μέτρησης: seconds.

Χρονικά αποτελέσματα για -O0

Threads	Serial	Static	Dynamic	Task1	Task2
1	43,74	42,92	43,26	46,65	51,29
2	-	42,07	23,34	34,83	38,76
4	-	39,96	12,59	24,63	31,84
8	-	34,46	9,38	21,83	27,08

Χρονικά αποτελέσματα για -O3

Threads	Serial	Static	Dynamic	Task1	Task2
1	11,09	11,30	11,25	13,16	13,69
2	-	11,14	5,91	11,18	14,07
4	-	10,61	3,29	8,57	12,63
8	-	9,11	2,42	8,96	12,04

Παρακάτω παρατίθενται τα διαγράμματα χρονοβελτίωσης για παράμετρο -O0 και -O3.