Παράλληλα και Διανεμημένα Συστήματα

Μάριος Πάκας
mariospakas@ece.auth.gr
9498

Εργασία 1

Ανάλυση της δομής δεδομένων

Το πρώτο βήμα στην ανάπτυξη κώδικα για τη συγκεκριμένη εργασία ήταν η ανάγνωση Market Matrix. Επειδή στην αρχή δεν έβρισκα δυαδικούς πίνακες, μέσω του command line terminal υπάρχει επιλογή να δώσει ο χρήστης την τιμή 0 για μη δυαδικό πίνακα και 1 για δυαδικό πίνακα (default 1). Έτσι γνωρίζει το πρόγραμμα αν ο coo πίνακας έχει 3 ή 2 στοιχεία ανά γραμμή και με αυτόν τον τρόπο μπορεί να τα διαβάσει δίχως πρόβλημα μνήμης. Ωστόσο δεν δουλεύει για πίνακες με μη μηδενικά στοιχεία στη διαγώνιο.

Ακόμη, ήταν σημαντικό για το v3 να έχω τους πίνακες σε άνω τριγωνική μορφή, οπότε ήθελα να συγκρίνω την στήλη και τη γραμμή του συμμετρικού πίνακα ώστε, ανάλογα με το τι ήταν μεγαλύτερο, να το περάσω σωστά ως όρισμα στην συνάρτηση coo2csc και να βγει άνω τριγωνικός. Στο v4 με βόλεψε να δουλέψω με ολόκληρο τον πίνακα και όχι μόνο τον συμμετρικό, οπότε διπλασίαζα όλα τα στοιχεία.

Ανάλυση αλγορίθμου V3

Για το V3, εφόσον έχουμε πλέον τον πίνακα σε δομή csc η σκέψη μου ήταν η εξής: Ξεκινούσα από ένα τυχαίο στοιχείο έστω row1, col1. Τότε γνωρίζω ότι:

(row1,col1) -> (col1,col2) -> (col2,row1)

Στη μορφή csc ο εύκολος τρόπος αναζήτησης ήταν ανά στήλη να βρίσκω ποια γραμμή είναι μη μηδενική, επομένως είχα στα χέρια μου το πρώτο στοιχείο (row1, col1). Στη συνέχεια (και επειδή, λόγω συμμετρίας του πίνακα, ίσχυε ότι στήλη = γραμμή και το αντίθετο) μπορούσα να ψάξω για το στήλη row1 και να βρω ποιες τιμές υπήρχαν ως γραμμές. Συνεπώς είχα πλέον στα χέρια μου και το στοιχείο (col2,row1). Προκειμένου να υπάρχει λοιπόν τρίγωνο έπρεπε να αναζητήσω αν στη στήλη col2 υπήρχε στοιχείο στη γραμμή με συντεταγμένη col1! Σε περίπτωση που υπήρχε αύξανα τον αριθμό τριγώνων, και τον πίνακα c3 για τις αντίστοιχες κορυφές row1, col1, col2.

Ανάλυση αλγορίθμου V4

Για τον αλγόριθμο στο V4 το πρώτο βήμα ήταν να υλοποιήσω τον πολλαπλασιασμό C=A⊙(AA). Στην αρχή έκανα ξεχωριστά τον κάθε πολλαπλασιασμό, αλλά στη συνέχεια το άλλαξα διότι ο A*A έβγαινε πυκνός και ήταν πιο περίπλοκο στο χειρισμό της μνήμης. Η ιδέα πίσω από αυτόν τον πολλαπλασιασμό είναι ότι τα μόνα στοιχεία που θα υπάρχουν θα βρίσκονται στην ίδια ακριβώς θέση με τον αρχικό πίνακα A. Επομένως ξεκινούσα περνώντας από κάθε στοιχείο (row, col). Στο επόμενο βήμα έπρεπε να υπολογίσω την τιμή (AA)[row,col]. Η τιμή αυτή ισούται με το άθροισμα των πολλαπλασιασμών ανάμεσα στις τιμές της γραμμής row και της στήλης col. Εκμεταλλευόμενοι την ιδιότητα της συμμετρίας όμως: γραμμή = στήλη. Επομένως κρατούσα σε έναν πίνακα k όλες τις συντεταγμένες των στηλών που υπήρχαν στην γραμμή row και σε έναν πίνακα l όλες τις συντεταγμένες των γραμμών στη στήλη col. Οι πίνακες αυτοί, λόγω της δομής csc ήταν και ταξινομημένοι, επομένως με ένα από merge sort σύγκρινα μεταξύ τους τις τιμές και εφόσον υπήρχε

στήλη ίση με γραμμή σήμαινε ότι το (AA)[row,col] έπρεπε να αυξηθεί κατά 1 (αφού A δυαδικός). Με αυτόν τον τρόπο έβρισκα τις τιμές του C, μιας και ισούταν με τις τιμές του (AA), απλώς μόνο για τις συντεταγμένες που ο A είχε ήδη στοιχεία.

Το επόμενο βήμα ήταν να υλοποιήσω c3=C*e όπου e ένα διάνυσμα μοναδιαίο. Υλοποιήθηκε με τη λογική ότι το i στοιχείο στο τελικό διάνυσμα είναι το άθροισμα της i-στης γραμμής (Hint: στήλης) με όλο το διάνυσμα c3. Ουσιαστικά το i-οστό στοιχείο του c3 ισούται με το άθροισμα όλων των στοιχείων που έχει ο C στην ι-στη γραμμή!

Cilk/Openmp

V3

Χρησιμοποίησα το cilk_for/#pragma omp parallel for για να παραλληλοποιήσω το εξωτερικό for loop και να δουλέψω για κάθε στήλη παράλληλα.

V4

Χρησιμοποίησα το cilk_for for για να παραλληλοποιήσω και το εξωτερικό for loop αλλά και το εσωτερικό, επομένως να δουλέψω με κάθε στοιχείο ξεχωριστά. Τη μεγαλύτερη διαφορά την έκανε η παραλληλοποιήση μόνο του πρώτου for loop, ωστόσο σε μερικά datasets (τα οποία ήταν πολύ μεγάλα) παρατήρησα μικρή βελτίωση, οπότε το κράτησα και εκεί.

Παρατήρησα ακόμη ότι ο παραλληλισμός του matrix*vector κομματιού, δεν είχε σημαντική διαφορά ή είχε χειρότερη επίδοση, για αυτό και δεν τον εφήρμοσα κι εκεί.

Για το Openmp χρησιμοποίησα το #pragma omp parallel μόνο στο εξωτερικό loop, σε αντίθεση με την cilk υλοποίηση διότι δεν παρατήρησα βελτίωση με τον παραλληλισμό του εσωτερικού loop, αλλά χειρότερη επίδοση.

Pthreads

Η ιδέα πίσω από την υλοποίηση μου στο pthreads είναι η εξής. Σκέφτηκα να παραλληλίσω το for loop και να το σπάσω σε κομμάτια ανάλογα με τον αριθμό των πυρήνων (στήλες/cores). Με αυτόν τον τρόπο καλούσα παράλληλα μια συνάρτηση πολλαπλασιασμού για κάθε ένα μικρότερο υποσύνολο της στήλης, αφού ο υπολογισμός της τιμής του C ήταν ανεξάρτητη διαδικασία, συνεπώς έδινα σε κάθε επεξεργαστή το δικό του κομμάτι να δουλέψει. Η λογική αυτή παρουσιάζει περιέργως για κλήση με έναν πυρήνα χειρότερη επίδοση από το κανονικό v4, πράγμα που με παραξένεψε. Σίγουρα υπάρχει βελτίωση όσο αυξάνεται ο αριθμός των πυρήνων, σε σχέση με την κλήση για έναν επεξεργαστή, ωστόσο για μερικά datasets ακόμα και με 8 πυρήνες, δεν καταφέρνει να ξεπεράσει την σειριακή επίδοση της v4, πράγμα απογοητευτικό. Συγκεκριμένα για το cielskian έκανε εμφανώς περισσότερη ώρα, για τα Belgium, dble, περίπου τα ίδια με το σειριακό, ωστόσο για το NACA παρουσίασε σημαντική βελτίωση. Δυστυχώς για το youtube δεν μπόρεσα ποτέ να βρω το bug που το έκανε να μην τερματίζει, επομένως δεν έχω μετρήσεις για αυτό. Τέλος αξίζει να σημειωθεί ότι δεν κατάφερα ποτέ να το κάνω να δουλέψει καλύτερα απ’ότι η cilk.

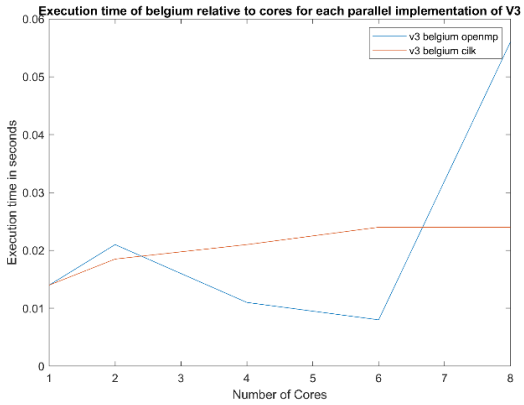
Μερικά Σχόλια

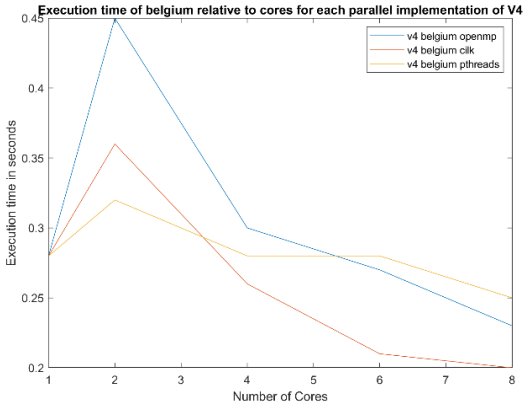
Παρατήρησα ότι ο αλγόριθμος v3 ήταν πιο γρήγορος σε όλα τα datasets πέρα από τον cielskian, στο οποίο υπερτερούσε κατά πολύ ο v4, μιας και στο cielskian δεν υπήρχαν καθόλου τρίγωνα.

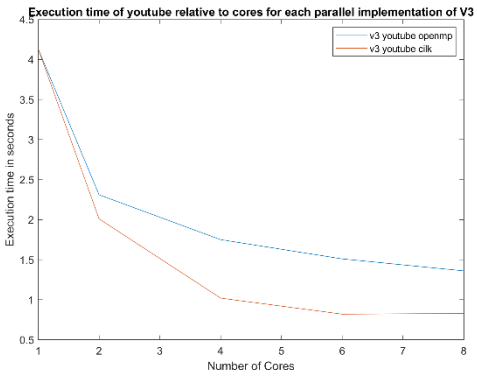
Σε μερικές παράλληλες υλοποιήσεις υπήρχε βελτίωση όσο αυξανόταν ο αριθμών των πυρήνων (προφανώς όχι γραμμική), ωστόσο από κάποιο σημείο και μετά μπορεί να υπήρχε και χειρότερη επίδοση με την αύξηση των πυρήνων.

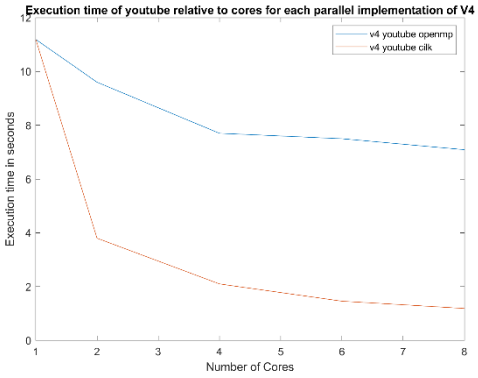
Ανάλογα το database κάποια υλοποίηση (ανάμεσα σε v3 και v4) παρουσίαζε βελτίωση με την παραλληλοποίηση, ενώ η άλλη χειρότερους χρόνους εκτέλεσης.

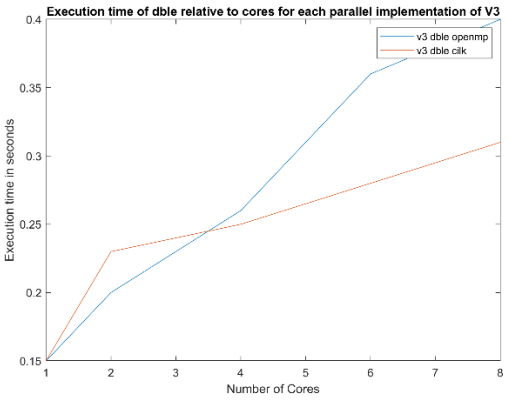
Διαγράμματα που απεικονίζουν τους χρόνους εκτέλεσης κάθε προγράμματος

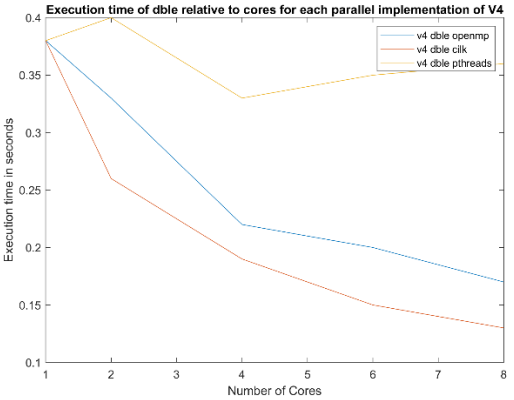


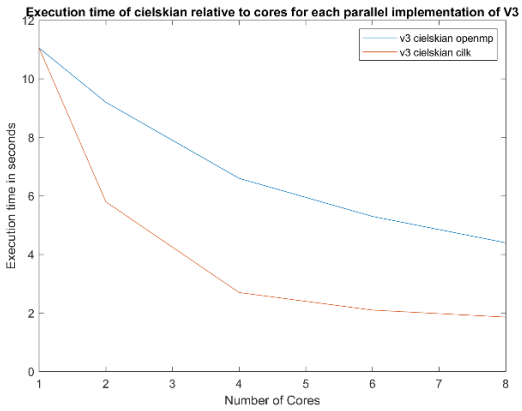


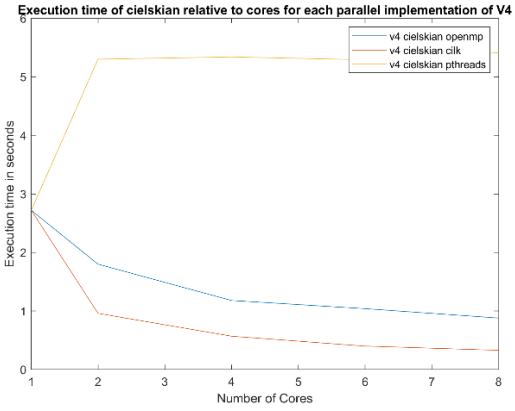


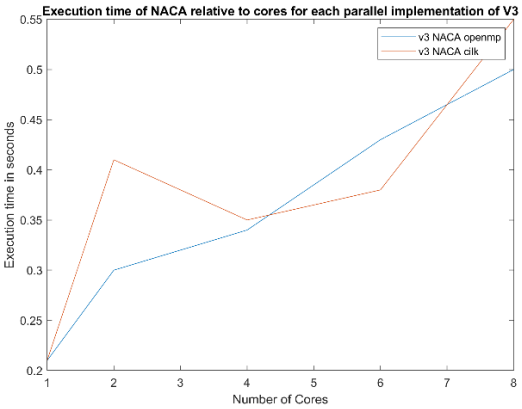


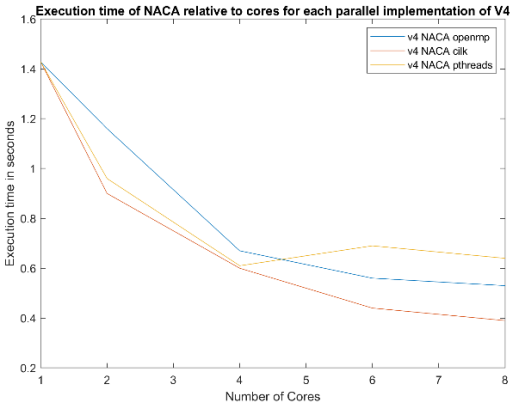









Execution time of NACA relative to cores for each parallel implementation of V3
v3 NACA openmp
v3 NACA cilk
Execution time in seconds
Number of Cores
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
1
2
3
4
5
6
7
8


Execution time of NACA relative to cores for each parallel implementation of V4
v4 NACA openmp
v4 NACA cilk
v4 NACA pthreads
Execution time in seconds
Number of Cores
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
1
2
3
4
5
6
7
8