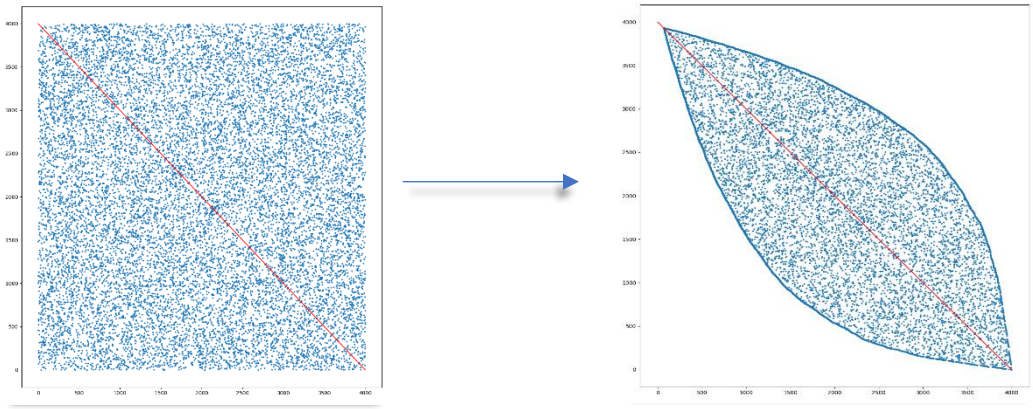
Παράλληλα και Διανεμημένα Συστήματα

Εργασία 4η: Parallel Implementation of Reverse Cuthill McKee Algorithm

Author

Όνομα: Πορτοκαλίδης Σταύρος, ΑΕΜ: 9334, email: stavport@ece.auth.gr
Code: https://github.com/sportokalidis/Parallel-Implementation-Reverse-Cuthill-Mckee-Algorithm



Περιγραφή προβλήματος

Σκοπός της 4ης εργασίας είναι η υλοποίηση του *Reverse Cuthill McKee Algorithm (RCM)*. Ο αλγόριθμος χρησιμοποιείται για την αναδιάταξη ενός αραιού τετραγωνικού πίνακα και είναι βασισμένος στον *Breadth First Search Algorithm* ενός γράφου. Ο αλγόριθμος παίρνει για είσοδο τον *adjacency matrix* του γράφου (που είναι και ο πίνακας που θέλουμε να αναδιατάξουμε) και απαριθμεί ξανά τους κόμβους του γράφου με κατάλληλή σειρά, με αποτέλεσμα συχνά να προκύπτει ένας νέος adjacency matrix με αρκετά μειωμένο bandwidth. Σκοπός της εργασίας είναι η υλοποίηση του αλγορίθμου αρχικά με σειριακό τρόπο και στην συνέχεια η εύρεση μιας μεθόδου παραλληλισμού του με χρήση της *openMP*.

Εξήγηση Υλοποίησης

Αρχικά σε κάθε version, γίνεται δήλωση με #def ο αριθμός των κόμβων του γράφου καθώς και το sparcity του αραιού πίνακα που θα δημιουργήσει το πρόγραμμα. Επιπλέον, στις παράλληλες υλοποιήσεις γίνεται και δήλωση του μέγιστου αριθμού threads που μπορούμε να χρησιμοποιήσουμε. Στην main function τις κάθε version, γίνεται αρχικά το initialization του adjacency matrix. Ανάλογα με την τιμή που έχουμε δηλώσει στο MODE, ο πίνακας είτε διαβάζεται από αρχείο για MODE=1, είτε δημιουργείται από το πρόγραμμα τυχαία ένας συμμετρικός πίνακας ανάλογα με το sparsity που δηλώσαμε για MODE=2. Σε κάθε περίπτωση, εκτελείται ο Cuthill McKee Algorithm και στο τέλος αντιστρέφουμε το αποτέλεσμα.

- Version 0: Sequential

  Πρόκειται για την sequential υλοποίηση του αλγορίθμου. Αρχικά, στην *Cuthill McKee()* function, γίνεται ο υπολογισμός των degrees, καθώς και οι αρχικοποιήσεις των *R vector*, που είναι ο πίνακας μεταθέσεων και μιας *ουράς Q*, η οποία μας βοηθάει να επιλέγουμε με την σωστή σειρά τους κόμβους που θα μπουν στον R. Ως πρώτο κόμβος επιλέγεται αυτός με το μικρότερο degree (δηλαδή αυτός με τους λιγότερους γείτονες), απ' ευθείας αυτός προστίθεται στην ουρά και γίνεται *visited*. Στη συνέχεια, μέχρι η *ουρά Q* να γίνει *empty*, αφαιρούμε ένα προς ένα τους κόμβους από αυτή, βρίσκουμε τους *no visited* γείτονες του κάθε κόμβου που μόλις αφαιρέσαμε, τους ταξινομούμε κατά αύξουσα σειρά με βάση το

degree (*sortByDegree()* function) και τους προσθέτουμε στο Q, τέλος προσθέτουμε το στοιχείο που αφαιρέσαμε στο *R vector*. Αν το Q γίνει *empty* και δεν έχουν προστεθεί όλοι οι κόμβοι στο R, δηλαδή έχουμε έναν *μη συνεκτικό γράφο (disjoint graph)*, τότε βρίσκουμε πάλι τον κόμβο με το μικρότερο degree που είναι *no visited* τον προσθέτουμε στο *Q*, και συνεχίζουμε την διαδικασία.

- Version 1: Find neighbors in parallel

  Στη version 1 εφαρμόζεται η πρώτη μέθοδος παραλληλισμού του αλγορίθμου με την χρήση της *openMP*. Αρχικά, ο υπολογισμός των degrees γίνεται παράλληλα (degreesCalculation Parallel() function), όπως και η εύρεση του μικρότερου degree (findMinIdxParallel(), function). Στη συνέχεια, για κάθε κόμβο που βγάζουμε από την ουρά, βρίσκουμε παράλληλα τους γείτονες του. Τέλος, η αντιστροφή του αποτελέσματος, γίνεται και αυτή παράλληλα.

- Version 2: Many nodes in parallel

  Η version 2 είναι η δεύτερη μέθοδος παραλληλισμού που υλοποιείται με χρήση της *openMP*. Όπως στην προηγούμενη έκδοση, ο υπολογισμός των degrees και η εύρεση του μικρότερου degree γίνεται παράλληλα. Σε αυτήν την περίπτωση, χρησιμοποιούμε περισσότερα από ένα thread για να εξυπηρετήσουμε περισσότερους από ένα κόμβους παράλληλα. Ο αριθμός των threads που χρησιμοποιούμε εξαρτάται από το μέγεθος της ουράς Q. Αν το μέγεθος της είναι μεγαλύτερο από τον μέγιστο αριθμό threads που μπορούμε να χρησιμοποιήσουμε, τότε χρησιμοποιούμε τον μέγιστο αριθμό threads, αν το μέγεθος της είναι μικρότερο, τότε ο αριθμός των threads είναι ίσος με το μέγεθος της ουράς. Όταν τερματίσουν όλα τα threads, το master thread είναι υπεύθυνο να προσθέσει όλα τα παιδιά των κόμβων στην ούρα με την σωστή σειρά (AddtoQueue() function).

- Version 3: Combination

  Η version 3 είναι η τρίτη μέθοδος παραλληλισμού που υλοποιείται με χρήση της *openMP* και είναι ένας συνδυασμός τον version 1 και version 2. Σε αυτή την version, όπως και στην version 2 χρησιμοποιούμε περισσότερα από ένα threads για να εξυπηρετήσουμε περισσότερους από ένα κόμβους ταυτόχρονα. Ωστόσο, όταν δεν υπάρχουν αρκετοί κόμβοι στην ουρά δεν αξιοποιούμε τον μέγιστο αριθμό threads που μπορούμε. Έτσι για να μειώσουμε τον αριθμό των idle threads όταν υπάρχουν, κάθε thread δημιουργεί επιπλέον threads και κάθε κόμβος εξυπηρετείται παράλληλα από πολλά threads όπως και στην version 1. Για παράδειγμα, όταν έχουμε τέσσερα threads στην διάθεση μας, και μόνο δύο κόμβους που μπορούμε να εξυπηρετήσουμε, τότε το κάθε thread δημιουργεί ένα επιπλέον thread και βρίσκουν τους γείτονες του κόμβος παράλληλα.

Testing

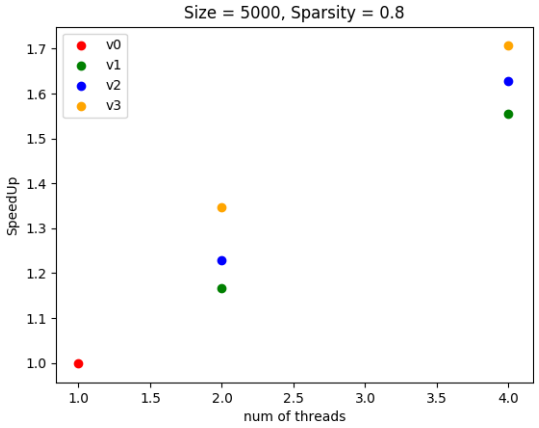
Για τον έλεγχο ορθότητας του αλγορίθμου έγινε χρήση του έτοιμου υλοποιημένου αλγορίθμου σε Python (python_tester.py). Ο έλεγχος γίνεται ως προς την έξοδο των προγραμμάτων, δηλαδή, ως προς τον πίνακα μεταθέσεων R. Το αποτέλεσμα κατά βάσει έβγαινε κοινό, ωστόσο διαφέρει σε μερικές περιπτώσεις και αυτό συμβαίνει όταν πρέπει να διαλέξουμε μεταξύ κόμβων με το ίδιο degree στην περίπτωση που έχουμε μη συνεκτικό γράφο. Επιπλέον, έγινε έλεγχος των εξόδων όλων των versions, βάζοντας τον ίδιο πίνακα για είσοδο, και το αποτέλεσμα ήταν κοινό σε όλες τις περιπτώσεις (χρήση του tester.py). Τέλος, με την χρήση του αρχείου *visualization.py* γίνεται μια αναπαράσταση του πίνακα και όντως ο πίνακας που προκύπτει έχει σημαντικά μειωμένο bandwidth και το αναμενόμενο σχήμα.

Μετρήσεις και Πειράματα

Hardware using: CPU: 4 x i7-7500U , RAM: 8 GB

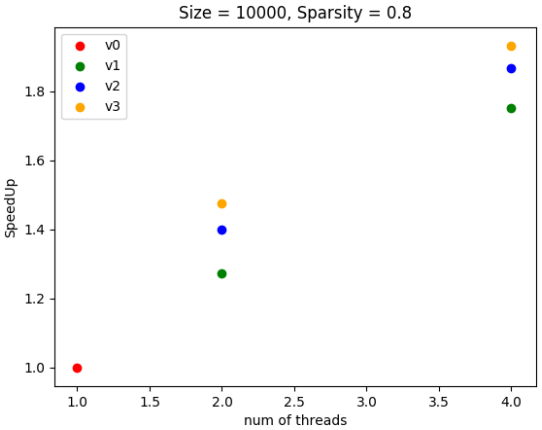
size=5.000, sparsity=0.8

	N =1	N = 2	N = 4
Version 0	0.07	-	-
Version 1	-	0.060	0.045
Version 2	-	0.057	0.043
Version 3	-	0.052	0.041



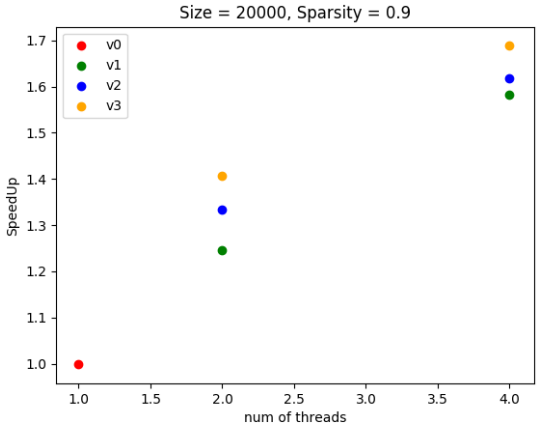
size=10.000, sparsity=0.8

	N =1	N = 2	N = 4
Version 0	0.28	-	-
Version 1	-	0.22	0.160
Version 2	-	0.20	0.150
Version 3	-	0.19	0.145



size=20.000, sparsity=0.8

	N =1	N = 2	N = 4
Version 0	1.09	-	-
Version 1	-	0.83	0.62
Version 2	-	0.75	0.59
Version 3	-	0.71	0.57



Σχολιασμός:

Παρατηρούμε ότι, με αύξηση του αριθμού των threads ο χρόνος για την εκτέλεση του αλγορίθμου μειώνεται. Παρατηρείται επίσης ότι, στην πλειοψηφία το v3 είναι γρηγορότερο. Ωστόσο, δε έχουμε μεγάλες διαφορές στις παράλληλες μεθόδους μεταξύ τους, οι οποίες φτάνουν να έχουν μέχρι και σε 2 φορές μεγαλύτερη ταχύτητα από τον σειριακό αλγόριθμο. Τέλος, αξίζει να σημειωθεί ότι, για πιο αραιούς πίνακες, ο χρόνος εκτέλεσης μειώνεται, πράγμα που επηρεάζει και τον παραλληλισμό. Αυτό είναι λογικό, αφού όσο πιο αραιός γίνεται ο πίνακας, τόσο μειώνεται και ο φόρτος που έχει να εξυπηρετήσει κάθε thread.

Περισσότερα αποτελέσματα και μετρήσεις εδώ