Δομές Δεδομένων – Εργασία 2

Τμήμα Πληροφορικής
Φθινοπωρινό Εξάμηνο 2021-2022
Υπεύθυνοι φοιτητές:
Μαρία Κονταράτου (3200078)
Γεώργιος Κουμουνδούρος (3200083)
Διδάσκων: Ε.Μαρκάκης

ΑΘΗΝΑ, ΔΕΚΕΜΒΡΙΟΣ 2021

ΜΕΡΟΣ Ε – ΑΝΑΦΟΡΑ ΠΑΡΑΔΟΣΗΣ

Α. Επεξήγηση ουράς προτεραιότητας μέρους Α

Στο πρώτο μέρος της εργασίας, κάναμε χρήση της ουράς προτεραιότητας MaxPQ για να μπορούμε να αποθηκεύουμε κάθε Processor ξεχωριστά με βάση τα δεδομένα που μας δίνονται από το input αρχείο txt του χρήστη. Στο main κομμάτι της κλάσης, διαβάζουμε το αρχείο και γνωρίζοντας ότι οι πρώτοι δύο αριθμοί είναι ο αριθμός των επεξεργαστών και των διεργασιών αντίστοιχα και τα αποθηκεύουμε στις αντίστοιχες μεταβλητές processorN και taskN. Στη συνέχεια, δημιουργούμε μια νέα ουρά προτεραιότητας με όνομα pcs μεγέθους processorN και καλούμε την μέθοδο greedyOne() καθώς και την printOutput().

Η greeedyOne() προσθέτει στην ουρά pcs μέσω της insert έναν νέο επεξεργαστή σύμφωνα με τη μεταβλητή processorN. Στη συνέχεια σε κάθε σειρά (όσες σειρές μας επιτρέπει το taskN) διαβάζουμε και αποθηκεύουμε το taskid της κάθε διεργασίας και συνέχεια το χρόνο επεξεργασίας time σε μεταβλητές. Στη συνέχεια βρίσκουμε τον επεξεργαστή μεγαλύτερη προτεραιότητα (μέσω της max() από την MaxPQ.java) και τοποθετούμε στις διεργασίες του τον χρόνο επεξεργασίας. Μετά κάνουμε sink, δηλαδή κατεβάζουμε τον κόμβο προς τα κάτω ώστε να έχει πλέον μικρότερη προτεραιότητα για την επόμενη επανάληψη. Εμφανώς, επιστρέφουμε την ουρά pcs ώστε να την καλέσουμε στην printOutput() και να τυπώσουμε το ζητούμενο output.

Η printOutput(), για κάθε επεξεργαστή που έχει δοθεί ως input, θα δημιουργήσει έναν προσωρινό επεξεργαστή ο οποίος θα παίρνει και θα αφαιρεί από την ουρά εκείνον με την υψηλότερη προτεραιότητα ώστε να βρίσκονται με αύξουσα σειρά ως προς τον χρόνο που ήταν ενεργός ο καθένας. Τυπώνουμε λοιπόν για καθέναν από αυτούς τα ζητούμενα outputs (με τη μορφή “id x, load=y: k,l,m…”). Ειδικά για το load, χρησιμοποιήσαμε τη μέθοδο getActiveTime() που υπάγεται στην Processor.java. Με τη χρήση ενός Node tmp, το οποίο αντιστοιχεί στο head στοιχείο της λίστας processedTasks, προσθέτουμε κάθε φορά τα data του στο άθροισμα του χρόνου. Τόσο την κλάση Node όσο και την List τις έχουμε δουλέψει στο εργαστήριο, από όπου αντλήσαμε μέρος του κώδικα.

Ως προς το makespan, για κάθε επανάληψη επεξεργαστή τσεκάρουμε εάν ο συνολικός χρόνος επεξεργασίας είναι μεγαλύτερος από το makespan και αν είναι θέτουμε το makespan να έχει την τιμή του.

Β. Αλγόριθμος ταξινόμησης Quicksort

Για τον αλγόριθμο Quicksort κάναμε χρήση πίνακα. Συγκεκριμένα, κάνουμε χρήση μίας μεθόδου διαμέρισης (partition) καθώς και μίας μεθόδου swap.

Στην quicksort έχουμε ένα στοιχείο του πίνακα και το τοποθετούμε στη μεταβλητή pivot στη θέση a[i]. Διατάσσουμε τον πίνακα σε δύο υπο-πίνακες ώστε ο πρώτος να έχει στοιχεία που είναι μεγαλύτερα ή ίσα και ο άλλος στοιχεία που είναι μικρότερα ή ίσα του pivot. Στη συνέχεια κάνουμε αναδρομές για κάθε υποπίνακα.

Για την υλοποίηση της διαμέρισης, σαρώνουμε τον πίνακα από αριστερά προς τα δεξιά μέχρι να βρούμε το στοιχείο μεγαλύτερο ή ίσο του a[r] (pivot), και σαρώνουμε από δεξιά προς τα αριστερά παράλληλα μέχρι να βρούμε το ίδιο στοιχείο. Τότε, κάνουμε swap και συνεχίζουμε μέχρι να συναντηθούν οι αριθμοδείκτες.

Ο παραπάνω αλγόριθμος μας βοήθησε στην υλοποίηση της greedyDec(). Δημιουργούμε έναν πίνακα με στοιχεία τους χρόνους επεξεργασίας κάθε task. Στη συνέχεια, εφαρμόζουμε την quicksort στον πίνακα καθώς και τον αλγόριθμο greedy και έτσι μας επιστρέφεται μία ουρά με διεργασίες με σειρά από μεγαλύτερη προς μικρότερη.

Γ. Πειραματική αξιολόγηση αλγορίθμων

Το αναμενόμενο αποτέλεσμα της αξιολόγησής μας είναι η Quicksort(αλγόριθμος Greedy-Decimal) να είναι προφανώς πιο γρήγορο από τον αλγόριθμο Greedy.

Για το μέρος Δ, δημιουργήσαμε τη συνάρτηση newFile, η οποία δημιουργεί αρχεία τύπου simple.txt και τα αποθηκεύει όταν τρέχει το πρόγραμμά μας, στον φάκελο data. Επιπλέον, έχουμε τοποθετήσει στο αρχείο Comparisons.java τις δύο συναρτήσεις Greedy και GreedyDec. Έχουμε, επίσης, δύο νέες μεταβλητές Gspan και GDspan για τα δύο makespan αντίστοιχα, οι οποίες αναθεωρούνται κάθε φορά για να βγάλουν το σωστό αποτέλεσμα. Ο τρόπος με τον οποίον δημιουργούμε και τεστάρουμε μέσω της Test() τα αρχεία είναι σχετικά απλός. Έχουμε έναν πίνακα με τις τιμές του N =100,500,1000 και κάνουμε επαναλήψεις για κάθε N δημιουργώντας 10 αρχεία και μετά βάζοντάς τα να κάνουν ένα τεστ ώστε να τρέξουν τους δύο αλγορίθμους και να τυπώσουμε τα makespans κάθε φορά. Η Test() παίρνει σαν όρισμα το συγκεκριμένο όνομα του txt file και καλεί τους αλγορίθμους και όπου al==1, τρέχει τον Greedy και όπου al==2 τον Greedy-Decimal.

Ο αλγόριθμος greedyDec() είναι αντίστοιχος σε λογική με τον greedyOne() με τη μόνη διαφορά ότι μετά την εισαγωγή των στοιχείων processor,taskN δημιουργούμε έναν πίνακα μεγέθους taskN και εκτελούμε την quicksort ώστε να γίνει η ταξινόμηση. Στη συνέχεια εκτελούμε κανονικά το πρόγραμμα και επιστρέφουμε την ουρά pcs.

Η υλοποίηση της Quicksort γνωρίζουμε ότι είναι αρκετά αποδοτική καθώς έχει πολυπλοκότητα O(NlogN). Η πολυπλοκότητα της Priority Queue είναι ίδιας τάξης, καθώς η insert() και η getmax() έχουν πολυπλοκότητα O(logN) και εκτελούνται για N processors συνεπώς συνολικά έχουμε πάλι O(NlogN). Όμως, στην quicksort δεν χρειάζεται πρόσθετος χώρος μνήμης, ενώ στον αλγόριθμο Greedy όπου χρησιμοποιούμε Priority Queues, υποχρεωτικά θα δεσμεύσουμε μνήμη όση το πλήθος των processors.

Δυστυχώς μας παρουσιάστηκε error NullPointerException στην MaxPQ όταν έτρεχε για τέτοιο capacity ο Processor και για το λόγο αυτό το πρόγραμμά μας Comparisons.java δεν κατάφερε να τρέξει όπως θα θέλαμε για να παρουσιαστούν τα δεδομένα μας. Έτσι, δεν έχουμε απτά παραδείγματα για να τεκμηριώσουμε τα λεγόμενά μας. Γενικά όμως, η Quicksort θα είναι λίγο πιο γρήγορη από την υλοποίηση με ουρά προτεραιότητας.

Δ. Πώς τρέχει ο κώδικας

Στο Β Μέρος κανονικά στην terminal με τις εντολές:
>javac Greedy.java
>java Greedy data/simple.txt

Στο Δ Μέρος τρέχει στην terminal με τις εντολές:
>javac Comparisons.java
>java Comparisons
Αν και όπως προαναφέραμε παρουσιάζει error ο κώδικάς μας.