ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

Τμημα ΗΜΜΥ. Τομεας Ηλεκτρονικης

Παραλληλα και Διανεμημενα Συστηματα

Εργασία 2: Distributed bitonic sort

Συστήματα κατανεμημένης μνήμης

Συντάκτης:
Χρήστος Χουτουρίδης [8997]
cchoutou@ece.auth.gr

Διδάσκων:
Νικόλαος Πιτσιάνης
nikos.pitsianis@eng.auth.gr

8 Ιανουαρίου 2025

1. Εισαγωγη

Η παρούσα εργασία αφορά τον παραλληλισμό συστημάτων με κατανεμημένη μνήμη και το μοντέλο μεταβίβασης μηνυμάτων (MPI). Το αντικείμενο με το οποίο ασχολούμαστε είναι ο αλγόριθμος ταξινόμησης **bitonic sort**, το οποίο έχουμε κατανέμει σε διαδικασίες που τρέχουνε σε διαφορετικούς υπολογιστές. Με αυτό τον τρόπο, κάθε διαδικασία έχει ένα μέρος των δεδομένων και τη θέση (διεύθυνση) που πρόκειται να ταξινομηθούν και ανταλλάσει δεδομένα με τις υπόλοιπες διαδικασίες. Αν και η συνήθης μορφή του αλγορίθμου είναι αναδρομική, στην παρούσα εργασία έχουμε ανοίξει την αναδρομή σε βρόχο επανάληψης. Ο βρόχος δημιουργεί το δίκτυο ταξινόμησης, όπου σε κάθε στάδιο οι διαδικασίες ανταλλάσουν δεδομένα με έναν, δύο, τέσσερις και ούτω καθεξής γείτονες, των οποίων οι διευθύνσεις διαφέρουν κατά hamming distance ίση με $1, 2, 4$ και ούτω καθεξής. Με άλλα λόγια ακολουθούν τις ακμές ενός υπερκύβου αντίστοιχης διάστασης με την δυαδική τάξη μεγέθους των διαδικασιών.

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.

3. Υλοποιηση

Πριν ξεκινήσουμε με τις λεπτομέρειες του αλγόριθμου και της υλοποίησης καλό θα ήταν να περιγράψουμε τις βασικές συναρτήσεις και δομές δεδομένων που χρησιμοποιούνται στην εργασία, ώστε να βοηθήσουμε στην καλύτερη κατανόηση της υλοποίησης.

- **`distBitonic()`**: Πρόκειται για την βασική συνάρτηση που υλοποιεί τον αλγόριθμο της εργασίας.
- **`distBubbletonic()`**: Πρόκειται για την βασική συνάρτηση που υλοποιεί τον αλγόριθμο της έκδοσης **v0.5**. Ο αναγνώστης μπορεί να πειραματιστεί με αυτή την έκδοση καθώς υπάρχουν make rules, αλλά δεν κρύβει κάποιον "άσο στο μανίκι", καθώς η βελτιστοποίηση έγινε με γνώμονα την *distBitonic()*.
- **`MPI_t`**: Ο τύπος αυτός δημιουργεί ένα επίπεδο αφαίρεσης γύρω από την MPI επικοινωνία, αρχικοποίηση και διαχείριση πόρων. Κάθε αντικείμενο αυτού του τύπου μπορεί να διαχειριστεί σύγχρονη ή ασύγχρονη επικοινωνία. Σε περίπτωση που παρουσιαστεί σφάλμα στην επικοινωνία, η εκτέλεση του προγράμματος τερματίζεται με κατάλληλη διαχείριση των πόρων του MPI.
- **`ShadowedVec_t`**: Ο τύπος αυτός προσφέρει το interface ενός **std::vector**, ενώ εσωτερικά περικλείει δύο. Ένα **`ενεργό`** και ένα ως **`σκιά`** του ενεργού. Ο λόγος είναι για να προσφέρει εναλλακτικό χώρο αποθήκευσης για τα εισερχόμενα δεδομένα κατά την ανταλλαγή στην MPI επικοινωνία, αλλά και για τον αλγόριθμο ταξινόμησης elbow-sort, ο οποίος δεν μπορεί να λειτουργήσει *"in-place"* καθώς χρειάζεται ξεχωριστό πίνακα προορισμού. Με την αφαίρεση αυτή προσφέρουμε ένα κοινό interface που καλύπτει και τις δύο περιπτώσεις.
- **`Timing`**: Η τάξη αυτή προσφέρει δυνατότητες χρονομέτρησης μίας ή και πολλαπλών κλήσεων αθροίζοντας τους χρόνους.

3.1. Model version

Η πρώτη έκδοση του αλγόριθμου ήταν μια ένα-προς-ένα αντιστοιχία με το μοντέλο που δημιουργήθηκε στη julia. Στην έκδοση αυτή όλες οι επικοινωνίες γίνονται με τις blocking εκδόσεις του MPI. Τα βασικά βήματα αυτής της έκδοσης για κάθε μία MPI διαδικασία είναι:

- Ταξινόμηση των δεδομένων τοπικά χρησιμοποιώντας κάποιον "ακριβό" αλγόριθμο.
- Ανταλλαγή όλων των δεδομένων με την διαδικασία - εταίρο (blocking).

- Διαχωρισμός ελαχίστων – μεγίστων, όπου η διαδικασία κρατάει τα μεν ή τα δε.
- Ταξινόμηση με την elbow-sort.

Έτσι σε κάθε επανάληψη του βρόχου η κάθε MPI διαδικασία ανταλλάσει **όλα** τα δεδομένα με τον εταίρο της και **αφού** τελειώσει η επικοινωνία, **τότε** εκτελείται ο διαχωρισμός και η ταξινόμηση. Προφανώς και μόνο από τη διατύπωση της προηγούμενης πρότασης κάποιος θα σκεφτόταν ότι υπάρχουν τόσες ευκαιρίες να κάνουμε τα πράγματα καλύτερα. Να μειώσουμε τις επικοινωνίες, να τις κάνουμε ασύγχρονες, κλπ... Παραθέτοντας τον Donald E. Knuth[1] που αναφέρει πως: *"premature optimization is the root of all evil"*, θα πρέπει να τονίσουμε πως για να επιλέξουμε τι από όλα πρέπει να βελτιστοποιήσουμε, πρέπει πρώτα να μετρήσουμε.

Αρχικά λοιπόν, αναλύσαμε την εκτέλεση του προγράμματος τοπικά με 4 MPI tasks χρησιμοποιώντας το perf, το οποίο έδειξε ότι όσο μεγάλωνε το μέγεθος των δεδομένων τόσο ο **κυρίαρχος παράγοντας γινόταν η full sort**. Για την ακρίβεια:

	Full Sort	Elbow Sort	MPI-exchange	Min-Max	Data Gen	Data Alloc
q=20	33.67%	9.07%	1.96%	<0.5%	2.96%	1.1%
q=26	**67.21%**	13.63%	4.63%	2.05%	5.2%	1.74%

Για να υποστηρίξουμε τα παραπάνω δεδομένα χρησιμοποιήσαμε και την συστοιχία batch. Για την ακρίβεια μετρήσαμε τόσο για $q = 20$, όσο και για $q = 27$, με 4 διεργασίες σε διάταξη 1node:4process και 4nodes:1process.

	Total	Full Sort	Elbow Sort	MPI-exchange	Min-Max
N1P4 - q=20	138 ms	94 ms	26 ms	14.5 ms	2.54 ms
N1P4 - q=27	20 s	**15.5 s**	3.44 s	**750.5 ms**	425.3 ms
N4P1 - q=20	151.7 ms	93.3 ms	26 ms	29.7 ms	1.56 ms
N4P1 - q=27	20 s	**15.5 s**	3.43 s	**531.3 ms**	336.3 ms

Τα παραπάνω δεδομένα επιβεβαιώνουν πως ο κυρίαρχος παράγοντας είναι η full sort που είναι $O(nlog(n))$, σε σχέση με όλα τα υπόλοιπα που είναι $O(n)$. Συμπεραίνουμε λοιπόν πως η αρχική "ακριβή" ταξινόμηση είναι πρακτικά το πρώτο πράγμα που πρέπει να βελτιστοποιήσουμε. Ακόμα τα δεδομένα φαίνεται να υποστηρίζουν πως η ανταλλαγή δεδομένων δεν επηρεάζεται τόσο από το αν οι διεργασίες επικοινωνούν σε κοινή μνήμη ή μέσω δικτύου. Το γεγονός ότι η συστοιχία "rome" έχει ακόμα γρηγορότερο interface δικτύου υποστηρίζει αυτή την υπόθεση.

3.2. Παραλληλοποίηση της full sort

Για την αρχική ταξινόμηση έχουμε χρησιμοποιήσει τη συνάρτηση βιβλιοθήκης std::sort(), η οποία χρησιμοποιεί την **introsort**. Μια υβριδική υλοποίηση quick-heap-insertion sort που εγγυάται $O(nlog(n))$ ακόμα και ως worst case. Για να τη βελτιστοποιήσουμε λοιπόν, απλώς την κάναμε παράλληλη. Θεωρώντας ότι η παραλληλοποίηση σε κοινή μνήμη δεν είναι το κύριο μέλημα της παρούσας εργασίας, απλώς χρησιμοποιήσαμε την openmp έκδοση της βιβλιοθήκης *__gnu_parallel::sort()*.

Με αυτή την αλλαγή και χρησιμοποιώντας 4 threads ανά MPI process, οι χρόνοι για το $q = 27$ πέσαν στην batch περίπου στα $4sec$ ρίχνοντας έτσι και το συνολικό χρόνο περίπου στα $7.5sec$. Με αυτό τον τρόπο μπορούμε να πάρουμε όση επιτάχυνση θέλουμε επιλέγοντας απλά αριθμό threads ανά MPI process. Ο περιοριστικός παράγοντας είναι οι πόροι που πρέπει να καταναλώσουμε στην συστοιχία για τις τελικές μετρήσεις. Με βάση τα παραπάνω, τα **4 threads** φαίνεται ένα καλό νούμερο, καθώς δεν αυξάνει πολύ τους πόρους και ταυτόχρονα προσφέρει μια ισορροπία μεταξύ της fullsort και του υπόλοιπου αλγόριθμου.

[1] Donald E. Knuth, *Structured Programming with go to Statements*, ACM Computing Surveys, vol. 6, no. 4, 1974

3.3. Βελτιστοποίηση του MPI