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

15 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**, τον οποίο έχουμε κατανέμει σε διαδικασίες που τρέχουνε σε διαφορετικούς υπολογιστές. Στη κάθε διαδικασία δεν έχουμε αναθέσει ένα αντικείμενο, αλλά ένα μέρος των δεδομένων και τη θέση (διεύθυνση) που αυτά πρόκειται να ταξινομηθούν. Η κάθε διαδικασία ανταλλάσει δεδομένα με τις υπόλοιπες διαδικασίες ακολουθώντας τη λογική του αλγόριθμου bitonic, αλλά όσον αφορά τα δεδομένα που βρίσκονται τοπικά, αυτά τα ταξινομεί με τον βέλτιστο δυνατό τρόπο. 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.

Όλοι οι παραπάνω τύποι είναι templates, και βρίσκονται στα αντίστοιχα headers (.hpp) αντί για αρχεία .cpp. Η συνάρτηση *main()* που υλοποιεί τον αλγόριθμο, αλλά και αυτή που καλεί τα unit tests, μαζί με τον validator και το command line interface βρίσκονται στο αρχείο main.cpp.

3.1. Model version

Η πρώτη έκδοση του αλγόριθμου που υλοποιήθηκε 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 της διτονικής ακολουθίας που προκύπτει με την elbow-sort.

Το πρώτο βήμα εκτελείται μόνο μία φορά, αλλά τα τελευταία τρία εκτελούνται σε βρόχο. 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 (68.1%)	26 ms	14.5 ms	2.54 ms
N1P4 - q=27	20 s	**15.5 s (77.5%)**	3.44 s	**750.5 ms**	425.3 ms
N4P1 - q=20	151.7 ms	93.3 ms (61.5%)	26 ms	29.7 ms	1.56 ms
N4P1 - q=27	20 s	**15.5 s (77,5%)**	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.

Φυσικά, αν ο αριθμός των διεργασιών μεγαλώσει πάρα πολύ τότε περιμένουμε ότι οι διαδικασίες elbow-sort, MPI-exchange και Min-Max, που είναι και το κομμάτι που αφορά τον αλγόριθμο bitonic sort, θα αρχίσει να παίζει σημαντικό ρόλο. Για την παρούσα εργασία βέβαια ο αριθμός των διεργασιών θα κινηθεί σε μικρά νούμερα.

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

3.2. Παραλληλοποίηση της full sort

Για την αρχική ταξινόμηση έχουμε χρησιμοποιήσει τη συνάρτηση βιβλιοθήκης std::sort(), η οποία είναι μια υλοποίηση της **introsort**. Μια υβριδική υλοποίηση quick-heap-insertion sort που εγγυάται $O(nlog(n))$ σε όλα τα σενάρια. Για να τη βελτιστοποιήσουμε λοιπόν, απλώς την κάναμε παράλληλη. Θεωρώντας ότι η παραλληλοποίηση σε κοινή μνήμη δεν είναι το κύριο μέλημα της παρούσας εργασίας, αντί να παραλληλοποιήσουμε έναν αλγόριθμο με το χέρι, 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 $\boldsymbol{q = 27}$ πέσαν στην batch περίπου στα $\boldsymbol{4sec}$ ρίχνοντας έτσι και το συνολικό χρόνο περίπου στα $\boldsymbol{7.5sec}$. 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, ο αριθμός των **4 threads** φαίνεται ένα καλό νούμερο, καθώς δεν αυξάνει πολύ τους πόρους και ταυτόχρονα προσφέρει μια ισορροπία μεταξύ της fullsort και του υπόλοιπου αλγόριθμου.

3.3. Βελτιστοποίηση του MPI

Όπως τονίσαμε και παραπάνω, το κομμάτι που αφορά τον αλγόριθμο bitonic sort, θα αρχίσει να παίζει ρόλο, όσο ο αριθμός των διαδικασιών μεγαλώνει αρκετά. Εκεί αναμένουμε πως το κρίσιμο μέρος είναι η MPI επικοινωνία, καθώς τα υπόλοιπα είναι $O(n)$ και δύσκολα μπορούμε να τα κάνουμε καλύτερα. Έτσι, παρόλο που με την βελτιστοποίηση της full-sort θα μπορούσαμε να έχουμε ένα τελικό "προϊόν", και δεδομένου πως η εργασία αφορά το MPI, μπήκαμε στη λογική να προσπαθήσουμε κάποιες βελτιστοποιήσεις.

3.4. MPI localization

Στο δίκτυο ταξινόμησης της bitonic, σε κάθε επανάληψη τα nodes επικοινωνούν με όλο και ποιο μακρινούς γείτονες, αλλά πάντα καταλύγουν να επικοινωνούν με τους άμεσους. Αυτό σημαίνει πως οι επικοινωνίες με τις κοντινές διευθύνσεις είναι περισσότερες από αυτές με τις μακρινές. Αν λοιπόν για το κάθε node, οι κοντινές διευθύνσεις ήταν στον ίδιο υπολογιστή της συστοιχίας τότε οι κοντινές επικοινωνίες θα γινόταν στη RAM, ενώ οι μακρινές μέσω ethernet. Εφόσον η κατανομή των διευθύνσεων του MPI δεν είναι στο χέρι μας, μπήκαμε στη λογική να δημιουργήσουμε ένα mapping, όπου οι διευθύνσεις των διαδικασιών για τον αλγόριθμο δεν είναι τα ranks του mpi, αλλά ένας άλλος αριθμός που τον δημιουργήσαμε εμείς, ώστε όλοι οι κοντινοί γείτονες να είναι στον ίδιο υπολογιστή. Αυτό έγινε πολύ εύκολα. Απλά δημιουργήσαμε μια λίστα με όλα τα ονόματα των υπολογιστών και τα rank τους, και την ταξινομήσαμε πρώτα ως προς τα ονόματα και μετά ως προς τα ranks. Μετά χρησιμοποιήσαμε αυτή τη λίστα για μετατροπή από την rank διεύθυνση στην localized σε $O(1)$.

Δυστυχώς τρέχοντας αυτή την έκδοση στη συστοιχία, δεν είδαμε **καμία διαφορά**. Μάλιστα σε πολλές περιπτώσεις η επικοινωνία μέσω ethernet ήταν πιο γρήγορη από αυτή στο ίδιο μηχάνημα. Αυτό ίσως δικαιολογείται από το γεγονός ότι η συστοιχία έχει πολύ γρήγορο ethernet interface και τα δεδομένα που ανταλλάσουμε δεν είναι πολύ μεγάλα. Σε κάθε περίπτωση, εγκαταλείψαμε τελείως αυτή τη βελτιστοποίηση.

3.5. Ελαχιστοποίηση της επικοινωνίας MPI

Κατά τη διάρκεια εκτέλεση τους αλγόριθμου και για να είναι σε θέση η κάθε διεργασία να διαχωρίσει τα μικρά ή τα μεγάλα στοιχεία της με κάποιο γείτονα, θα πρέπει να δημιουργήσει ένα αντίγραφο των δεδομένων του γείτονα, τοπικά. Φυσικά κατά την διαδικασία ανταλλαγής και διαχωρισμού, κάποια από τα δεδομένα της ακολουθίας παραμένουν ως είχαν. Είναι ήδη μεγαλύτερα ή μικρότερα από τα αντίστοιχα του γείτονα. Θα περίμενε λοιπόν κάποιος να υπάρχει ένας τρόπος να αποφευχθεί η μεταφορά δεδομένων που θα παραμείνουν. Πράγματι, όπως αποδεικνύεται και στο παράρτημα A′.1, αν έχουμε δύο **ταξινομημένες** ακολουθίες που η μία θα κρατήσει τα μέγιστα L_i και η άλλη θα κρατήσει τα ελάχιστα S_i, τότε τα δεδομένα που θα καταλήξουν να χρειάζονται ανταλλαγή **για την κάθε ακολουθία** βρίσκονται στο υποσύνολο επικάλυψης:

$$E = \{l_x\}_{x=i}^{j} : \quad L_{min} \leq l_x \leq S_{max}$$

Δηλαδή στο κομμάτι της ακολουθίας με τα στοιχεία που είναι ***μεγαλύτερα από το μικρότερο αυτής που κρατάει τα μεγάλα*** L_{min} ***και μικρότερα από το μεγαλύτερο αυτής που κρατάει τα μικρά*** S_{max}. Αν το σύνολο επικάλυψης E_1, E_2 των ακολουθιών:

- Είναι **κενό** και για τις δύο ακολουθίες, **καμία ανταλλαγή δεν χρειάζεται να γίνει**, και κατ' επέκταση ούτε να εκτελεστεί ο διαχωρισμός μικρών-μεγάλων *keepMinOrMax()*.
- Εμπεριέχει όλα τα στοιχεία των ακολουθιών, τότε η ανταλλαγή πρέπει να γίνει με όλα τα στοιχεία κανονικά.
- Είναι **υποσύνολο** των ακολουθιών, τότε οι ακολουθίες πρέπει να ανταλλάξουν και να εφαρμόσουν τον διαχωρισμό μικρών-μεγάλων στην **τομή** $E_1 \cap E_2$.

Δυστυχώς για να δουλέψει η παραπάνω βελτιστοποίηση οι ακολουθίες **πρέπει να είναι ταξινομημένες**. Στη δική μας περίπτωση, κάθε φορά που εκτελείται η exchange τα δεδομένα έχουν τη μορφή διτονικής, αλλά δεν είναι απαραίτητα ταξινομημένα σε όλα τα βήματα. Αυτό σημαίνει ότι δεν μπορούμε να εφαρμόσουμε πλήρως τον παραπάνω συλλογισμό για την περίπτωση που το σύνολο επικάλυψης είναι υποσύνολο των ακολουθιών. Μπορούμε όμως να τον εφαρμόσουμε στη περίπτωση που είναι κενό.

Για το σκοπό αυτό, πριν από την ανταλλαγή δεδομένων, οι διαδικασίες ανταλλάσουν τις πληροφορίες των ορίων τους (min/max) και με βάση αυτές υπολογίζουν αν η ανταλλαγή χρειάζεται να γίνει ή όχι. Αυτό μειώνει ναι μεν τις επικοινωνίες, χωρίς να σημαίνει ότι μειώνει κατά την ίδια αναλογία και τον συνολικό χρόνο ανταλλαγής του αλγόριθμου. Αυτό γιατί ενώ μια διαδικασία μπορεί να αποφύγει μια ανταλλαγή και να πάει στο επόμενο στάδιο ξεκινώντας ανταλλαγή με τον γείτονα του επόμενου βήματος, αυτός ο γείτονας μπορεί να μην είχε γλιτώσει την ανταλλαγή και άρα η διαδικασία να περιμένει αυτό τον χρόνο ούτως ή άλλως.

Θεωρούμε πως αν σε κάθε βήμα οι ακολουθίες δεν ήταν απλά διτονικές, αλλά ταξινομημένες, τότε η πιθανότητα οι επικοινωνίες να ήταν μειωμένες θα μεγάλωνε και άρα ο αντίκτυπος θα ήταν μεγαλύτερος. Αυτό όμως θα το πληρώναμε εκτελώντας elbow-sort σε κάθε βήμα. Το ποια από τις δύο λύσεις είναι αποδοτικότερη δεν το διερευνήσαμε, καθώς θεωρούμε ότι ξεφεύγει από τα πλαίσια αυτής της εργασίας.

3.6. MPI pipeline

Η τελευταία βελτιστοποίηση που δοκιμάσαμε για την επικοινωνία ήταν η δημιουργία ενός pipeline για την επικοινωνία και τον διαχωρισμό μεγίστων-ελαχίστων. Καθώς μεγαλώνει ο αριθμός των διαδικασιών, ο συνολικός χρόνος που αφιερώνει ο αλγόριθμός μας στην επικοινωνία και στον διαχωρισμό αυξάνεται. Στον αλγόριθμό μας ο διαχωρισμός λαμβάνει χώρα **μετά** την ολοκλήρωση ολόκληρης της επικοινωνίας. Μπήκαμε λοιπόν στη λογική να δημιουργήσουμε μια επικάλυψη αυτών των δύο λειτουργιών. Για το σκοπό αυτό χρησιμοποιήσαμε τις ασύγχρονες εκδόσεις της επικοινωνίας *MPI_ Isend()*, *MPI_ Irecv()* και *MPI_ Wait()*. Έτσι χωρίζουμε την επικοινωνία σε **τμήματα** και μετά την μεταφορά δεδομένων του πρώτου τμήματος, εκκινούμε την μεταφορά για το δεύτερο και εκτελούμε τον διαχωρισμό μεγίστων-ελαχίστων στο κομμάτι που παραλάβαμε.

Στην περίπτωση που επιλεγεί ένα στάδιο, τότε ο αλγόριθμος καταλήγει να δουλεύει όπως ο blocking, με τη μόνη διαφορά ότι αντί για την εντολή *MPI_ SendRecv()*, χρησιμοποιείται το ζευγάρι *{MPI_ Isend(), MPI_ Irecv()} - MPI_ Wait()*, χωρίς να εκτελείται κάποιος κώδικας ενδιάμεσα.

3.7. Σύγκριση υλοποιήσεων

Φυσικά, πριν υιοθετήσουμε και εφαρμόσουμε στις τελικές εκτελέσεις κάποια από τις παραπάνω βελτιστοποιήσεις, θα πρέπει να μετρήσουμε και να επιβεβαιώσουμε τον αντίκτυπό της. Γιαυτό το λόγο όλες οι παραπάνω λύσεις υλοποιήθηκαν μία μία και δοκιμάστηκαν ξεχωριστά στη συστοιχία. Στον παρακάτω πίνακα παραθέτουμε συγκεντρωτικά τα σημαντικότερα αποτελέσματα από τις μετρήσεις.

Ο συμβολισμός N**x**P**y** σημαίνει **x**:Nodes, **y**:Process per node και άρα $p = \log_2(xy)$.

Έκδοση	Configuration	Total	Full Sort	Elbow Sort	MPI-exchange	Min-Max
Vanila	N1P2 - q=20	28 ms	19.5 ms	4.67 ms	2.7 ms	1.1 ms
	N2P4 - q=20	54.4 ms	19 ms	13 ms	15.6 ms	5.62 ms
	N4P8 - q=20	126.5 ms	19.3 ms	22.2 ms	69.6 ms	14.2 ms
	N1P2 - q=27	4824 ms	3645 ms	633.5 ms	379.5 ms	165 ms
	N2P4 - q=27	5643 ms	3254 ms	1129.6 ms	1129.6 ms	883.6 ms
	N4P8 - q=27	**15.37 s**	3459 ms	3090 ms	6792 ms	2262.8 ms
Exch. Optimization	N1P2 - q=20	~~191 ms~~	~~72.5 ms~~	~~4.76 ms~~	~~109 ms~~	~~1.56 ms~~
	N2P4 - q=20	**70.6 ms**	20.7 ms	14.7 ms	20.1 ms	5.22 ms
	N4P8 - q=20	126.9 ms	19.8 ms	22.9 ms	**52.5** ms	14.2 ms
	N1P2 - q=27	**4906 ms**	3518 ms	661 ms	181.5 ms	179 ms
	N2P4 - q=27	**9569 ms**	3652 ms	2058.7 ms	2026 ms	721 ms
	N4P8 - q=27	15.84 s	3512 ms	3187 ms	**5908** ms	**1731** ms
Pipeline = 1	N1P2 - q=20	**33 ms**	22 ms	6.11 ms	3.78 ms	1.47 ms
	N2P4 - q=20	**258.6 ms**	20.5 ms	14.5 ms	214.5 ms	6.41 ms
	N4P8 - q=20	**211 ms**	21.4 ms	24.4 ms	146.9 ms	16.7 ms
	N1P2 - q=27	**4500 ms**	3474 ms	630 ms	216.5 ms	178 ms
	N2P4 - q=27	**10.62 s**	3934 ms	2111 ms	3253 ms	1325.1 ms
Pipeline = 8	N4P8 - q=27	**21.67 s**	3837 ms	3412 ms	14.1 s	3390 ms
	N4P16 - q=27	27.8 s	3980 ms	3986 ms	19.2 s	4952 ms

Από τα παραπάνω συμπεραίνουμε πως:

- Στην ουσία για το μέγεθος των δεδομένων που εκτελέσαμε $q = 20 : 27, p = 1 : 6$, η έκδοση χωρίς καμία βελτιστοποίηση στην επικοινωνία είναι η καλύτερη επιλογή.
- Αν μεγαλώνει αρκετά ο αριθμός των επικοινωνιών (δηλαδή όσο μεγαλώνει το p), η ελαχιστοποίηση της επικοινωνίας δείχνει πως αρχίζει να παίζει ρόλο.
- Το pipeline ακόμα και στην περίπτωση που είναι 1, δηλαδή θεωρητικά ισοδύναμο με τον blocking αλγόριθμο, δείχνει ότι είναι πολύ αργό. Αυτό θεωρούμε ότι οφείλεται στο γεγονός ότι για την ασύγχρονη επικοινωνία το MPI δημιουργεί ξεχωριστά requests στην ουρά, ενώ στην επιλογή της *MPI_SendRecv()*, όλη η ανταλλαγή δεδομένων γίνεται με τρόπο διαφανή στο MPI και άρα με τρόπο που μπορεί να το βελτιστοποιήσει εσωτερικά.
- Οι μετρήσεις παρουσιάζουν πολύ μεγάλες παρεκκλίσεις. Αυτό θεωρούμε ότι συμβαίνει γιατί κατά τη διάρκεια των μετρήσεων, η συστοιχία είχε μεγάλο φόρτο.

4. Τελικη εκδοση

Έτσι λοιπόν, για την τελική έκδοση του αλγόριθμού μας επιλέγουμε να έχουμε **κατ' επιλογή** είτε το exchange optimization είτε το pipeline, με τέτοιο τρόπο ώστε όταν δεν επιλέγονται ο αλγόριθμος να συμπεριφέρεται όπως η απλή έκδοση. Για τις μετρήσεις αποφασίσαμε να **μην χρησιμοποιήσουμε pipeline** και να χρησιμοποιήσουμε **exchange optimization μόνο για** τις περιπτώσεις που $\boldsymbol{p = 6 : 7}$.

Στον κατάλογο hpc, βρίσκονται τα sbatch scripts που χρησιμοποιήθηκαν για τις τελικές μετρήσεις. Ο αναγνώστης μπορεί επίσης τοπικά να μεταγλωττίσει, να τρέξει τα tests και να εκτελέσει τον αλγόριθμο bitonic 8 φορές για 2^{20} δεδομένα για παράδειγμα, δίνοντας από το root directory:

```
$ make -j hpc-build
```

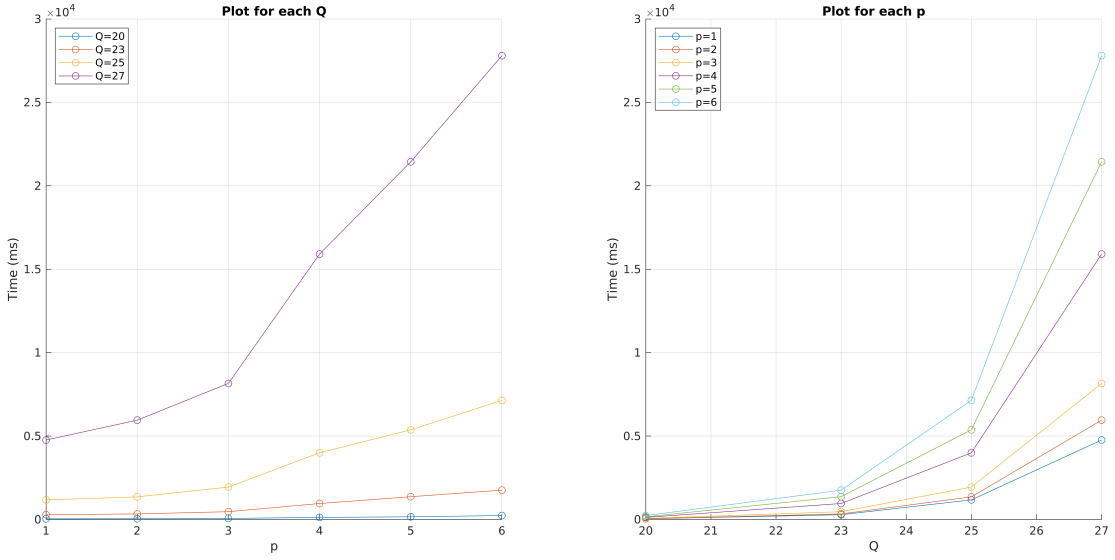
```
$ export OMP_NUM_THREADS=4     # optional to enable 4 threads per MPI task
$ mpirun -np 4 ./out/tests
$ mpirun -np 4 ./out/distbitonic -q 20 --perf 8 --validation
```

Η επαλήθευση του αλγόριθμου έγινε σε δύο στάδια. Πρώτα, κατά τη διάρκεια της ανάπτυξης δημιουργήθηκαν unit tests (κατάλογος tests) που χρησιμοποιήθηκαν για τη διαρκή επιβεβαίωση ότι κάποιες συναρτήσεις και ο βασικός αλγόριθμος συμπεριφερόταν σωστά. Έπειτα υλοποιήθηκε ένας validator που ενσωματώθηκε στο εκτελέσιμο και μπορεί να καλεστεί από το command line argument `--validation`, ο οποίος συλλέγει δεδομένα από όλα τα MPI process και επιβεβαιώνει ότι η ταξινόμηση έγινε σωστά.

Παρακάτω παραθέτουμε τα αποτελέσματα στους τελικούς χρόνους από τη συστοιχία rome, όπου τρέξαμε τη bitonic 8 φορές και κρατήσαμε τον ενδιάμεσο χρόνο.

	N1P2	N1P4	N2P4	N4P4	N4P8	N4P16	N4P32[2]
$Q = 20$	40 ms	42 ms	62.87 ms	123.25 ms	158.81 ms	233.52 ms	–[2]
$Q = 23$	285 ms	331.7 ms	465.5 ms	956.7 ms	1364.5 ms	1755.5 ms	–[2]
$Q = 25$	1174 ms	1354.5 ms	1940.4 ms	3994.2 ms	5373.8 ms	7139.9 ms	–[2]
$Q = 27$	4761.5 ms	5952.5 ms	8154.7 ms	15.91 sec	21.44 sec	27.80 sec	–[2]

Στο σχήμα 1 φαίνεται πως επηρεάζονται οι συνολικοί χρόνοι εκτέλεσης από τον αριθμό των MPI process και από το μέγεθος των δεδομένων ξεχωριστά.

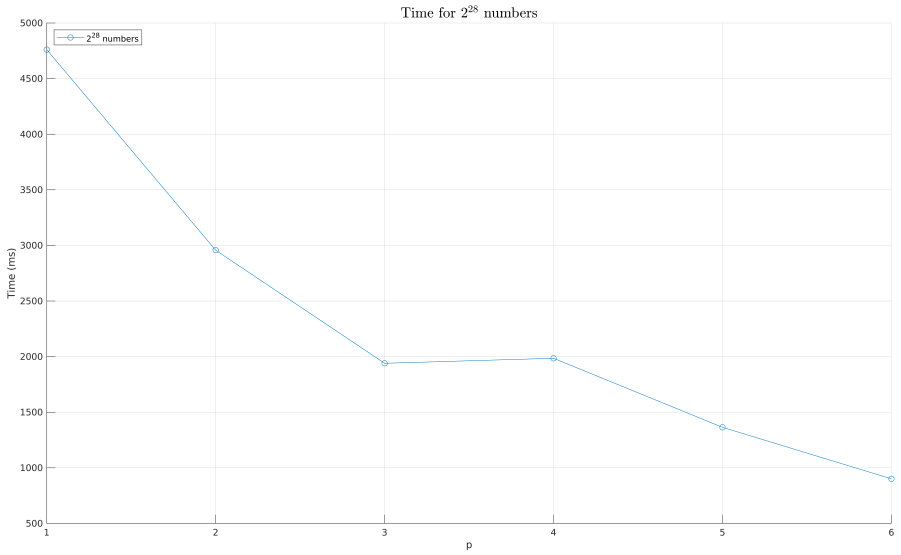


Σχήμα 1: Χρόνοι εκτέλεσης ως προς **p** και **q**.

Ενδιαφέρον παρουσιάζει να δούμε τον συνολικό χρόνο εκτέλεσης για σταθερό συνολικό αριθμό δεδομένων. Για παράδειγμα για 2^{28} 32-bit αριθμούς διαμοιρασμένους σε $2, 4, 8, ..., 128$ MPI processes έχουμε:

N1P2-Q27	N1P4-Q26	N2P4-Q25	N4P4-Q24	N4P8-Q23	N4P16-Q22	N4P32-Q21 [2]
4761.5 ms	2957.5 ms	1940.4 ms	1985.2 ms	1364.5 ms	901.1 ms	– [2]

[2]Μέχρι και την ημερομηνία της σύνταξης της αναφοράς η συστοιχία δεν έχει εκτελέσει τα sbatch scripts για τις εν λόγο μετρήσεις. Ο αναγνώστης παρ' όλα αυτά μπορεί να βρει εδώ μια *μελλοντική έκδοση* της αναφοράς η οποία όταν εκτελεστούν τα scripts θα συμπεριλαμβάνει τις μετρήσεις αυτές.



Σχήμα 2: Χρόνοι εκτέλεσης 2^{28} αριθμών ως προς **p**.

Α′. Παραρτημα

Α′.1. Υπολογισμός συνόλου ανταλλαγής ταξινομημένων ακολουθιών

Έστω L, S δύο ταξινομημένες ακολουθίες μεγέθους N, μία αύξουσα και μία φθίνουσα.

$$L = \{l_i\}_{i=1}^{N}, S = \{s_i\}_{i=1}^{N}, \quad l_i, s_i \in \mathbf{R}$$

Έστω f συνάρτηση η οποία παίρνει δύο ταξινομημένες ακολουθίες $L = \{l_i\}_{i=1}^{N}$ και $S = \{s_i\}_{i=1}^{N}$, και επιστρέφει δύο ακολουθίες $L' = \{l'_i\}_{i=1}^{N}$ και $S' = \{s'_i\}_{i=1}^{N}$:

$$f\colon (\mathbf{R}^N, \mathbf{R}^N) \to (\mathbf{R}^N, \mathbf{R}^N), \qquad f(L,S) = (L', S') = \left(\{\max(l_i, s_i)\}_{i=1}^{N}, \{\min(l_i, s_i)\}_{i=1}^{N}\right).$$

Όπου, η L' περιέχει τα μεγαλύτερα στοιχεία από τη σύγκριση στοιχείο με στοιχείο l_i και s_i, ενώ η S' περιέχει τα μικρότερα. Αν l_{min} το μικρότερο στοιχείο της L και s_{max} το μεγαλύτερο στοιχείο της S, τότε:

$$\exists E_L = \{e_i\}_{i=1}^{m} \subseteq L : e_i > s_{max} \geq s_i \forall i$$
$$\exists E_S = \{s_i\}_{i=1}^{m} \subseteq S : s_i < l_{min} \leq e_1 \forall i$$

Άρα:

$$f(E_L, S_L) = (\{\max(e_i, s_i)\}_{i=1}^{m}, \{\min(e_i, s_i)\}_{i=1}^{m}) = (\{e_i\}_{i=1}^{m}, \{s_i\}_{i=1}^{m}) = (E_L, E_S)$$

Επομένως υπάρχουν τα υποσύνολα $E_L \subseteq L, E_S \subseteq S$, τα οποία δεν επηρεάζονται από τη συνάρτηση f, και άρα δεν ανταλλάσουν τα στοιχεία μεταξύ τους. Έτσι τα υποσύνολα τα οποία αλλάζουν στοιχεία είναι:

$$L_{ex} = L \setminus E_L = \{l_{min}, \ldots, s_{max}\}, l_{min} \leq s_{max}$$
$$S_{ex} = S \setminus E_S = \{l_{min}, \ldots, s_{max}\}, l_{min} \leq s_{max}$$