Quicksort Parallel Implementation using MPI Library

Περιεχόμενα:

Εκφώνηση του προβλήματος:

Υλοποίηση αλγορίθμου Quicksort:
Η γρήγορη ταξινόμηση (quicksort) αποτελεί ένα κλασικό παράδειγμα αλγορίθμου της οικογενείας «Διαίρει και Βασίλευε». Στην τεχνική αυτή για να λυθεί ένα πρόβλημα το «σπάμε» σε υποπροβλήματα, λύνουμε κάθε υποπρόβλημα ξεχωριστά και συνθέτουμε τις επιμέρους λύσεις για να παράξουμε την τελική λύση του προβλήματος. Ένα ακόμα όνομα για την εν λόγω ταξινόμηση είναι το «ταξινόμηση με διαμερισμό και ανταλλαγή» (partition exchange sort). Όπως δηλώνει το πρώτο όνομα του αλγορίθμου, πρόκειται για μία πολύ αποτελεσματική μέθοδο ταξινόμησης. Όπως δηλώνει το δεύτερο όνομά του, τα βασικά χαρακτηριστικά του είναι δύο: η ανταλλαγή και ο διαμερισμός. Πρώτον, εκτελούνται ανταλλαγές μεταξύ των στοιχείων του πίνακα, έτσι ώστε τα στοιχεία με μικρότερες τιμές να μετακινηθούν προς τη μία πλευρά, ενώ τα στοιχεία με μεγαλύτερες τιμές να μετακινηθούν προς την άλλη πλευρά του πίνακα. Έτσι, ακολουθεί η εφαρμογή του διαμερισμού του πίνακα σε δύο μικρότερους που ταξινομούνται ανεξάρτητα μεταξύ τους. Είναι προφανές ότι στους δύο υποπίνακες επιφυλάσσεται η ίδια αντιμετώπιση: ανταλλαγή και διαμερισμός.

Θεωρητική παρουσίαση

Η Ταχυταξινόμηση (Quicksort) είναι αλγόριθμος ταξινόμησης ο οποίος αναπτύχθηκε από τον βρετανό Tony Hoare. Ο αλγόριθμος είναι βασισμένος στην τεχνική«διαίρει και βασίλευε». Σε κάθε βήμα του αλγορίθμου:

- Επιλέγεται ένα τυχαίο στοιχείο(pivot) A[q] του πίνακα A
- Διαίρει: τα στοιχεία του A χωρίζονται σε αυτά που είναι μικρότερα από το pivot και σε αυτά που είναι μεγαλύτερα από το pivot.
- Βασίλευε: Εφαρμόζεται αναδρομικά η ίδια διαδικασία σε κάθε ένα από τα τμήματα του A που προέκυψαν

Ψευδοκώδικας Ταχυταξινόμησης:

```
1.quickSort(A, p, r) {
2.     if (p < r) {
3.          q = Partition(A, p, r);
4.          quickSort(A, p, q-1);
5.          quickSort(A, q+1, r);
6.     }
7.}
```

Η επιλογή pivot και ο διαμερισμός(partitioning) μπορούν να γίνουν με πολλούς διαφορετικούς τρόπους. Η επιλογή συγκεκριμένων σχεδίων υλοποίησης επηρεάζει σημαντικά την απόδοση του αλγορίθμου.

Lomuto Partition:

Αυτό το σχήμα αποδίδεται στον Nico Lomuto. Σαν pivot επιλέγεται το τελευταίο στοιχείο του πίνακα. Αυτός ο αλγόριθμος διατηρεί έναν δείκτη i καθώς σαρώνει τη συστοιχία χρησιμοποιώντας ένα άλλον δείκτη j έτσι ώστε τα στοιχεία στο low έως i-1 να είναι μικρότερα από το pivot και τα στοιχεία στο i έως j να είναι ίσα ή μεγαλύτερα από το pivot.
Δεδομένου ότι αυτό το σχήμα είναι πιο συμπαγές και κατανοητό, χρησιμοποιείται συχνά για εισαγωγικό εκπαιδευτικό υλικό, αν και είναι λιγότερο αποτελεσματικό από το αρχικό σχήμα του Hoare π.χ. Στην περίπτωση που όλα τα στοιχεία είναι ίδια.
Στην χείριστη περίπτωση που ο πίνακας είναι ήδη ταξινομημένος η πολυπλοκότητα είναι O(n^2)

Ψευδοκώδικας Lomuto Partition:

```
1. partition_lomuto(A, low, high)
2.     pivot := A[high]
3.     i := low
4.    for j := low to high do
5.          if A[j] < pivot then
6.               swap A[i] with A[j]
7.               i := i + 1
8.     swap A[i] with A[high]
9.    return i
```

Hoare Partition:

Το αρχικό σχήμα που περιγράφεται από τον Tony Hoare χρησιμοποιεί δύο δείκτες που ξεκινούν από τα άκρα του πίνακα που χωρίζεται, και μετά μετακινούνται μεταξύ τους, έως ότου εντοπίσουν μια αντιστροφή: ένα ζευγάρι στοιχείων, ένα μεγαλύτερο από ή ίσο με το pivot, και ένα μικρότερο ή ίσο του pivot, που είναι σε λάθος σειρά το ένα με το άλλο. Στη συνέχεια, τα στοιχεία ανταλλάσσονται. Όταν συναντώνται οι δείκτες, ο αλγόριθμος σταματά και επιστρέφει τον τελικό δείκτη.
Το σχήμα του Hoare είναι πιο αποτελεσματικό από το σύστημα διαμερισμάτων του Lomuto επειδή κάνει τρεις φορές λιγότερα swaps κατά μέσο όρο και δημιουργεί αποδοτικά διαμερίσματα ακόμη και όταν όλες οι τιμές είναι ίσες.

Ψευδοκώδικας Lomuto Partition:

```
1. partition_hoare(A, low, high)
2.     pivot := A[⌊(high + low) / 2⌋]
3.     i := low - 1
4.           j := high + 1
5.           loop forever
6.                 do
7.                       i := i + 1
8.                  while A[i] < pivot
9.                 do
10.                      j := j - 1
11.                while A[j] > pivot
12.    if i ≥ j then
13.                      return j
14.          swap A[i] with A[j]
```

Πολυπλοκότητα

- **Χειρότερη περίπτωση**: Η χειρότερη συμπεριφορά για γρήγορη ταξινόμηση εμφανίζεται όταν η ρουτίνα διαμέρισης παράγει ένα υποπρόβλημα/υποπίνακα με στοιχεία n-1 και ένα με 0 στοιχεία. Ας υποθέσουμε ότι προκύπτει μη ισορροπημένο διαμέρισμα σε κάθε αναδρομική κλήση. Το κόστος διαμέρισης είναι O(n). Η αναδρομική κλήση σε έναν πίνακα μεγέθους 0 επιστρέφει:

  $$T(0) = \Theta(1)$$

  Η αναδρομή για τον χρόνο εκτέλεσης είναι:

  $$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

  Εάν αθροίσουμε το κόστος που προκύπτει σε κάθε επίπεδο της αναδρομής, παίρνουμε μια αριθμητική σειρά,που αξιολογεί την πολυπλοκότητα ως $O(n^2)$.

- **Ιδανική Περίπτωση:** Η ιδανική συμπεριφορά για γρήγορη ταξινόμηση εμφανίζεται όταν η ρουτίνα διαμέρισης παράγει δύο υποπροβλήματα/υποπίνακες καθένα από τα οποία έχει μέγεθος όχι μεγαλύτερο από n/2, αφού το ένα έχει μέγεθος [n/2] και ένα μέγεθος [(n/2)-1]. Η αναδρομή για τον χρόνο εκτέλεσης είναι:

  $$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

  Αυτή η αναδρομή καταλήγει σε πολυπλοκότητα $O(nlogn)$.

- **Μέση/Αναμενόμενη περίπτωση:**
  Μπορούμε να πάρουμε μια ιδέα της μέσης περίπτωσης λαμβάνοντας υπόψιν μια περίπτωση όπου το διαμέρισμα βάζει στοιχεία O(n/9) σε ένα σύνολο και στοιχεία O(9n/10) σε άλλο σύνολο. Ακολουθεί επανάληψη για αυτήν την περίπτωση.

  $$T(n) = T(\frac{n}{9}) + T(\frac{9n}{10}) + \Theta(n)$$

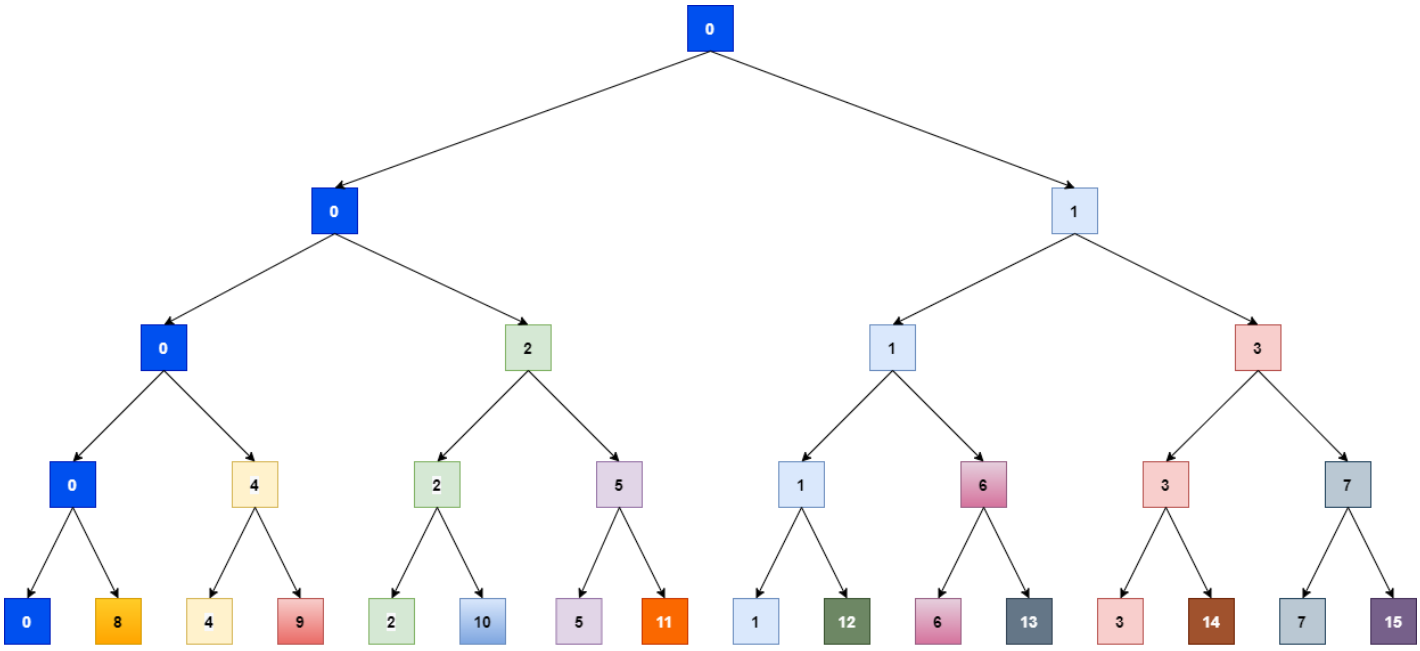
Η λύση της παραπάνω επανάληψης είναι επίσης πολυπλοκότητα $O(nlogn)$, όπου ο συμβολισμός $O(nlogn)$ κρύβει μια σταθερά λίγο μεγαλύτερη από ότι στην ιδανική περίπτωση.

Παράλληλη Προσέγγιση

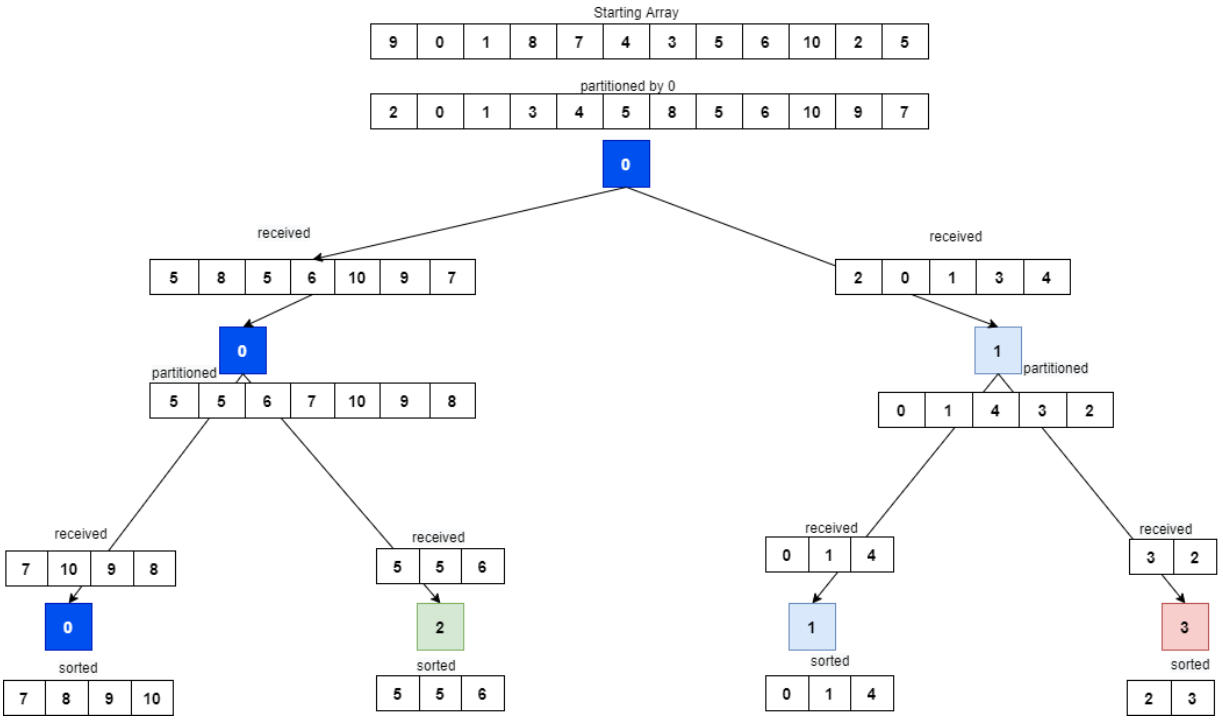
Στην παρούσα εργασία υλοποιήθηκε η παράλληλη εκδοχή της quicksort με δύο μεθόδους, την αναδρομική και με την ένωση(merge).

Quicksort Recursive - Αναδρομική Ταχυταξινόμηση

Κατά την εκδοχή αυτή, ακολουθούμε την φιλοσοφία “διαίρει και βασίλευε”. Ο κεντρικός υπολογιστής(Master) είναι αυτός που θα ξεκινήσει την διαδικασία και θα μοιράσει ένα κομμάτι του πίνακα σε έναν άλλον υπολογιστή και θα κρατήσει το υπόλοιπο. Η διαδικασία αυτή επαναλαμβάνεται για όλους τους υπολογιστές μέχρις ότου να μην υπάρχει κάποιος διαθέσιμος για να δεχτεί κάποιο υποπίνακα. Σε αυτό το σημείο, όλοι οι υπολογιστές ταξινομούν το κομμάτι τους με τον σειριακό αλγόριθμο και επιστρέφουν το αποτέλεσμα που έλαβαν πίσω στον υπολογιστή τους το έστειλε. Το παρακάτω διάγραμμα εξηγεί την διαδικασία:

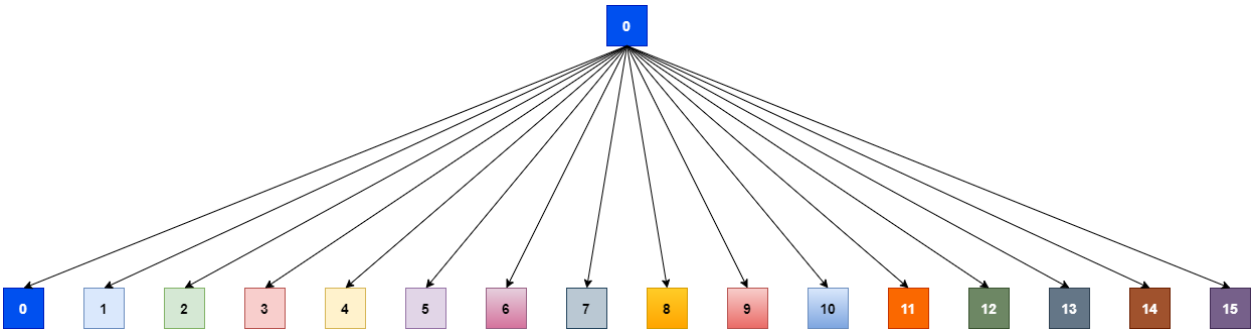


Για καλύτερη κατανόηση της ροής της διαδικασίας, δίνεται το παρακάτω παράδειγμα:



Merge Quicksort - Ταχυταξινόμηση με ένωση

Στην υλοποίηση Merge Quicksort διαιρούμε τον αρχικό πίνακα σε ύπο-πίνακες τους οποίους αποστέλλουμε στους αντίστοιχους υπολογιστές . Αφού κάθε υπολογιστής ταξινομήσει τον δικό του πίνακα, τον αποστέλλει στον αρχικό υπολογιστή όπου και οι πίνακες διαδοχικά ενώνονται για να διαμορφώσουν τον τελικό, πλέον ταξινομημένο, πίνακα.



Απόδοση και Μετρικές

Κατά την υλοποίηση του παράλληλου αλγορίθμου μετρήθηκαν όλες οι απαραίτητες μετρικές σε σχέση με τις μέγιστες δυνατότητες της πλατφόρμας, προκειμένου να προσδιοριστεί η απόδοση του. Συνεπώς για Cluster Size = 16, μελετούνται τρεις διαφορετικές διαστάσεις του προβλήματος, για πίνακα μεγέθους 100.000, 500.000 και 1.000.000 θέσεων.

Recursive Implementation

Χρονική Πολυπλοκότητα

Για την σειριακή εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων απαιτείται χρόνος **T_s=2.383101 seconds**
Για την παράλληλη εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων από 16 υπολογιστές απαιτείται χρόνος **T_p=1.317556 seconds**

Επιτάχυνση - Speedup

Η επιτάχυνση που έχουμε με την παράλληλη εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων από 16 υπολογιστές:

$$S_p = \frac{T_s}{T_p} = \frac{2.383101}{1.1317556} = 1.808728434$$

Αποδοτικότητα - Efficiency

Η αποδοτικότητα που έχουμε με την παράλληλη εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων από 16 υπολογιστές:

$$E = \frac{S_p}{p} = \frac{1.808728434}{16} = 0.1130455271$$

Μέγεθος Προβλήματος - Size

Ο σειριακός μας αλγόριθμος, Quicksort με Lomuto Partitioning, στην μέση/γενική περίπτωση έχει πολυπλοκότητα $O(nlogn)$.
Επομένως για πίνακα δεδομένων 1000000 στοιχείων το μέγεθος του προβλήματος θα είναι $10^6 log 10^6$.

Επικοινωνιακό Κόστος - Communication Cost

Κάνοντας μετρήσεις, μετράμε τον χρόνο που απαιτείται για ένα round trip επικοινωνίας μεταξύ δύο υπολογιστών για την αποστολή ενός πίνακα μεγέθους x. Η γραμμική καμπύλη η οποία ταιριάζει και περιγράφει καλύτερα τα δεδομένα είναι η εξής:

$$y = 1.6668538 * 10^{-7}x + 0.0007423201$$

Συνεπώς θέλοντας να μετρήσουμε το επικοινωνιακό κόστος για την εκτέλεση του αλγορίθμου με 16 υπολογιστές(στην ιδανική περίπτωση), συμβουλευόμαστε το παραπάνω διάγραμμα όπου προκύπτει το εξής.
Στο 1ο επίπεδο του δεντρου επικοινωνίας γίνεται **μία** ανταλλαγή πίνακα μεγέθους περίπου N/2.
Στο 2ο επίπεδο του δεντρου επικοινωνίας γίνονται **δύο** ανταλλαγές πινάκων μεγέθους περίπου N/4.
Στο 3ο επίπεδο του δεντρου επικοινωνίας γίνονται **τέσσερις** ανταλλαγές πινάκων μεγέθους περίπου N/8.
Στο 4ο επίπεδο του δεντρου επικοινωνίας γίνονται **οκτώ** ανταλλαγές πινάκων μεγέθους περίπου N/16.

Συνεπώς για τον πίνακα 100.000 θέσεων το επικοινωνιακό κόστος είναι:

$$CommCost = 0.009 + 2 * 0.0049 + 4 * 0.0028 + 8 * 0.00178 = 0.044\ s$$

Για τον πίνακα 500.000 θέσεων το επικοινωνιακό κόστος είναι:

$$CommCost = 0.0424 + 2 * 0.0215 + 4 * 0.0111 + 8 * 0.0059 = 0.177\ s$$

Για τον πίνακα 1.000.000 θέσεων το επικοινωνιακό κόστος είναι:

$$CommCost = 0.084 + 2 * 0.0424 + 4 * 0.0215 + 8 * 0.0111 = 0.3436\ s$$

Συνεπώς οι τιμές για το επικοινωνιακό κόστος της συγκεκριμένης υλοποίησης περιγράφονται από την παρακάτω καμπύλη :

$$y = 3.329016 * 10^{-7}x + 0.01065246$$

Merge Implementation

Χρονική Πολυπλοκότητα

Για την σειριακή εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων απαιτείται χρόνος **T_s=2.752389 seconds**
Για την παράλληλη εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων από 16 υπολογιστές απαιτείται χρόνος **T_p=0.397279 seconds**

Επιτάχυνση - Speedup

Η επιτάχυνση που έχουμε με την παράλληλη εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων από 16 υπολογιστές:

$$S_p = \frac{T_s}{T_p} = \frac{2.752389}{0.397279} = 6.928100906$$

Αποδοτικότητα - Efficiency

Η αποδοτικότητα που έχουμε με την παράλληλη εκτέλεση του αλγορίθμου με πίνακα δεδομένων 10000000 στοιχείων από 16 υπολογιστές:

$$E = \frac{S_p}{p} = \frac{6.928100906}{16} = 0.4330063067$$

Μέγεθος Προβλήματος - Size

Ο σειριακός μας αλγόριθμος, Quicksort με Hoare Partitioning, στην μέση/γενική περίπτωση έχει πολυπλοκότητα $O(nlogn)$.
Επομένως για πίνακα δεδομένων 1000000 στοιχείων το μέγεθος του προβλήματος θα είναι $10^6 log 10^6$.

Επικοινωνιακό Κόστος - Communication Cost

΄Εχοντας υπολογίσει γραμμική καμπύλη η οποία ταιριάζει και περιγράφει καλύτερα τα δεδομένα είναι η εξής:

$$y = 1.6668538 * 10^{-7}x + 0.0007423201$$

Στην περίπτωση της Merge, από το διάγραμμα φαίνεται πως έχουμε **15** ανταλλαγές πινάκων μεγέθους N/16. (15 επειδή το πρώτο κομμάτι το κρατάει ο υπολογιστής 0).
Συνεπώς για 16 υπολογιστές και πίνακα μεγέθους 100.000 θέσεων, το κόστος είναι:

$$CommCost = 15 * 0.00178 = 0.0267\text{s}$$

Συνεπώς για 16 υπολογιστές και πίνακα μεγέθους 500.000 θέσεων, το κόστος είναι:

$$CommCost = 15 * 0.00595 = 0.08925\text{s}$$

Συνεπώς για 16 υπολογιστές και πίνακα μεγέθους 1.000.000 θέσεων, το κόστος είναι:

$$CommCost = 15 * 0.0111 = 0.1665\text{s}$$

Συνεπώς οι τιμές για το επικοινωνιακό κόστος της συγκεκριμένης υλοποίησης περιγράφονται από την παρακάτω καμπύλη :

$$y = 1.552992 * 10^{-7}x + 0.01132377$$

Ανάλυση Πολυπλοκότητας του Προβλήματος

Με τη γενική παραδοχή της πολυπλοκότητας του χρόνου για τη sequential quicksort= n log n,
Ας πάρουμε την ιδανική ανάλυση για την παραπάνω εφαρμογή, υποθέτοντας ότι κάθε διαμέρισμα μπορεί να δημιουργήσει δύο ίσες ενότητες.
Για δύο επεξεργαστές:

$$T(n) = \frac{n}{2} log(\frac{n}{2})$$

Για τρεις επεξεργαστές:

$$T(n) = max(\frac{n}{2} log(\frac{n}{2}) + \frac{n}{4} log(\frac{n}{4})) \ <=>$$

$$T(n) = \frac{n}{2} log(\frac{n}{2})$$

Για τέσσερις επεξεργαστές:

$$T(n) = \frac{n}{4} log(\frac{n}{4})$$

Ως εκ τούτου, **για p αριθμό επεξεργαστών**:

$$T(n) = \frac{n}{k} log(\frac{n}{k}), \text{ όπου } 2^{k-1} < p < 2^{k+1}$$

Αλλά δεδομένου ότι η εφαρμογή λειτουργεί με κοινή χρήση του μικρότερου διαμερίσματος και μεταβλητό μέγεθος διαμέρισης, αυτή η ανάλυση χρόνου δεν μπορεί να αποδείξει τον μέσο χρόνο εκτέλεσης για την παράλληλη αναδρομική quicksort(Quicksort Recursive). Επίσης, τα πειραματικά αποτελέσματα απέδειξαν επίσης ότι αυτή η χρονική πολυπλοκότητα δεν διατηρείται για τη μέση εκτέλεση της εφαρμογής.

Στην περίπτωση της 2ης μεθόδου, την Parallel Quicksort with Merge, ο διαμοιρασμός των πινάκων δεν είναι αναγκαίο να γίνεται σε δυνάμεις του 2, αλλά διαμοιράζεται σε όλους τους διαθέσιμους υπολογιστές. Δηλαδή, η χρονική πολυπλοκότητα αυτής της μεθόδου εξαρτάται από το μέγεθος του υποπίνακα που θα λάβουν όλοι οι υπολογιστές. Έτσι, κάθε υπολογιστής θα εκτελέσει την σειριακή quicksort για τον υποπίνακα της.
Για **πίνακα n θέσεων και p διαθέσιμους υπολογιστές,** η χρονική πολυπλοκότητα είναι:

$$T(n) = \frac{n}{p} log(\frac{n}{p})$$

Βέβαια, δεν πρόκειται για αντιπροσωπευτικό μέγεθος καθώς μεγάλο μέρος του υπολογισμού καταλαμβάνεται και στην ένωση όλων των υποπινάκων πίσω στον τελικό πίνακα.

Καθώς οι πίνακες ενώνονται ο 1ος με τον 2ο, ο 3ος με την ένωση των δύο προηγούμενων κ.ο.κ. Αυτή η διεργασία πραγματοποιείται (p-1) φορές, όπου το μέσο μέγεθος του πίνακα είναι (n/2 + n/p) και συνεπώς:

$$T(n) = \frac{n}{p} log(\frac{n}{p}) + (\frac{n}{2} + \frac{n}{p})(p-1)$$

Πειραματικά αποτελέσματα

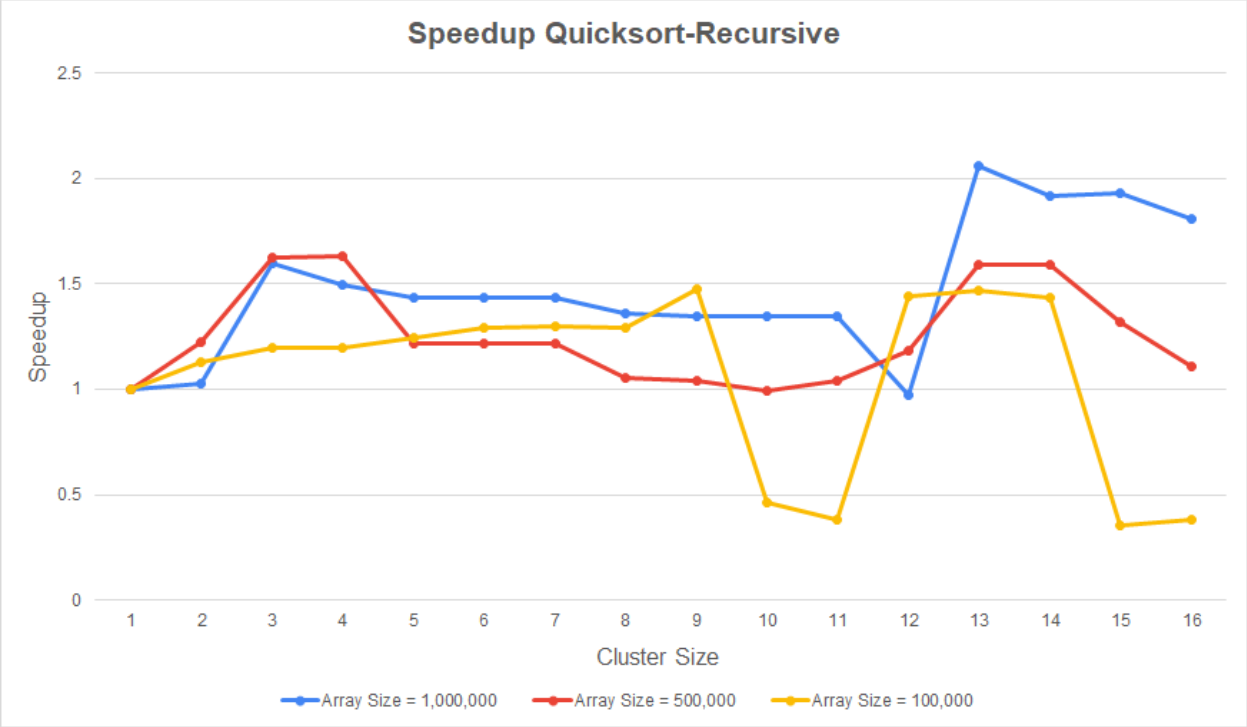
Recursive Implementation

	Array Size = 100,000	
Size of Cluster	Speedup	Efficiency(Sp/p)
1	1	1
2	1.127207518	0.563603759
3	1.19390282	0.397967607
4	1.193529779	0.298382445
5	1.241915718	0.248383144
6	1.293278121	0.215546353
7	1.301890255	0.185984322
8	1.294998175	0.161874772
9	1.476466566	0.164051841
10	0.459958133	0.045995813
11	0.379712184	0.034519289
12	1.441074901	0.120089575
13	1.469475927	0.11303661
14	1.43371059	0.102407899
15	0.355948362	0.023729891
16	0.379126584	0.023695412

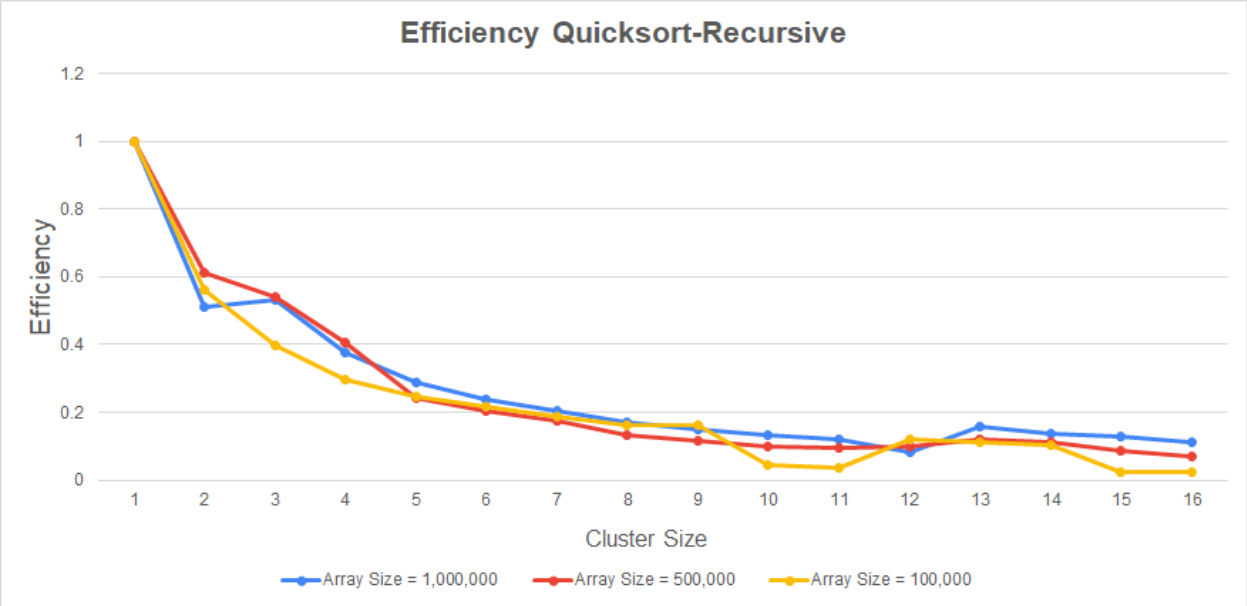
Size of Cluster	Array Size = 500,000 Speedup	Efficiency(Sp/p)
1	1.00000	1
2	1.227394784	0.613697392
3	1.626107888	0.542035963
4	1.62901	0.407253303
5	1.214000268	0.242800054
6	1.217235312	0.202872552
7	1.21551	0.173644198
8	1.053031161	0.131628895
9	1.043787784	0.11597642
10	0.98992	0.098991644
11	1.042411073	0.094764643
12	1.182560938	0.098546745
13	1.59008	0.122313769
14	1.589139287	0.113509949
15	1.319996779	0.087999785
16	1.11066	0.069416355

Size of Cluster	Array Size = 1,000,000 Speedup	Efficiency (Sp/p)
1	1	1
2	1.02633517	0.513167585
3	1.59962961	0.53320987
4	1.499136602	0.37478415
5	1.437593126	0.287518625
6	1.435759562	0.23929326
7	1.433059763	0.204722823
8	1.359093421	0.169886678
9	1.344569198	0.149396578
10	1.348793523	0.134879352
11	1.345199147	0.122290832
12	0.973787689	0.081148974
13	2.059217047	0.158401311
14	1.919835173	0.137131084
15	1.929705188	0.128647013
16	1.808728434	0.113045527

Επιτάχυνση - Speedup($S_p=T_s/T_p$)



Αποδοτικότητα - Efficiency($E=S_p/p$)



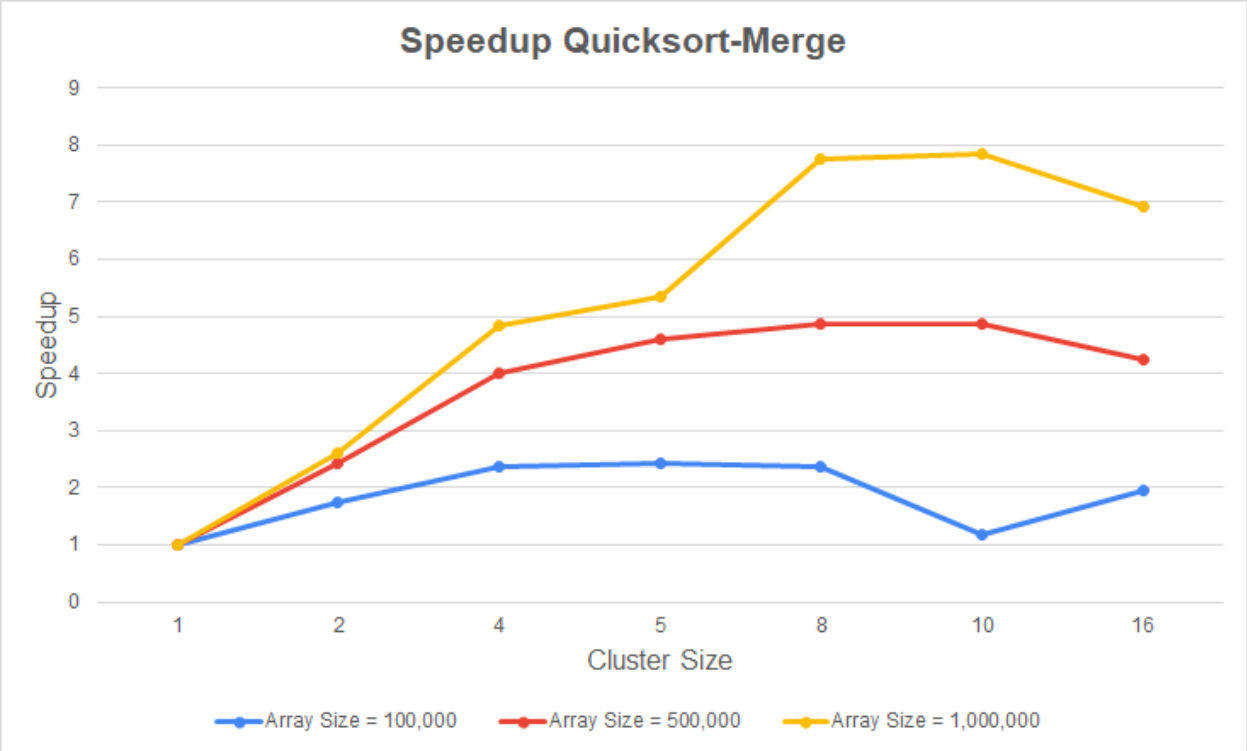
Merge Implementation

Processors	Array Size = 100,000 Speedup	Efficiency (Sp/p)
1	1	1
2	1.733937485	0.866968743
4	2.353924905	0.588481226
5	2.434428202	0.48688564
8	2.371999648	0.296499956
10	1.169716152	0.116971615
16	1.939213149	0.121200822

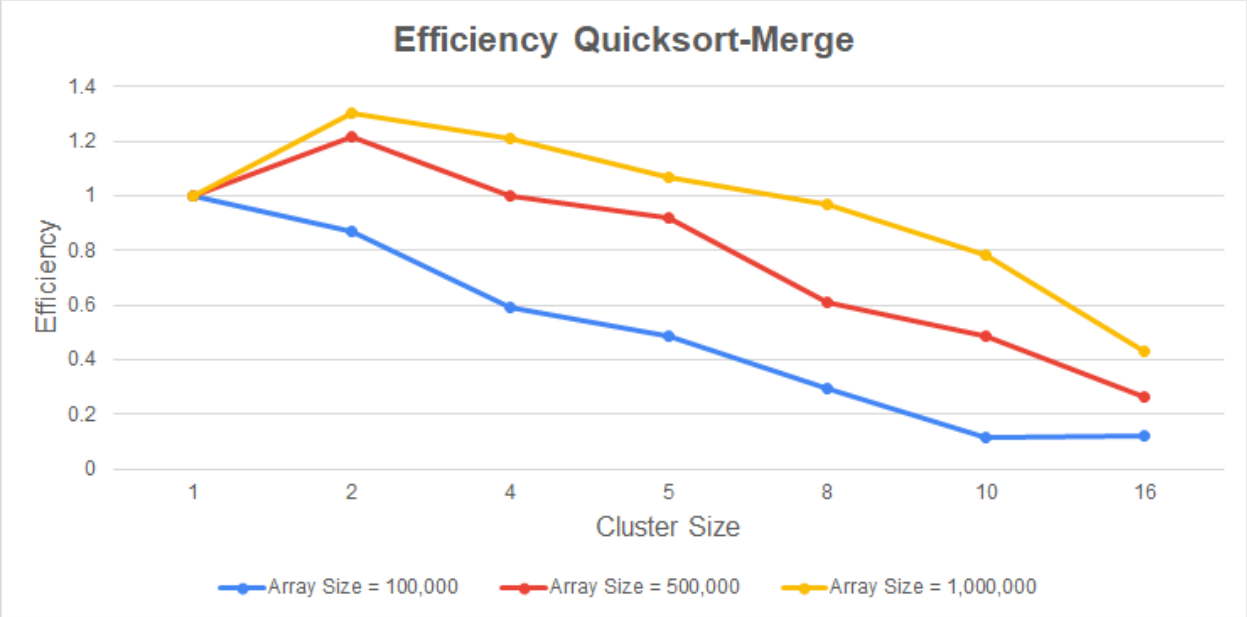
Processors	Array Size = 500,000 Speedup	Efficiency (Sp/p)
1	1	1
2	2.432054183	1.216027091
4	3.998815323	0.999703831
5	4.607969159	0.921593832
8	4.874180524	0.609272566
10	4.863757925	0.486375793
16	4.246359309	0.265397457

Processors	Array Size = 1,000,000 Speedup	Efficiency (Sp/p)
1	1	1
2	2.602060177	1.301030089
4	4.846412278	1.211603069
5	5.349472125	1.069894425
8	7.768022398	0.9710028
10	7.849323976	0.784932398
16	6.928100906	0.433006307

Επιτάχυνση - Speedup($S_p=T_s/T_p$)



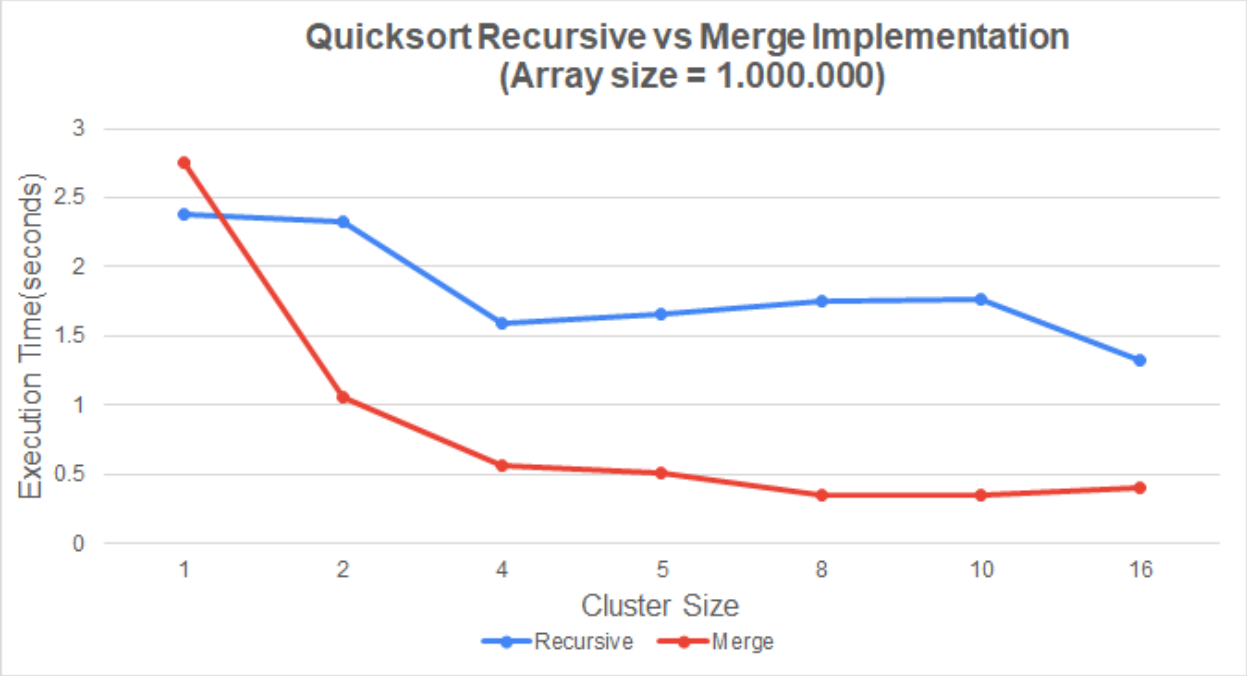
Αποδοτικότητα - Efficiency($E=S_p/p$)



Συμπεράσματα

Στο διάγραμμα που ακολουθεί παρουσιάζονται οι χρόνοι εκτέλεσης των δύο αλγορίθμων ανάλογα με το πλήθος των υπολογιστών που συμμετέχουν στο cluster, για πίνακες δεδομένων

μεγέθους 1.000.000 στοιχείων. Παρατηρούμε πως η υβριδική υλοποίηση (Merge Implementation) είναι πιο αποδοτική.



Καθώς τα δεδομένα που διαμοιράζουν οι αλγόριθμοι, σε κάθε στάδιο χρησιμοποιούνται το πολύ από έναν επεξεργαστή, δεν εμφανίζονται προβλήματα συνοχής μνήμης και δεν είναι αναγκαίος ο έλεγχος του χρονισμού για την προστασία αυτής.
Συνεπώς πλατφόρμες shared memory όπως το Openmp είναι ικανές να υποστηρίξουν υλοποιήσεις των αλγορίθμων μας δίχως κάποια αλλαγή της ροής τους, μειώνοντας σημαντικά τα επικοινωνιακά κόστη.

Υλοποίηση Quicksort Recursive (γλώσσα C πλατφόρμα MPI)

```c
#include "mpi.h"
#include<stdio.h>
#include <stdlib.h>
#include "math.h"
#include <stdbool.h>
#define SIZE 1000000

/*
    Divides the array given into two partitions
        - Lower than pivot
        - Higher than pivot
    and returns the Pivot index in the array
*/
int partition(int *arr, int low, int high){
    int pivot = arr[high];
    int i = (low - 1);
    int j,temp;
    for (j=low;j<=high-1;j++){
    if(arr[j] < pivot){
         i++;
             temp=arr[i];
             arr[i]=arr[j];
             arr[j]=temp;
    }
    }
    temp=arr[i+1];
    arr[i+1]=arr[high];
    arr[high]=temp;
    return (i+1);
}

/*
    Hoare Partition - Starting pivot is the middle point
    Divides the array given into two partitions
        - Lower than pivot
        - Higher than pivot
    and returns the Pivot index in the array
*/
int hoare_partition(int *arr, int low, int high){
    int middle = floor((low+high)/2);
    int pivot = arr[middle];
    int j,temp;
    // move pivot to the end
    temp=arr[middle];
    arr[middle]=arr[high];
    arr[high]=temp;

    int i = (low - 1);
    for (j=low;j<=high-1;j++){
        if(arr[j] < pivot){
            i++;
            temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
        }
```

```
    }
    // move pivot back
    temp=arr[i+1];
    arr[i+1]=arr[high];
    arr[high]=temp;

    return (i+1);
}

/*
    Simple sequential Quicksort Algorithm
*/
void quicksort(int *number,int first,int last){
    if(first<last){
        int pivot_index = partition(number, first, last);
        quicksort(number,first,pivot_index-1);
        quicksort(number,pivot_index+1,last);
    }
}

/*
    Functions that handles the sharing of subarrays to the right clusters
*/
int quicksort_recursive(int* arr, int arrSize, int currProcRank, int maxRank, int rankIndex) {
    MPI_Status status;

    // Calculate the rank of the Cluster which I'll send the other half
    int shareProc = currProcRank + pow(2, rankIndex);
    // Move to lower layer in the tree
    rankIndex++;

    // If no Cluster is available, sort sequentially by yourself and return
    if (shareProc > maxRank) {
        MPI_Barrier(MPI_COMM_WORLD);
        quicksort(arr, 0, arrSize-1 );
        return 0;
    }
    // Divide array in two parts with the pivot in between
    int j = 0;
    int pivotIndex;
    pivotIndex = hoare_partition(arr, j, arrSize-1 );

    // Send partition based on size(always send the smaller part),
    // Sort the remaining partitions,
    // Receive sorted partition
    if (pivotIndex <= arrSize - pivotIndex) {
        MPI_Send(arr, pivotIndex , MPI_INT, shareProc, pivotIndex, MPI_COMM_WORLD);
        quicksort_recursive((arr + pivotIndex+1), (arrSize - pivotIndex-1 ), currProcRank,
maxRank, rankIndex);
        MPI_Recv(arr, pivotIndex , MPI_INT, shareProc, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }
    else {
        MPI_Send((arr + pivotIndex+1), arrSize - pivotIndex-1, MPI_INT, shareProc, pivotIndex
+ 1, MPI_COMM_WORLD);
        quicksort_recursive(arr, (pivotIndex), currProcRank, maxRank, rankIndex);
        MPI_Recv((arr + pivotIndex+1), arrSize - pivotIndex-1, MPI_INT, shareProc,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }
}
```

```
int main(int argc, char *argv[]) {
    int unsorted_array[SIZE];
    int array_size = SIZE;
    int size, rank;
    // Start Parallel Execution
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank==0){
        // --- RANDOM ARRAY GENERATION ---
        printf("Creating Random List of %d elements\n", SIZE);
        int j = 0;
        for (j = 0; j < SIZE; ++j) {
            unsorted_array[j] =(int) rand() % 1000;
        }
        printf("Created\n");
    }

    // Calculate in which layer of the tree each Cluster belongs
    int rankPower = 0;
    while (pow(2, rankPower) <= rank){
        rankPower++;
    }
    // Wait for all clusters to reach this point
    MPI_Barrier(MPI_COMM_WORLD);
    double start_timer, finish_timer;
    if (rank == 0) {
        start_timer = MPI_Wtime();
        // Cluster Zero(Master) starts the Execution and
        // always runs recursively and keeps the left bigger half
        quicksort_recursive(unsorted_array, array_size, rank, size - 1, rankPower);
    }else{
        // All other Clusters wait for their subarray to arrive,
        // they sort it and they send it back.
        MPI_Status status;
        int subarray_size;
        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        // Capturing size of the array to receive
        MPI_Get_count(&status, MPI_INT, &subarray_size);
        int source_process = status.MPI_SOURCE;
        int subarray[subarray_size];
        MPI_Recv(subarray, subarray_size, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        quicksort_recursive(subarray, subarray_size, rank, size - 1, rankPower);
        MPI_Send(subarray, subarray_size, MPI_INT, source_process, 0, MPI_COMM_WORLD);
    };

    if(rank==0){
        finish_timer = MPI_Wtime();
        printf("Total time for %d Clusters : %2.2f sec \n",size, finish_timer-start_timer);

        // --- VALIDATION CHECK ---
        printf("Checking.. \n");
        bool error = false;
        int i=0;
        for(i=0;i<SIZE-1;i++) {
            if (unsorted_array[i] > unsorted_array[i+1]){
                error = true;
                printf("error in i=%d \n", i);
            }
        }
```

```
        if(error)
            printf("Error..Not sorted correctly\n");
        else
            printf("Correct!\n");
    }

    MPI_Finalize();
    // End of Parallel Execution
    return 0;
}
```

Υλοποίηση Quicksort Merge (γλώσσα C πλατφόρμα MPI)

```c
#include "mpi.h"
#include<stdio.h>
#include <stdlib.h>
#include "math.h"
#include <stdbool.h>
#define SIZE 1000000


/*
    Hoare Partition - Starting pivot is the middle point
    Divides the array given into two partitions
        - Lower than pivot
        - Higher than pivot
    and returns the Pivot index in the array
*/
int hoare_partition(int *arr, int low, int high){
    int middle = floor((low+high)/2);
    int pivot = arr[middle];
    int j,temp;
    // move pivot to the end
    temp=arr[middle];
    arr[middle]=arr[high];
    arr[high]=temp;

    int i = (low - 1);
    for (j=low;j<=high-1;j++){
        if(arr[j] < pivot){
            i++;
            temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
        }
    }
    // move pivot back
    temp=arr[i+1];
    arr[i+1]=arr[high];
    arr[high]=temp;

    return (i+1);
}


/*
    Simple sequential Quicksort Algorithm
*/
void quicksort(int *number,int first,int last){
    if(first<last){
        int pivot_index = hoare_partition(number, first, last);
        quicksort(number,first,pivot_index-1);
        quicksort(number,pivot_index+1,last);
    }
}

/*
    Function that handles the merging of two sorted subarrays
```

```
    and returns one bigger sorted array
*/
void merge(int *first,int *second, int *result,int first_size,int second_size){
    int i=0;
    int j=0;
    int k=0;

    while(i<first_size && j<second_size){

        if (first[i]<second[j]) {
            result[k]=first[i];
            k++;
            i++;
        }else{
            result[k]=second[j];
            k++;
            j++;
        }

        if(i == first_size){
            // if the first array has been sorted
            while(j<second_size){
                result[k]=second[j];
                k++;
                j++;
            }
        } else if (j == second_size){
            // if the second array has been sorted
            while(i < first_size){
                result[k]=first[i];
                i++;
                k++;
            }
        }
    }
}

int main(int argc, char *argv[]) {

    int *unsorted_array = (int *)malloc(SIZE * sizeof(int));
    int *result = (int *)malloc(SIZE * sizeof(int));
    int array_size = SIZE;
    int size, rank;
    int sub_array_size;

    MPI_Status status;
    // Start parallel execution
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if(rank==0){
        // --- RANDOM ARRAY GENERATION ---
        printf("Creating Random List of %d elements\n", SIZE);
        int j = 0;
        for (j = 0; j < SIZE; ++j) {
            unsorted_array[j] =(int) rand() % 1000;
        }
        printf("Created\n");
    }
```

```
    // Number of computers in cluster
    int iter_count = size;
    // Determine the size of the subarray each computer receives
    sub_array_size=(int)SIZE/iter_count;

    // Computer 0 (Master) splits the array and sends each subarray to the respective machine
    if( rank == 0 ){
        double start_timer;
        start_timer=MPI_Wtime();
        int i =0;
        if(iter_count > 1){
            // ================================================SENDING
DATA================================================
            for(i=0;i<iter_count-1;i++){
                int j;
                //send the subarray

MPI_Send(&unsorted_array[(i+1)*sub_array_size],sub_array_size,MPI_INT,i+1,0,MPI_COMM_WORLD);
            }

            // ==========================================CALCULATE FIRST
SUBARRAY==========================================
            int i =0;
            int *sub_array = (int *)malloc(sub_array_size*sizeof(int));
            for(i=0;i<sub_array_size;i++){
                // Passing the first sub array since rank 0 always calculates the first sub
array
                sub_array[i]=unsorted_array[i];
            }
            // Sequentially sorting the first array
            quicksort(sub_array,0,sub_array_size-1);

            // ============================================RECEIVING
DATA============================================
            for (i=0;i<iter_count;i++){
                if(i > 0){
                    int temp_sub_array[sub_array_size];
                    // Receive each subarray

MPI_Recv(temp_sub_array,sub_array_size,MPI_INT,i,777,MPI_COMM_WORLD,&status);
                    int j;
                    int temp_result[i*sub_array_size];
                    for(j=0;j<i*sub_array_size;j++){
                        temp_result[j]=result[j];
                    }
                    int temp_result_size = sub_array_size*i;
                    // Merge it back into the result array
                    merge(temp_sub_array,temp_result,result,sub_array_size,temp_result_size);

                }else{
                    // On first iteration we just pass the sorted elements to the result array
                    int j;
                    for(j=0;j<sub_array_size;j++){
                        result[j]=sub_array[j];
                    }
                    free(sub_array);
                }
            }
        }else{
            // if it runs only in a single computer
            quicksort(unsorted_array,0,SIZE-1);
```

```
            for(i=0;i<SIZE;i++){
                result[i]=unsorted_array[i];
            }
        }
        double finish_timer;
        finish_timer=MPI_Wtime();
        printf("End Result: \n");
        printf("Cluster Size %d, execution time measured : %2.7f sec \n",size, finish_timer-
start_timer);
    }else{
        // All the other computers have to sort the data and send it back
        sub_array_size=(int)SIZE/iter_count;
        int *sub_array = (int *)malloc(sub_array_size*sizeof(int));
        MPI_Recv(sub_array,sub_array_size,MPI_INT,0,0,MPI_COMM_WORLD,&status);
        quicksort(sub_array,0,sub_array_size-1);
        int i=0;
        MPI_Send(sub_array,sub_array_size,MPI_INT,0,777,MPI_COMM_WORLD);//sends the data back
to rank 0
        free(sub_array);
    }

    if(rank==0){
        // --- VALIDATION CHECK ---
        printf("Checking.. \n");
        bool error = false;
        int i=0;
        for(i=0;i<SIZE-1;i++) {
            if (result[i] > result[i+1]){
                error = true;
                printf("error in i=%d \n", i);
            }
        }
        if(error)
            printf("Error..Not sorted correctly\n");
        else
            printf("Correct!\n");
    }
    free(unsorted_array);
    // End of Parallel Execution
    MPI_Finalize();
}
```

Βιβλιογραφικές Αναφορές

1. Παύλος Εφραιμίδης. Διαλέξεις Μαθήματος Αλγόριθμοι και Δομές Δεδομένων. Δημοκρίτειο Πανεπιστήμιο Θράκης. https://eclass.duth.gr/courses/TMA566/
2. Wikipedia contributors. Quicksort. Wikipedia. https://en.wikipedia.org/wiki/Quicksort
3. Cormen, T. H. (2009). Introduction to Algorithms, 3rd Edition (The MIT Press) (3rd ed.). MIT Press.
4. Foster, J. (1995). Designing and Building Parallel Programs. Addison-Wesley.
5. Miller, R., & Boxer, L. (2012). Algorithms Sequential & Parallel: A Unified Approach (3rd ed.). Cengage Learning.
6. Μιχαήλ Π. Μπεκάκος. Διαλέξεις Μαθήματος Υπολογισμοί Υψηλής Απόδοσης: Παράλληλοι Αλγόριθμοι και Υπολογιστική Πολυπλοκότητα. Δημοκρίτειο Πανεπιστήμιο Θράκης. https://eclass.duth.gr/courses/TMA503/