ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΙΑ 2Α

Multisort

ΣΤΟΙΧΕΙΑ ΦΟΙΤΗΤΗ / ΕΡΓΑΣΙΑΣ

ΟΝΟΜΑΤΕΠΩΝΥΜΟ : ΑΘΑΝΑΣΙΟΥ ΒΑΣΙΛΕΙΟΣ ΕΥΑΓΓΕΛΟΣ
ΑΡΙΘΜΟΣ ΜΗΤΡΩΟΥ : 19390005
ΕΞΑΜΗΝΟ ΦΟΙΤΗΤΗ : 11
ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ : ΠΑΔΑ

ΥΠΕΥΘΥΝΟΣ ΕΡΓΑΣΤΗΡΙΟΥ: ΙΟΡΔΑΝΑΚΗΣ ΜΙΧΑΛΗΣ
ΥΠΕΥΘΥΝΟΣ ΘΕΩΡΙΑΣ: ΜΑΜΑΛΗΣ ΒΑΣΙΛΕΙΟΣ

ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

ΠΕΡΙΕΧΟΜΕΝΑ

1. Εισαγωγή

1.1 Σκοπός της άσκησης

Ο σκοπός της άσκησης είναι η υλοποίηση και η αξιολόγηση του παράλληλου αλγορίθμου ταξινόμησης 1Δ πίνακα **multisort**, όπου αποτελεί μια εναλλακτική επιλογή του αλγορίθμου **mergesort**. Ο παράλληλος υπολογισμός γίνεται με χρήση του OpenMP που αποτελεί σε χαμηλό επίπεδο το πολυνηματικό (multithreaded) πρότυπο παράλληλου προγραμματισμού, με τη λογική του μοντέλου παράλληλης εκτέλεσης fork-join.

Συγκεκριμένα, ο προγραμματιστής εξοικειώνεται με τις high level τεχνικές που του παρέχει το OpenMP, όπου διακρίνει το τμήμα κώδικα που θα εκτελεστεί παράλληλα και κατανοεί την ανάθεση εργασιών (tasks) στα νήματα με την ιδιότητα της αναδρομής.

Τέλος, η άσκηση αποσκοπεί στο να αναδείξει και τους χρόνους εκτέλεσης του παράλληλου αλγορίθμου, ώστε να υπολογιστούν και να συγκριθούν οι επιταχύνσεις (speed-up) τόσο κατά την περίπτωση που το πρόγραμμα εκτελείται ακολουθιακά (με 1 νήμα), όσο παράλληλα (> 1 νήματα).

1.2 Συνοπτική περιγραφή του προβλήματος που επιλύεται

Ο αλγόριθμος επιλύει το πρόβλημα της ταξινόμησης ενός 1Δ πίνακα N ακεραίων A[1...N-1], χρησιμοποιώντας το OpenMP για την παράλληλη εκτέλεση του. Η μέθοδος αυτή στηρίζεται στην τεχνική **διαίρει και βασίλευε**, όπου το πρόβλημα διασπάται σε υποπροβλήματα και υπολογίζονται τοπικά οι λύσεις των υποπροβλημάτων, ώστε να συγκεντρωθούν στο τέλος για την βέλτιστη δυνατή. Άλλωστε, γι' αυτό αποτελεί εναλλακτική επιλογή του **mergesort**, όπου ο πίνακας σπάει σε υπο-πίνακες, ταξινομούνται τοπικά και στο τέλος συγχωνεύονται σ' έναν ενιαίο ο οποίος είναι και ο τελικός ταξινομημένος.

Ο αλγόριθμος **multisort** διαχωρίζει αρχικά την προς ταξινόμηση ακολουθία σε τέσσερα ισομεγέθη τμήματα, και συνεχίζει εφαρμόζοντας αναδρομικά την παραπάνω διαδικασία διαχωρισμού σε κάθε τμήμα. Με την ολοκλήρωση κάθε αναδρομικής κλήσης, τα τέσσερα επιμέρους τμήματα τα οποία επιστρέφονται στο κυρίως σώμα ταξινομημένα, συγχωνεύονται σε μία ενιαία ταξινομημένη ακολουθία σε δύο βήματα (πρώτα συγχωνεύονται – παράλληλα – ανά δύο σε δύο τμήματα διπλάσιου μεγέθους, και στη συνέχεια τα δύο εναπομείναντα τμήματα συγχωνεύονται μεταξύ τους)

Όσον αφορά την απόδοση, λαμβάνονται υπόψιν οι εξής παράμετροι:

- Μέγεθος πίνακα N
- Αριθμός νημάτων T
- Όριο πίνακα LIMIT όπου σημάνει και την λήξη του παράλληλου υπολογισμού, όπου εκτελείται ο γνωστός σειριακός αλγόριθμος ταξινόμησης **quicksort**

2. Σχεδιασμός

2.1 Περιγραφή της προσέγγισης που ακολουθήθηκε

Η υλοποίηση βασίζεται στην κατανομή των υπολογισμών σε tasks χρησιμοποιώντας το OpenMP και αναδρομή για την επίτευξη παράλληλης επεξεργασίας, καθώς, 1 task αναλαμβάνεται από μόνο 1 thread. Η προσέγγιση διαχωρίζεται σε διακριτές φάσεις:

- **Ανάθεση εργασιών (tasks) στα νήματα:** Με χρήση OpenMP και αναδρομή το κάθε task αναλαμβάνεται από κάποιο διαθέσιμο ενεργό thread.
- **Αξιοποίηση της δυνατότητας του OpenMP:** Στην κατανομή εργασιών και στον συγχρονισμό με οδηγίες wait.
- **Ανάλυση της δομής του προβλήματος:** Τα δεδομένα οργανώνονται ώστε να διευκολύνουν την παράλληλη επεξεργασία, ελαχιστοποιώντας την επικοινωνία μεταξύ νημάτων.

2.2 Ανάλυση της λογικής και των μεθοδολογιών

Η λογική βασίζεται στα εξής:

1. **Διαχωρισμός του πίνακα Α σε 4 ισομεγέθη τμήματα:** Η 1η φάση του αλγορίθμου είναι ο πίνακας να διαχωριστεί σε 4 ισομεγέθη τμήματα. Αυτό επιτυγχάνεται με τον υπολογισμό των κατάλληλων δεικτών, ώστε κάθε τμήμα να έχει το ίδιο περίπου μέγεθος N/4
2. **Αναδρομική κλήση της multisort για τα 4 τμήματα με χρήση OpenMP tasks και παράλληλη εκτέλεση από 4 διαθέσιμα νήματα:** Η 2η φάση του αλγορίθμου είναι η αναδρομική κλήση για καθένα από τα 4 τμήματα του πίνακα με χρήση OpenMP tasks που αναθέτουν κάθε αναδρομική κλήση σε ξεχωριστό νήμα. Έτσι, η ταξινόμηση των 4 τμημάτων πραγματοποιείται ταυτόχρονα
3. **Παράλληλη συγχώνευση ανά δύο σε δύο τμήματα διπλάσιου μεγέθους:** Η 3η φάση είναι η παράλληλη συγχώνευση ανά δύο σε δύο τμήματα με τη βοήθεια της μεθόδου merge.
4. **Συγχώνευση των δύο εναπομείναντων τμημάτων μεταξύ τους:** Η 4η φάση είναι η συγχώνευση των δύο εναπομείναντων τμημάτων σ’ ένα ενιαίο ταξινομημένο πίνακα με τη βοήθεια και πάλι της μεθόδου merge.
5. **Έλεγχος κριτηρίου τερματισμού:** Εφόσον, από την αναδρομική κλήση το μέγεθος του πίνακα δεν υπερβεί το όριο που θέσαμε στην παράμετρο LIMIT, τότε η αναδρομή τερματίζει και εκτελείται ο σειριακός αλγόριθμος ταξινόμησης με την μέθοδο quicksort.

2.3 Περιγραφή των δομών δεδομένων και των αλγορίθμων

2.3.1 Δομές δεδομένων και μεταβλητές

Για την επίλυση του προβλήματος χρησιμοποιήθηκαν οι παρακάτω δομές δεδομένων και μεταβλητές:

Όνομα Μεταβλητής	Περιγραφή Μεταβλητής
`int main(int argc, char *argv[])`	
int	
*A	Δυναμικός 1Δ πίνακας που θα ταξινομηθεί
*Space	Δυναμικός 1Δ που θα χρησιμοποιηθεί ως προσωρινός πίνακας αποθήκευσης για την ταξινόμηση του A
threads	Το πλήθος των νημάτων που θα δημιουργηθούν
size	Το μέγεθος του πίνακα
i	Δείκτης επανάληψης
double	
start_time	Έναρξη χρόνου μέτρησης της παράλληλης επεξεργασίας του αλγορίθμου
end_time	Λήξη χρόνου μέτρησης της παράλληλης επεξεργασίας του αλγορίθμου
FILE	
*fpA_unsort	Αρχείο εξόδου για την αποθήκευση του πίνακα A προτού ταξινομηθεί
*fpA_sort	Αρχείο εξόδου για την αποθήκευση του πίνακα A αφού ταξινομηθεί
`void multisort(int *start, int *space, int size)`	
int	
*start	Δείκτης στην αρχή του πίνακα που πρέπει να ταξινομηθεί
*space	Δείκτης σ' έναν βοηθητικό πίνακα ίδιου μεγέθους με τον αρχικό, που χρησιμοποιείται για την συγχώνευση των τμημάτων
size	Μέγεθος του πίνακα που θα ταξινομηθεί
quarter	Το μέγεθος του τμήματος που θα διαχωριστεί
*startA	Δείκτης στην αρχή του 1^{ου} υπο-τμήματος
*startB	Δείκτης στην αρχή του 2^{ου} υπο-τμήματος
*startC	Δείκτης στην αρχή του 3^{ου} υπο-τμήματος
*startD	Δείκτης στην αρχή του 4^{ου} υπο-τμήματος
*spaceA	Δείκτης στην αρχή του βοηθητικού πίνακα του 1^{ου} υπο-τμήματος
*spaceB	Δείκτης στην αρχή του βοηθητικού πίνακα του 2^{ου} υπο-τμήματος
*spaceC	Δείκτης στην αρχή του βοηθητικού πίνακα του 3^{ου} υπο-τμήματος

*spaceD	Δείκτης στην αρχή του βοηθητικού πίνακα του 4^{ου} υπο-τμήματος
`void quicksort(int *start, int *end)`	
int	
*start	Δείκτης στην αρχή του τμήματος του πίνακα που πρέπει να ταξινομηθεί
*end	Δείκτης στο τέλος του τμήματος του πίνακα που πρέπει να ταξινομηθεί
*pvt	Το στοιχείο οδηγός που θα τοποθετηθεί στην σωστή θέση ταξινόμησης στον πίνακα
`int* pivotPartition(int *start, int *end)`	
int	
*start	Δείκτης στην αρχή του τμήματος του πίνακα που πρέπει να διαχωριστεί
*end	Δείκτης στο τέλος του τμήματος του πίνακα που πρέπει να διαχωριστεί
*pvt	Το στοιχείο οδηγός που θα τοποθετηθεί στην σωστή θέση ταξινόμησης στον πίνακα
*i	Δείκτης που δείχνει σε θέση πριν από αυτήν που δείχνει ο δείκτης j
*j	Δείκτης που δείχνει σε θέση μετά από αυτήν που δείχνει ο δείκτης i
`void swap(int *a, int *b)`	
int	
*a	Δείκτης στο πρώτο στοιχείο που θα ανταλλαγεί
*b	Δείκτης στο δεύτερο στοιχείο που θα ανταλλαγεί
temp	Προσωρινός χώρος αποθήκευσης
`void merge(int *startA, int *endA, int *startB, int *endB, int *space)`	
int	
*startA	Δείκτης στο πρώτο στοιχείο του πρώτου υποπίνακα (A)
*endA	Δείκτης στο τελευταίο στοιχείο του πρώτου υποπίνακα (A)
*startB	Δείκτης στο πρώτο στοιχείο του δεύτερου υποπίνακα (B)
*endB	Δείκτης στο τελευταίο στοιχείο του δεύτερου υποπίνακα (B)
*space	Δείκτης σε προσωρινό πίνακα αποθήκευσης
*i	Δείκτης αρχικοποίησης για τον υπο-πίνακα A
*j	Δείκτης αρχικοποίησης για τον υπο-πίνακα B
*k	Δείκτης αρχικοποίησης για τον προσωρινό πίνακα αποθήκευσης

2.3.2 Πρόγραμμα γεννήτρια για παραγωγή 1Δ πινάκων

Ο αλγόριθμος αυτός δημιουργεί έναν πίνακα Α διαστάσεων Ν με ψευδοτυχαίους ακέραιους αριθμούς με βάση ένα διάστημα τιμών.

- **Κώδικας:**

```
for (i = 0; i < size; i++)
{
        A[i] = rand() % 199 - 99;
        A[i] = A[i] >= 0 ? A[i] + 10 : A[i] - 10;
}
```

- **Λειτουργία:**
    - Με βρόχο κάνουμε προσπέλαση τον 1Δ πίνακα
    - Επιλέγουμε τιμές από -99 εώς 99
    - Αλλάζουμε την τιμή κατά 10 ανάλογα το πρόσημο

2.3.3 Multisort

Η συνάρτηση **multisort** υλοποιεί τον εναλλακτικό παράλληλο αλγόριθμο ταξινόμησης σε σχέση με τον **mergesort**. Ο αλγόριθμος χωρίζει τον πίνακα σε τέσσερα ισομεγέθη τμήματα (quarters), ταξινομεί κάθε τμήμα αναδρομικά και στη συνέχεια συγχωνεύει τα ταξινομημένα τμήματα. Όταν το μέγεθος του τμήματος είναι μικρότερο από το όριο LIMIT, χρησιμοποιείται ο ακολουθιακός αλγόριθμος **quicksort** για την τοπική ταξινόμηση.

2.3.4 Quicksort

Η συνάρτηση **quicksort** υλοποιεί τον αναδρομικό αλγόριθμο γρήγορης ταξινόμησης

1. Επιλέγει ένα στοιχείο οδηγός (pivot) από το τμήμα του πίνακα που ταξινομείται, δηλαδή, ένα στοιχείο που βρίσκεται στην σωστή θέση ταξινόμησης.
2. Τοποθετεί όλα τα στοιχεία μικρότερα ή ίσα με τον οδηγό στα αριστερά του και όλα τα μεγαλύτερα στα δεξιά του, διαχωρίζοντας τον πίνακα σε δύο υποπίνακες.
3. Επαναλαμβάνει αναδρομικά τη διαδικασία για τον αριστερό και δεξιό υποπίνακα μέχρι να επιτευχθεί η πλήρης ταξινόμηση.

2.3.5 Pivot Partition

Η συνάρτηση **pivotPartition** με την τεχνική δύο δεικτών εκτελεί τον διαχωρισμό του πίνακα γύρω από ένα στοιχείο οδηγός (pivot).

1. Το pivot ορίζεται αρχικά ως το τελευταίο στοιχείο του τμήματος του πίνακα.
2. Μετακινεί όλα τα στοιχεία μικρότερα ή ίσα με το pivot στην αριστερή πλευρά του πίνακα, ενώ τα μεγαλύτερα παραμένουν στη δεξιά πλευρά.
3. Τέλος, το pivot τοποθετείται στη σωστή του θέση, δηλαδή μεταξύ των μικρότερων και των μεγαλύτερων στοιχείων.

2.3.6 Swap

Η συνάρτηση **swap** πραγματοποιεί την ανταλλαγή (swap) των τιμών δύο μεταβλητών που δείχνονται από τους δείκτες a και b.

2.3.7 Merge

Η συνάρτηση **merge** συγχωνεύει δύο ταξινομημένα υποτμήματα (A και B) σε έναν ενιαίο ταξινομημένο πίνακα. Ο συγχωνευμένος πίνακας αποθηκεύεται αρχικά στον προσωρινό πίνακα space και στη συνέχεια επιστρέφεται στον αρχικό πίνακα.

3. Υλοποίηση

3.1 Αναφορά στις βασικές λειτουργίες του κώδικα

Οι βασικές λειτουργίες του κώδικα περιλαμβάνουν:

- **Αρχικοποίηση του πίνακα A με τυχαίες τιμές:**
  Με πρόγραμμα γεννήτρια που αναφέρεται στο κεφάλαιο 2.3.2 Πρόγραμμα γεννήτρια για παραγωγή 1Δ πινάκων δημιουργείται ο προς ταξινόμηση 1Δ πίνακας.
- **Αλγόριθμος multisort:**
  Ο αλγόριθμος περιλαμβάνει:
  - Διαίρεση του πίνακα σε υπο-πίνακες: Ο αρχικός πίνακας χωρίζεται σε σε 4 ισομεγέθη τμήματα
  - Παράλληλη ταξινόμηση των υπο-πινάκων: Κάθε υπο-πίνακας ταξινομείται ταυτόχρονα από διαφορετικό νήμα. Αν το μέγεθος του υπο-πίνακα είναι μικρότερο από ένα όριο που θέτει ο χρήστης, τότε εκτελείται ο ακολουθιακός αλγόριθμος ταξινόμησης quicksort.
  - Παράλληλη συγχώνευση των ταξινομημένων υπο-πινάκων: Μετά την τοπική ταξινόμηση, οι υπο-πίνακες συγχωνεύονται για να σχηματίσουν τον τελικό ταξινομημένο πίνακα.
- **Διαχείριση διακριτών εργασιών:**
  Για τον συγχρονισμό των διακριτών εργασιών που αναλαμβάνουν αναδρομικές κλήσεις του αλγορίθμου, λαμβάνονται χώρα και μηχανισμοί συγχρονισμού για τον ορθό παράλληλο υπολογισμό που πραγματοποιούν τα νήματα.

- **Μετρήσεις χρόνου:**
  Για την εκτέλεση του αλγορίθμου λαμβάνεται ο χρόνος έναρξης του παράλληλου υπολογισμού και ο χρόνος λήξης για να υπολογιστούν οι επιταχύνσεις και η απόδοση του αλγορίθμου.
- **Εκτυπώσεις αποτελεσμάτων:**
  Τα τελικά αποτελέσματα αποθηκεύονται σε αρχείο.

3.2 Επεξήγηση παράλληλων τμημάτων του κώδικα

Οι παράλληλες ενότητες εξηγούνται παρακάτω:

- **Κλήση αλγορίθμου multisort:**
  - Κώδικας:

```
#pragma omp parallel
{
   #pragma omp single
   multisort(A, Space, size);
}
```

  - Λειτουργία:
    - Η **#pragma omp parallel** ενεργοποιεί παράλληλα νήματα
    - Η **#pragma omp single** διασφαλίζει ότι μόνο ένα νήμα που ανήκει στην παράλληλη περιοχή θα εκτελέσει τον αλγόριθμο multisort.

- **Αναδρομική κλήση της multisort για τα τέσσερα τμήματα με χρήση tasks για εκτέλεση από διαφορετικά νήματα:**
  - Κώδικας:

```
#pragma omp task firstprivate(start, space, size)
multisort(startA, spaceA, quarter);

#pragma omp task firstprivate(start, space, size)
multisort(startB, spaceB, quarter);

#pragma omp task firstprivate(start, space, size)
multisort(startC, spaceC, quarter);

#pragma omp task firstprivate(start, space, size)
multisort(startD, spaceD, size - 3 * quarter);

#pragma omp taskwait
```

  - Λειτουργία:

- Η **#pragma omp task** ορίζει μία διακριτή εργασία που εκτελείται από ένα νήμα.
- Δημιουργούνται 4 εργασίες για να ταξινομηθεί το κάθε τμήμα του πίνακα, με το τελευταίο να έχει μεγαλύτερο μέγεθος λόγω ότι ο πίνακας μπορεί να μην διαιρείται ακριβώς σε 4 ισομεγέθη τμήματα.
- Κάθε εργασία θα εκτελεστεί ανεξάρτητα, αξιοποιώντας τα διαθέσιμα νήματα.
- Ο μηχανισμός αυτός προσφέρει πιο αποδοτική εκτέλεση σε αναδρομικές κλήσεις παράλληλων αλγορίθμων.
- Η οδηγία **firstprivate** ορίζει ότι κάθε νήμα που εκτελεί την διακριτή εργασία, θα έχει ιδιωτικές μεταβλητές (private) όπου θα είναι και αρχικοποιημένες (firstprivate).
- Η οδηγία **#pragma omp taskwait** συγχρονίζει τα νήματα, ώστε να περιμένουν να ολοκληρωθούν όλες οι διακριτές εργασίες για να προχωρήσει ο παράλληλος υπολογισμός.
- Χωρίς συγχρονισμό, θα είχαμε εσφαλμένα αποτελέσματα.

- **Παράλληλη συγχώνευση ανά δύο σε δύο τμήματα διπλάσιου μεγέθους:**
  - Κώδικας:

```
#pragma omp task firstprivate(start, space, size)
merge(startA, startA + quarter - 1, startB, startB + quarter - 1, spaceA);

#pragma omp task firstprivate(start, space, size)
merge(startC, startC + quarter - 1, startD, start + size - 1, spaceC);

#pragma omp taskwait
```

  - Λειτουργία:
    - Οι οδηγίες OpenMP ισχύουν όπως αναφέρθηκαν στα προαναφερόμενα.
    - Δημιουργούνται 2 εργασίες, για να συγχωνευτούν τα υπο-τμήματα ανά δύο σε δύο τμήματα διπλάσιου μεγέθους
    - Χρειαζόμαστε πάλι συγχρονισμό εργασιών με την οδηγία **#pragma omp taskwait** για να μην έχουμε λανθασμένα αποτελέσματα.

3.3 Περιγραφή της επικοινωνίας και του συγχρονισμού μεταξύ νημάτων

Η επικοινωνία και ο συγχρονισμός μεταξύ των νημάτων διασφαλίζεται με διάφορους μηχανισμούς που παρέχει η OpenMP. Παρακάτω περιγράφονται οι μέθοδοι που χρησιμοποιήθηκαν στο πρόγραμμα:

- **Ιδιωτική Μνήμη:**
  - Ιδιωτικές αρχικοποιημένες μεταβλητές (firstprivate):

Για κάθε εργασία, το νήμα που την αναλαμβάνει διαθέτει τα δικά του αντίγραφα μεταβλητών τα οποία είναι και αρχικοποιημένα, π.χ. *start, space, size* κλπ.

- Χρήση στο πρόγραμμα:

```
#pragma omp task firstprivate(start, space, size)
```

  - Οι ιδιωτικές μεταβλητές εξασφαλίζουν ότι κάθε νήμα εκτελεί ανεξάρτητες υπολογιστικές πράξεις

- **Μηχανισμοί Συγχρονισμού**
  Η σωστή λειτουργία του προγράμματος βασίζεται σε συγχρονισμό των διακριτών εργασιών, ώστε τα νήματα να περιμένουν να ολοκληρωθούν όλες οι εργασίες για να προχωρήσει η εκτέλεση του αλγορίθμου.

  - Χρήση στο πρόγραμμα:

```
#pragma omp taskwait
```

4. Δοκιμές και Αποτελέσματα

4.1 Αναφορά των συνθηκών εκτέλεσης

Η εκτέλεση γίνεται για διαφορετικά μεγέθη πίνακα N, αριθμούς νημάτων T και όριο LIMIT που αποτελεί το κριτήριο τερματισμού της αναδρομικής κλήσης του αλγορίθμου multisort.

Η μεταγλώττιση του προγράμματος γίνεται μέσω command line σε περιβάλλον Linux, με τον compiler GNU **gcc** και τον διακόπτη **–fopenmp** για την σύνδεση του με την βιβλιοθήκη **omp.h.**

```
gcc –o omp_msort omp_msort.c -fopenmp
```

Η εκτέλεση του προγράμματος γίνεται μέσω command line σε περιβάλλον Linux και πρέπει ο χρήστης να περάσει παραμετρικά 2 αρχεία txt, ώστε να αποθηκευτούν αντίστοιχα ο προς ταξινόμηση πίνακας A και ο ταξινομημένος. Ενδεικτική εντολή εκτέλεσης:

```
./omp_msort A_unsort.txt A_sort.txt
```

4.2 Παρουσίαση των αποτελεσμάτων σε μορφή κειμένου

Τα αποτελέσματα είναι αποθηκευμένα στον φάκελο Output και οι πίνακες A_unsort και A_sort εκάστως στους αντίστοιχους φακέλους. Για εξοικονόμηση χώρου δεν παρουσιάζονται όλα τα αποτελέσματα σε μορφή κειμένου στην παρούσα τεκμηρίωση. Για να ανακατευθυνθείτε στα αρχεία εξόδου πατάτε στον σύνδεσμο που είναι στις παρακάτω υπο-κεφαλίδες και αντίστοιχα για τους πίνακες που το όνομα τους βρίσκεται τόσο στους αντίστοιχους φακέλους όσο και στα αποτελέσματα σε μορφή κειμένου.

Το πρόγραμμα απαιτεί από τον χρήστη να περάσει παραμετρικά 2 .txt αρχεία εξόδου με όνομα της επιλογής του, στα οποία θα αποθηκευτούν ο μη-ταξινομημένος πίνακας A και ο ταξινομημένος. Σε περίπτωση που ο χρήστης δεν βάλει τον απαιτούμενο αριθμό παραμέτρων, το πρόγραμμα τερματίζεται και εμφανίζεται χαρακτηριστικό μήνυμα

4.2.1 Output_no_args.txt

```
Usage: ./omp_msort A_unsort.txt A_sort.txt
```

Για να ελέγξουμε την ορθότητα του αλγορίθμου δοκιμάσαμε για αρχή έναν πίνακα μικρού μεγέθους, ώστε να επιβεβαιώσουμε ότι επιτυγχάνεται η ταξινόμηση.

4.2.2 Output.txt

```
Threads             : 6
Matrix size         : 36
Limit for quicksort : 4
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort/A_unsort.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort/A_sort.txt

-----------------------------------
Multisort finished in 0.000301 sec.
-----------------------------------
```

Προκειμένου να υπολογίσουμε τις επιταχύνσεις, εκτελέσαμε τον αλγόριθμο ακολουθιακά.

4.2.3 Output_T1_N1000_L100.txt

```
Threads             : 1
Matrix size         : 1000
Limit for quicksort : 100
```

```
----------------------------------
Before sorting
----------------------------------
The A has been stored in A_unsort/A_unsort_T1_N1000_L100.txt

----------------------------------
After sorting
----------------------------------
The A has been stored in A_sort/A_sort_T1_N1000_L100.txt

----------------------------------
Multisort finished in 0.000120 sec.
```

4.2.4 Output_T1_N500000_L500.txt

```
Threads             : 1
Matrix size         : 500000
Limit for quicksort : 500
----------------------------------
Before sorting
----------------------------------
The A has been stored in A_unsort/A_unsort_T1_N500000_L500.txt

----------------------------------
After sorting
----------------------------------
The A has been stored in A_sort/A_sort_T1_N500000_L500.txt

----------------------------------
Multisort finished in 0.079525 sec.
----------------------------------
```

4.2.5 Output_T1_N100000000_L1000.txt

```
Threads             : 1
Matrix size         : 100000000
Limit for quicksort : 1000
----------------------------------
Before sorting
----------------------------------
The A has been stored in A_unsort/A_unsort_T1_N100000000_L1000.txt

----------------------------------
After sorting
----------------------------------
The A has been stored in A_sort/A_sort_T1_N100000000_L1000.txt

----------------------------------
Multisort finished in 19.709039 sec.
```

```
---------------------------------
```

4.2.6 Output_T2_N1000_L100.txt

```
Threads             : 2
Matrix size         : 1000
Limit for quicksort : 100
---------------------------------
Before sorting
---------------------------------
The A has been stored in A_unsort/A_unsort_T2_N1000_L100.txt

---------------------------------
After sorting
---------------------------------
The A has been stored in A_sort/A_sort_T2_N1000_L100.txt

---------------------------------
Multisort finished in 0.000197 sec.
---------------------------------
```

4.2.7 Output_T3_N5000_L100.txt

```
Threads             : 3
Matrix size         : 5000
Limit for quicksort : 100
---------------------------------
Before sorting
---------------------------------
The A has been stored in A_unsort/A_unsort_T3_N5000_L100.txt

---------------------------------
After sorting
---------------------------------
The A has been stored in A_sort/A_sort_T3_N5000_L100.txt

---------------------------------
Multisort finished in 0.000470 sec.
---------------------------------
```

4.2.8 Output_T4_N1000000_L500.txt

```
Threads             : 4
Matrix size         : 1000000
Limit for quicksort : 500
---------------------------------
Before sorting
---------------------------------
The A has been stored in A_unsort/A_unsort_T4_N1000000_L500.txt
```

```
----------------------------------
After sorting
----------------------------------
The A has been stored in A_sort/A_sort_T4_N1000000_L500.txt

----------------------------------
Multisort finished in 0.057440 sec.
----------------------------------
```

4.2.9 Output_T6_N500000_L500.txt

```
Threads             : 6
Matrix size         : 500000
Limit for quicksort : 500
----------------------------------
Before sorting
----------------------------------
The A has been stored in A_unsort/A_unsort_T6_N500000_L500.txt

----------------------------------
After sorting
----------------------------------
The A has been stored in A_sort/A_sort_T6_N500000_L500.txt

----------------------------------
Multisort finished in 0.021820 sec.
----------------------------------
```

4.2.10 Output_T8_N100000000_L1000.txt

```
Threads             : 8
Matrix size         : 100000000
Limit for quicksort : 1000
----------------------------------
Before sorting
----------------------------------
The A has been stored in A_unsort/A_unsort_T8_N100000000_L1000.txt

----------------------------------
After sorting
----------------------------------
The A has been stored in A_sort/A_sort_T8_N100000000_L1000.txt

----------------------------------
Multisort finished in 6.076829 sec.
----------------------------------
```

4.2.11 Output_T12_N50000000_L1000.txt

```
Threads             : 12
Matrix size         : 50000000
Limit for quicksort : 1000
---------------------------------
Before sorting
---------------------------------
The A has been stored in A_unsort/A_unsort_T12_N50000000_L1000.txt

---------------------------------
After sorting
---------------------------------
The A has been stored in A_sort/A_sort_T12_N50000000_L1000.txt

---------------------------------
Multisort finished in 2.945010 sec.
---------------------------------
```

4.2.12 Output_T16_N100000000_L1000.txt

```
Threads             : 16
Matrix size         : 100000000
Limit for quicksort : 1000
---------------------------------
Before sorting
---------------------------------
The A has been stored in A_unsort/A_unsort_T16_N100000000_L1000.txt

---------------------------------
After sorting
---------------------------------
The A has been stored in A_sort/A_sort_T16_N100000000_L1000.txt

---------------------------------
Multisort finished in 6.174742 sec.
---------------------------------
```

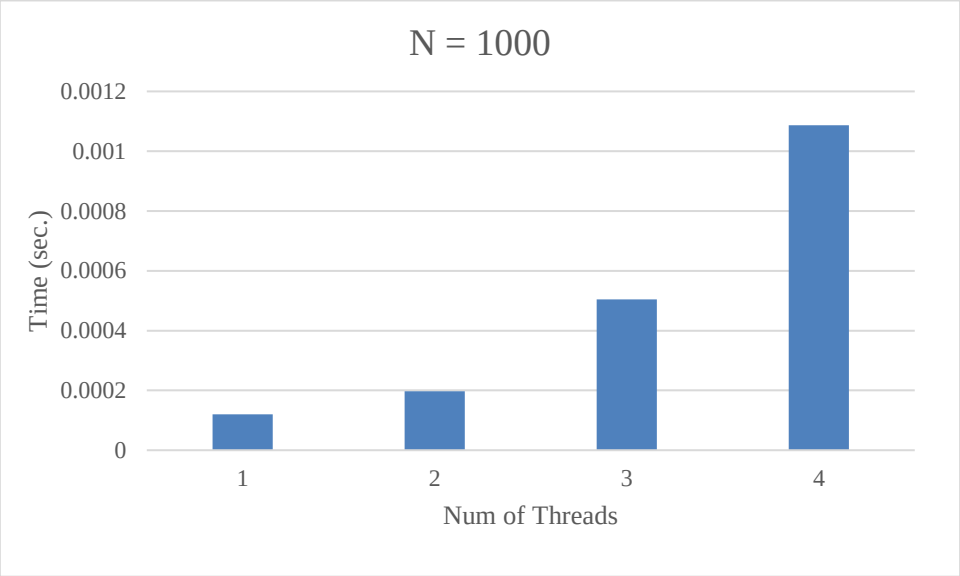
4.3 Ανάλυση της αποδοτικότητας

4.3.1 Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου

Οι χρόνοι που καταγράφηκαν για διαφορετικό αριθμό νημάτων T και μεγέθους πίνακα N παρουσιάζονται στα παρακάτω διαγράμματα. Συγκεκριμένα, τα δεδομένα των αποτελεσμάτων είναι:

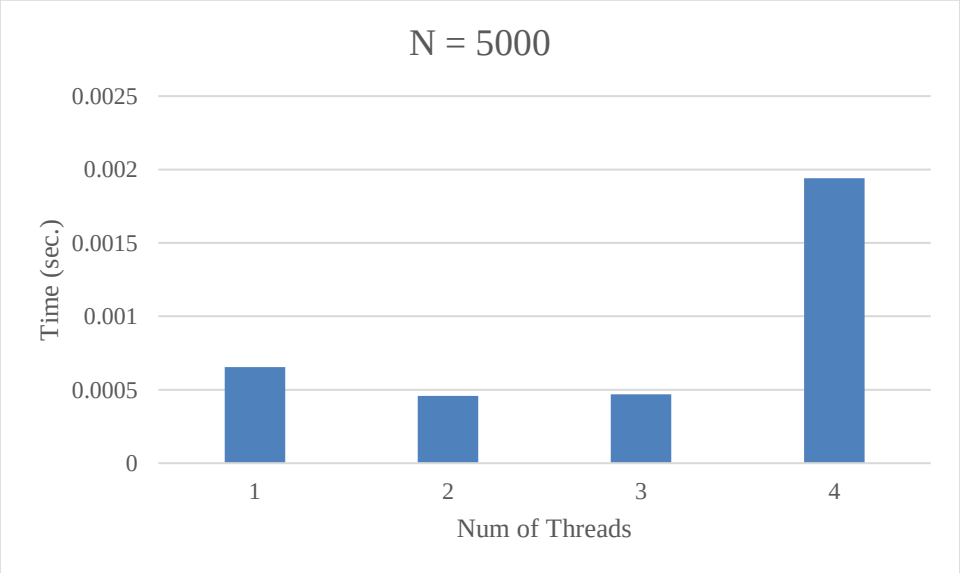
- **Αριθμός νημάτων:** 1, 2, 3, 4, 6, 8, 12 και 16
- **Μέγεθος Πίνακα:** 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000

Threads	Total Time
1	0,00012
2	0,000197
3	0,000504
4	0,001087



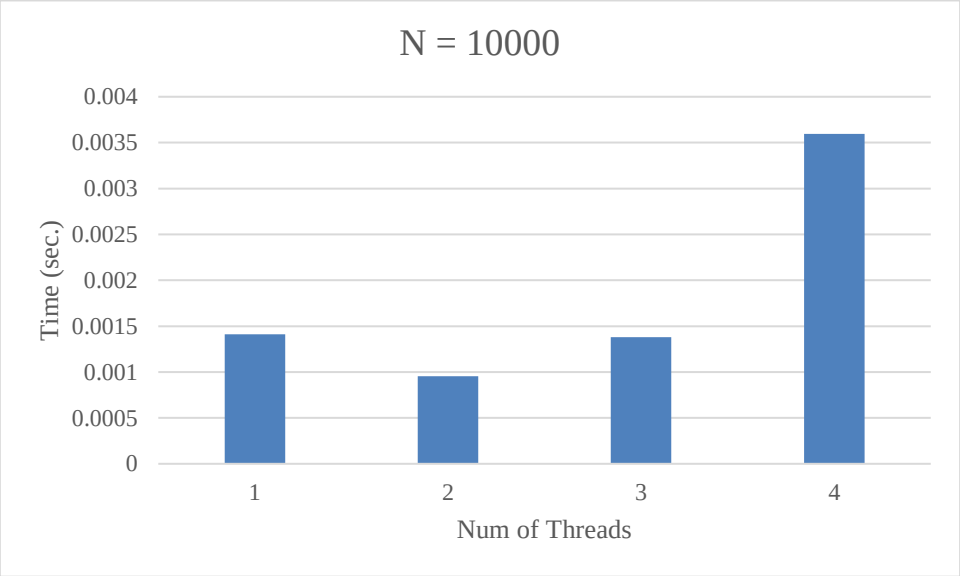
Εικόνα 1. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 1000

Threads	Total Time
1	0,000654
2	0,000459
3	0,00047
4	0,00194



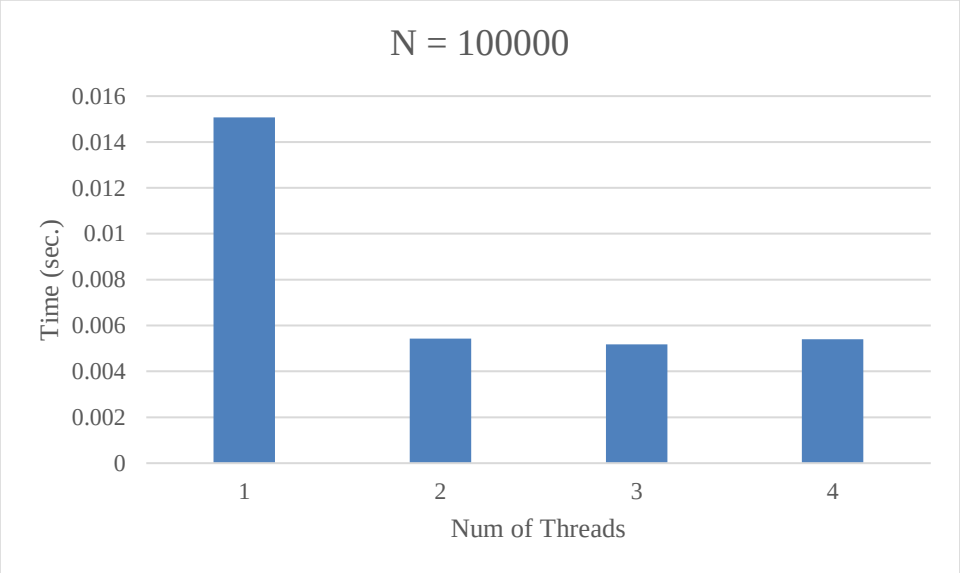
Εικόνα 2. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 5000

Threads	Total Time
1	0,001412
2	0,000955
3	0,00138
4	0,003595



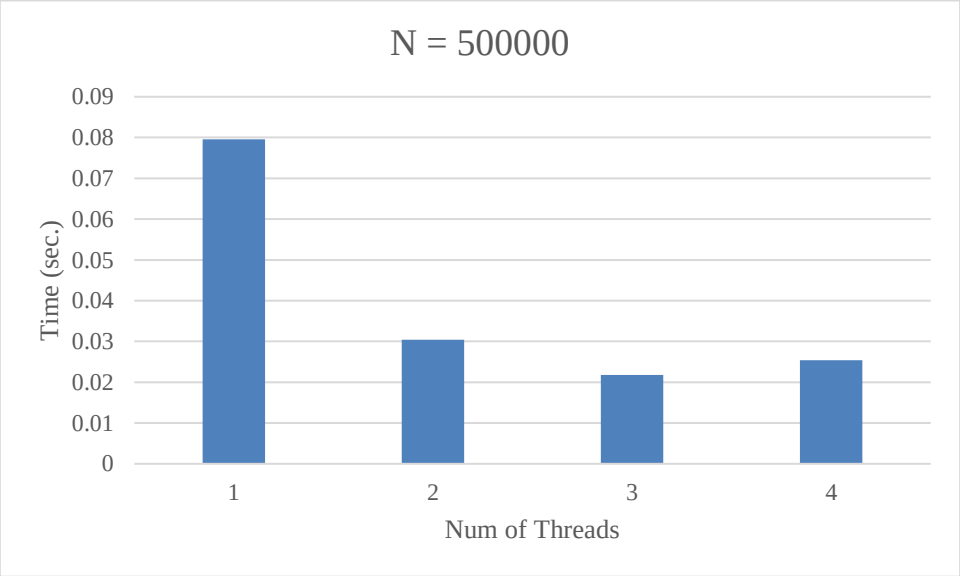
Εικόνα 3. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 10000

Threads	Total Time
1	0,015074
4	0,005436
6	0,005173
8	0,005402



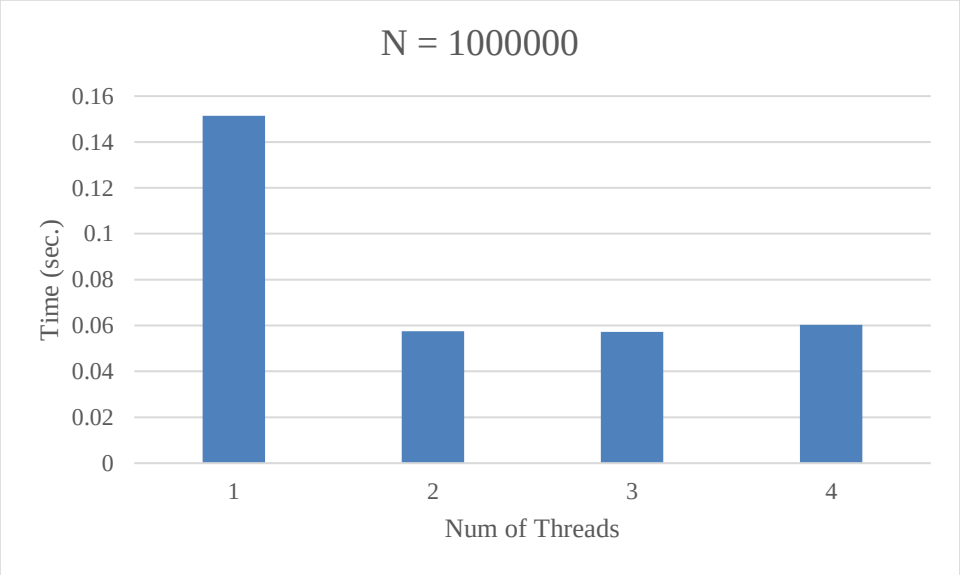
Εικόνα 4. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 100000

Threads	Total Time
1	0,079525
4	0,030407
6	0,02182
8	0,025397



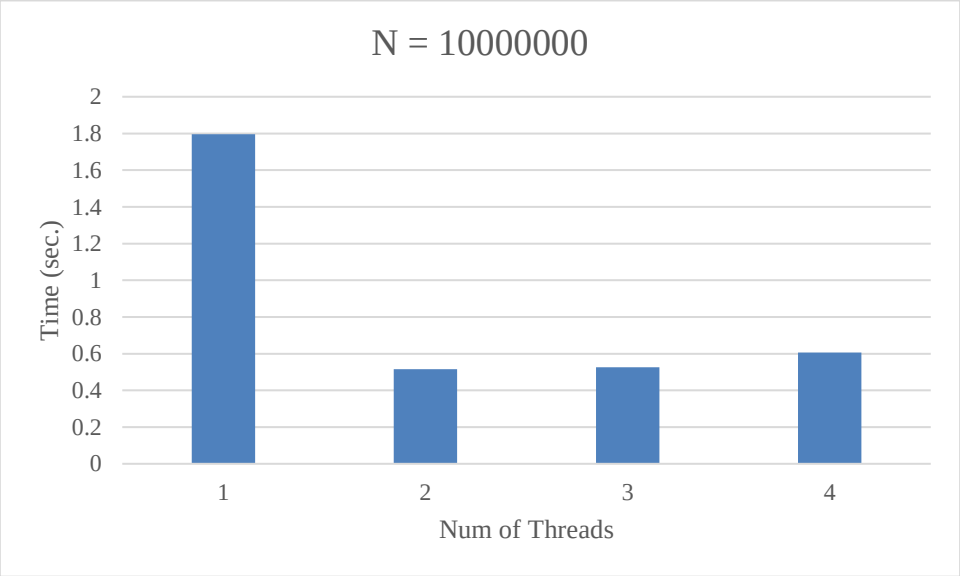
Εικόνα 5. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 500000

Threads	Total Time
1	0,151469
4	0,05744
6	0,057255
8	0,060294



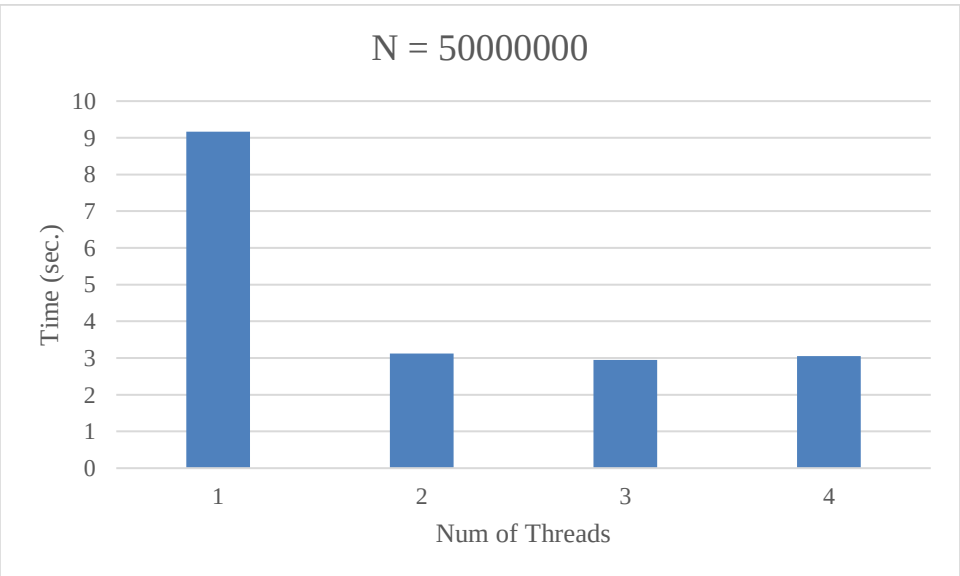
Εικόνα 6. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 1000000

Threads	Total Time
1	1,795719
8	0,515042
12	0,525958
16	0,607034



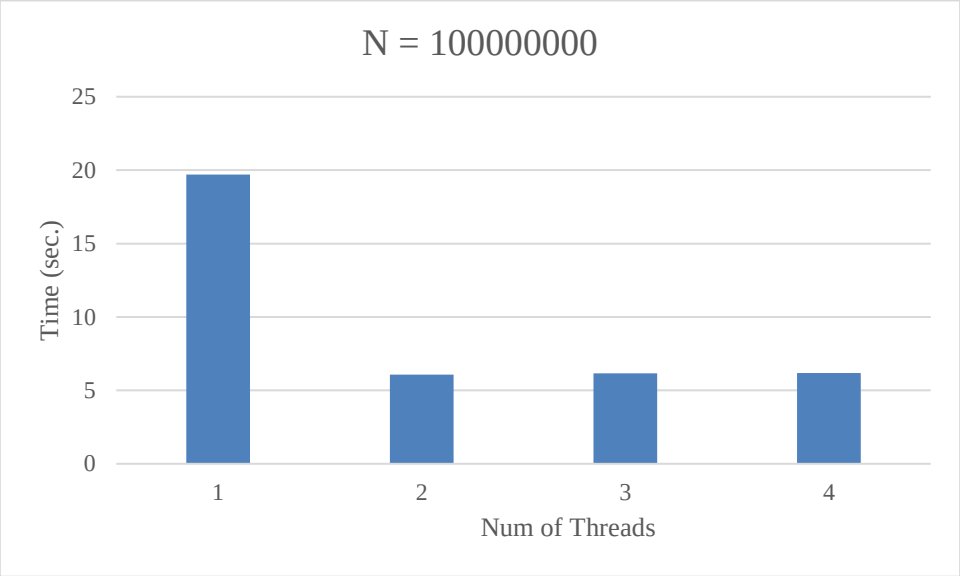
Εικόνα 7. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 10000000

Threads	Total Time
1	9,168969
8	3,122941
12	2,94501
16	3,050295



Εικόνα 8. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 50000000

Threads	Total Time
1	19,70904
8	6,076829
12	6,154808
16	6,174742



Εικόνα 9. Χρόνοι εκτέλεσης του παράλληλου αλγορίθμου για N = 100000000

4.3.2 Επιταχύνσεις

Οι επιταχύνσεις υπολογίζονται από τον παρακάτω μαθηματικό τύπο:

$$\text{Speedup} = {}^{t_s}/_{t_n}$$

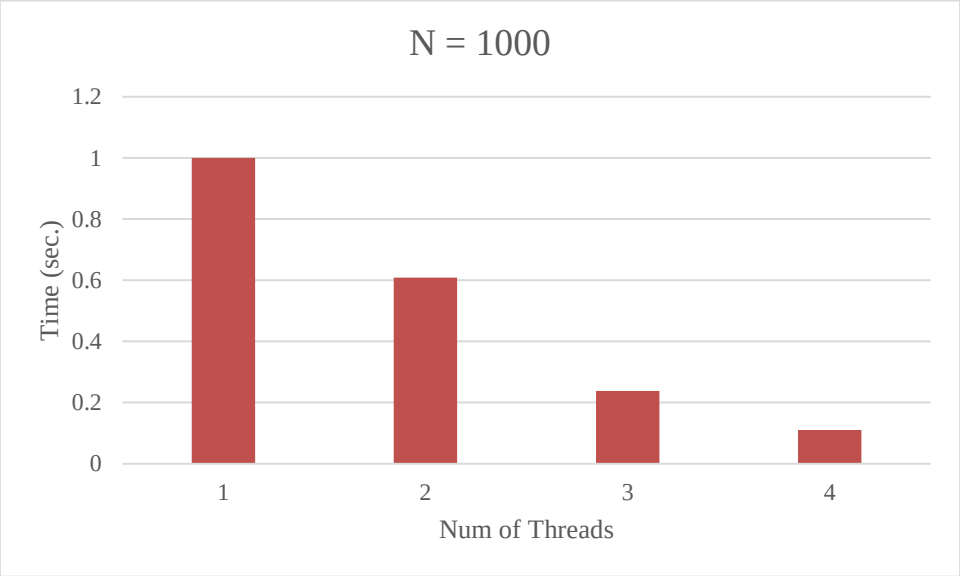
t_s: Ο χρόνος εκτέλεσης του ακολουθιακού προγράμματος, δηλαδή, εκτέλεση μ' ένα 1 νήμα
t_n: Ο χρόνος εκτέλεσης του παράλληλου προγράμματος, δηλαδή, εκτέλεση με n νήματα

Πηγή: Μάθημα 2ο – Παράλληλος Υπολογισμός – Εισαγωγή στον Παράλληλο Υπολογισμό – σελ. 7

Οι επιταχύνσεις παρουσιάζονται στα παρακάτω διαγράμματα. Συγκεκριμένα, τα δεδομένα των αποτελεσμάτων είναι:

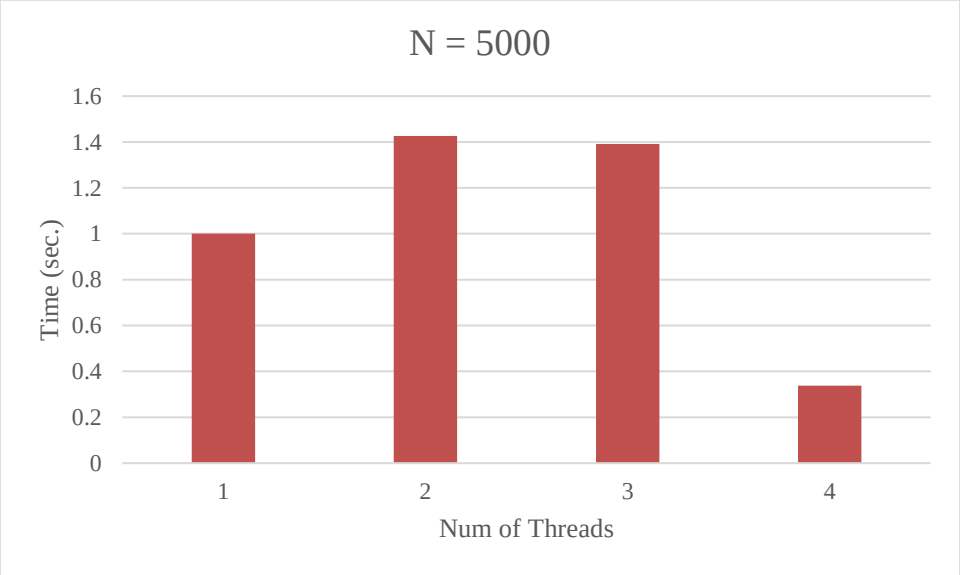
- **Αριθμός νημάτων:** 1, 2, 3, 4, 6, 8, 12 και 16
- **Μέγεθος Πίνακα:** 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000

Threads	Speed-up
1	1
2	0,609
3	0,238
4	0,11



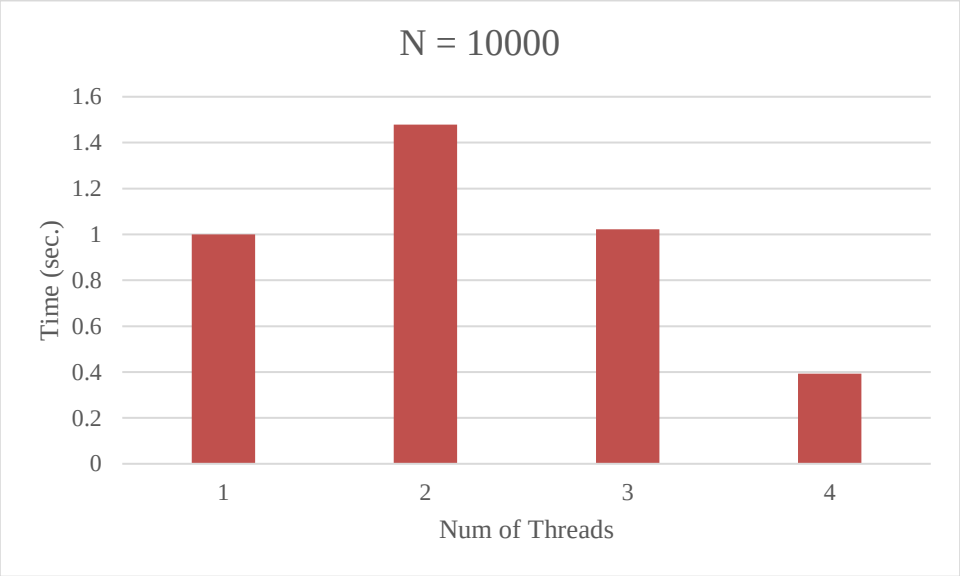
Εικόνα 10. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 1000

Threads	Speed-up
1	1
2	1,426
3	1,392
4	0,338



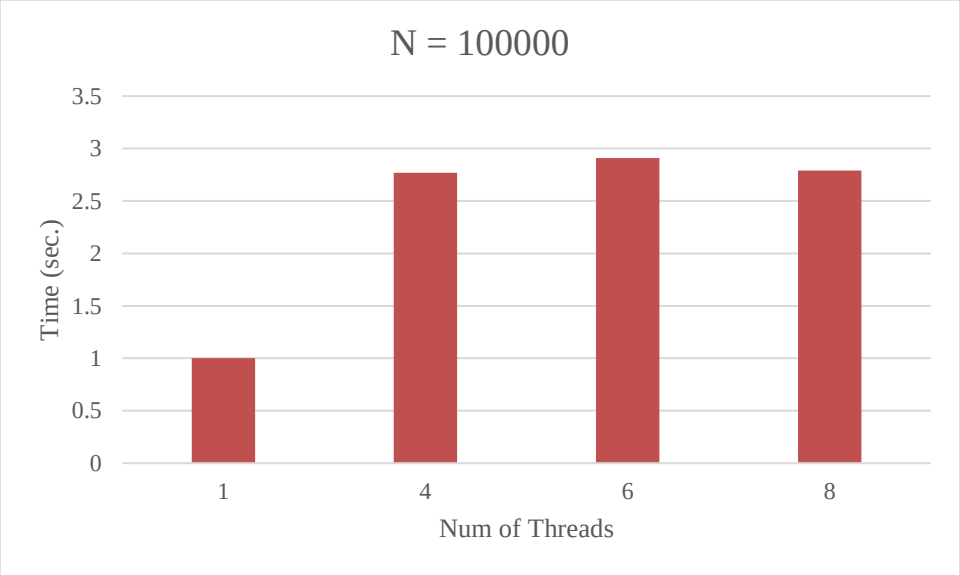
Εικόνα 11. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 5000

Threads	Speed-up
1	1
2	1,479
3	1,023
4	0,393



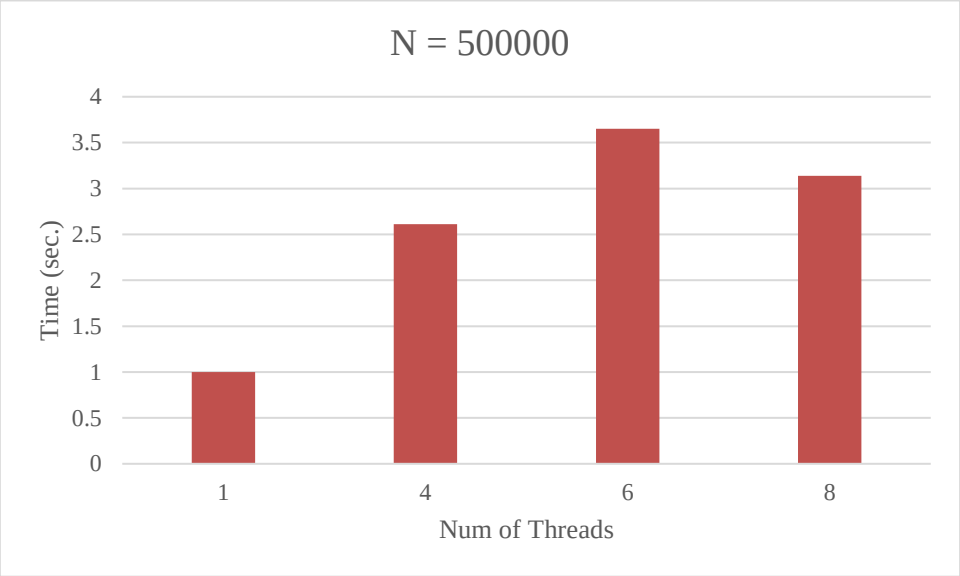
Εικόνα 12. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 10000

Threads	Speed-up
1	1
4	2,77
6	2,91
8	2,79



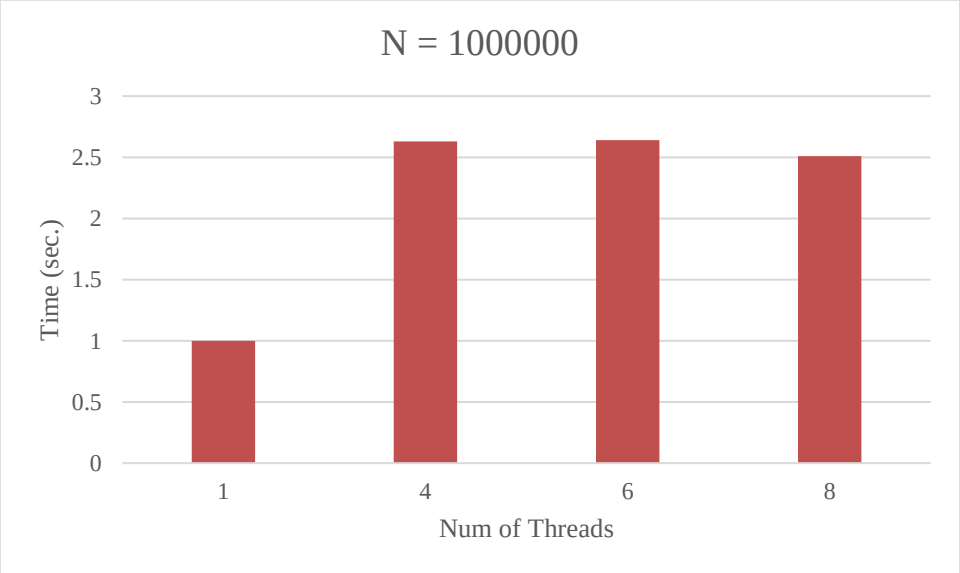
Εικόνα 13. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 100000

Threads	Speed-up
1	1
4	2,61
6	3,65
8	3,14



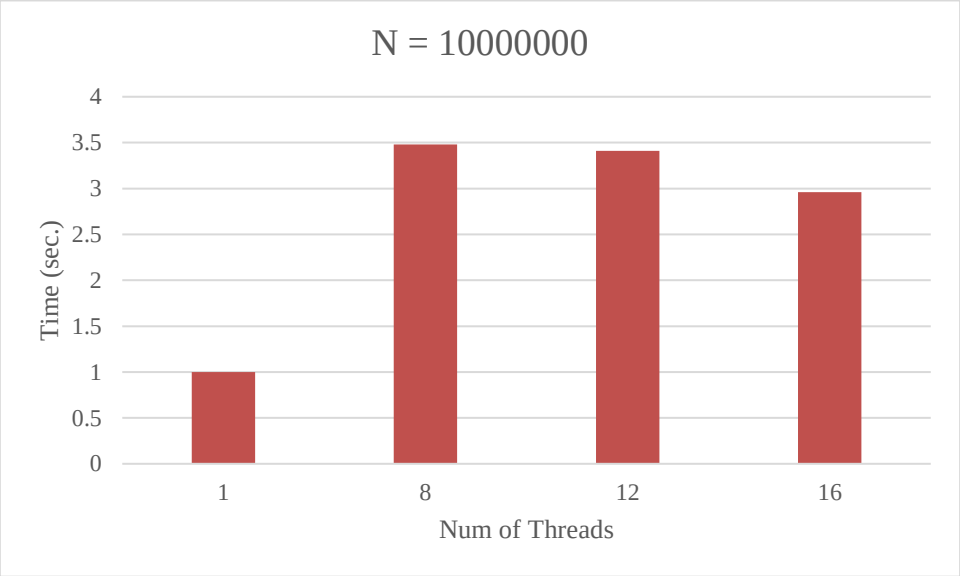
Εικόνα 14. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 500000

Threads	Speed-up
1	1
4	2,63
6	2,64
8	2,51



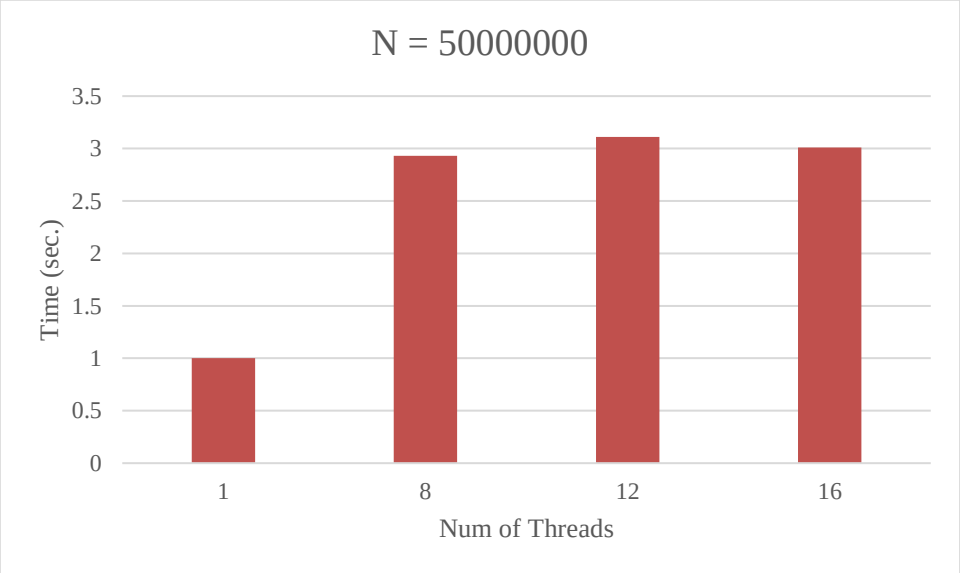
Εικόνα 15. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 1000000

Threads	Speed-up
1	1
8	3,48
12	3,41
16	2,96



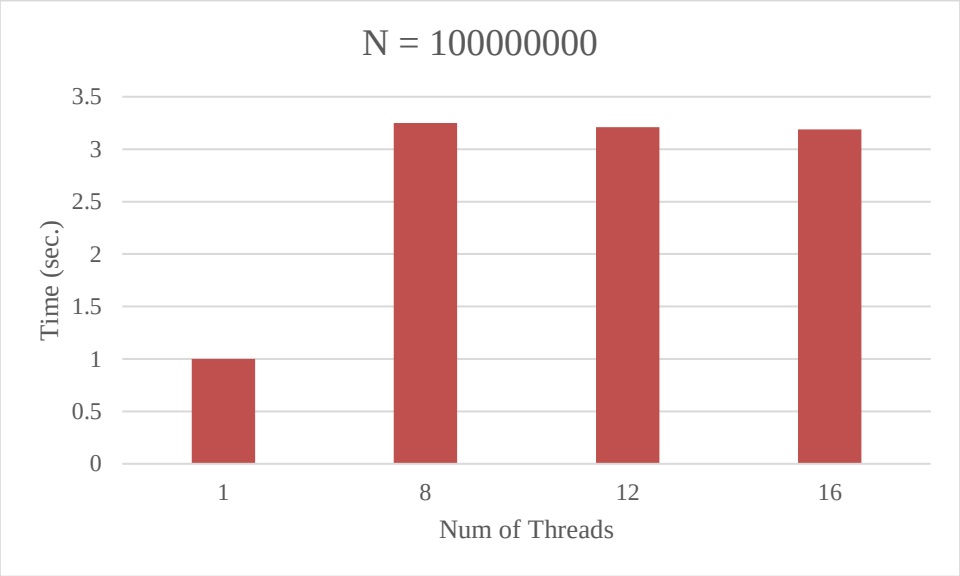
Εικόνα 16. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 10000000

Threads	Speed-up
1	1
8	2,93
12	3,11
16	3,01



Εικόνα 17. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 50000000

Threads	Speed-up
1	1
8	3,25
12	3,21
16	3,19



Εικόνα 18. Επιταχύνσεις του παράλληλου αλγορίθμου για N = 100000000

4.3.3 Παρατηρήσεις

Παρατηρούμε ότι για μικρά N, η παραλληλία δεν αποδίδει καλύτερους χρόνους από την ακολουθιακή. Αυτό οφείλεται και στο γεγονός ότι η δημιουργία και η διαχείριση περισσότερων νημάτων μπορεί να επιφέρει πρόσθετη επιβάρυνση παρά λιγότερη. Ο χρόνος αυτός αντικρούει τα οφέλη του παράλληλου υπολογισμού, καθώς, τα νήματα ανταγωνίζονται για μνήμη λόγω του μικρού όγκου δεδομένων προκαλώντας καθυστερήσεις. Επίσης, υπάρχουν και καθυστερήσεις στον συγχρονισμό, ειδικά όταν έχουμε να κάνουμε με διακριτές εργασίες που αναλαμβάνει 1 νήμα και μηχανισμούς συγχρονισμού με οδηγίες taskwait.

Για παράδειγμα, στα δεδομένα που συλλέξαμε για **N = 1000**, παρατηρούμε ότι οι χρόνοι εκτέλεσης αυξάνονται όσο αυξάνουμε τον αριθμό των ενεργών νημάτων και γι' αυτό δεν πετυχαίνουμε επιταχύνσεις > 1 sec, καθώς, ο χρόνος εκτέλεσης των εργασιών με 1 νήμα είναι μικρότερος από τον αντίστοιχο με 2, 3 ή και 4 νήματα.

Ωστόσο, τα οφέλη του παράλληλου υπολογισμού διακρίνονται για μεγάλα N, όπως παρατηρούμε στα δεδομένα που συλλέξαμε για **N = 100000000**, καθώς, εκεί πετυχαίνουμε καλύτερους χρόνους με περισσότερα νήματα και συνεπώς και καλύτερες επιταχύνσεις. Αυτό οφείλεται και στην ανάθεση των εργασιών στα διαθέσιμα νήματα, όπου ελαχιστοποιείται ο χρόνος αδράνειας λόγω του μεγάλου όγκου δεδομένων.

Συνεπώς, τα οφέλη του παράλληλου υπολογισμού διακρίνονται πιο αποδοτικά για μεγάλο όγκο δεδομένων, καθώς έτσι δεν έχουμε νήματα που παραμένουν ανενεργά.

5. Προβλήματα και Αντιμετώπιση

5.1 Αναφορά προβλημάτων

Πρόβλημα εντοπίστηκε στην διαχείριση των δεικτών στην ρουτίνα multisort και merge. Συγκεκριμένα, δεν κατανοήθηκε πλήρως ο ρόλος του δείκτη space και πώς θα αποθηκεύαμε τον τελικό ταξινομημένο πίνακα. Πρόβλημα εντοπίσαμε και στο 4ο τμήμα του πίνακα, όπου αν δεν διαιρούτανε ο πίνακας διά 4 ακριβώς, τότε περίσσευαν στοιχεία του πίνακα που δεν ταξινομήθηκαν.

Επίσης, πρόβλημα εντοπίσαμε και στους χρόνους εκτέλεσης των παράλληλων εργασιών, όταν το περιβάλλον εργασίας ήταν το WSL (Windows Subsystem for Linux), καθώς, έπαιρνε παραπάνω χρόνος να ολοκληρωθεί ο παράλληλος αλγόριθμος από το αναμενόμενο.

5.2 Λύσεις που δοκιμάστηκαν και εφαρμόστηκαν

Εν τέλη, ο δείκτης space έδειχνε σ' έναν προσωρινό πίνακα αποθήκευσης όπου θα τον χρησιμοποιούσαμε για αντιγράψουμε τον ταξινομημένο τελικό πίνακα στον δείκτη που έδειχνε στον αρχικό. Όσο για το πρόβλημα του διαμοιρασμού στο 4ο τμήμα του πίνακα αλλάξαμε το μέγεθος που θα εξασφαλίσει ότι το 4ο τμήμα θα έχει και τα περίσσεια στοιχεία σε περίπτωση που το μέγεθος δεν διαιρείται με το 4:

```
multisort(startD, spaceD, size - 3 * quarter);
```

Για τους χρόνους, επιλέξαμε να αλλάξουμε περιβάλλον και να πάμε σε καθαρή διανομή Linux (Ubuntu) και τα αποτελέσματα ήτανε σαφώς καλύτερα.

WSL

```
Threads            : 6
Matrix size        : 10000000
Limit for quicksort : 100
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort.txt

-----------------------------------
Multisort finished in 0.670413 sec.
-----------------------------------
```

Linux

```
Threads            : 6
Matrix size        : 10000000
Limit for quicksort : 100
---------------------------------
Before sorting
---------------------------------
The A has been stored in A_unsort.txt

---------------------------------
After sorting
---------------------------------
The A has been stored in A_sort.txt

---------------------------------
Multisort finished in 0.562977 sec.
---------------------------------
```

Έπειτα από σχετική έρευνα στο διαδίκτυο ανακαλύψαμε τον λόγο που στο WSL οι χρόνοι εκτέλεσης παράλληλων εργασιών είναι μεγαλύτεροι:

OMP is incredibly slow in WSL2 due to filesystem boundary

https://github.com/JanDeDobbeleer/oh-my-posh/issues/1268

Το πρόβλημα απόδοσης με το OpenMP στην WSL2 μπορεί συχνά να αποδοθεί στον τρόπο με τον οποίο το WSL2 χειρίζεται το σύστημα αρχείων. Το WSL2 χρησιμοποιεί ένα εικονικοποιημένο περιβάλλον που αλληλεπιδρά με το σύστημα αρχείων των Windows και αυτή η αλληλεπίδραση μπορεί να εισάγει σημαντική καθυστέρηση όταν υπάρχουν συχνές λειτουργίες εισόδου/εξόδου. Αυτό γίνεται ιδιαίτερα αισθητό με εργαλεία όπως το OpenMP, όπου τα παράλληλα νήματα μπορεί να αλληλεπιδρούν σε μεγάλο βαθμό με το σύστημα αρχείων για αποθήκευση δεδομένων ή συγχρονισμό.

6. Συμπεράσματα

6.1 Ανακεφαλαίωση

Ανακεφαλαιώνοντας, ο παράλληλος αλγόριθμος multisort με OpenMP tasks είναι μια αποδοτική προσέγγιση για την επίτευξη γρήγορης ταξινόμησης μεγάλων συνόλων δεδομένων σε παράλληλο περιβάλλον, εκμεταλλευόμενος τις δυνατότητες των σύγχρονων επεξεργαστών.

Η τεχνική του διαίρει και βασίλευε σε συνδυασμό με τον παράλληλο υπολογισμό επιτυγχάνουν εκπληκτικές επιδόσεις για μεγάλο όγκο δεδομένων και αυτό φαίνεται και στις επιταχύνσεις που καταγράψαμε για μεγάλα μεγέθη σε σύγκριση με μικρά.

ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

Σας ευχαριστώ για την προσοχή σας.