

Παραλληλισμος αλγορίθμων Bubble sort, Bucket sort και της μεθόδου του Otsu

Δημοκρίτειο Πανεπιστήμιο Θράκης
Τμήμα ηλεκτρολόγων και ηλεκτρολόγων μηχανικών
Εργαστήριο Αρχιτεκτονικής Υπολογιστών
και Συστημάτων Υψηλών επιδόσεων
Τεχνολογία παράλληλης Επεξεργασίας
Ευάγγελος Λουρμπάκης
evanlour@ee.duth.gr

May 7, 2025

Περιεχόμενα

1	Εισαγωγή	2
2	Ανάλυση λειτουργίας των αλγορίθμων	2
2.1	Ο αλγόριθμος Bubble sort	2
2.2	Ο αλγόριθμος Odd even sort	3
2.3	Ο αλγόριθμος Bucket sort	4
2.4	Ο αλγόριθμος Merge sort	5
3	Otsu's thresholding method	7
3.1	Η σημασία του thresholding στην ψηφιακή επεξεργασία	7
3.2	Ο τρόπος λειτουργίας του αλγορίθμου του Otsu	7
3.3	Ο αλγόριθμος του Otsu από τη μεριά του προγραμματιστή	7
4	Απόδοση αλγορίθμων με τη βιβλιοθήκη OpenMP	8
4.1	Το περιβάλλον εκτέλεσης	8
4.2	Τα αποτελέσματα για τον Bubble sort	9
4.3	Τα αποτελέσματα για τον Bucket sort	11
4.4	Τα αποτελέσματα για τη μέθοδο του Otsu	12
5	Συμπεράσματα	14

1 Εισαγωγή

Τη σήμερων ημέρα παρατηρείται μια συνεχής ανοδική τάση στον αριθμό των επεξεργαστών και στον επαγγελματικό τομέα αλλά και στο χώρο του καταναλωτή. Καθώς δεν μπορεί να γίνει αυτόματη ανάθεση των διεργασιών για να αξιοποιηθούν όλοι οι πυρήνες, δημιουργείται η ανάγκη για ανακατασκευή της πλειονότητας των προγραμμάτων με σκοπό να αξιοποιηθεί πλήρως όλη η διαθέσιμη υπολογιστική δύναμη. Επιδιώκεται να προσδιοριστεί το ποσοστό επιτάχυνσης των αλγορίθμων *Bubble sort* και *Bucket sort*, με τη χρήση του αλγορίθμου *Merge sort* για την εσωτερική αναδιάταξη των δεξαμενών του αλγορίθμου. Επίσης θα εξεταστεί η επιτάχυνση της μεθόδου του Otsu. Για την παραλληλοποίηση των παρακάτω αλγορίθμων θα χρησιμοποιηθεί η βιβλιοθήκη [2] *OpenMP* στη γλώσσα προγραμματισμού C.

2 Ανάλυση λειτουργίας των αλγορίθμων

2.1 Ο αλγόριθμος Bubble sort

Ο αλγόριθμος *Bubble sort* είναι από τους πιο απλούς και διαδεδομένους αλγορίθμους που υπάρχουν. Ο αλγόριθμος ξεκινάει από την αρχή του πίνακα και συγκρίνει διαδοχικά όλους τους αριθμούς για να δει εάν βρίσκεται στη θέση που πρέπει. Μετα συνεχίζει στο δεύτερο στοιχείο του πίνακα κοκ. Όπως είναι προφανές, ο αλγόριθμος εμφανίζει υψηλή πολυπλοκότητα χρόνου και συγκεκριμένα εμφανίζει $O(N^2)$ πολυπλοκότητα.

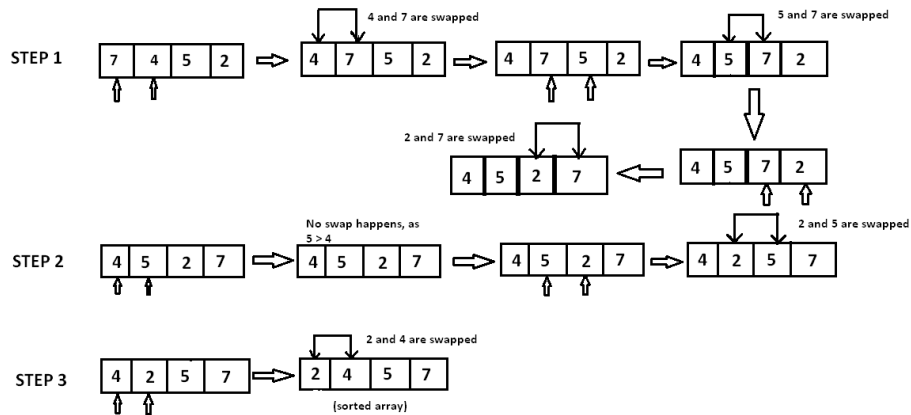


Figure 1: Ο τρόπος λειτουργίας του Bubble sort [3].

Ενα ακόμη πρόβλημα που προκύπτει είναι πως λόγω της φύσης του αλγορίθμου είναι αρκετά δύσκολο να παραλληλοποιηθεί κατάλληλα χωρίς να χάσει αρκετή απόδοση λόγω του χρόνου συγχρονισμού. Για να αποφευχθούν οι καθυστερήσεις

Algorithm 1 Ο αλγόριθμος Bubble sort

```
 $i \leftarrow 0$   
 $N \leftarrow \text{array size}$   
 $\text{array}[N]$   
for  $i \leftarrow 0$  to  $N - 1$  do  
  for  $j \leftarrow 0$  to  $N - i - 1$  do  
    if  $\text{array}[j] > \text{array}[j + 1]$  then  
       $\text{swap}(\text{array}[j], \text{array}[j + 1])$ 
```

λόγω συγχρονισμού, θα χρησιμοποιηθεί μια παραλλαγή του αλγορίθμου *Bubble sort*: Τον αλγόριθμο *Odd even sort*.

2.2 Ο αλγόριθμος Odd even sort

Ο αλγόριθμος *Odd even sort* είναι μια ευνοϊκή παραλλαγή του *Bubble sort* κατά την οποία ελέγχεται εναλλάξ το κάθε στοιχείο του πίνακα με το δεξιά του και με το αριστερά του. Ενώ σε πολυπλοκότητα παρουσιάζει ίδιο θεωρητικό χρόνο εκτέλεσης, λόγω του ότι ελέγχεται το κάθε στοιχείο με το διπλανό του είναι πολύ πιο εύκολο και αποδοτικό να παραλληλοποιηθεί η διαδικασία: N πυρήνες ελέγχουν ανά 2 τα στοιχεία στα αριστερά τους και έπειτα αυτά στα δεξιά τους. Επειδή κάθε φορά για κάθε νήμα τα στοιχεία θα είναι ανεξάρτητα των υπολοίπων, δεν χάνουμε χρόνο για συγχρονισμό, το οποίο μας επιτρέπει να παρατηρήσουμε σημαντικές βελτιώσεις στο χρόνο εκτέλεσης.

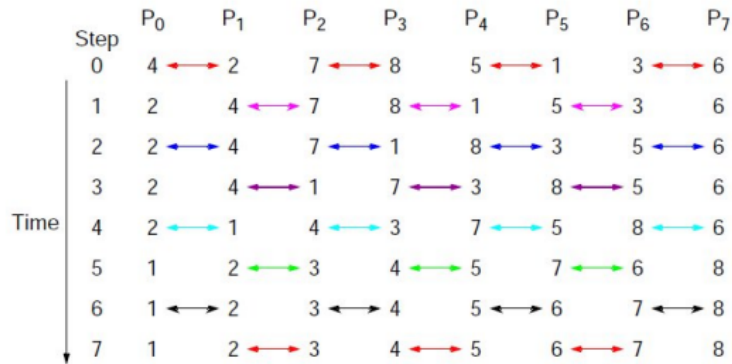


Figure 2: Ο τρόπος λειτουργίας του Odd even sort [7].

Algorithm 2 Ο αλγόριθμος Odd even sort

```
 $i \leftarrow 0$   
 $j \leftarrow 0$   
 $N \leftarrow \text{array size}$   
 $\text{array}[N]$   
while array is not sorted do  
  for  $i \leftarrow 1$  to  $N - 1$  with jumpsize: 2 do  
    if  $\text{array}[i] > \text{array}[i + 1]$  then  
       $\text{swap}(\text{array}[i], \text{array}[i + 1])$   
  for  $j \leftarrow 1$  to  $N - 1$  with jumpsize: 2 do  
    if  $\text{array}[j] > \text{array}[j + 1]$  then  
       $\text{swap}(\text{array}[j], \text{array}[j + 1])$ 
```

Είναι προφανές πως η μετατροπή η οποία υλοποιήθηκε στον αλγόριθμο επιτρέπει την αποδοτική αξιοποίηση περισσότερων πυρήνων του επεξεργαστή, ακόμη και GPU [4]

2.3 Ο αλγόριθμος Bucket sort

Ο αλγόριθμος *Bucket sort* διαχωρίζει τα στοιχεία του πίνακα σε N δεξαμενές. Έπειτα, στη κάθε δεξαμενή ανακατατάσει τα στοιχεία και στο τέλος αποθηκεύει με τη σειρά τα στοιχεία των containers στον πίνακα εξόδου. Εσωτερικά του container μπορούν να επιλεγθούν διάφοροι αλγόριθμοι. Επιλέχθηκε να χρησιμοποιηθεί ο αλγόριθμος *Merge sort* διότι παρουσιάζει λογαριθμική χρονική πολυπλοκότητα στο χρόνο. Η διαδικασία παραλληλοποίησης είναι σχετικά απλή: Γίνεται διαμοιρασμός των αριθμών σε N threads και τοποθετούνται τα στοιχεία στις ανάλογες τοπικές δεξαμενές. Έπειτα αρκεί να ενωθούν οι ανάλογες δεξαμενές μεταξύ τους και έπειτα να συνδεθούν σε σειρά για να δημιουργηθεί τελικά ο πίνακας εξόδου. Έτσι αποφεύγεται η εκτεταμένη χρήση εντολών συγχρονισμού μεταξύ των νημάτων και επιτρέπει την πλήρη αξιοποίηση του διαθέσιμου υλικού.

Algorithm 3 Ο αλγόριθμος Bucket sort

```
 $N \leftarrow \text{array size}$   
 $M \leftarrow \text{number of containers}$   
 $\text{array}[N]$   
 $\text{containers}[\text{number of containers}]$   
 $i \leftarrow 0$   
 $j \leftarrow 0$   
for  $i \leftarrow 0$  to  $N - 1$  do  
  Send  $\text{array}[i]$  To the corresponding container  
for  $j \leftarrow 0$  to  $M$  do  
  merge sort ( $\text{containers}[i]$ )  
 $\text{array} \leftarrow \text{Concatenate}(\text{containers})$ 
```

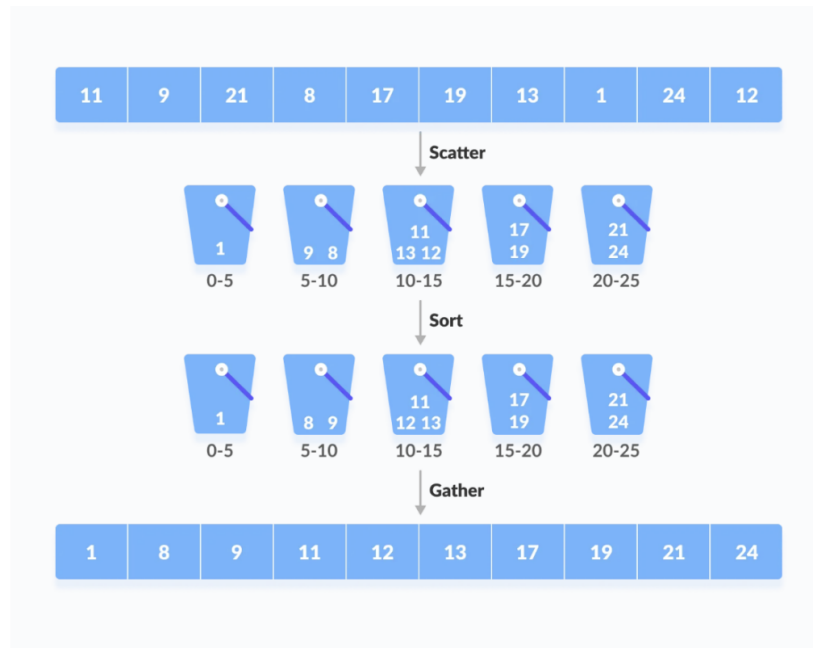


Figure 3: Ο τρόπος λειτουργίας του Bucket sort [5].

2.4 Ο αλγόριθμος Merge sort

Για την εσωτερική ανακατανομή της κάθε δεξαμενής έχει επιλεχθεί να χρησιμοποιηθεί ο αλγόριθμος *merge sort*. Ο *merge sort* είναι ιδανικός γιατί είναι ευσταθής αλγόριθμος, το οποίο είναι προαπαιτούμενο για τον *bucket sort* και επίσης παρουσιάζει γρήγορους χρόνους εκτέλεσης. Αξίζει να σημειωθεί πως στα πειραματικά δεδομένα επιλέχθηκε να μην υλοποιηθεί παράλληλα διότι αρχικά είναι γνωστή η επιτάχυνση του αλγορίθμου όταν αξιοποιούνται παράλληλες μέθοδοι [8], καθώς επίσης και για φυσιολογικούς αριθμούς δεξαμενών το overhead θα υποβάθμιζε σημαντικά την οποιαδήποτε επιτάχυνση λόγω παραλληλοποίησης. Ο *merge sort* έχει την εξής αρχή λειτουργίας: Διασπά τον αρχικό μεγάλο πίνακα σε ζεύγη των 2, τα ανακατατάσει εσωτερικά και μετά κατασκευάζει σταδιακά τον υπόλοιπο πίνακα παίρνοντας κάθε φορά τον μικρότερο διαθέσιμο αριθμό. Με αυτόν τον τρόπο επιτυγχάνει ταχύτατους χρόνους εκτέλεσης.

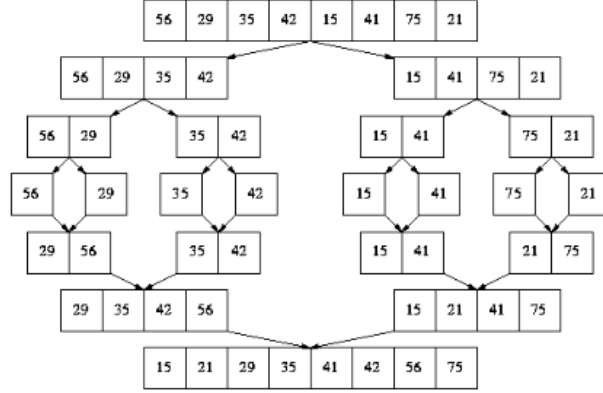


Figure 4: Ο τρόπος λειτουργίας του merge sort [5].

Algorithm 4 Ο αλγόριθμος Merge sort

```

MergeSortArray(array, size)
if size  $\leq 1$  then
   $\lfloor$  return
half  $\leftarrow$  size / 2
left  $\leftarrow$  new array of size half
right  $\leftarrow$  new array of size (size - half)
for i = 0 to half - 1 do
   $\lfloor$  left[i]  $\leftarrow$  arr[i]
for i = 0 to size - half - 1 do
   $\lfloor$  right[i]  $\leftarrow$  arr[half + i]
MergeSortArray(left, half)
MergeSortArray(right, size - half)
leftPointer  $\leftarrow$  0
rightPointer  $\leftarrow$  0
while leftPointer < half and rightPointer < size - half do
  if left[leftPointer] < right[rightPointer] then
    array[leftPointer + rightPointer]  $\leftarrow$  left[leftPointer]
    leftPointer ++
  else
    array[leftPointer + rightPointer]  $\leftarrow$  right[rightPointer]
    rightPointer ++
while leftPointer < half do
  array[leftPointer + rightPointer]  $\leftarrow$  left[leftPointer]
  leftPointer ++
while rightPointer < size - half do
  array[leftPointer + rightPointer]  $\leftarrow$  right[rightPointer]
  rightPointer ++

```

3 Otsu's thresholding method

3.1 Η σημασία του thresholding στην ψηφιακή επεξεργασία

Στην επεξεργασία εικόνας thresholding ονομάζεται η διαδικασία δυαδικοποίησης μιας ασπρόμαυρης εικόνας με κάποιο ανώτατο όριο. Η διαδικασία αυτή ορίζει την εικόνα κατάλληλη για περαιτέρω αναλύσεις, όπως για παράδειγμα τη τροφοδότησή της σε κάποιο σύγχρονο πρόγραμμα μετάφρασης εικόνας σε κείμενο. Είναι απαραίτητο η διαδικασία να είναι αποδοτική και γρήγορη καθώς η επεξεργασία εικόνας είναι εξαιρετικά ακριβή υπολογιστικά, καθώς και πολύ ευαίσθητη σε θόρυβο και δυσμορφίες στην εικόνα εισόδου.

3.2 Ο τρόπος λειτουργίας του αλγορίθμου του Otsu

Συγκεκριμένα ο αλγόριθμος του Otsu χρησιμοποιεί τον παρακάτω τύπο για να μπορέσει να επιλέξει το κατάλληλο threshold για τη δυαδικοποίηση της εικόνας.

$$\sigma_w^2 = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t) \quad (1)$$

Όπου ω_0 και ω_1 είναι τα εκάστοτε βάρη των 2 κλάσεων που καθορίζει το threshold και υπολογίζονται ως εξής:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i), \quad \omega_1(t) = \sum_{i=t}^{L-1} p(i) \quad (2)$$

Με t να είναι το threshold και το L να είναι ο αριθμός των δεξαμενών που χρησιμοποιούνται. Συνήθως είναι 255. Επιδιώκεται να επιτευχθεί η μέγιστη διακύμανση μεταξύ των 2 κλάσεων που δημιουργούνται από τη τιμή threshold.

$$\begin{aligned} \sigma_b^2 &= \sigma^2 - \sigma_w^2(t) = \omega_0(t)(\mu_0 - \mu_T)^2 + \omega_1(t)(\mu_1 - \mu_T)^2 \Rightarrow \\ &\Rightarrow \sigma_b^2 = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2 \end{aligned}$$

Οι τιμές μ_0 , μ_1 , μ_T είναι οι μέσες τιμές των κλάσεων και υπολογίζονται ως εξής:

$$\mu_0 = \frac{\sum_{i=0}^{t-1} ip(i)}{\omega_0(t)} \quad \mu_1 = \frac{\sum_{i=t}^{L-1} ip(i)}{\omega_1(t)} \quad \mu_T = \frac{\sum_{i=0}^{L-1} ip(i)}{L} \quad (3)$$

3.3 Ο αλγόριθμος του Otsu από τη μεριά του προγραμματιστή

Η πιο διαδεδομένη μέθοδος thresholding που υπάρχει είναι η μέθοδος του Otsu. Σε αυτή τη μέθοδο αρχικά γίνεται εύρεση του ιστογράμματος της εικόνας. Έπειτα, χωρίζει την εικόνα σε 2 κλάσεις: Την κλάση με όλες τις τιμές κάτω από ένα προκαθορισμένο όριο και τις υπόλοιπες. Τότε, γίνεται εύρεση του αριθμού ο οποίος θα ελαχιστοποιήσει τη διαφορά μεταξύ των 2 κλάσεων.

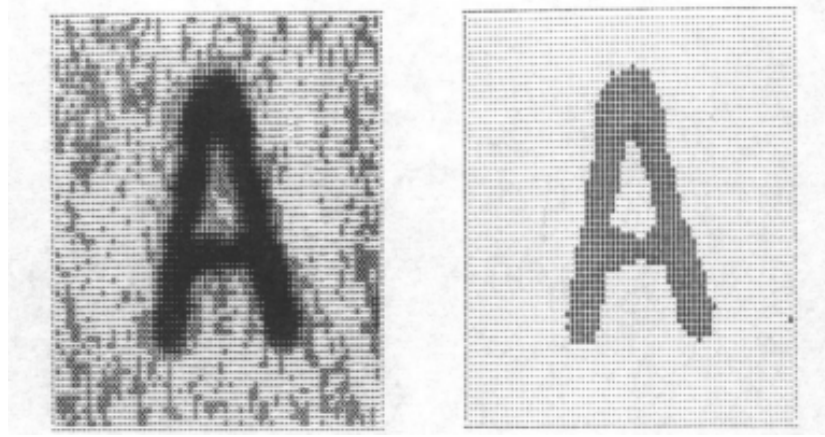


Figure 5: Παράδειγμα λειτουργίας της μεθόδου του Otsu[6].

Algorithm 5 Ο αλγόριθμος του Otsu

```

procedure OTSUTHRESHOLD(image)
  Compute histogram of image
  Compute total number of pixels
  for  $t = 0$  to 255 do
    Compute background weight  $\omega_B(t)$ 
    Compute foreground weight  $\omega_F(t)$ 
    Compute background mean  $\mu_B(t)$ 
    Compute foreground mean  $\mu_F(t)$ 
    Compute between-class variance  $\sigma_b^2(t)$ 
  return threshold  $t$  that maximizes  $\sigma_b^2(t)$ 

```

4 Απόδοση αλγορίθμων με τη βιβλιοθήκη OpenMP

4.1 Το περιβάλλον εκτέλεσης

Ο κώδικας εκτέλεσης γράφτηκε στη γλώσσα προγραμματισμού C και η παραλληλοποίηση των αποτελεσμάτων υλοποιήθηκε με τη βοήθεια της βιβλιοθήκης OpenMP [2]. Το υλικό εκτέλεσης αποτελείται από ένα σύστημα xeon με 64 πυρήνες και 128GB RAM. Κατά την εκτέλεση και εξόρυξη πειραματικών δεδομένων από τον κώδικα δεν εκτελούνταν καμία άλλη διεργασία για να υπάρξει ισοτιμία στο περιβάλλον εκτέλεσης. Επιπρόσθετα, η χρονομέτρηση έγινε εσωτερικά στον κώδικα μέσω της εντολής `omp_get_wtime()` και μετρήθηκε μόνο ο χρόνος εκτέλεσης του αλγορίθμου. Επιπρόσθετα, αξίζει να αναφερθεί πως έγινε σύγκριση των επιπέδων βελτιστοποίησης του προγράμματος από τον compiler. Τα levels O1, O2 και O3 παρουσίασαν πολύ κοντινούς χρόνους εκτέλεσης.

4.2 Τα αποτελέσματα για τον Bubble sort

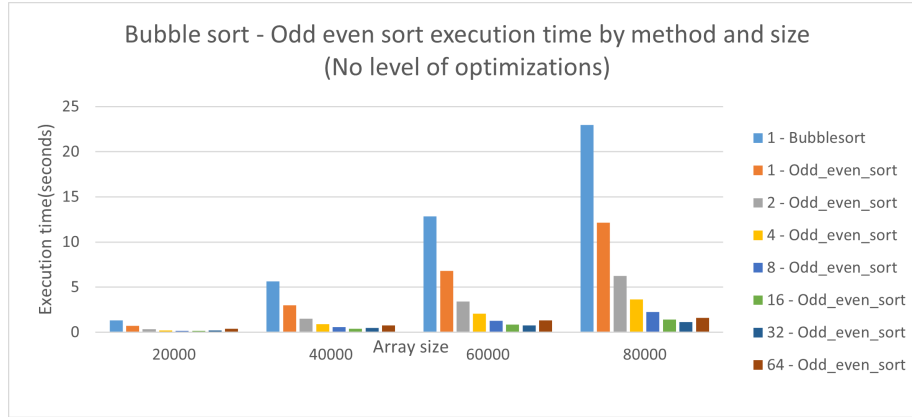


Figure 6: Οι χρόνοι εκτέλεσης Bubble sort και Odd even sort χωρίς optimization levels.

Στα Figure 6 και 7 απεικονίζονται οι χρόνοι εκτέλεσης των αλγορίθμων μαζί με τα threads τα οποία αξιοποίησαν ανάλογα με το μέγεθος του πίνακα. Η ειδοποιός διαφορά είναι η βελτιστοποίηση του προγράμματος κατά τη κατασκευή του προγράμματος. Όπως αναγράφεται και στις περιγραφές το Figure 6 δεν αξιοποιεί βελτιστοποιήσεις από τον compiler, ενώ το Figure 7 αξιοποιεί επιπέδου 2. Διακρίνονται πολλά ενδιαφέροντα αποτελέσματα: Για πολύ μικρά μεγέθη πινάκων η παραλληλοποίηση δεν προσδίδει καθόλου speedup, το οποίο είναι λογικό λόγω του overhead που προστίθεται με την εισαγωγή των επιπλέον συναρτήσεων για την αξιοποίηση των παραπάνω πόρων του συστήματος. Επιπρόσθετα, ο αλγόριθμος *odd even sort* παρουσιάζει χαμηλότερους εκτέλεσης ακόμη και όταν χρησιμοποιούμε μόνο ένα thread, το οποίο σημαίνει πως ο γενικός χρόνος εκτέλεσης είναι χαμηλότερος, άρα ο αλγόριθμος είναι αποδοτικότερος. Σε μεγαλύτερα μεγέθη πινάκων παρατηρούμε πως τα επιπλέον threads προσφέρουν αρκετά μεγάλο speedup και δεν επηρεάζονται από το επιπλέον overhead που εισάγεται κατά την παραλληλοποίηση. Τέλος, παρατηρείται πως ο χρόνος βελτιώνεται σημαντικά σε μικρότερους αριθμούς νημάτων με χρήση βελτιστοποίησης O2. Σε μεγαλύτερους αριθμούς threads η διαφορά είναι πολύ μικρή.

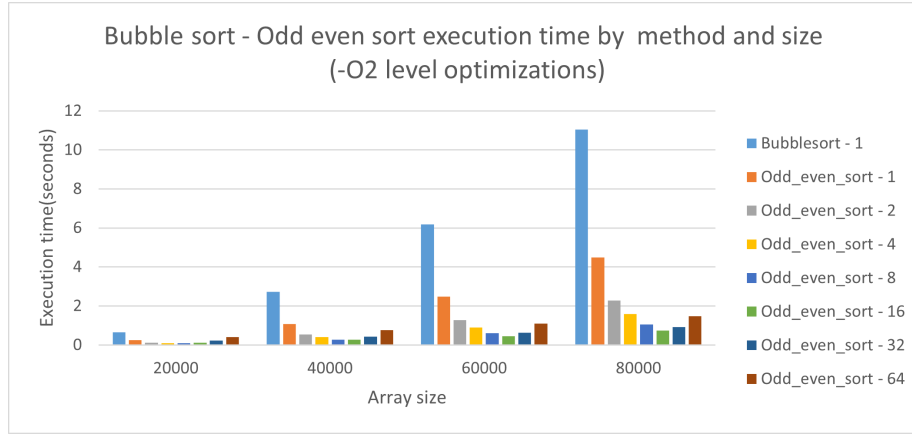


Figure 7: Οι χρόνοι εκτέλεσης Bubble sort και Odd even sort με O2 optimizations.

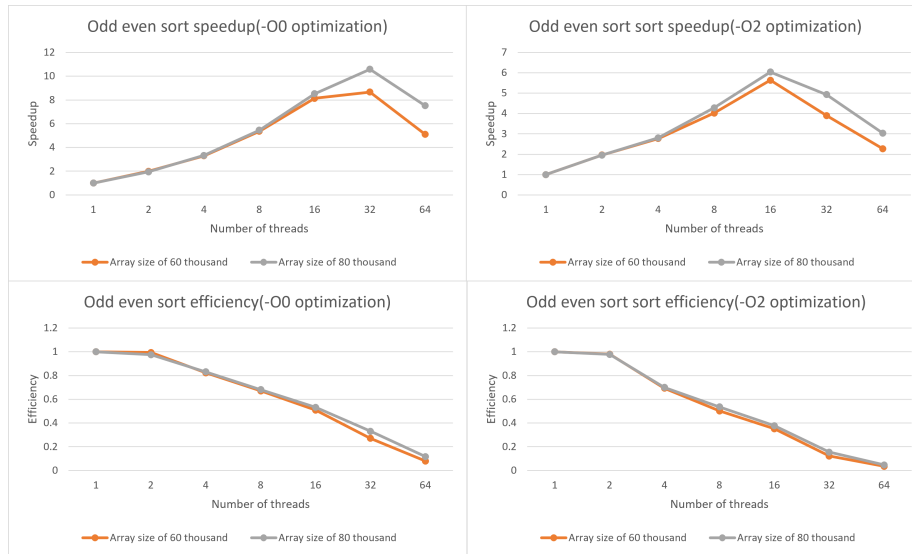


Figure 8: Οι τιμές επιτάχυνσης και απόδοσης του Odd even sort.

4.3 Τα αποτελέσματα για τον Bucket sort

Για την αξιολόγηση του αλγορίθμου πέραν του χρόνου εκτέλεσης, του μεγέθους του πίνακα και τον αριθμό των νημάτων θα γίνει αξιολόγηση της ταχύτητας και με βάση τον αριθμό των δεξαμενών. Παρατηρώντας το Figure 8 διακρίνεται πως για μεγαλύτερο αριθμό δεξαμενών έχουμε μια μικρή μείωση του χρόνου εκτέλεσης. Επίσης, είναι εμφανής η διαφορά μεταξύ τον αριθμό των νημάτων και υπάρχει βελτίωση έως και για μέγεθος πίνακα 15,000,000. Τέλος, στο figure 9 παρατηρείται μια μεγάλη σχετικά μείωση στο χρόνο εκτέλεσης του αλγορίθμου για χρήση βελτιστοποίησης επιπέδου 2, υποδεικνύοντας πως υπάρχει περιθώριο βελτίωσης.

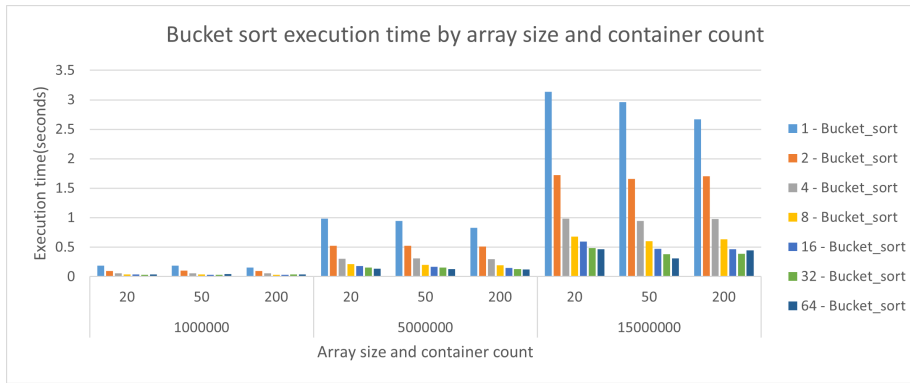


Figure 9: Οι χρόνοι εκτέλεσης για τους Bucket sort και Merge sort για διάφορα μεγέθη δεξαμενών (καμία βελτιστοποίηση).

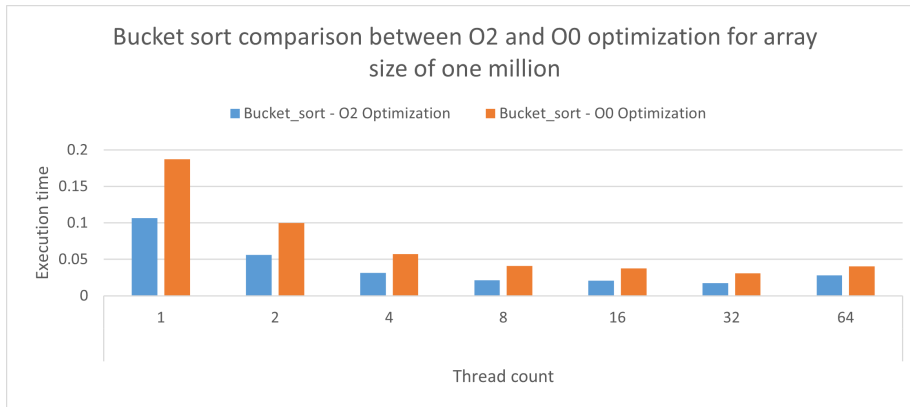


Figure 10: Η διαφορά του χρόνου εκτέλεσης του bucket sort με και χωρίς compiler optimization (20 δεξαμενές).

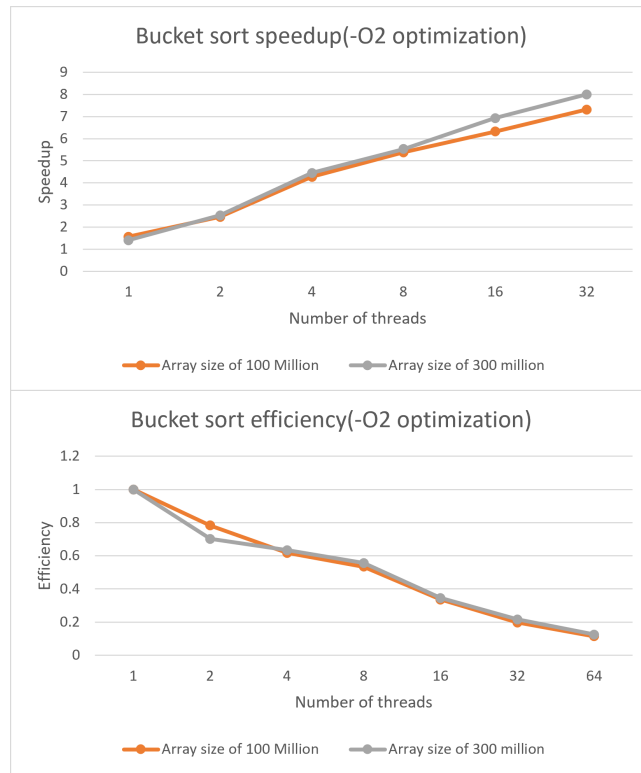


Figure 11: Οι τιμές επιτάχυνσης και απόδοσης του Bucket sort.

4.4 Τα αποτελέσματα για τη μέθοδο του Otsu

Για την μέθοδο του Otsu επιλέχθηκε για να είναι εμφανές η αποδοτικότητα της παραλληλοποίησης του αλγορίθμου η χρήση εικόνας με μέγεθος 30000x20000. Και πάλι, όπως φαίνεται και απο το Figure 10, χρειάζεται ακόμη μεγαλύτερη εικόνα για να αξιοποιηθούν περισσότερα νήματα. Επίσης, όπως και αναφέρθηκε παραπάνω, επιλέχθηκε να απεικονιστεί η διαφορά μεταξύ των διάφορων επιπέδων optimization, η όπως συμπεραίνεται από τα πειραματικά στοιχεία η πανομοιότυπη απόδοση των επιπέδων. Παρ'όλα αυτά, παρατηρείται ο υποδιπλασιασμός του χρόνου εκτέλεσης και για πολυπληθές νήματα και όχι μόνο για ένα νήμα, το οποίο είναι το πιο συνηθισμένο αποτέλεσμα. Τέλος, κρίνεται επιτακτικής ανάγκης να αναφερθεί πως με τη χρήση του επιπέδου βελτιστοποίησης 2 ο αλγόριθμος παρουσιάζει πανομοιότυπους χρόνους και αποτελέσματα με αυτούς της μεθόδου της βιβλιοθήκης OpenCV [1].

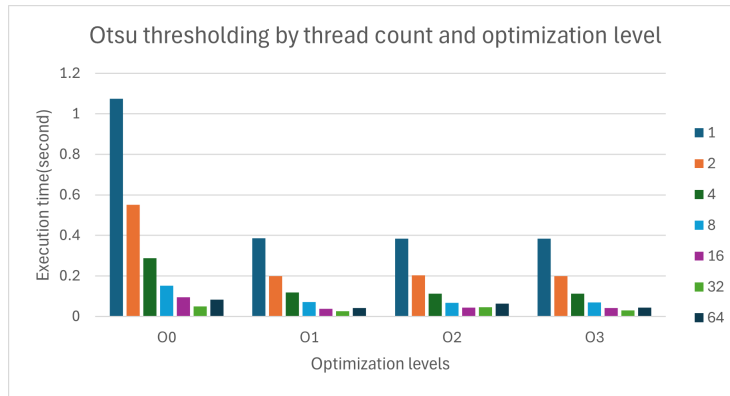


Figure 12: Ο χρόνος εκτέλεσης της μεθόδου του Otsu.

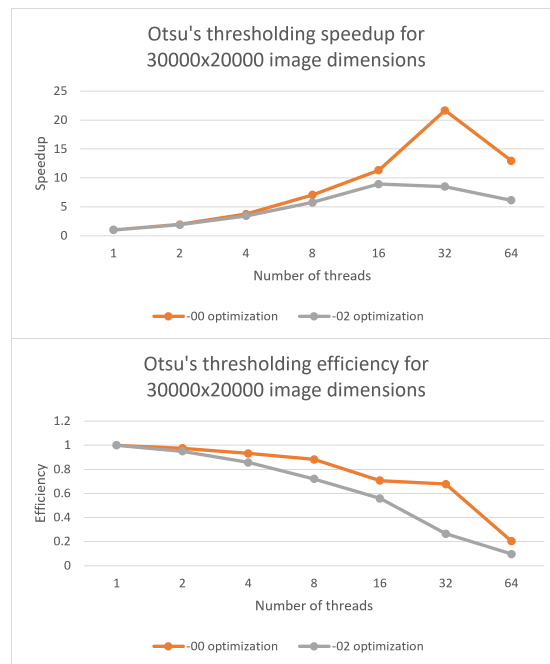


Figure 13: Οι τιμές επιτάχυνσης και απόδοσης της μεθόδου του Otsu.

5 Συμπεράσματα

Σύμφωνα με τα πειραματικά δεδομένα αλλά και το θεωρητικό υπόβαθρο είναι εμφανές πώς με την αξιοποίηση σύγχρονων μεθόδων, βιβλιοθηκών και εργαλείων μπορούμε με μεγάλη επιτυχία να επιταχύνουμε σημαντικά την αποδοχή των αλγορίθμων *Bubble sort*, *Bucket sort* και της μεθόδου του *Otsu* και να επιτύχουμε χρόνους εκτέλεσης αδύνατους για σειριακές εκτελέσεις κώδικα. Παράλληλα, αξίζει να σημειωθεί πώς ο προγραμματιστής παίζει ακόμη μεγάλο μέρος στη κατασκευή και την απόδοση του προγράμματος, καθώς είναι υπεύθυνος για την βέλτιστη αξιοποίηση και εφόσον χρειαστεί, όπως στην περίπτωση του αλγορίθμου *Bubble sort* να τροποποιήσει τη μαθηματική μέθοδο με σκοπό να ευνοεί την παράλληλη επεξεργασία. Τέλος, είναι σημαντικό να σημειωθεί πως η σειριακή εκτέλεση έχει ακόμη ρόλο στη κατασκευή προγραμμάτων σε τομείς που δεν παρουσιάζουν ιδιαίτερη ανάγκη για χρήση έντονων πόρων αλλά και σε τομείς οι οποίοι δεν μπορούν να παραλληλοποιηθούν αποτελεσματικά.

References

- [1] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).
- [2] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
- [3] HackerEarth. *Bubble Sort Algorithm Tutorial*. URL: <https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial>.
- [4] Yernar Kumashev. *GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm*. URL: https://essay.utwente.nl/80585/1/Final_Research_Paper.pdf.
- [5] Rashita Mehta. *Bucket Sort Algorithm*. URL: <https://www.scaler.in/bucket-sort-algorithm/>.
- [6] NOBUYUKI OTSU. *A Threshold Selection Method from Gray-Level Histograms*. URL: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4310076>.
- [7] Ricardo Rocha and Fernando Silva. *Parallel Sorting Algorithms*. URL: https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf.
- [8] Christopher Zelenka. *Parallel Merge Sort*. URL: <https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf>.