Παραλληλισμος αλγορίθμων Bubble sort και Bucket sort

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

March 25, 2025

Περιεχόμενα

1 Εισαγωγή

Τη σήμερων ημέρα παρατηρείται μια συνεχή 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, δημιουγείται 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 *B*ubble sort και *B*ucket sort, με τη χρήση του αλγορίθμου *M*erge

sort για την εσωτερική αναδιάταξη των δεξαμενών του αλγορίθμου. Για την παραλληλοποίηση των παρακάτω αλγορίθμων θα χρησιμοποιηθεί η βιβιοθήκη [1] OpenMP στη γλώσσα προγραμματισμού C.

2 Ανάλυση λειτουργίας των αλγορίθμων

2.1 Ο αλγόριθμος Bubble sort

Ο αλγόριθμος *B*ubble sort είναι από τους πιό απλούς και διαδεδομένους αλγορίθμους που υπάρχουν. Ο αλγόριθμος ξεκινάει απο την αρχή του πίνακα και συγκρίνει διαδοχικά όλους τους αριθμούς για να δεί 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.

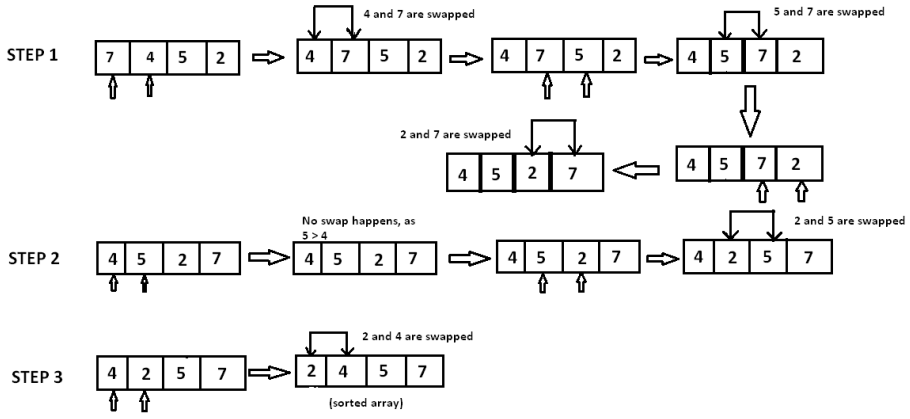

Figure 1: Ο τρόπος λειτουργίας του Bubble sort [2].

Algorithm 1 Ο αλγόριθμος Bubble sort

```
i ← 0
N ← array size
array[N]
for i ← 0 to N − 1 do
    for j ← 0 to N − i − 1 do
        if array[j] > array[j + 1] then
            swap(array[j], array[j + 1])
```

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 αποφεχθούν οι καθυστερήσεις λόγω συγχρονισμού, θα χρησιμοποιηθεί μια παραλλαγή του αλγορίθμου *B*ubble sort: Τον αλγόριθμο *O*dd even sort.

2.2 Ο αλγόριθμος Odd even sort

Ο αλγόριθμος *O*dd even sort είναι μια ευνοική παραλλαγή του *B*ubble 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.

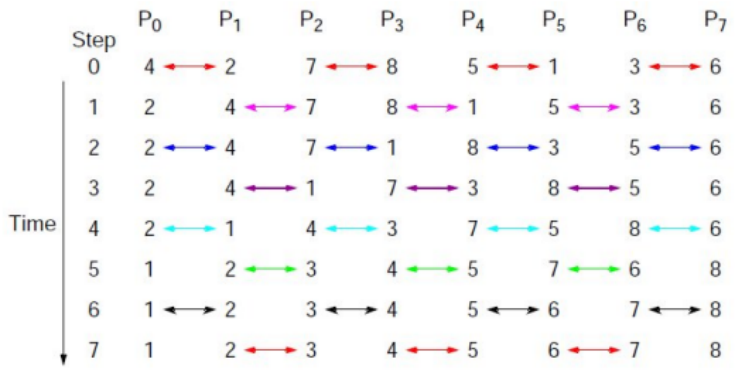

Figure 2: Ο τρόπος λειτουργίας του Odd even sort [5].

Algorithm 2 Ο αλγόριθμος Odd even sort

```
i ← 0
j ← 0
N ← array size
array[N]
while array is not sorted do
    for i ← 1 to N − 1 with jumpsize: 2 do
        if array[i] > array[i + 1] then
            swap(array[i], array[i + 1])
    for j ← 1 to N − 1 with jumpsize: 2 do
        if array[j] > array[j + 1] then
            swap(array[j], array[j + 1])
```

Είναι προφανές πως η μετατροπή η οποία υλοποιήθηκε στον αλγόριθμο επιτρέπει την αποδοτική αξιοποίηση περισσότερων πυρήνων του επεξεργαστή, ακόμη και GPU [3]

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 επιλεχθούν 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 δημιμουργηθεί τελικά ο πίνακας εξόδου. Έτσι αποφεύγεται η εκτεταμένη χρήση εντολών συγχρονισμού μεταξύ των νημάτων και επιτρέπει την πλήρη αξιοποίηση του διαθέσιμου υλικού.

Figure 3: Ο τρόπος λειτουργίας του Bucket sort [4].

Algorithm 3 Ο αλγόριθμος Bucket sort

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

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 [6], καθώς επίσης και για φυσιολογικούς αριθμούς δεξαμενών το overhead θα υποβάθμιζε σημαντικά την οποιαδήποτε επιτάχυνση λόγω παραλληλοποίησης. Ο *merge sort* εχει 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.

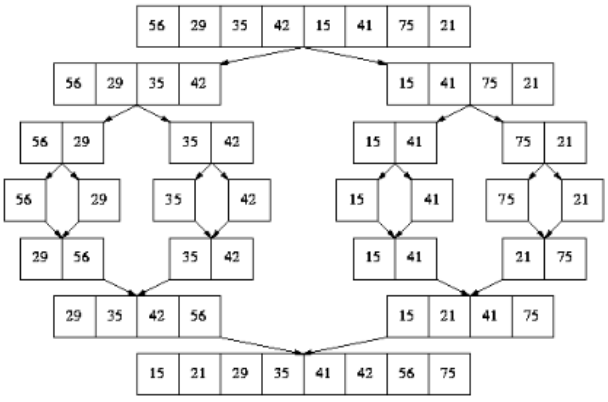

Figure 4: Ο τρόπος λειτουργίας του merge sort [4].

Algorithm 4 Ο αλγόριθμος Merge sort

```
MergeSortArray(array, size)
if size ≤ 1 then
    return
half ← size / 2
left ← new array of size half
right ← new array of size (size - half)
for i = 0 to half − 1 do
    left[i] ← arr[i]
for i = 0 to size − half − 1 do
    right[i] ← arr[half + i]
MergeSortArray(left, half)
MergeSortArray(right, size − half)
leftPointer ← 0
rightPointer ← 0
while leftPointer < half and rightPointer < size − half do
    if left[leftPointer] < right[rightPointer] then
        array[leftPointer + rightPointer] ← leftPointer
        leftPointer + +
    else
        array[leftPointer + rightPointer] ← rightPointer
        rightPointer + +
while leftPointer < half do
    array[leftPointer + rightPointer] ← leftPointer
    leftPointer + +
while rightPointer < size − half do
    array[leftPointer + rightPointer] ← rightPointer
    rightPointer + +
```

3 Απόδοση αλγορίθμων στο πειραματικό περιβάλλον

3.1 Το περιβάλλον εκτέλεσης

Ο κώδικας εκτέλεσης γράφτηκε στη γλώσσα προγραμματισμού C και η παραλληλοποίηση των αποτελεσμάτων υλοποιήθηκε με τη βοήθεια της βιβλιοθήκης OpenMP [1]. Το υλικό εκτέλεσης αποτελείται από τον επεξεργαστή AMD Ryzen 5800H στα 3.4GHZ και 16GB RAM σε συχνότητα 3200MHZ. 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. Τέλος, πρέπει να αναφερθει πως ο επεξεργαστής έχει 8 φυσικούς πυρήνες και 16 νήματα, το οποίο σημαίνει πως δεν

θα παρατηρηθεί η ίδια επιτάχυνση χρησιμοποιόντας 16 threads αντί 8, σε σύγκριση με μεταβολές όπως για παράδειγμα 8 αντί για 4.

3.2 Τα αποτελέσματα για τον Bubble sort

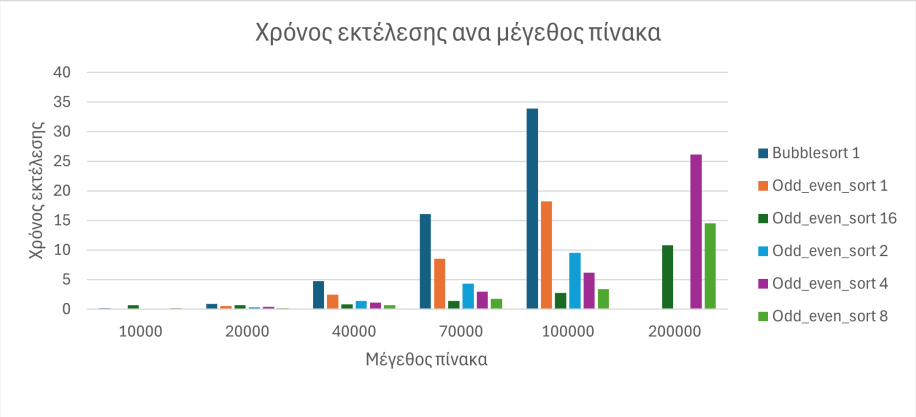

Figure 5: Οι χρόνοι εκτέλεσης Bubble sort και Odd even sort.

Στο Figure 5 απεικονίζονται οι χρόνοι εκτέλεσης των αλγορίθμων μαζί με τα threads τα οποία αξιοποίησαν ανάλογα με το μέγεθος του πίνακα. 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 επιπλέων 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 αποδοτικοτερος. 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.

3.3 Τα αποτελέσματα για τον Bucket sort

Εδώ επιλέχθηκε πέραν της επιθεώρησης του speedup που προσφέρουν το επιπλέον υλικό να παρατηρηθεί η σημασία των δεξαμενών στη ταχύτητα του Bucket sort. Ξεκινόντας από τον αριθμό των δεξαμενών, παρατηρείται πως γενικότερα οι επιπρόσθετες δεξαμενές σε μεγάλα μεγέθη πινάκων προσφέρουν ένα μικρό ποσοστό speedup, εφόσον είναι διαθέσιμα επαρκή threads για να τις αξιοποιήσουν κατάλληλα. Προχωρώντας στη σύγκριση των μεθόδων, αρχικά παρατηρείται πως ο Bucket sort είναι πιο γρήγορος ακόμη και με ένα νήμα σε σύγκριση με τον merge sort, το οποίο είναι το προβλεπόμενο αποτέλεσμα καθώς έχουν πολυπλοκότητα O(N) και O(Nlog(N)) αντίστοιχα. Έπειτα, παρατηρούμε πτώση περίπου 40% ανά

τον διπλασιασμό των διαθέσιμων νημάτων. Έχοντας στο νου πως η ιδανική πτώση θα ήταν 50%, αυτό σημαίνει πως η υλοποίηση του αλγορίθμου, καθώς και ο ίδιος ο αλγόριθμος, προσφέρουν ένα πολύ υψηλό ποσοστό απόδοσης παραλληλοποίησης.

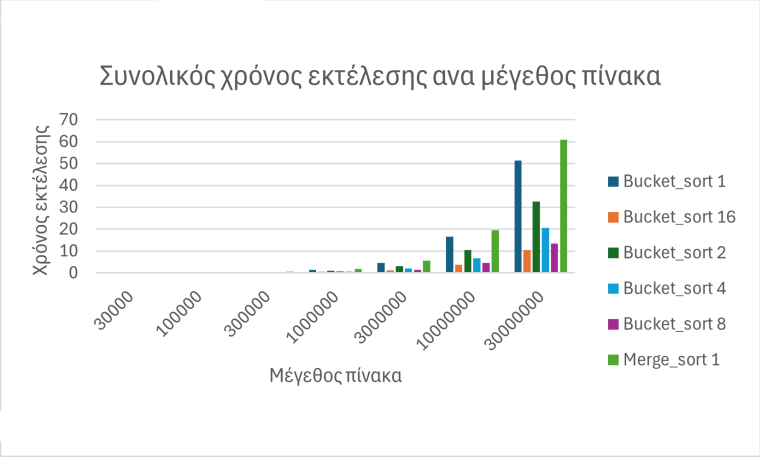

Figure 6: Οι χρόνοι εκτέλεσης για τους Bucket sort και Merge sort.

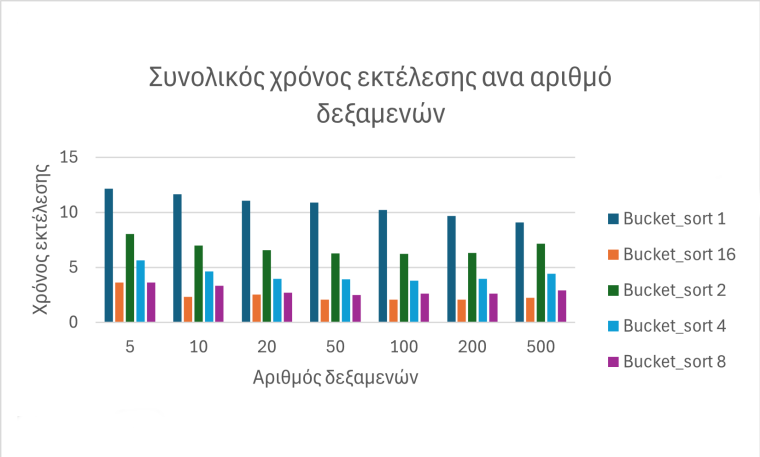

Figure 7: Οι χρόνοι εκτέλεσης για τους Bucket sort για δίαφορα μεγέθη δεξαμενών.

4 Συμπεράσματα

Σύμφωνα με τα πειραματικά δεδομένα αλλά και το θεωρητικό υπόβαθρο είναι εμφανές πώς με την αξιοποίηση σύγχρονων μεθόδων, βιβλιοθηκών και εργαλειών μπορούμε με μεγάλη επιτυχία να επιταγχύνουμε σημαντικά την αποδοση των αλγορίθμων *Bubble sort* και *Bucket sort* και να επιτύγχουμε 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] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[2] HackerEarth. *Bubble Sort Algorithm Tutorial*. URL: `https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial`.

[3] Yernar Kumashev. *GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm*. URL: `https://essay.utwente.nl/80585/1/Final_Research_Paper.pdf`.

[4] Rashita Mehta. *Bucket Sort Algorithm*. URL: `https://www.scaler.in/bucket-sort-algorithm/`.

[5] Ricardo Rocha and Fernando Silva. *Parallel Sorting Algorithms*. URL: `https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf`.

[6] Christopher Zelenka. *Parallel Merge Sort*. URL: `https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf`.