

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 1^{ης} ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 19.11.2025

▪ Ενότητα 2.1 – Παραλληλοποίηση και Βελτιστοποίηση του Αλγορίθμου K-means

Στόχος της άσκησης είναι η ανάπτυξη δύο παράλληλων εκδόσεων του αλγορίθμου K-means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με τη χρήση του προγραμματιστικού εργαλείου OpenMP.

Αρχικά, μελετήσαμε το υλικό και τα αρχεία του εργαστηρίου (στον φάκελο του kmeans που μας δίνεται), όπως και το αντίστοιχο υλικό των διαλέξεων του μαθήματος. Έτσι, τροποποιήσαμε τα περιφερειακά αρχεία (Makefile, make_on_queue.sh, run_on_queue.sh, file_io.sh) που εξυπηρετούν την ορθή μεταγλώττιση και λειτουργία των κυρίων αρχείων με τις παράλληλες εκδόσεις του αλγορίθμου μας (omp_naive_kmeans.c, omp_reduction_kmeans.c), ως εξής:

1. **Makefile:** Μετονομάσαμε τα αρχεία, ώστε να ανταποκρίνονται στο εργαστηριακό υλικό που μας δόθηκε, κάναμε uncommment τα σχόλια που μεταγλωττίζουν τα αρχεία με τον παράλληλο κώδικα και συμπεριλάβαμε το -forenmp, ώστε να δηλώσουμε ότι τα αρχεία μας χρησιμοποιούν τη βιβλιοθήκη OpenMP.
2. **make_on_queue.sh:** Αλλάξαμε τη διεύθυνση του φακέλου src σε «/home/parallel/parlab05/a2/kmeans», ώστε να εξυπηρετεί τις ανάγκες της άσκησης, όπως ζητήθηκε. Καθώς η αλλαγή αυτή είναι μικρή, το συγκεκριμένο αρχείο δεν θα συμπεριληφθεί στην αναφορά.
3. **run_on_queue.sh:** Το αρχείο προσαρμόστηκε ώστε να επιτρέπει την εκτέλεση των πειραμάτων με διαφορετικές πολιτικές δέσμευσης νημάτων (affinity), μέσω παραμέτρων στην εντολή qsub. Υποστηρίζονται οι δύο κύριες φάσεις που ζητούνται στην άσκηση:
 - (α) εκτέλεση χωρίς καμία πολιτική δέσμευσης (noaff) και
 - (β) εκτέλεση με προκαθορισμένη πολιτική affinity (aff), όπου τα N νήματα του OpenMP δένονται ρητά στα N πρώτα λογικά CPU slots του κόμβου (0, 1, ..., N-1).

Η επιλογή αυτή πάρθηκε κατόπιν συζήτησης με τον διδάσκοντα και καθώς συνιστά την standard επιλογή της βιβλιοθήκης (κατόπιν αναζήτησης στο διαδίκτυο). Μια άλλη επιλογή με την οποία πειραματιστήκαμε ήταν ο διαμοιρασμός των threads στα 4 nodes του μηχανήματος ισάριθμα, ώστε να

αξιοποιήσουμε στο μέγιστο το διαθέσιμο memory bandwidth (η εφαρμογή μας είναι memory bound), παρά να είναι τοποθετημένα κοντά, στο ίδιο node και αυξάνοντας τη συμφόρηση στον δίαυλο μνήμης. Ωστόσο, για λόγους απλότητας και έκτασης της άσκησης, αφήσαμε την πολιτική στο default.

Σημειώνουμε ότι η επιλογή της πολιτικής γίνεται αυτόματα κατά την υποβολή του πειράματος, ενώ τα αποτελέσματα αποθηκεύονται σε ξεχωριστούς φακέλους, οργανωμένους ανά εκτελέσιμο και ανά επιλεγμένη πολιτική affinity, ώστε να διευκολύνεται η σύγκριση των μετρήσεων.

4. **file io.c**: Συμπληρώσαμε το header που ζητούνταν, ως: `#include <omp.h>`. Καθώς η αλλαγή αυτή είναι μικρή, το συγκεκριμένο αρχείο δεν θα συμπεριληφθεί στην αναφορά.

Τα αρχεία αυτά βρίσκονται στον orion και στον scirouter της ομάδας μας και παρουσιάζονται (αυτά που άλλαξαν σημαντικά, για λόγους πληρότητας) ακολούθως:

2.1.1 – Shared Clusters

Στην πρώτη παράλληλη υλοποίηση του αλγορίθμου K-means υιοθετήσαμε το μοντέλο shared clusters, χωρίς καμία βελτιστοποίηση στη συλλογή των μερικών αποτελεσμάτων. Πρόκειται για μια «αφελή» (naive) προσέγγιση, όπου όλα τα νήματα ενημερώνουν απευθείας τους κοινόχρηστους πίνακες `newClusters[]` και `newClusterSize[]`. Η πρόσβαση σε αυτά τα κοινόχρηστα δεδομένα απαιτεί συγχρονισμό, προκειμένου να αποφευχθούν πιθανά race conditions, ο οποίος στη συγκεκριμένη εκδοχή υλοποιείται αποκλειστικά με `#pragma omp atomic` για κάθε ενημέρωση-πρόσβαση. Σημειώνουμε ότι θα μπορούσε να έχει χρησιμοποιηθεί και `#pragma omp critical`, ωστόσο η εντολή αυτή είναι πιο αργή και υποβέλτιστη σε απλές προσβάσεις-πράξεις, όπως αυτή.

Η προσέγγιση αυτή επιτρέπει την εύκολη και άμεση παραλληλοποίηση του βρόχου, αλλά εισάγει σημαντικό κόστος εξαιτίας των συχνών πράξεων που γίνονται με `atomic` και της υψηλής πιθανότητας contention, ειδικά για μικρό αριθμό συντεταγμένων (`numCoords`) ή για μεγάλο αριθμό νημάτων. Δηλαδή, η ευκολία υλοποίησης έρχεται με υψηλό κόστος συγχρονισμού. Έτσι, η 2.1.1 συμβάλλει κυρίως ως σημείο αναφοράς για τη σύγκριση με πιο αποδοτικές τεχνικές συγχώνευσης που αναπτύσσονται στην επόμενη ενότητα (2.1.2 – copied clusters and reduce).

Όσον αφορά τις λεπτομέρειες της υλοποίησής μας, επισημαίνουμε τα εξής:

- Ο παράλληλος βρόχος (`#pragma omp parallel for`) αναθέτει σε κάθε νήμα ένα υποσύνολο αντικειμένων για επεξεργασία.
- Η αύξηση της μεταβλητής `delta` προστατεύεται με `atomic operation`, καθώς εδώ δεν χρησιμοποιείται `reduction`. Ωστόσο, ακόμα και χωρίς `atomic` (δεν ζητούταν με σχόλιο) η ορθότητα του προγράμματος δεν θα άλλαζε, αφού ναι μεν η `delta` θα είχε λάθος τιμή, αλλά σίγουρα `delta >= 1 > threshold`, άρα ο έλεγχος θα ήταν πάντα σωστός και αληθής και θα μπορούσαμε να αποφύγουμε αυτό το `atomic` (η σχετική συζήτηση έγινε και με τον διδάσκοντα και επιλέξαμε να το βάλουμε).
- Η ενημέρωση των δομών `newClusterSize[]` και `newClusters[]` γίνεται επίσης με `atomic` σε κάθε πρόσβαση, γεγονός που καθιστά την υλοποίηση μεν εύκολη, αλλά δε καθόλου αποδοτική, αφού έχουμε μεγάλο κόστος συγχρονισμού, σε πολλαπλά σημεία.

Ο ολοκληρωμένος κώδικας της υλοποίησης (`omp_naive_kmeans.c`) βρίσκεται στον `scirouter` και στον `ορίον` της ομάδας μας, αλλά παρατίθεται και στην παρούσα αναφορά για λόγους πληρότητας.

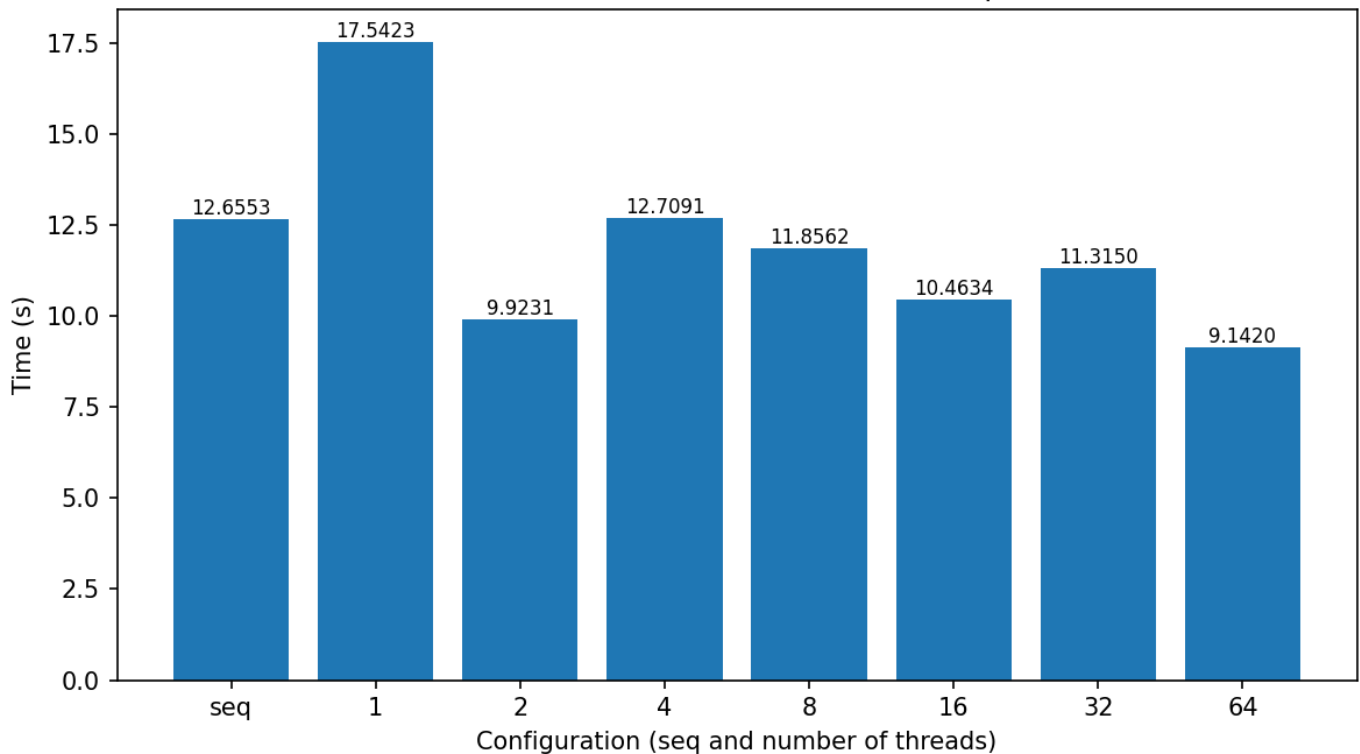
1. No Affinity

Το παραπάνω παράλληλο πρόγραμμα (omp_naive_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και χωρίς affinity. Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

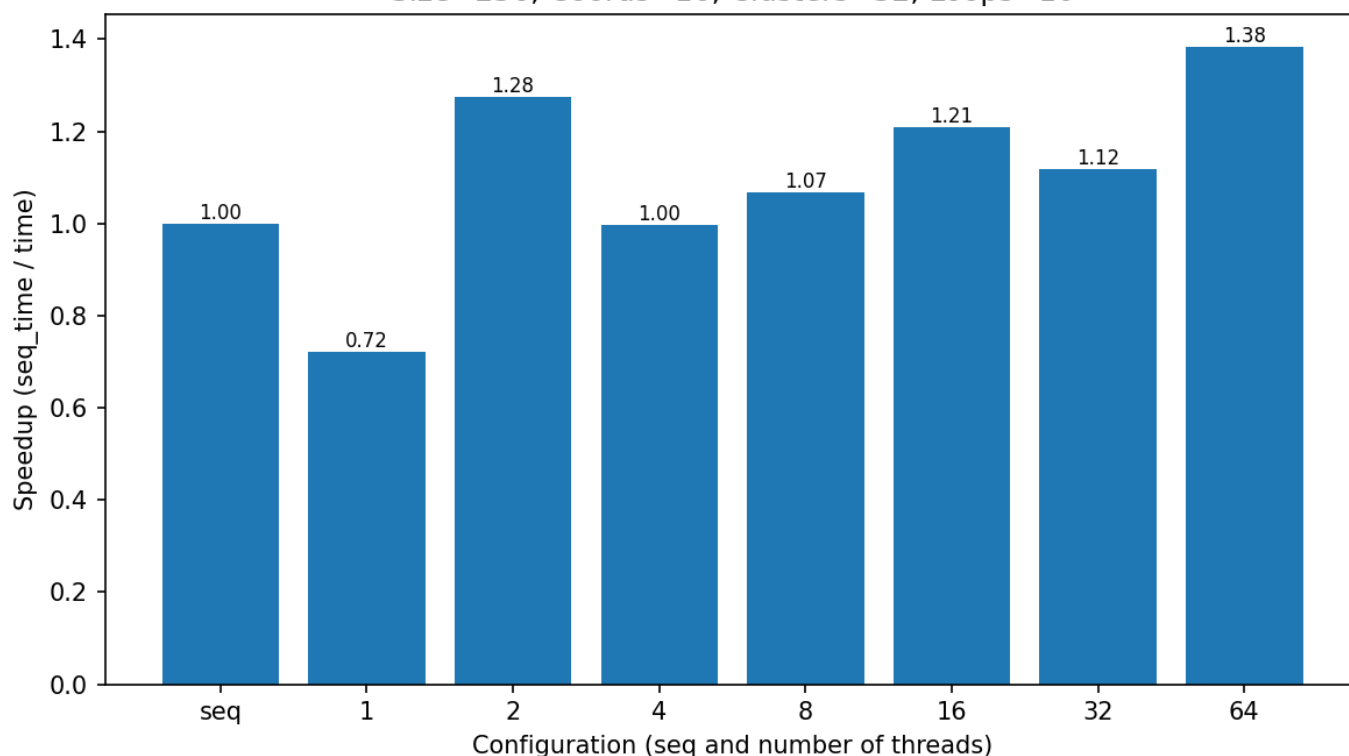
THREADS	TIME
seq	12.65
1	17.54
2	9.92
4	12.70
8	11.85
16	10.46
32	11.31
64	9.14

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:

K-means (shared clusters, naive) — Time vs Threads (no affinity)
Size=256, Coords=16, Clusters=32, Loops=10



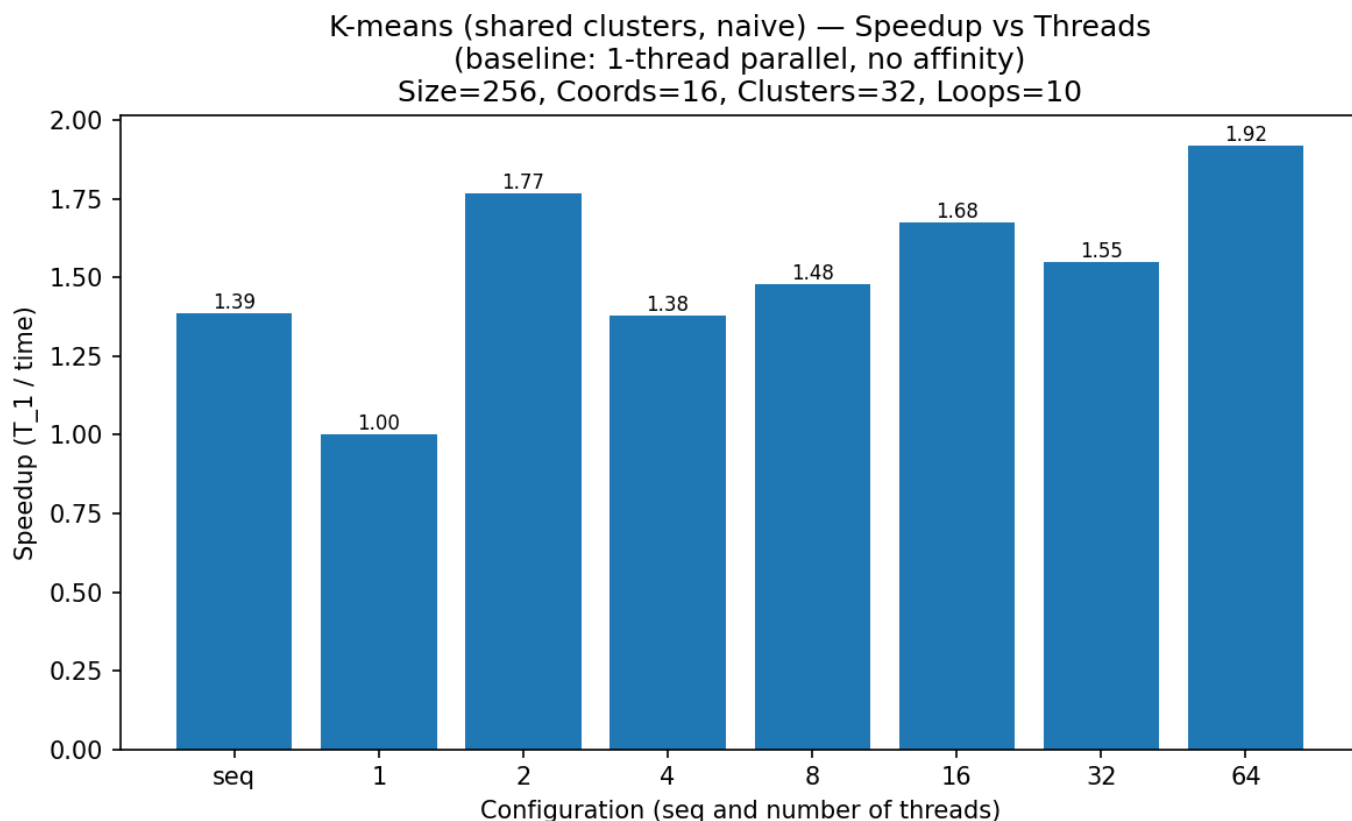
K-means (shared clusters, naive) — Speedup vs Threads (no affinity)
Size=256, Coords=16, Clusters=32, Loops=10



Σύμφωνα με τα παραπάνω, συμπεραίνουμε ότι υλοποίηση χωρίς affinity δεν κλιμακώνει ικανοποιητικά. Ο σειριακός χρόνος είναι περίπου 12.7s, ενώ η παράλληλη έκδοση με 1 νήμα είναι σαφώς χειρότερη ($\approx 17.5s$, speedup ≈ 0.72), κάτι που αποδίδεται στο κόστος δημιουργίας του παράλληλου προγράμματος-κώδικα (δημιουργία/συγχρονισμός νημάτων, επιπλέον κώδικας OpenMP), όπως αναφέρεται και στις διαφάνειες. Για περισσότερα νήματα, τα διαγράμματα χρόνου και speedup δείχνουν μικρές μόνο βελτιώσεις: γύρω στο $1.28\times$ στα 2 νήματα (αν και παρατηρείται κλιμάκωση σχεδόν $2\times$ από το 1 νήμα του παράλληλου προγράμματος), τιμές κοντά στο $1.0\text{--}1.2\times$ για 4–32 νήματα και μέγιστο περίπου $1.38\times$ στα 64 νήματα, πολύ μακριά από την ιδανική γραμμική κλιμάκωση.

Η συμπεριφορά αυτή ταιριάζει ακριβώς με τη θεωρία για synchronization bottlenecks: στη naive shared έκδοση όλες οι ενημερώσεις των πινάκων `newClusterSize[]` και `newClusters[]` γίνονται με `#pragma omp atomic`, άρα πολλές προσπελάσεις σε λίγες κοινόχρηστες μεταβλητές σειριοποιούνται και δημιουργούν έντονο contention, όπως στα παραδείγματα των διαφανειών για fine-grained synchronization. Έτσι, μεγάλο μέρος του χρόνου δαπανάται σε συγχρονισμό αντί για πραγματικό υπολογισμό, ενώ και το σειριακό τμήμα του αλγορίθμου (σύμφωνα με τον νόμο του Amdahl) βάζει χαμηλό άνω φράγμα στο speedup. Τα παραπάνω αποτελέσματα και τα διαγράμματα, λοιπόν, επιβεβαιώνουν ότι η λύση αυτή είναι μεν ορθή και εύκολη στην υλοποίηση, αλλά καθόλου αποδοτική.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



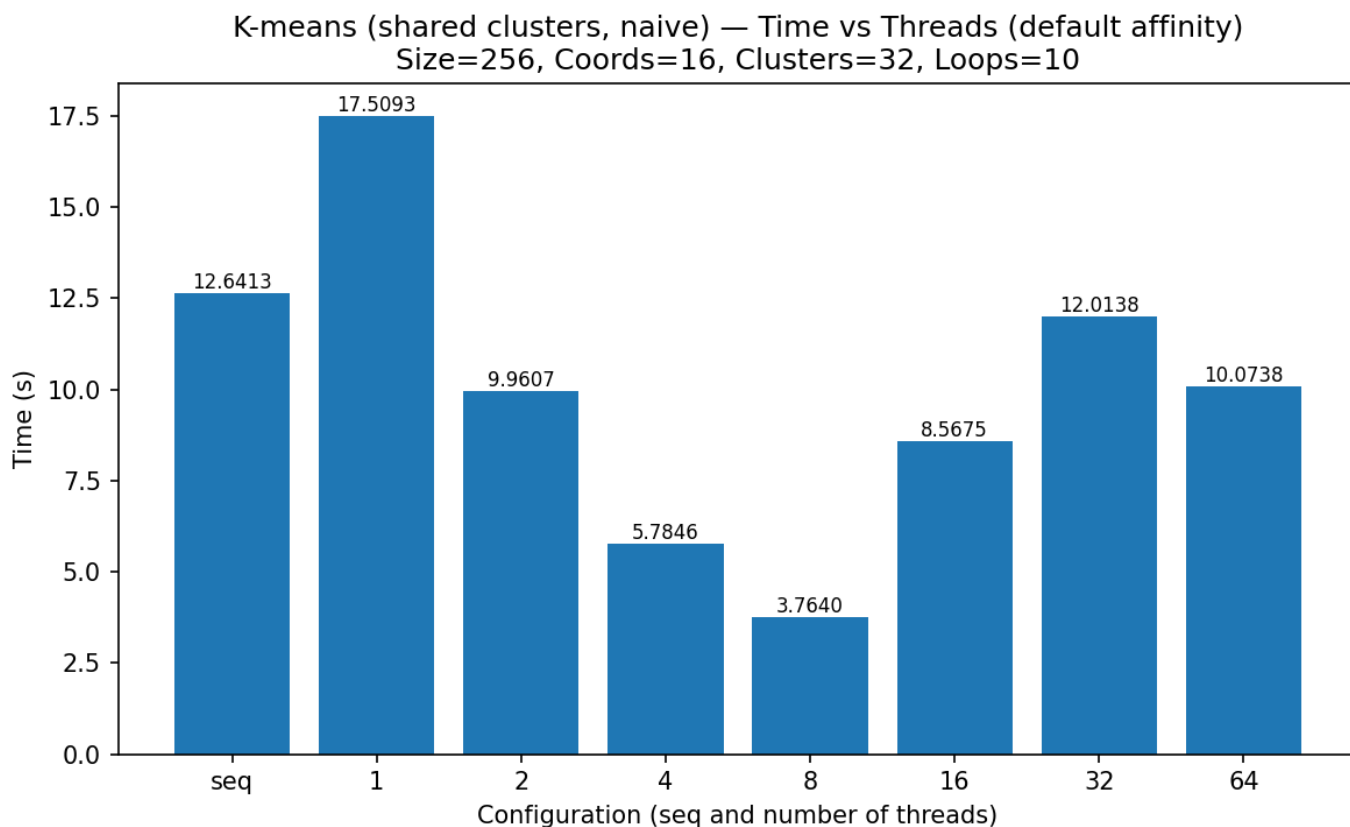
Από το παραπάνω διάγραμμα παρατηρούμε ότι, αν θεωρήσουμε ως βάση το παράλληλο πρόγραμμα με 1 νήμα, όλες οι εκτελέσεις με περισσότερα νήματα εμφανίζουν πλέον speedup μεγαλύτερο της μονάδας. Ενδεικτικά, για 2 νήματα το κέρδος είναι περίπου $1.8\times$ σε σχέση με το 1-thread (σχεδόν γραμμικό και το μόνο ικανοποιητικό), ενώ για 64 νήματα φτάνει σχεδόν το $2\times$ (πολύ κακή κλιμάκωση γενικότερα). Αυτό επιβεβαιώνει ότι ένα σημαντικό τμήμα του κόστους στην περίπτωση του 1 νήματος οφείλεται αποκλειστικά στο parallel overhead του OpenMP (δημιουργία ομάδας νημάτων, συγχρονισμοί κ.λπ.), το οποίο «απλώνεται» σε περισσότερα νήματα και αντισταθμίζεται μερικώς όταν αυξάνουμε τον βαθμό παραλληλίας και ότι η εφαρμογή είναι memory bound. Παρ' όλα αυτά, η κλιμάκωση παραμένει εξαιρετικά κακή και σε απόλυτους όρους ως προς το σειριακό πρόγραμμα και τα κέρδη παραμένουν μικρά, γεγονός που δείχνει ότι το synchronization bottleneck της naive shared υλοποίησης δεν επιτρέπει ουσιαστική κλιμάκωση.

2. Default Affinity

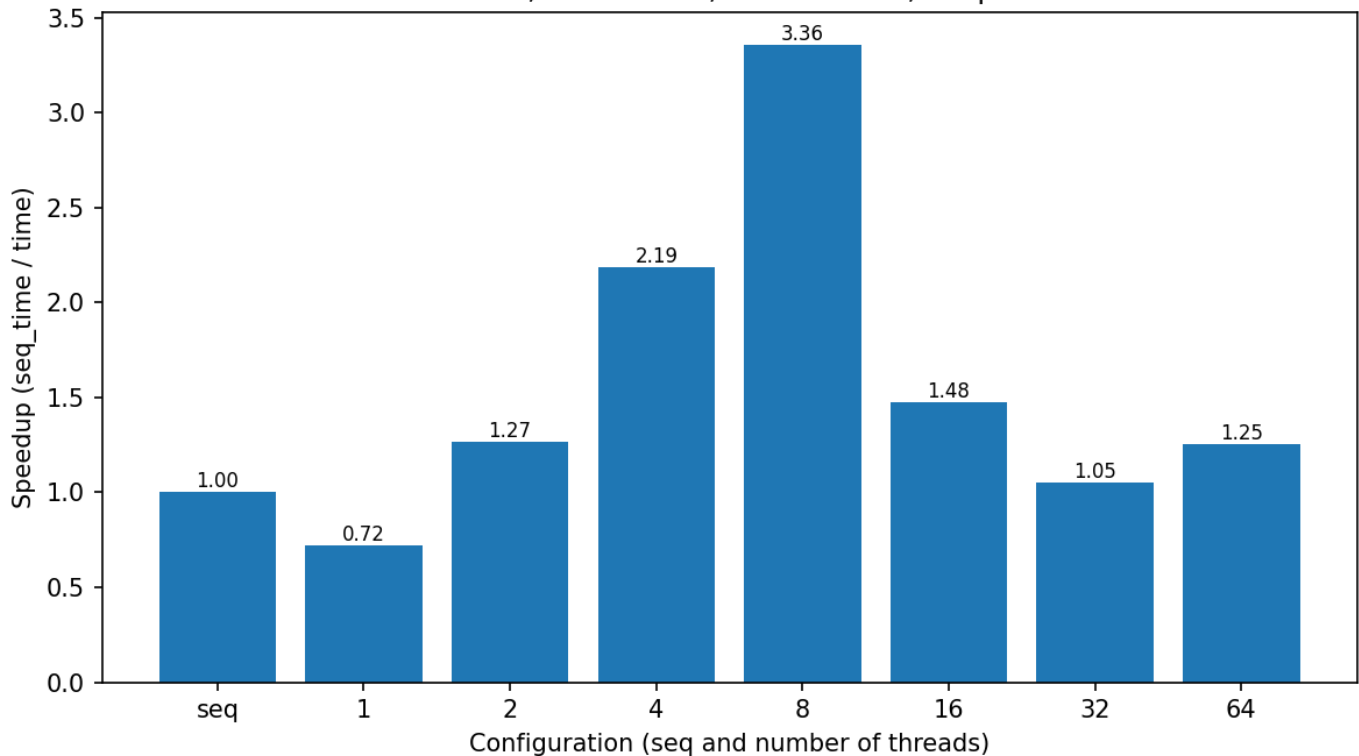
Το παραπάνω παράλληλο πρόγραμμα (omp_naive_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64, αλλά αυτή τη φορά με affinity. Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	12.64
1	17.50
2	9.96
4	5.78
8	3.76
16	8.56
32	12.01
64	10.07

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



K-means (shared clusters, naive) — Speedup vs Threads (default affinity)
Size=256, Coords=16, Clusters=32, Loops=10

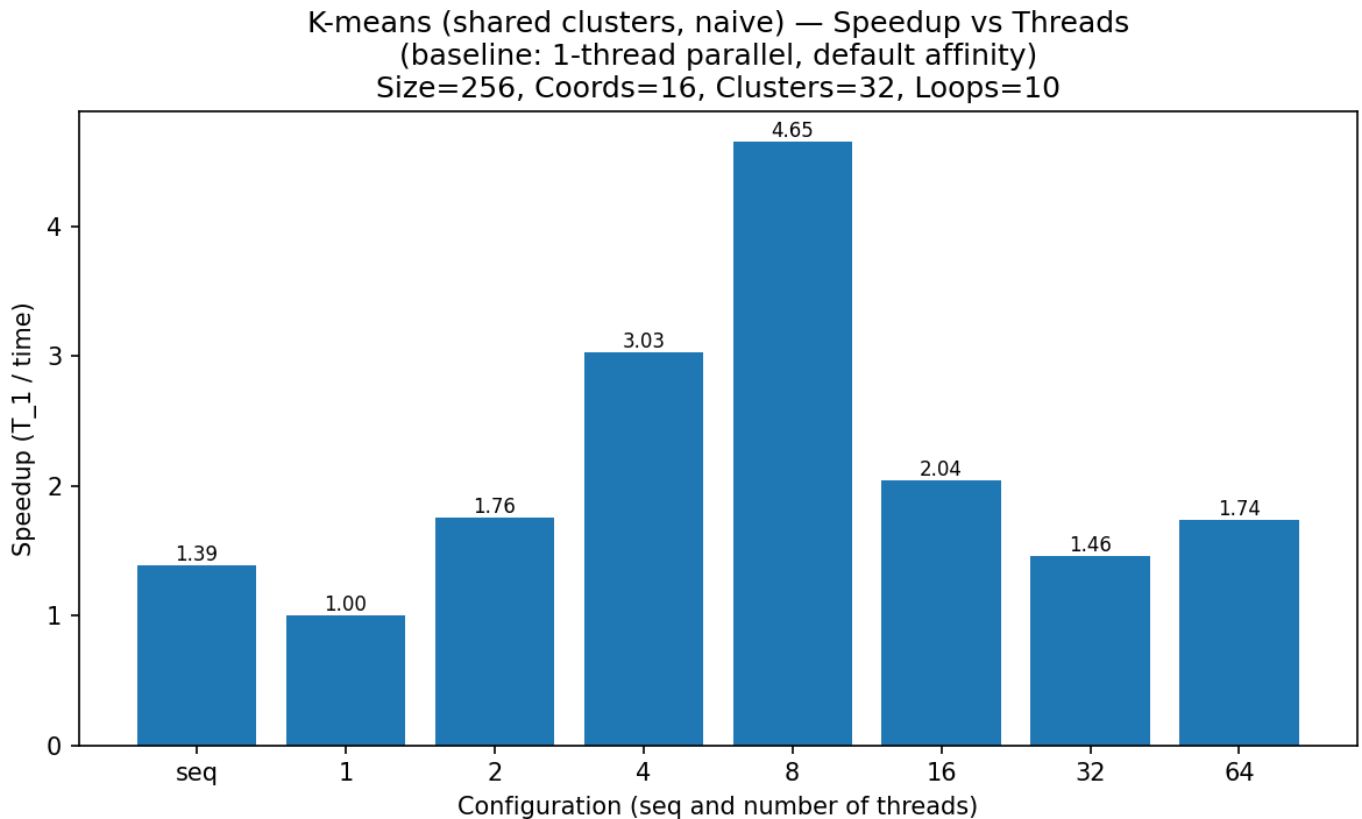


Παρατηρούμε ότι με ενεργοποιημένο το default affinity η συμπεριφορά της naive shared υλοποίησης βελτιώνεται σημαντικά σε σχέση με την περίπτωση χωρίς affinity. Ο χρόνος εκτέλεσης μειώνεται πολύ καλά οριακά γραμμικά ως προς το παράλληλο πρόγραμμα με 1 νήμα) μέχρι τα 8 νήματα (από ~17.5s στο 1 νήμα σε ~3.8s στα 8 νήματα), με αντίστοιχο speedup ~3.4× σε σχέση με το σειριακό πρόγραμμα, γεγονός που δείχνει ότι η δέσμευση των νημάτων σε σταθερούς πυρήνες αξιοποιεί καλύτερα την τοπικότητα cache και μνήμης μέσα στο ίδιο NUMA node, όπως επισημαίνεται και στις διαφάνειες για affinity και locality. Ωστόσο, για 16, 32 και 64 νήματα η επίδοση υποβαθμίζεται (ο χρόνος αυξάνεται ξανά και το speedup πέφτει κοντά στη μονάδα), κάτι που είναι αναμενόμενο για έναν κατά βάση memory-bound αλγόριθμο με έντονο synchronization μέσω atomic πράξεων.

Πιο συγκεκριμένα, όταν ξεπερνάμε τα νήματα που «χωράει άνετα» ένα socket/NUMA node, η εκτέλεση αρχίζει να μοιράζεται σε πολλαπλούς κόμβους μνήμης και αυξάνονται οι απομακρυσμένες προσπελάσεις (remote NUMA accesses) και η συμφόρηση στον δίαυλο μνήμης. Ταυτόχρονα, οι atomic ενημερώσεις στους κοινόχρηστους πίνακες newClusters[] και newClusterSize[] δημιουργούν έντονο contention στις ίδιες cache lines, με αποτέλεσμα η θεωρητική παραλληλία να χάνεται από τον συγχρονισμό, όπως ακριβώς περιγράφεται στις διαφάνειες για synchronization bottlenecks και NUMA αρχιτεκτονικές. Συνολικά, το affinity εκμεταλλεύεται καλά τη δομή του κόμβου μέχρι τα 8 νήματα, αλλά τα αρχιτεκτονικά

και αλγοριθμικά όρια της naive shared λύσης δεν επιτρέπουν ουσιαστική κλιμάκωση πέρα από αυτό το σημείο.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



Από το τελευταίο διάγραμμα, όπου ως βάση λαμβάνουμε πλέον την παράλληλη εκτέλεση με 1 νήμα, βλέπουμε ότι τα speedups για 2, 4 και 8 νήματα είναι ιδιαίτερα υψηλά (της τάξης του 1.7–1.8×, ~3× και ~4.5× αντίστοιχα). Αυτό δείχνει ότι το σημαντικό parallel overhead της OpenMP (δημιουργία και οργάνωση της ομάδας νημάτων, συγχρονισμοί, barriers) κατανέμεται αποτελεσματικά σε λίγα νήματα όταν αυτά «μένουν» σε κοντινούς πυρήνες του ίδιου NUMA node, με αποτέλεσμα η αύξηση του βαθμού παραλληλίας από 1 σε 2–8 threads να αποδίδει καθαρό κέρδος εντός της ίδιας αρχιτεκτονικής (κάπως κοντά σε γραμμικά, ειδικά αρχικά).

Παρ' όλα αυτά, όταν συνεχίζουμε πέραν από τα 8 νήματα, τα speedups ως προς το 1-thread παράλληλο πρόγραμμα μειώνονται αισθητά, γεγονός που υποδηλώνει ότι έχουμε φτάσει πρακτικά το όριο των πόρων (πυρήνων, cache, memory bandwidth) ενός sockets και αρχίζουμε να «πατάμε» σε δεύτερο NUMA

node ή/και να ενεργοποιούμε hardware multithreading στους ίδιους φυσικούς πυρήνες. Σε έναν αλγόριθμο όπως ο K-means, που είναι memory-bound και επιβαρυνμένος με atomic operations και συγχρονισμό σε κοινόχρηστες δομές, η περαιτέρω αύξηση των νημάτων δεν μπορεί να εκμεταλλευτεί αποτελεσματικά τις διαθέσιμες memory lanes και οδηγεί σε κορεσμό και σε περισσότερη εμπλοκή μεταξύ των νημάτων. Έτσι, το affinity βελτιώνει σημαντικά την απόδοση μέχρι τα 8 threads, αλλά τα εγγενή NUMA και synchronization bottlenecks της naive shared υλοποίησης εξακολουθούν να περιορίζουν την κλιμάκωση σε μεγαλύτερο αριθμό νημάτων.

2.1.1 – Copied Clusters and Reduce

Στη δεύτερη παράλληλη υλοποίηση του αλγορίθμου K-means υιοθετούμε και πάλι το μοντέλο shared clusters, αλλά αυτή τη φορά με τεχνική copied clusters και reduction για τη συλλογή των μερικών αποτελεσμάτων. Αντί όλα τα νήματα να ενημερώνουν απευθείας τους κοινόχρηστους πίνακες newClusters[] και newClusterSize[], κάθε νήμα διατηρεί δικά του, τοπικά αντίγραφα (local_newClusters[tid], local_newClusterSize[tid]) τα οποία ενημερώνει ελεύθερα, χωρίς atomic operations ή άλλον συγχρονισμό. Στο τέλος του παράλληλου βρόχου, τα τοπικά αυτά αντίγραφα συγχωνεύονται σε έναν κοινό πίνακα μέσω μιας φάσης reduction, η οποία εκτελείται από ένα νήμα (ή σε ένα μικρό, καλά οριοθετημένο σειριακό τμήμα κώδικα).

Η προσέγγιση αυτή αυξάνει λίγο τη χρήση μνήμης και προσθέτει ένα επιπλέον βήμα συγχώνευσης, αλλά μειώνει δραστικά το κόστος συγχρονισμού σε σχέση με τη naïve εκδοχή, καθώς αποφεύγονται οι χιλιάδες ατομικές ενημερώσεις πάνω στις ίδιες cache lines. Έτσι, η 2.1.2 στοχεύει σε πολύ καλύτερη κλιμάκωση με τον αριθμό νημάτων, ειδικά σε NUMA αρχιτεκτονικές, όπου η μείωση του contention στη μνήμη παίζει καθοριστικό ρόλο στην επίδοση.

Όσον αφορά τις λεπτομέρειες της υλοποίησής μας, επισημαίνουμε τα εξής:

- Ο παράλληλος βρόχος (`#pragma omp parallel`) αναθέτει σε κάθε νήμα ένα υποσύνολο αντικειμένων, όπως και πριν, αλλά οι ενημερώσεις των clusters γίνονται αποκλειστικά στα τοπικά arrays `local_newClusterSize[tid]` και `local_newClusters[tid]`, χωρίς χρήση `atomic`.
- Η μεταβλητή `delta`, που μετράει τις αλλαγές στα memberships, υπολογίζεται πλέον με κατάλληλο reduction μέσα στο `parallel`, ώστε να αποφεύγονται επιπλέον ατομικές προσπελάσεις και να διατηρείται η ορθότητα του κριτηρίου σύγκλισης.
- Στο τέλος του βρόχου, μια `#pragma omp single` περιοχή εκτελεί τη φάση reduction, αθροίζοντας τα τοπικά αντίγραφα όλων των νημάτων στους κοινόχρηστους πίνακες `newClusterSize[]` και `newClusters[]`. Με αυτόν τον τρόπο συγκεντρώνονται τα μερικά αποτελέσματα με ελάχιστο συγχρονισμό, σε ένα καλά ελεγχόμενο σημείο του προγράμματος.

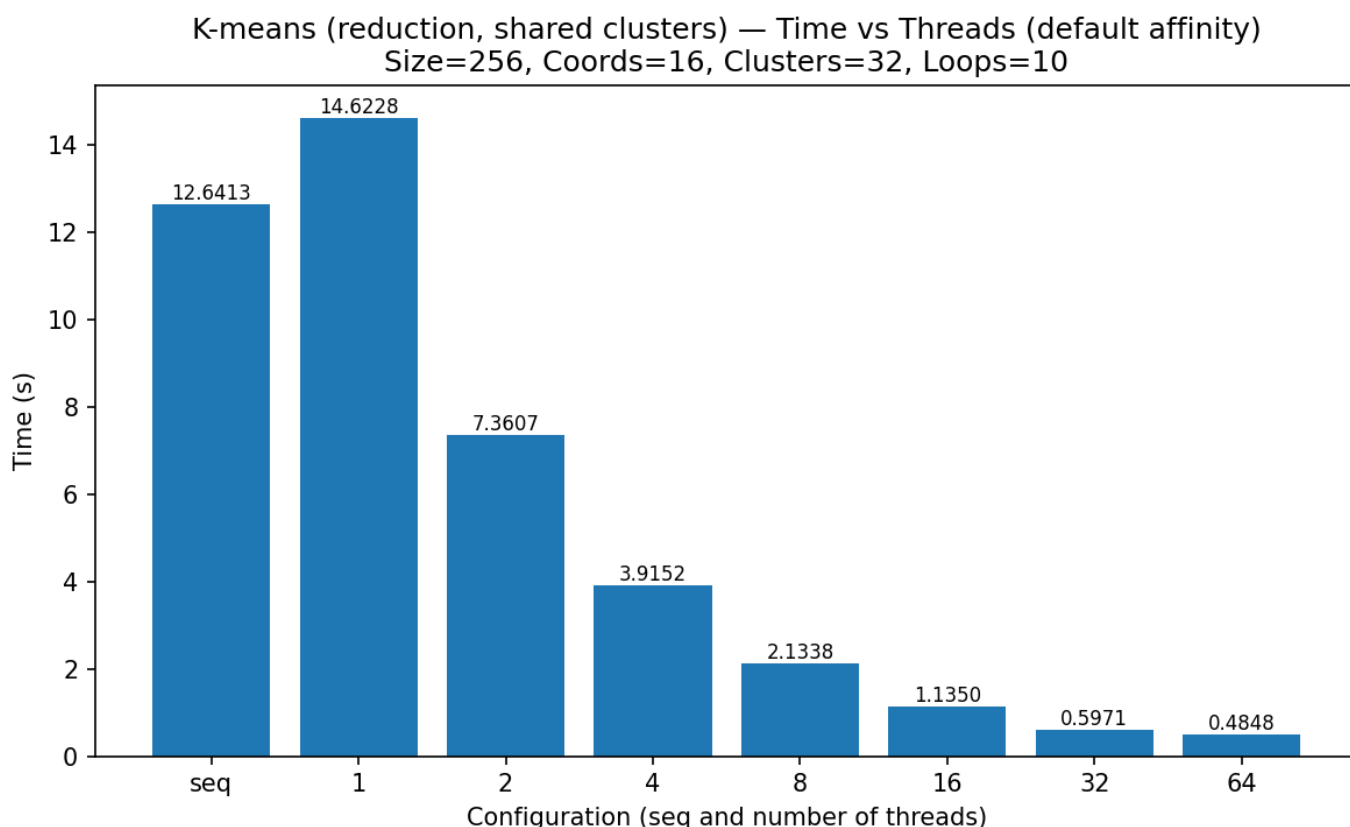
Ο ολοκληρωμένος κώδικας της υλοποίησης (`omp_reduction_kmeans.c`) βρίσκεται στον `scirouter` και στον `orion` της ομάδας μας, αλλά παρατίθεται και στην παρούσα αναφορά για λόγους πληρότητας.

1. Παραλληλοποίηση για το αρχικό grid size και διαγράμματα

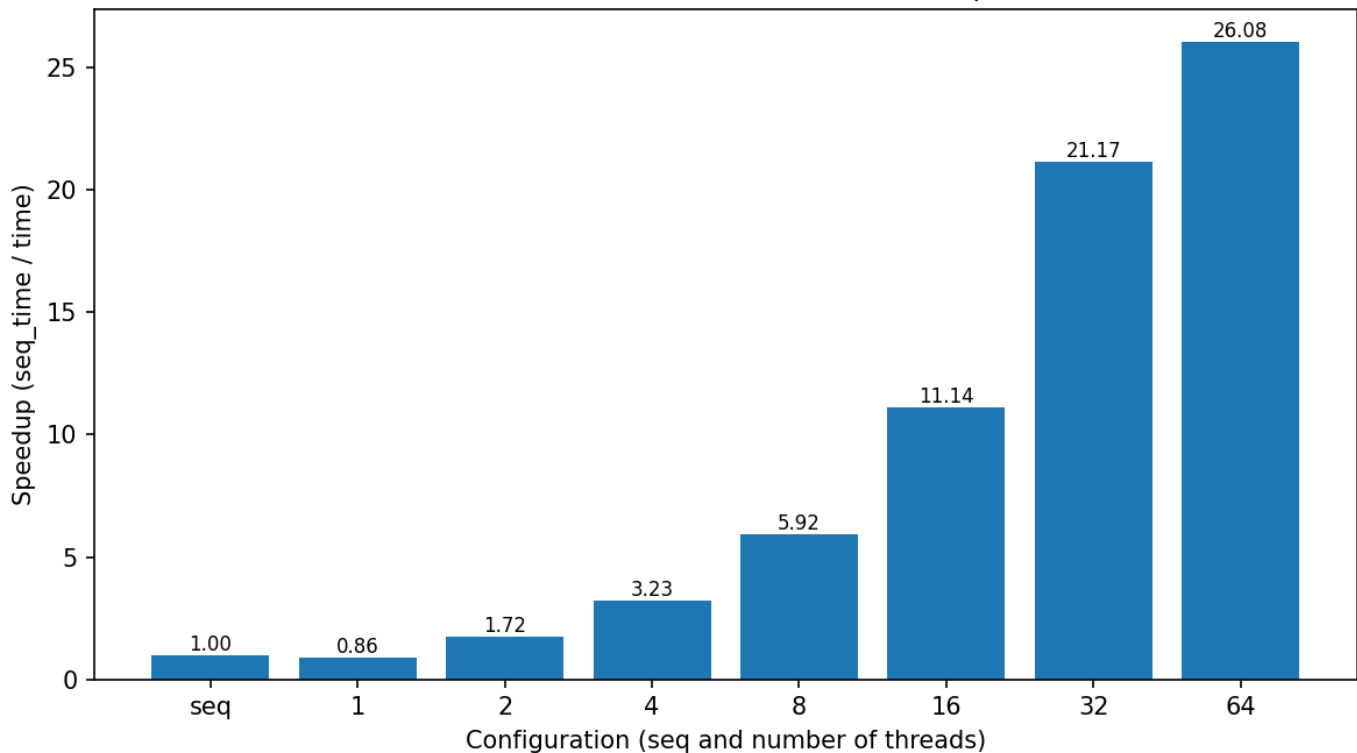
Το παραπάνω παράλληλο πρόγραμμα (omp_reduction_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και με affinity (όπως και πριν). Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	12.64
1	14.62
2	7.36
4	3.92
8	2.13
16	1.14
32	0.60
64	0.48

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



K-means (reduction, shared clusters) — Speedup vs Threads (default affinity)
Size=256, Coords=16, Clusters=32, Loops=10

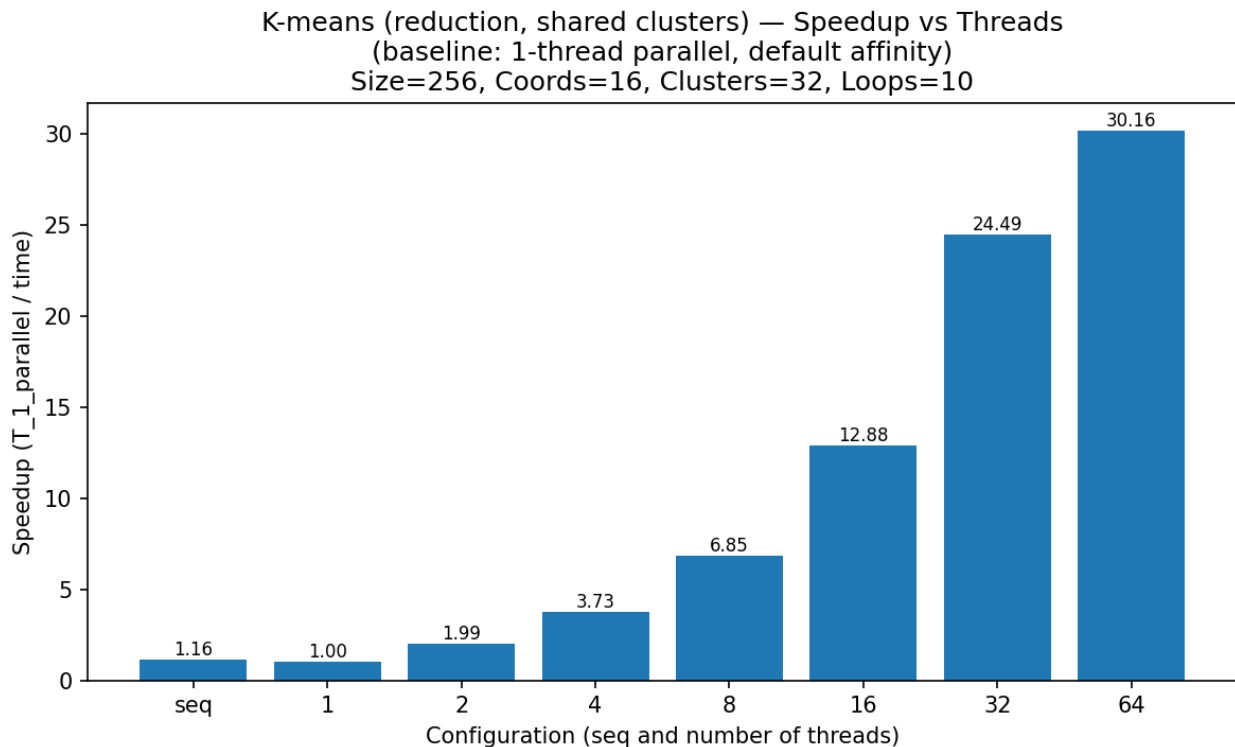


Από τα δύο πρώτα διαγράμματα παρατηρούμε ότι η έκδοση με copied clusters και reduction κλιμακώνει πλέον πολύ ικανοποιητικά σε σχέση με τη naive shared προσέγγιση. Ο σειριακός χρόνος είναι περίπου 12.6s, ενώ η παράλληλη εκτέλεση με 1 νήμα παραμένει λίγο χειρότερη (~14.6s), λόγω του parallel overhead του OpenMP, όπως και πριν. Ωστόσο, από τα 2 νήματα και πάνω ο χρόνος μειώνεται μονοτονικά και σχεδόν γραμμικά: ~7.4s στα 2 threads, ~3.9s στα 4, ~2.1s στα 8, ~1.1s στα 16, ~0.6s στα 32 και ~0.5s στα 64 threads. Αντίστοιχα, το speedup ως προς το σειριακό πρόγραμμα φτάνει περίπου τις 1.7x, 3.2x, 5.9x, 11x, 21x και 26x για 2, 4, 8, 16, 32 και 64 νήματα αντίστοιχα, πολύ κοντά στην ιδανική κλιμάκωση που παρουσιάζεται και στις διαφάνειες.

Η διαφορά σε σχέση με τη naive υλοποίηση εξηγείται από την αρχιτεκτονική της reduction λύσης: κάθε νήμα ενημερώνει αποκλειστικά τα δικά του local αντίγραφα (local_newClusters, local_newClusterSize), αποφεύγοντας atomic ενημερώσεις σε κοινές cache lines και μειώνοντας δραστικά το synchronization bottleneck. Έτσι, το μεγαλύτερο μέρος του χρόνου ξοδεύεται σε πραγματικό υπολογισμό και όχι σε συγχρονισμό, κάτι που επιτρέπει στον αλγόριθμο να κλιμακώνεται πολύ καλύτερα σε ένα NUMA σύστημα όπως ο sandman. Μέχρι τα 32 threads (ένας hardware thread ανά φυσικό πυρήνα) αξιοποιούνται αποτελεσματικά οι πόροι όλων των sockets, ενώ η μικρή “κάμψη” της κλιμάκωσης από τα 32 στα 64 νήματα αποδίδεται κυρίως στο hardware multithreading και στον κορεσμό του

memory bandwidth: δύο λογικά νήματα ανά πυρήνα μοιράζονται την ίδια εκτέλεση και τις ίδιες memory lanes σε έναν ήδη memory-bound αλγόριθμο.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



Το τελευταίο διάγραμμα, όπου ως βάση λαμβάνουμε την παράλληλη εκτέλεση με 1 νήμα, αναδεικνύει ακόμη πιο καθαρά το όφελος της reduction υλοποίησης. Σε αυτή τη σύγκριση, το σειριακό πρόγραμμα εμφανίζεται ήδη ταχύτερο από το 1-thread parallel (speedup $\approx 1.2\times$), επιβεβαιώνοντας ότι το κόστος δημιουργίας και οργάνωσης της ομάδας νημάτων είναι σημαντικό όταν χρησιμοποιείται μόνο ένα νήμα. Από τα 2 threads και πάνω, όμως, η κλιμάκωση γίνεται εντυπωσιακή: το speedup φτάνει περίπου τις $2\times$ στα 2 νήματα, $\sim 3.7\times$ στα 4, $\sim 6.8\times$ στα 8, $\sim 12.9\times$ στα 16, $\sim 24.5\times$ στα 32 και πάνω από $30\times$ στα 64 νήματα σε σχέση με το 1-thread parallel.

Τα αποτελέσματα αυτά δείχνουν ότι, μόλις “ξεπληρωθεί” το parallel overhead, η copied clusters and reduction προσέγγιση εκμεταλλεύεται πλήρως την παραλληλία που προσφέρει ο κόμβος: μέχρι τα 32 threads αξιοποιείται ουσιαστικά κάθε φυσικός πυρήνας όλων των NUMA nodes, με πολύ μικρό synchronization cost χάρη στα local arrays, ενώ η περαιτέρω αύξηση στα 64 threads δίνει μικρότερο, αλλά υπαρκτό, πρόσθετο κέρδος λόγω του hardware multithreading. Σε αντίθεση με τη naive shared υλοποίηση, εδώ η κλιμάκωση περιορίζεται κυρίως από το διαθέσιμο memory

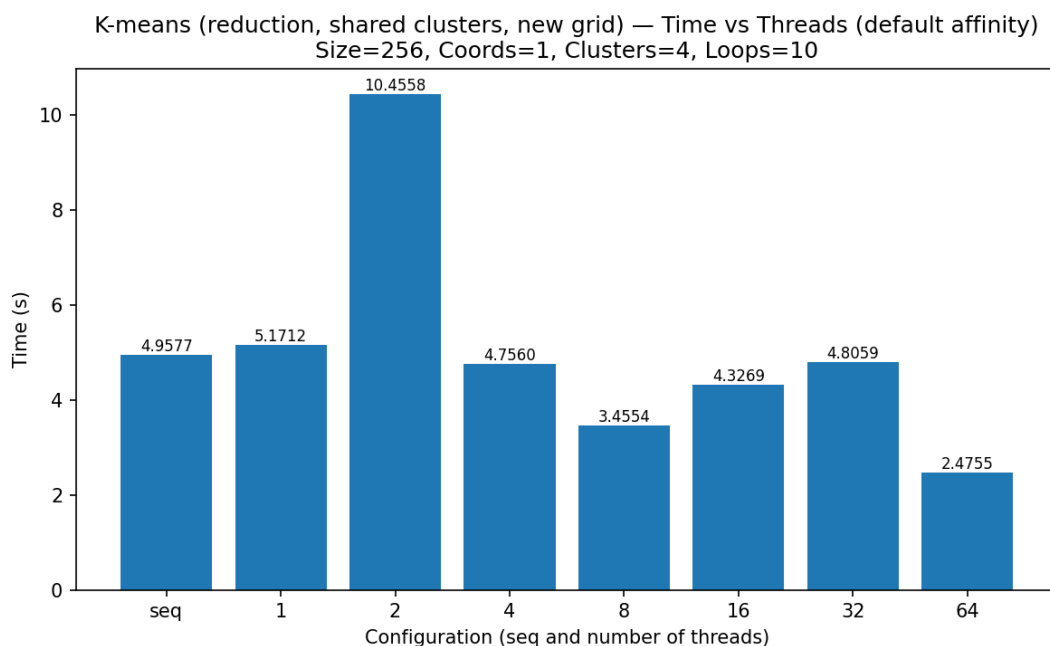
bandwidth και το μικρό σειριακό τμήμα του κώδικα (νόμος του Amdahl), ενώ το synchronization bottleneck έχει ουσιαστικά εξαλειφθεί.

2. Παραλληλοποίηση για το μειωμένο grid size και διαγράμματα

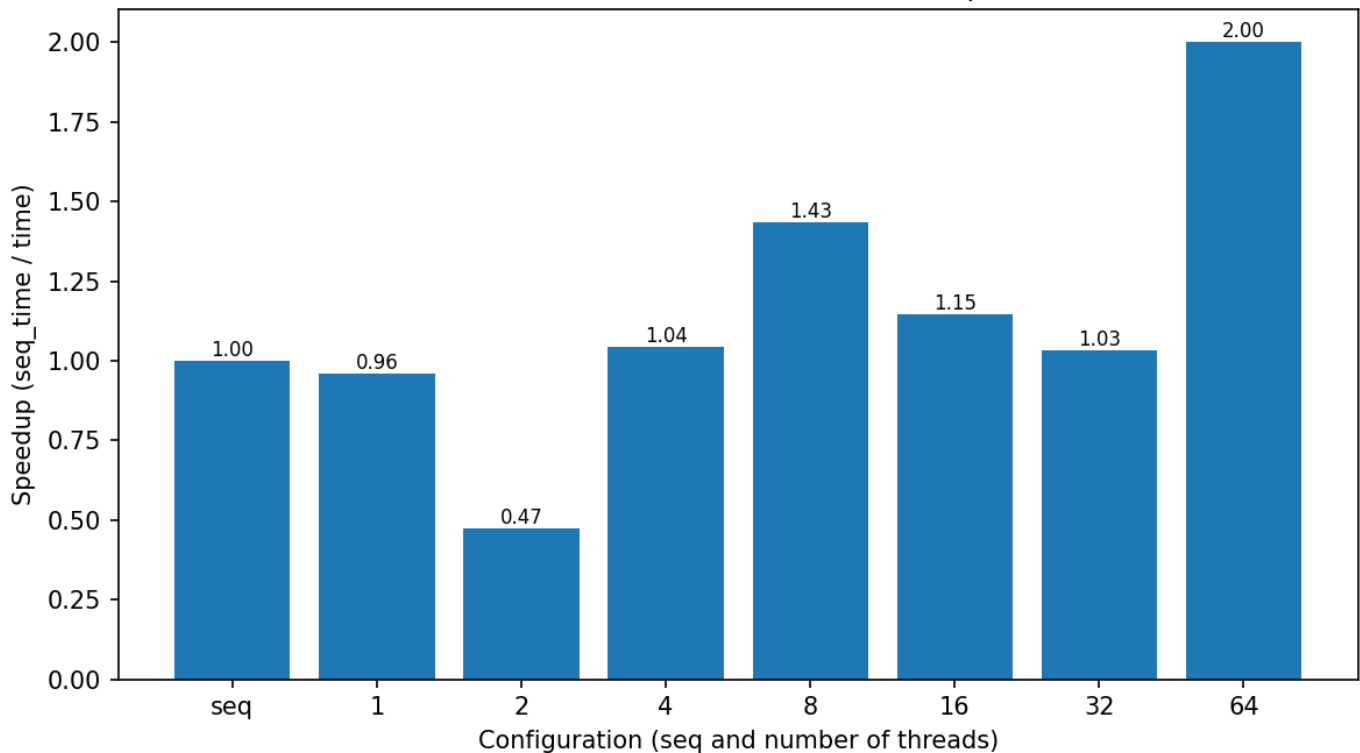
Το παραπάνω παράλληλο πρόγραμμα (omp_reduction_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και με affinity (όπως και πριν). Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	4.96
1	5.17
2	10.46
4	4.76
8	3.46
16	4.33
32	4.81
64	2.48

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



K-means (reduction, shared clusters, new grid) — Speedup vs Threads (default affinity)
Size=256, Coords=1, Clusters=4, Loops=10

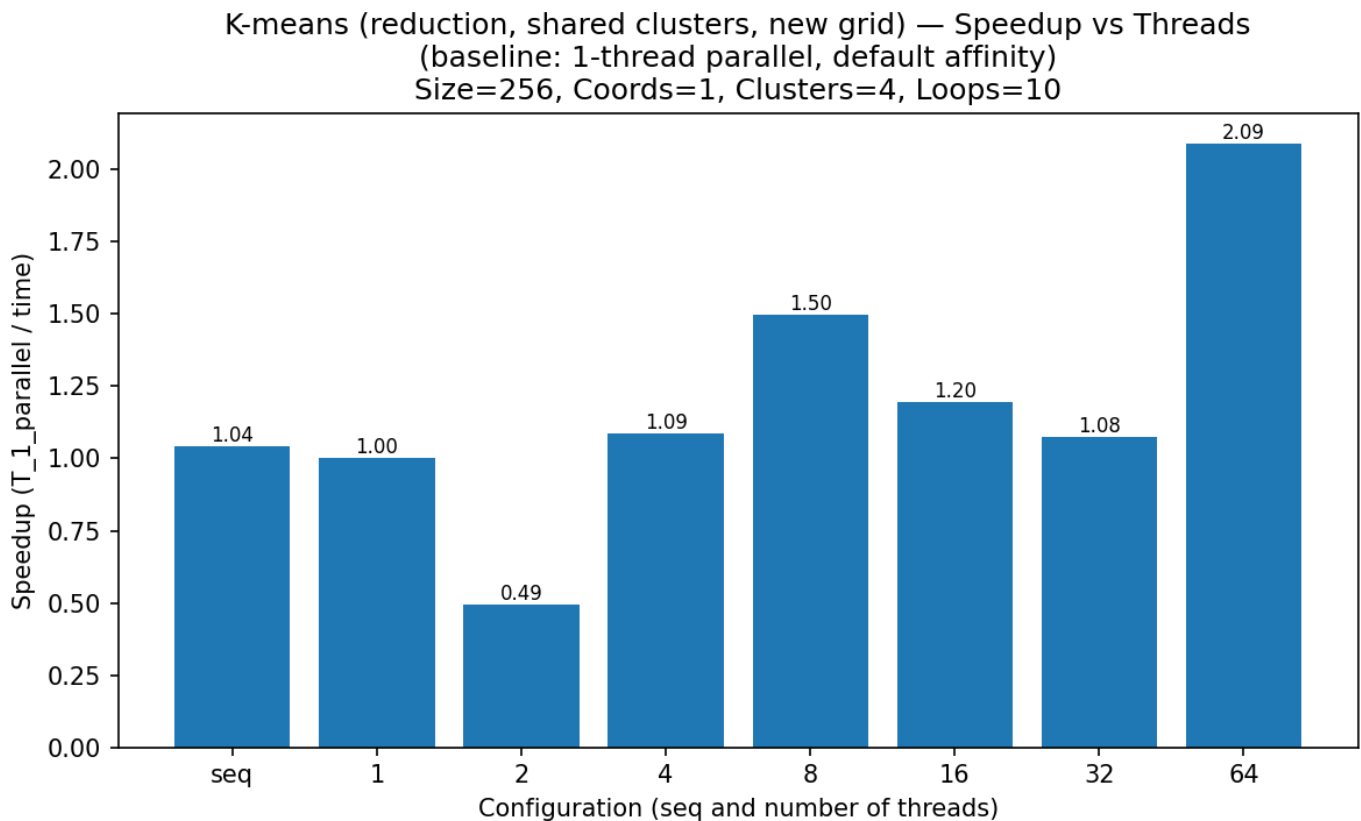


Από τα δύο πρώτα διαγράμματα για το μειωμένο grid παρατηρούμε ότι η συμπεριφορά της έκδοσης με reduction είναι σαφώς χειρότερη από αυτή στο αρχικό, πιο «υπολογιστικά βαρύ» (compute intensive) grid ({256,16,32,10}). Εδώ ο σειριακός χρόνος είναι περίπου 4.96s και η παράλληλη εκτέλεση με 1 νήμα λίγο χειρότερη (~5.17s), αλλά η κλιμάκωση με περισσότερα νήματα δεν είναι πλέον καλή: στα 2 threads ο χρόνος μάλιστα χειροτερεύει σημαντικά (~10.46s), στα 4 και 8 νήματα έχουμε μια μικρή βελτίωση (4.76s και 3.46s αντίστοιχα), ενώ στα 16 και 32 νήματα ο χρόνος ξανανεβαίνει κοντά στον σειριακό (4.33s και 4.81s) και μόνο στα 64 threads πέφτει στα ~2.48s. Αντίστοιχα, το speedup ως προς το σειριακό πρόγραμμα μόλις που ξεπερνά το 1.4× στα 8 νήματα και φτάνει περίπου το 2× στα 64 νήματα.

Η βασική διαφορά στα scalability plots, σε σχέση με το προηγούμενο grid, είναι ότι εδώ η κλιμάκωση «επιπεδώνει» πολύ νωρίς και είναι έντονα ακανόνιστη. Στο αρχικό grid με 16 συντεταγμένες και 32 clusters, ο αλγόριθμος ήταν πολύ πιο compute-intensive και η έκδοση με reduction κατάφερε να φτάσει speedup ~26× στα 64 threads. Αντίθετα, στο μειωμένο grid με μόνο 1 συντεταγμένη και 4 clusters, το workload ανά αντικείμενο είναι πολύ μικρότερο και η εφαρμογή είναι ακόμη πιο έντονα memory-bound: κάθε νήμα κάνει ελάχιστες πράξεις ανά προσπέλαση μνήμης, με αποτέλεσμα το κόστος πρόσβασης σε μνήμη και το overhead του OpenMP να κυριαρχούν. Έτσι, ο διαθέσιμος υπολογισμός δεν αρκεί για να «κρύψει»

τη latency της μνήμης και το scalability περιορίζεται σημαντικά, όπως φαίνεται στα διαγράμματα.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



Στο τελευταίο διάγραμμα, όπου ως βάση λαμβάνουμε την παράλληλη εκτέλεση με 1 νήμα, γίνεται ακόμη πιο εμφανές ότι στο μειωμένο grid το scaling είναι περιορισμένο και ασταθές. Ο σειριακός κώδικας παραμένει ελαφρώς ταχύτερος από το 1-thread parallel, ενώ τα speedups ως προς το 1-thread πρόγραμμα κινούνται γύρω στη μονάδα για τα περισσότερα T: το 2-thread run είναι σαφώς χειρότερο ($\text{speedup} < 1$), στα 4 και 8 νήματα έχουμε κάποια βελτίωση, ενώ στα 16 και 32 νήματα ουσιαστικά δεν κερδίζουμε τίποτα. Μόνο στα 64 threads παρατηρείται πιο αξιοσημείωτο κέρδος, της τάξης περίπου του 2x, αλλά και πάλι πολύ μακριά από τα speedups που είδαμε στο αρχικό grid και την επιθυμητή κλιμάκωση.

Η διαφορά στα scalability plots σε σχέση με την περίπτωση {256,16,32,10} εξηγείται από το ότι εδώ το πρόβλημα είναι πλέον «πολύ μικρό» υπολογιστικά ανά στοιχείο και κυριαρχείται από τις προσπελάσεις στη μνήμη (memory-bound). Στο αρχικό grid, ο μεγάλος αριθμός συντεταγμένων και clusters παρείχε αρκετό

υπολογισμό ώστε η έκδοση με reduction να εκμεταλλεύεται ουσιαστικά όλους τους πυρήνες μέχρι τα 32 threads και να έχει πολύ καλή κλιμάκωση. Στο νέο grid, το per-object work είναι μικρό και η ταυτόχρονη πρόσβαση πολλών νημάτων στα ίδια δεδομένα καταναλώνει γρήγορα το memory bandwidth, με αποτέλεσμα ο επιπλέον παραλληλισμός να μην μπορεί να μεταφραστεί σε αντίστοιχο speedup. Ουσιαστικά, έχουμε ένα χαρακτηριστικό παράδειγμα από τις διαφάνειες: όσο πιο memory-bound είναι μια εφαρμογή και όσο μικρότερο το διαθέσιμο υπολογιστικό workload, τόσο πιο γρήγορα «σκάει» η κλιμάκωση και τα scalability plots γίνονται ρηχά και ασταθή.

Στο Linux, η πολιτική first-touch σε NUMA συστήματα ορίζει ότι οι φυσικές σελίδες μνήμης δεσμεύονται στο NUMA node του πυρήνα που τις «ακουμπά» πρώτος (πρώτη εγγραφή). Στο πρόγραμμά μας, οι πίνακες `local_newClusters` και `local_newClusterSize` δεσμεύονται αρχικά σειριακά (μέσα σε loops στο main thread), οπότε οι αντίστοιχες σελίδες μνήμης τοποθετούνται κατά κανόνα στο NUMA node όπου εκτελείται το main thread. Στη συνέχεια, όταν άλλα νήματα OpenMP που τρέχουν σε διαφορετικούς πυρήνες και nodes προσπελούν αυτές τις δομές, μεγάλο μέρος των προσβάσεων είναι «απομακρυσμένο» (remote NUMA), με αυξημένη latency και μειωμένο effective bandwidth. Αυτό περιορίζει την επίδοση, ειδικά στη reduction υλοποίηση, όπου η πρόσβαση στα τοπικά clusters δεδομένα είναι πολύ συχνή.

Επιπλέον, εμφανίζεται και το φαινόμενο false-sharing: παρόλο που κάθε νήμα ενημερώνει διαφορετικά elements μέσα στο `local_newClusters[tid]`, οι επιμέρους πίνακες για διαφορετικά tid μπορεί να τοποθετούνται σε συνεχόμενες διευθύνσεις και να μοιράζονται τις ίδιες cache lines. Έτσι, όταν δύο threads γράφουν σε διαφορετικά στοιχεία που τυχαίνει να κατοικούν στην ίδια γραμμή cache, οι γραμμές αυτές κάνουν συνεχώς invalidate μεταξύ των πυρήνων, δημιουργώντας σημαντικό overhead, χωρίς να υπάρχει πραγματικό data sharing σε επίπεδο προγράμματος.

Για να αντιμετωπίσουμε προβλήματα NUMA τοποθέτησης (first-touch), μπορούμε να αφήσουμε το κάθε νήμα να κάνει τη δική του δέσμευση μνήμης, π.χ. με malloc μέσα στην παράλληλη περιοχή: κάθε thread εκτελεί το malloc και την αρχικοποίηση των δικών του `local_newClusters[tid]` και `local_newClusterSize[tid]`, οπότε οι σελίδες του κάθε πίνακα first-touched από το αντίστοιχο νήμα καταλήγουν στον «σωστό» NUMA node. Με αυτόν τον τρόπο, οι επαναλαμβανόμενες προσπελάσεις σε τοπικά δεδομένα γίνονται κυρίως σε τοπική μνήμη, μειώνοντας σημαντικά τα remote NUMA accesses.

Για να περιορίσουμε το false-sharing, κάναμε χρήση padding στα τοπικά arrays: φροντίζουμε κάθε «γραμμή» `local_newClusters[tid]` να ευθυγραμμίζεται σε μέγεθος cache line (π.χ. 64 bytes) και να μεσολαβεί αρκετό κενό (padding) ανάμεσα στα δεδομένα διαφορετικών νημάτων, ώστε καμία cache line να μην περιέχει ταυτόχρονα δεδομένα από δύο διαφορετικά tid. Με αυτόν τον τρόπο, κάθε γραμμή cache ανήκει ουσιαστικά σε ένα μόνο thread και δεν υπάρχει αλληλοεπικάλυψη που θα προκαλούσε invalidations. Συνδυάζοντας την τεχνική του per-thread malloc (για σωστό first-touch ανά thread και NUMA node) με το κατάλληλο padding, μειώνουμε τόσο τα προβλήματα NUMA τοποθέτησης όσο και τα φαινόμενα false-sharing, όπως προτείνεται και στο hint της εκφώνησης.

Γενικές Παρατηρήσεις

- Η υλοποίηση με reduction αποδείχθηκε σαφώς πιο αποδοτική από αυτή με atomic operations, καθώς μεταφέρει το κόστος συγχρονισμού σε ένα μικρό, καλά οριοθετημένο στάδιο συγχώνευσης (reduction) και αφήνει τον κυρίως βρόχο να εκτελείται χωρίς locks. Αντίθετα, στην naïve υλοποίηση μεγάλο μέρος του χρόνου χάνεται σε atomic ενημερώσεις πάνω στις ίδιες cache lines. Παρ' όλα αυτά, το reduction δεν είναι πάντα προτιμότερο: σε σενάρια με μικρό πρόβλημα ή λίγες συγκρούσεις στα shared δεδομένα, το επιπλέον κόστος αντιγραφής και συγχώνευσης μπορεί να εξανεμίσει τα κέρδη.
- Πέραν των 32 threads δεν παρατηρείται ποτέ ουσιαστική (σχεδόν γραμμική) βελτίωση, καθώς στο sandman έχουμε 32 φυσικούς πυρήνες και 64 λογικά νήματα μέσω hardware multithreading (hyperthreading). Σε έναν κατά βάση memory-bound αλγόριθμο, όπως ο K-means με shared clusters, δύο λογικά νήματα στον ίδιο πυρήνα μοιράζονται τους ίδιους execution πόρους και τις ίδιες memory lanes, οπότε η περαιτέρω αύξηση των νημάτων δεν μεταφράζεται σε αντίστοιχο speedup και μπορεί να οδηγήσει ακόμη και σε υποβάθμιση της επίδοσης.
- Η επιλογή μιας κατάλληλης πολιτικής affinity βελτιώνει αισθητά την επίδοση. Η δέσμευση των νημάτων σε συγκεκριμένους πυρήνες (ώστε να «μένουν κοντά» σε δεδομένα και cache) μειώνει το κόστος επικοινωνίας με τη μνήμη και εκμεταλλεύεται καλύτερα την τοπικότητα. Παράλληλα, η προσεκτική διασπορά των threads στα NUMA nodes μπορεί να αυξήσει το διαθέσιμο memory bandwidth για memory-bound εφαρμογές. Τα πειράματά μας δείχνουν ξεκάθαρα ότι με ενεργό affinity η επίδοση μέχρι τα 8–16 threads βελτιώνεται σημαντικά σε σχέση με την πλήρως “noaff” περίπτωση.

▪ Ενότητα 2.2 – Παραλληλοποίηση του Αλγορίθμου Floyd-Warshall

- **Υλοποίηση και ανάλυση εξαρτήσεων**

Για την παραλληλοποίηση του recursive Floyd-Warshall αλγορίθμου χρησιμοποιήθηκαν OpenMP tasks. Η βασική πρόκληση προέκυψε από το γεγονός ότι ο αλγόριθμος παίρνει ως ορίσματα τον ίδιο πίνακα (A) τρεις φορές (A,B,C), αν και με διαφορετικά offsets. Αυτό έχει ως αποτέλεσμα, παρόλο που τα A,B,C έχουν διαφορετικό όνομα, να αναφέρονται συχνά στην ίδια θέση μνήμης, δημιουργώντας εξαρτήσεις τύπου RAW (Read After Write) και WAW (Write After Write).

- **Μελέτη των Αναδρομικών Κλήσεων**

Η μελέτη των εξαρτήσεων ανά αναδρομική κλήση βασίστηκε στις ακόλουθες παρατηρήσεις:

1. Κάθε αναδρομική κλήση γίνεται για blocks μισού μεγέθους σε σχέση με αυτά της εισόδου (3 blocks/arrays ίδιων διαστάσεων)
2. Κάθε πίνακας/block χωρίζεται σε 4 ίσα μέρη, επιτρέποντας σε διαφορετικά blocks ίδιων διαστάσεων να μην έχουν μερικές επικαλύψεις.
3. Σε πιο "βαθύ επίπεδο" της αναδρομής, ο αλγόριθμος περιορίζεται σε εγγραφή στον A (και υπό-blocks του) και ανάγνωση από τους B,C (και υπό-blocks τους) στην δεδομένη κλήση. Αυτό εξασφαλίζει ότι σε βαθύτερο επίπεδο δεν υπάρχει επικάλυψη πρόσβασης μνήμης που δεν φαίνεται στο ψηλότερο.

Μελετώντας τις κλήσεις, το πρόβλημα χωρίστηκε σε 4 περιπτώσεις ανάλογα με τις σχέσεις των blocks A,B,C.

- **Σενάρια Παραλληλισμού (Task Chains)**

Θεωρώντας ως call_i την i-οστή σε σειρά αναδρομική κλήση του παρακάτω αλγορίθμου βγάζουμε τα συμπεράσματα του πίνακα:

FWR (A00, B00, C00);

FWR (A01, B00, C01);

FWR (A10, B10, C00);

FWR (A11, B10, C01);

FWR (A11, B10, C01);

FWR (A10, B10, C00);

FWR (A01, B00, C01);

FWR (A00, B00, C00);

Case ID	Σχέση Blocks	Αλυσίδα Εξαρτήσεων	Παραλληλισμός
1	A=B=C (όλα ταυτίζονται)	1 -> {2,3} -> 4 -> 5 -> {6,7}->8	Τα ζευγάρια {2, 3} και {6, 7} εκτελούνται παράλληλα10.
2/3	A=B ή A=C (αλλά B ≠ C)	Αν A=B: 1->2->7->8. και 3->4->5->6. Αν A=C: 1->3->6->8 και 2->4->5->6->8	Προκύπτουν 2 παράλληλες και εντελώς ανεξάρτητες αλυσίδες εκτέλεσης
4	A≠ B≠ C (όλα διαφορετικά)	1-> 8, 2->7, 3->6, 4->5	Προκύπτουν 4 παράλληλες και εντελώς ανεξάρτητες αλυσίδες

Στην αρχική κλήση (A,0,0,A,0,0,A,0,0), εφαρμόζεται η Case 1. Η εξάρτηση RAW των κλήσεων 2, 3 από το 1, και η εξάρτηση του 4 από 2, 3, καθορίζουν τη σειρά εκτέλεσης.

- **Επιλογή block size (B)**

Για την επιλογή του μεγέθους του ελάχιστου block (bsize) όπου γίνεται ο υπολογισμός του min (base case), δοκιμάστηκαν $B=\{16,32,64,128,256\}$ για $N=1024$:

- ✓ Παρατηρήθηκε ότι η επίδοση βελτιωνόταν σημαντικά με την αύξηση του B.
- ✓ Στο $B=128$ δεν σημειώθηκε βελτίωση, ενώ στο $B=256$ ο χρόνος εκτέλεσης μάλιστα αυξήθηκε.
- ✓ Επιλέχθηκε $B=64$ για την υπόλοιπη άσκηση. Αν και το $B=128$ ήταν αποδοτικό, το $B=64$ προτιμήθηκε για να αξιοποιηθεί περισσότερο η παραλληλία, καθώς μεγαλύτερο B αντιστοιχεί σε λιγότερα layers παράλληλου προγράμματος. Η επιλογή του B επηρεάζει ελάχιστα το speedup, καθώς επηρεάζει με τον ίδιο τρόπο το σειριακό και το παράλληλο πρόγραμμα.

- **Παράμετροι Εκτέλεσης και Περιβάλλοντος**

Η εκτέλεση και οι μετρήσεις του παράλληλου αλγορίθμου Recursive Floyd-Warshall (FW_SR) πραγματοποιήθηκαν μέσω του script `run_on_queue.sh` στο περιβάλλον PBS (Portable Batch System) του συστήματος sandman. Η εργασία υποβλήθηκε στην ουρά serial ζητώντας 64 πυρήνες (`ppn=64`) στον κόμβο sandman. Το πρόγραμμα εκτελέστηκε επαναληπτικά για τρία μεγέθη πίνακα (N): 1024, 2048, και 4096, χρησιμοποιώντας την βελτιστοποιημένη τιμή $B=64$ για το μέγεθος του ελάχιστου block. Ο παραλληλισμός OpenMP ελέγχθηκε μέσω της μεταβλητής περιβάλλοντος `OMP_NUM_THREADS`, η οποία διατρήθηκε στις τιμές $T=\{1,2,4,8,16,32,64\}$. Κάθε συνδυασμός N και T καταγράφηκε σε ξεχωριστά αρχεία εξόδου (`.out` και `.err`), εξασφαλίζοντας ακριβή δεδομένα για την ανάλυση της κλιμάκωσης του αλγορίθμου.

Τα `run_on_queue.sh` και το κυρίως πρόγραμμα `fw_sr_p.c` φαίνονται ακολούθως:

- **Αποτελέσματα Μετρήσεων Επίδοσης**

Αρχικά δοκιμάσαμε να τρέξουμε τον σειριακό (επαναληπτικό) αλγόριθμο. Ο χρόνος εκτέλεσης για $N=1024$ ήταν 1.3954.

Στη συνέχεια, τρέχοντας το πρόγραμμα με τις παραμέτρους της προηγούμενης παραγράφου στο sandman πήραμε τα εξής αποτελέσματα:

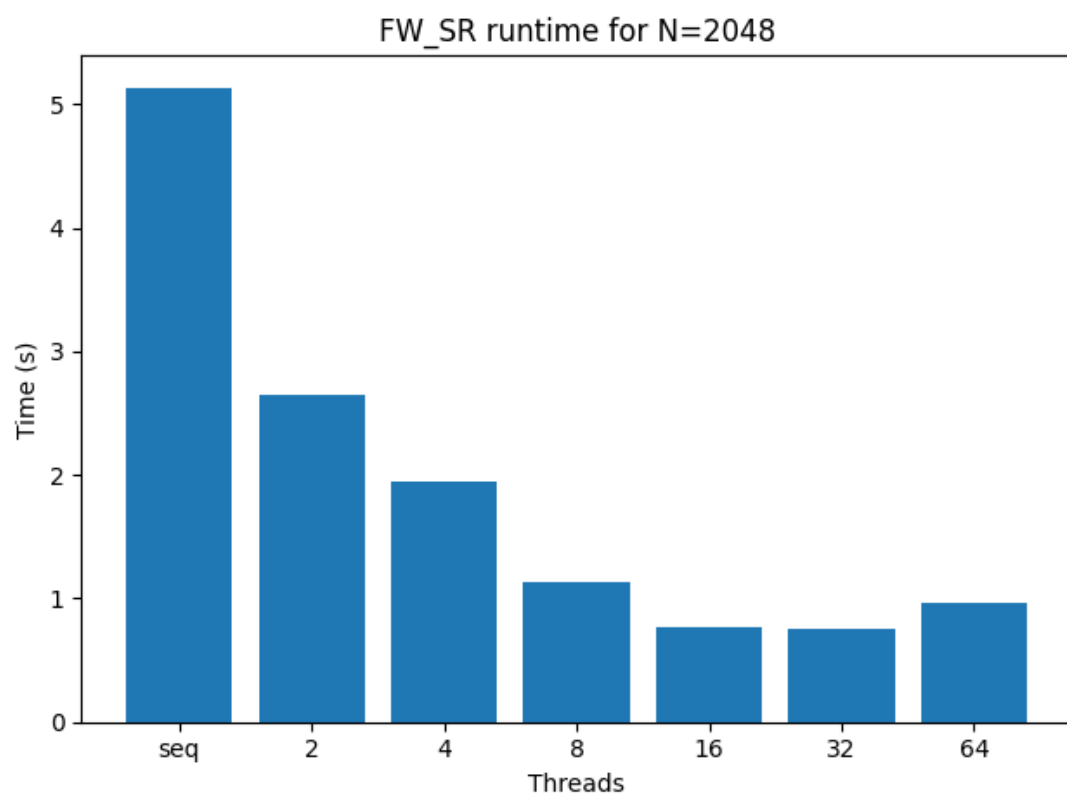
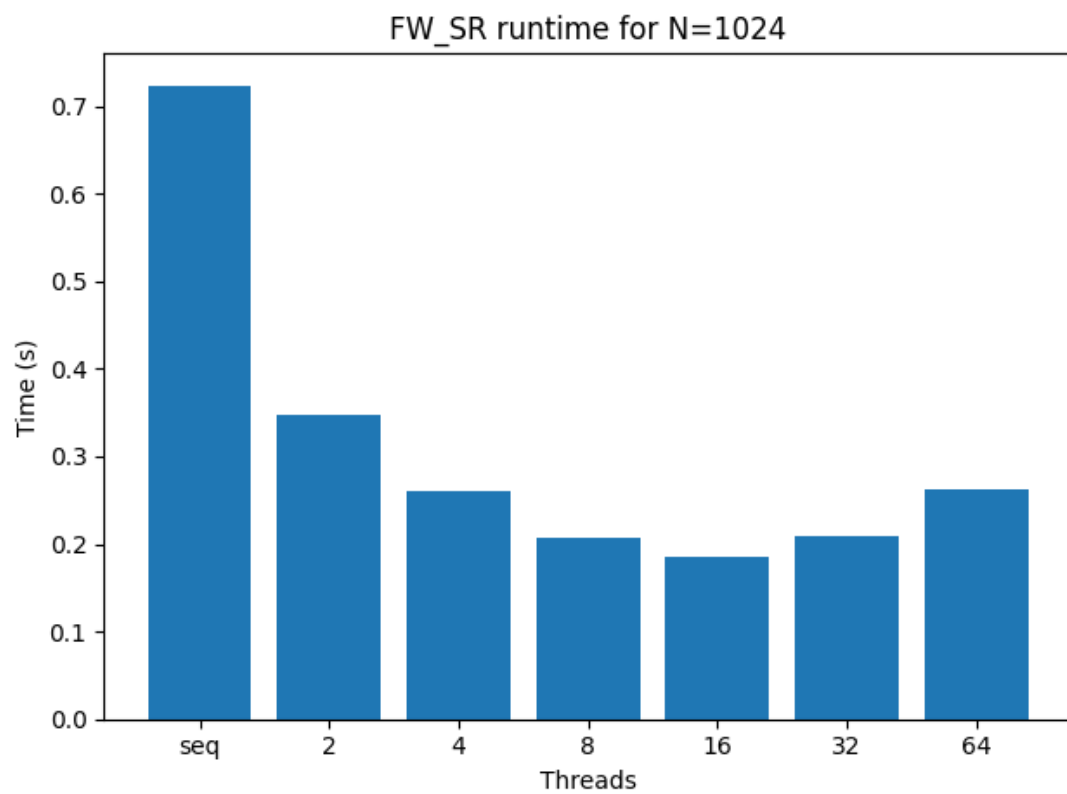
Χρόνοι εκτέλεσης

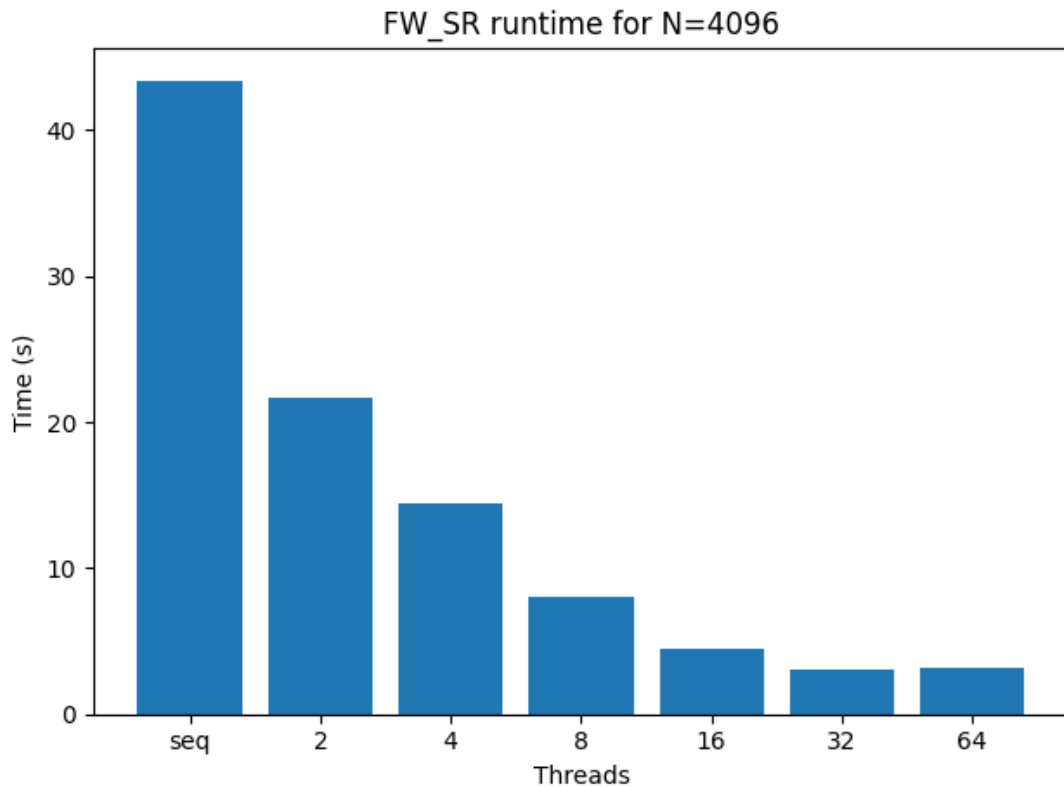
N	Tseq (T=1)	T=2	T=4	T=8	T=16	T=32	T=64
1024	0.7237	0.3467	0.2604	0.2078	0.1850	0.2097	0.2627
2048	5.1359	2.6523	1.9513	1.1305	0.7720	0.7496	0.9577
4096	43.4211	21.7275	14.3879	7.9806	4.4654	3.0423	3.1596

Συντελεστής επιτάχυνσης (speedup)

N	T=2	T=4	T=8	T=16	T=32	T=64	Max S
1024	2.09	2.78	3.48	3.91	3.45	2.75	3.91
2048	1.94	2.63	4.54	6.65	6.85	5.36	6.85
4096	2.00	3.02	5.44	9.72	14.27	13.74	14.27

Ακολουθούν και τα barplots για τα runs των διαφορετικών N:





- **Συμπεράσματα και παρατηρήσεις**

- ✓ Αρχικά παρατηρούμε ότι η χρήση του σειριακού recursive αλγορίθμου έφερε από μόνη της σημαντική βελτίωση (από 1.3954 σε 0.7237). Αυτή οφείλεται πιθανώς στην βελτίωση της cache locality που επιτυγχάνεται με τη recursive δομή του αλγορίθμου. Αυτή η αναδρομική διάσπαση του πίνακα σε μικρότερα blocks επιτρέπει την πιο αποδοτική χρήση της ιεραρχίας της κρυφής μνήμης (cache hierarchy), μειώνοντας δραστικά τα misses.
- ✓ Η κλιμάκωση (speedup) είναι καλύτερη για το μεγαλύτερο dataset (N=4096), φτάνοντας το 14.27× στους 32 threads. Αυτό συμβαίνει επειδή, για μεγάλο N, ο τεράστιος όγκος των υπολογισμών ($O(N^3)$) κυριαρχεί έναντι του overhead του OpenMP tasking και των καθυστερήσεων της μνήμης, επιτρέποντας την καλύτερη αξιοποίηση της παραλληλίας.

- ✓ Το ταβάνι στο scalability (Max Speedup) οφείλεται στους thread overheads και στη δομή του taskgraph, η οποία εισάγει περιορισμένη παραλληλία λόγω των εξαρτήσεων ιδιαίτερα στο case 1, όπου οι σειριακές εκτελέσεις μειώνονται απλά σε 6 από τις 8 αρχικές.
- ✓ Η αύξηση των threads στους 64 οδηγεί σε αύξηση του χρόνου εκτέλεσης σε όλες τις περιπτώσεις, υποδηλώνοντας ότι το overhead του tasking υπερβαίνει το όφελος της παραλληλοποίησης.

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Νοέμβριος 2025