Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Συστήματα Παράλληλης Επεξεργασίας
9ο Εξάμηνο
Ακ. Έτος 2022-2023

Parlab16: Χρήστος Παπαδημητρίου (**03118017**), Μάρκος Ράμος(**03118841**), Αλέξανδρος Μοίρας(**03118081**)

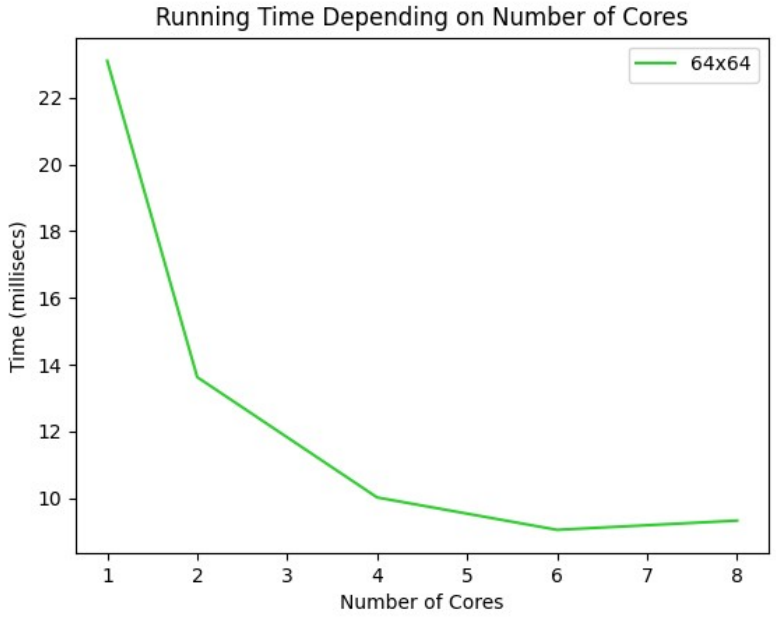
Άσκηση 1: Εισαγωγή στην Παραλληλοποίηση σε Αρχιτεκτονικές Κοινής Μνήμης

Αρχικά Παραθέτουμε τον Πίνακα με τις Μετρήσεις για τον Χρόνο (σε seconds) που έκανε να τρέξει το παραλληλοποιημένο πρόγραμμα για τα διαφορετικά πλήθη πυρήνων καθώς και για τα διαφορετικά μεγέθη ταμπλώ:

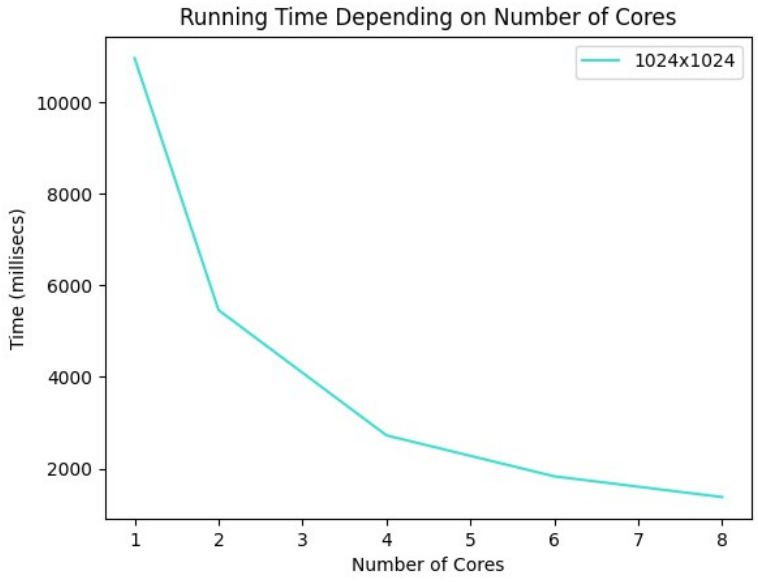
Πυρήνες \ Ταμπλώ	64x64	1024x1024	4096x4096
1	0.023103	10.964798	175.910355
2	0.013629	5.461097	95.425191
4	0.010025	2.725209	51.817218
6	0.009055	1.829950	36.341021
8	0.009332	1.376615	33.838737

Παρουσιάζουμε τα παραπάνω δεδομένα με μορφή διαγραμμάτων:

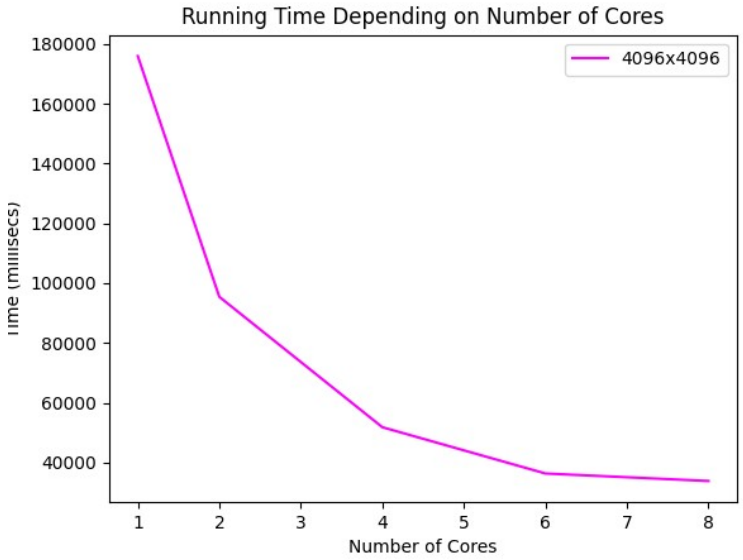
- Ο Χρόνος Εκτέλεσης του Προγράμματος για τα διάφορα πλήθη πυρήνων για grid 64x64



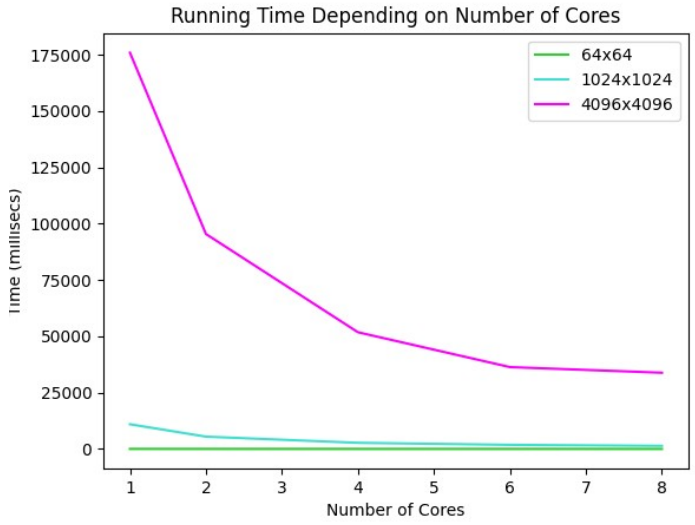
- Ο Χρόνος Εκτέλεσης του Προγράμματος για τα διάφορα πλήθη πυρήνων για grid 1024x1024



- Ο Χρόνος Εκτέλεσης του Προγράμματος για τα διάφορα πλήθη πυρήνων για grid 4096x4096



- Ο Χρόνος Εκτέλεσης του Προγράμματος για τα διάφορα πλήθη πυρήνων για τα διαφορετικά είδη grid σε κοινό διάγραμμα:

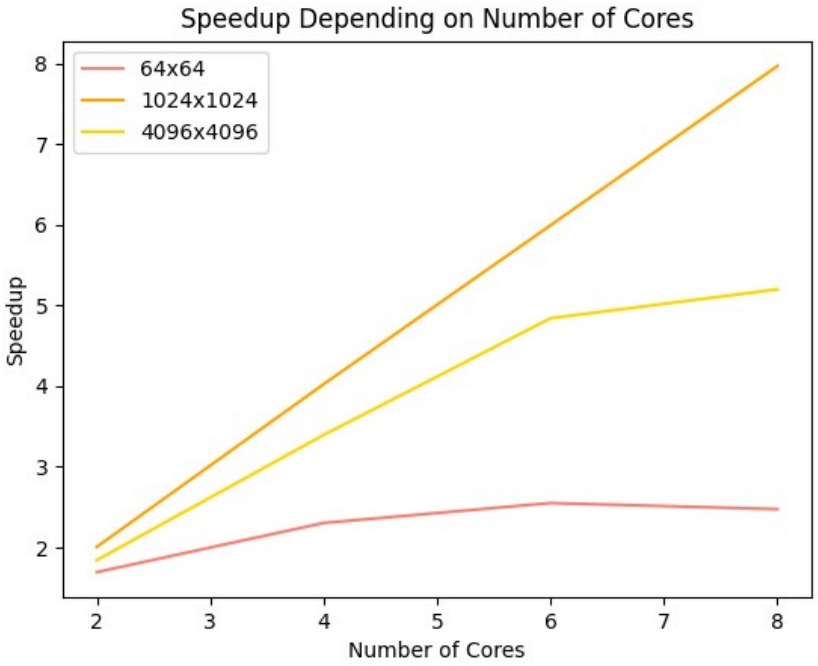


Παρατηρούμε ότι το σχήμα της γραφικής παράστασης μένει σταθερό, ανεξάρτητα από το μέγεθος του Grid. Σε όλες τις περιπτώσεις βλέπουμε μία εκθετικού τύπου πτώση του χρόνου καθώς αυξάνουμε το πλήθος των επεξεργαστών. Βέβαια, όσο μεγαλύτερο είναι το grid, τόσο μεγαλύτερη γίνεται η βελτίωση

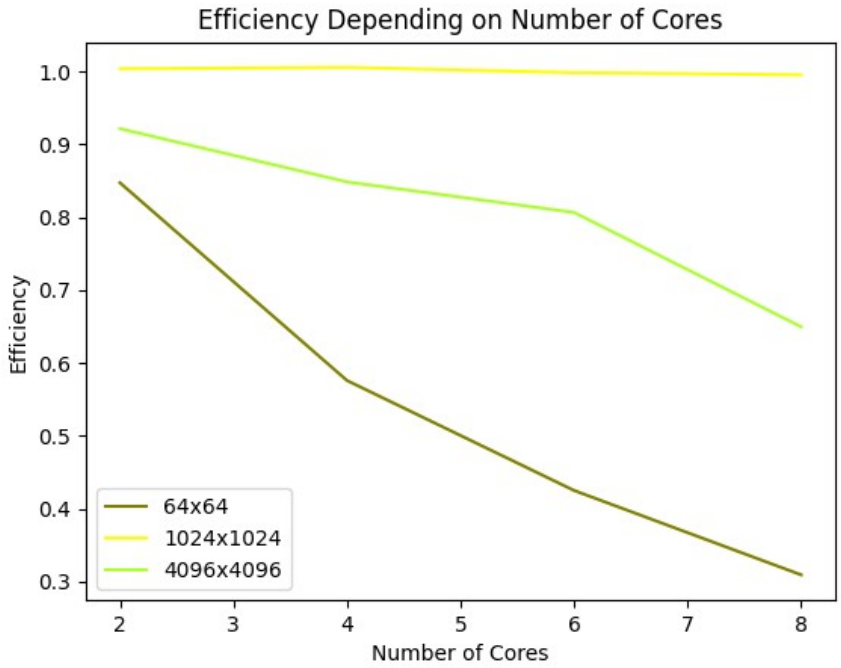
σε απόλυτα νούμερα (Έχουμε μεγαλύτερο εύρος στον κατακόρυφο άξονα). Η πτώση αυτή συναντά κορεσμό μεταξύ των 6 και των 8 πυρήνων, στην περίπτωση του 64x64. Όταν αυξάνουμε δηλαδή τους πυρήνες από 6 σε 8 δεν υπάρχει βελτίωση του χρόνου. Στις άλλες δύο περιπτώσεις η μείωση του χρόνου είναι μικρότερη για την αλλαγή από 6 σε 8 πυρήνες. Από αυτήν την άποψη, στις περισσότερες περιπτώσεις, δεν έχει νόημα να χρησιμοποιήσουμε περισσότερους από 6 πυρήνες για να παραλληλοποιήσουμε αυτό το πρόγραμμα.

Για καλύτερη εκτίμηση του κόστους της παραλληλοποίησης σε σχέση με την βελτίωση της επίδοσης παραθέτουμε παρακάτω και τα διαγράμματα Speedup και Efficiency:

- Speedup



- Efficiency



Με βάση τα τελευταία διαγράμματα γίνεται φανερό ότι η παραλληλοποίηση δεν έχει νόημα για μικρό μέγεθος grid. Πράγματι για το 64x64 grid το πρόγραμμα παρουσιάζει (σχεδόν) σταθερό speedup και ραγδαία φθίνον Efficiency. Αυτό ενδεχομένως συμβαίνει γιατί το overhead της επικοινωνίας είναι σημαντικό σε σχέση με το κέρδος από την παραλληλοποίηση των υπολογισμών για τόσο μικρό μέγεθος εισόδου.

Για μεγαλύτερα μεγέθη εισόδου η παραλληλοποίηση παρουσιάζει μεγαλύτερο ενδιαφέρον. Συγκεκριμένα, για το grid 1024x1024 οι μετρήσεις μας δίνουν ιδανική συμπεριφορά γραμμικού Speedup με Efficiency σταθερά ίσο με 1 για όλο το εύρος πλήθους πυρήνων που εξετάσαμε. Για το grid 4096x4096, έχουμε επίσης καλή συμπεριφορά με γραμμική αύξηση του Speedup και Efficiency περι του 0.9 μέχρι τους 6 πυρήνες, ενώ όταν πηγαίνουμε από τους 6 στους 8 πυρήνες ο ρυθμός βελτίωσης πέφτει (το Speedup αυξάνεται σχετικά λίγο και το Efficiency πέφτει σημαντικά). Η καλύτερη συμπεριφορά του παραλληλοποιημένου προγράμματος για είσοδο 1024x1024 σε σχέση με την συμπεριφορά για είσοδο 4096x4096 θα μπορούσε εξηγηθεί από την συμφόρηση εξαιτίας της μνήμης που προκαλεί η μεγαλύτερη ποσότητα δεδομένων της δεύτερης περίπτωσης.

Άσκηση 2: Παραλληλοποίηση και Βελτιστοποίηση Αλγορίθμων σε αρχιτεκτονικές Κοινής Μνήμης

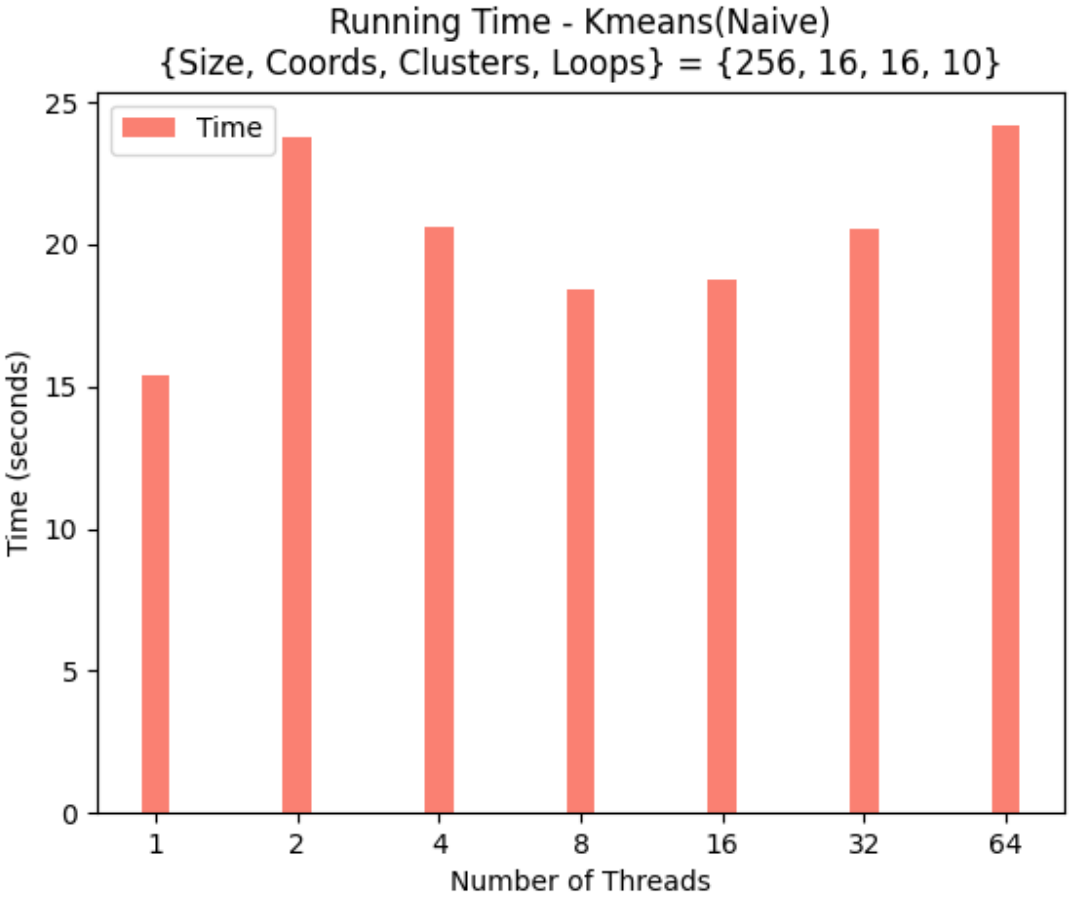
2.1 Παραλληλοποίηση και Βελτιστοποίηση του Αλγορίθμου K-means

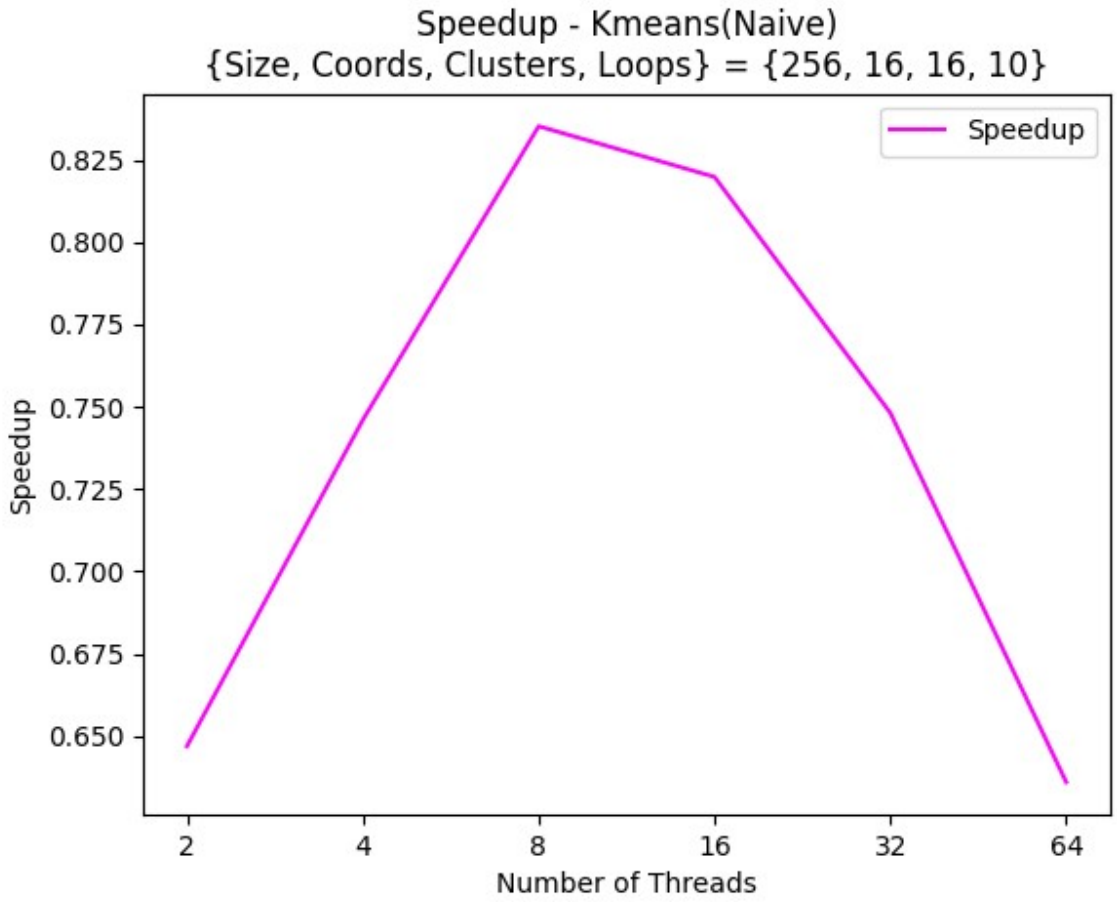
Μελετήσαμε την Παραλληλοποίηση σε δύο εκδόσεις του παραπάνω αλγορίθμου. Μία που παραλληλοποιεί με "αφελή" τρόπο τον σειριακό αλγόριθμο, εξασφαλίζοντας απλώς ότι τα κοινά δεδομένα διαμοιράζονται με ορθό τρόπο (με ατομικές προσβάσεις όταν πρόκειται να συμβεί εγγραφή). Η δεύτερη έκδοση χρησιμοποιεί copied δεδομένα (πίνακες) για κάθε νήμα ώστε να απαλλαγούμε από την ανάγκη συγχρονισμού κατά την εγγραφή. Τα copied δεδομένα συνδυάζονται (reduction) ανα διαστήματα στις global δομές.

Έκδοση 1: Shared Clusters

Ακολουθούν τα διαγράμματα χρόνου εκτέλεσης και Speedup για threads = {1, 2, 4, 8, 16, 32, 64}:

- Διάγραμμα Χρόνου Εκτέλεσης


Running Time - Kmeans(Naive)
{Size, Coords, Clusters, Loops} = {256, 16, 16, 10}
Time
Time (seconds)
0
5
10
15
20
25
1
2
4
8
16
32
64
Number of Threads

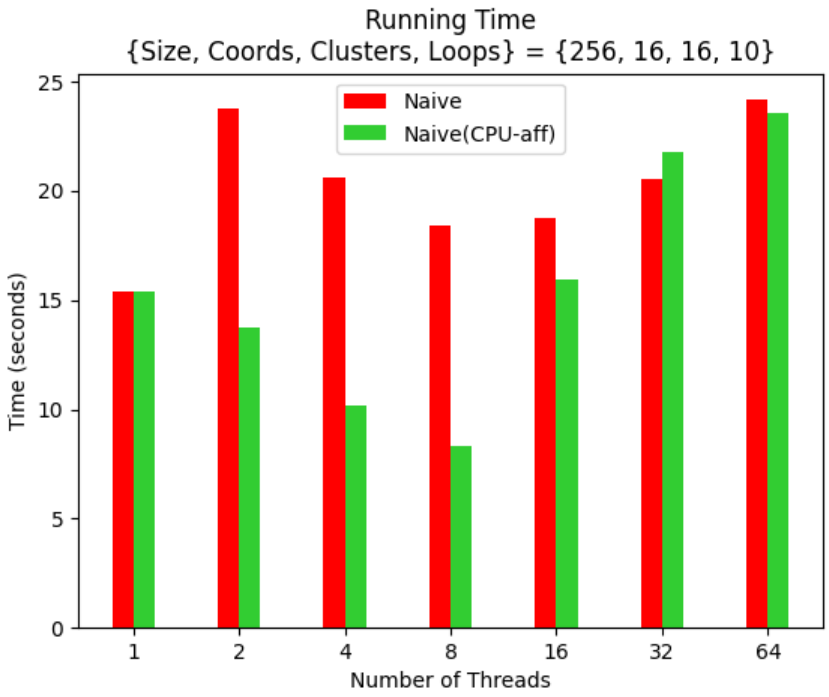


- Διάγραμμα Speedup

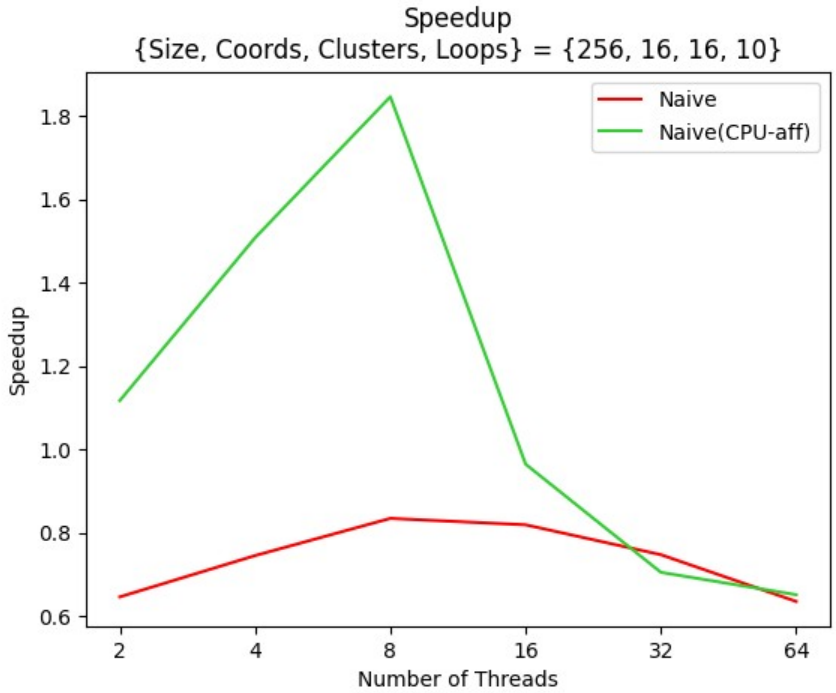
Είναι φανερό ότι με την συγκεκριμένη υλοποίηση, ο παραλληλισμός δεν βελτιώνει τις επιδόσεις. Πράγματι το Speedup είναι λιγότερο από 1 για όλα τα πλήθη threads με τα οποία πειραματιστήκαμε. Χρησιμοποιώντας ατομικές προσβάσεις για να πετύχουμε την ορθότητα κατά τον παραλληλισμό εισάγουμε τόσο μεγάλο βαθμό σειριοποίησης (τα writes πρέπει να γίνονται σειριακά-atomically), όσο και φόρτο στον δίαυλο λόγω cache invalidates κατά την εγγραφή σε κοινά δεδομένα (newClusterSize, new Clusters) από πολλά threads τα οποία τρέχουν σε πολλούς πυρήνες. Επίσης ένας άλλος λόγος για τον οποίο δεν αξιοποιούνται αποδοτικά οι caches είναι τα thread migrations. Συγκεκριμένα όταν μετά από ένα context switch ένα thread τοποθετείται σε άλλον επεξεργαστή από αυτόν στον οποίο εκτελούταν αρχικά, πιθανότατα θα χρειαστεί να ξαναδιαβάσει από χαμηλότερα στην ιεραρχεία μνήμης τις εισόδους του (κυρίως τα objects αλλα πιθανόν και το membership). Τα παραπάνω φαινόμενα προσθέτουν τόσο overhead στον αλγόριθμο ώστε το αποτέλεσμα να είναι χειρότερο από το σειριακό.

Μπορούμε να μειώσουμε αυτό το overhead επιλύωντας το τελευταίο από τα προβλήματα που αναφέρθηκαν, χρησιμοποιώντας την μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY για να προσδέσουμε κάθε thread (λογισμικού) σε έναν συγκεκριμένο πυρήνα Παρακάτω παρουσιάζουμε τα αποτελέσματα σε σύγκριση με αυτά που είδαμε στην προηγούμενη περίπτωση:

- Διάγραμμα Χρόνου Εκτέλεσης



- Διάγραμμα Speedup

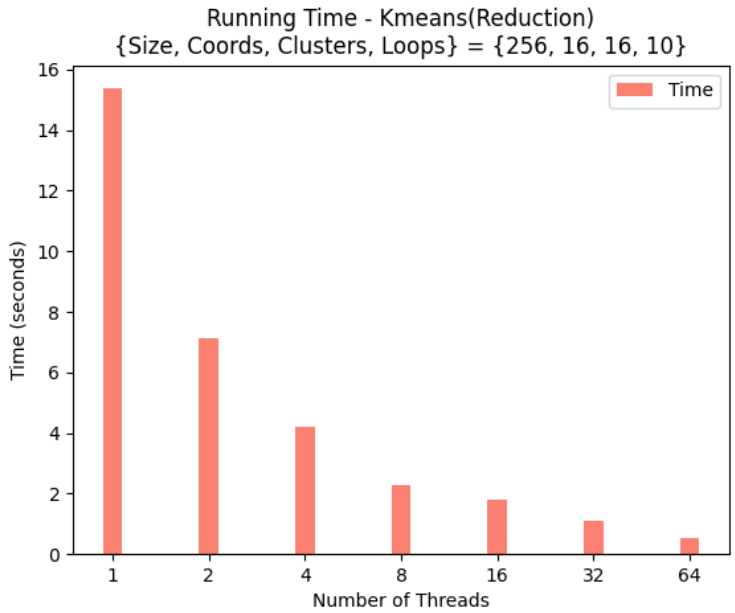


Μετά την πρόσδεση Threads σε πυρήνες παρατηρούμε ότι έχουμε βελτίωση της τις επίδοσης μέχρι και τα 8 threads και το καλύτερο Speedup που αγγίζουμε είναι περί του 1.8.

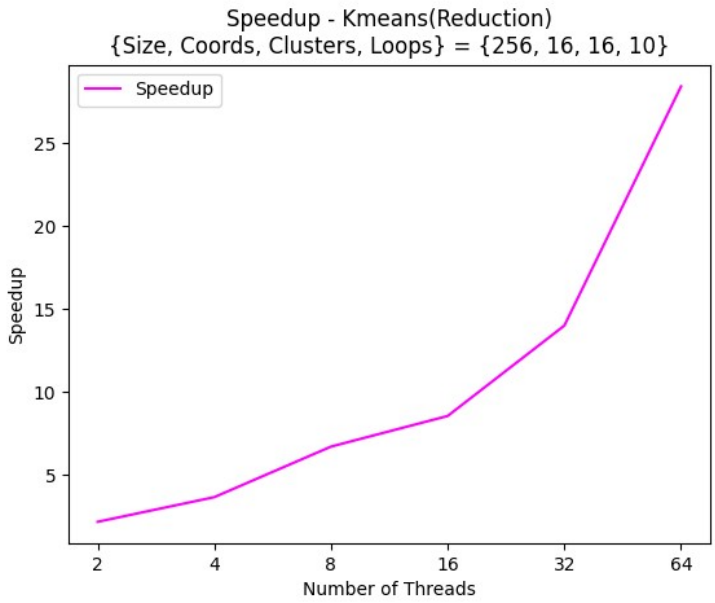
Έκδοση 2: Copied Clusters & Reduction

Παρουσιάζουμε και για αυτήν την έκδοση τα ίδια διαγράμματα όπως και πριν:

- Διάγραμμα χρόνου εκτέλεσης



- Διάγραμμα Speedup

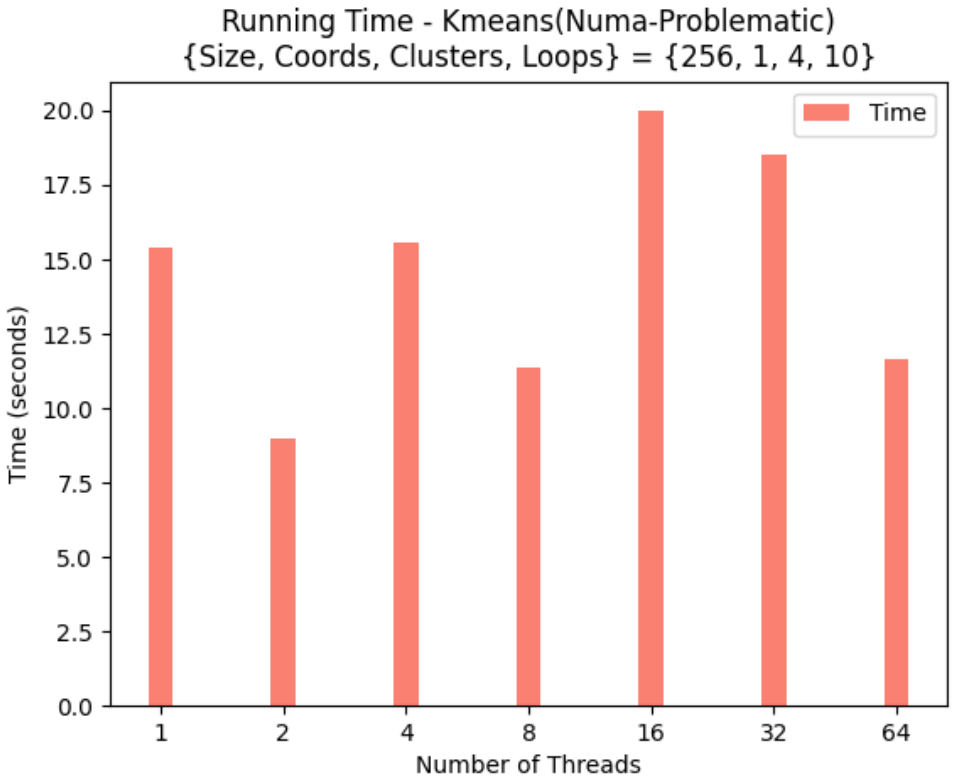


Είναι φανερό ότι η συμπεριφορά αυτής της έκδοσης είναι πολύ πιο ικανοποιητική. Η επίδοση βελτιωνεται σταθερά καθώς αυξάνουμε τα threads, για όλα τα πλήθη threads που δοκιμάζουμε. Μάλιστα ο ρυθμός βελτίωσης μένει (χοντρικά) σταθερός. Δηλαδή κάθε φορά που διπλασιάζουμε τα threads, ο χρόνος εκτέλεσης υφίσταται (χοντρικά) υποδιπλασιασμό.

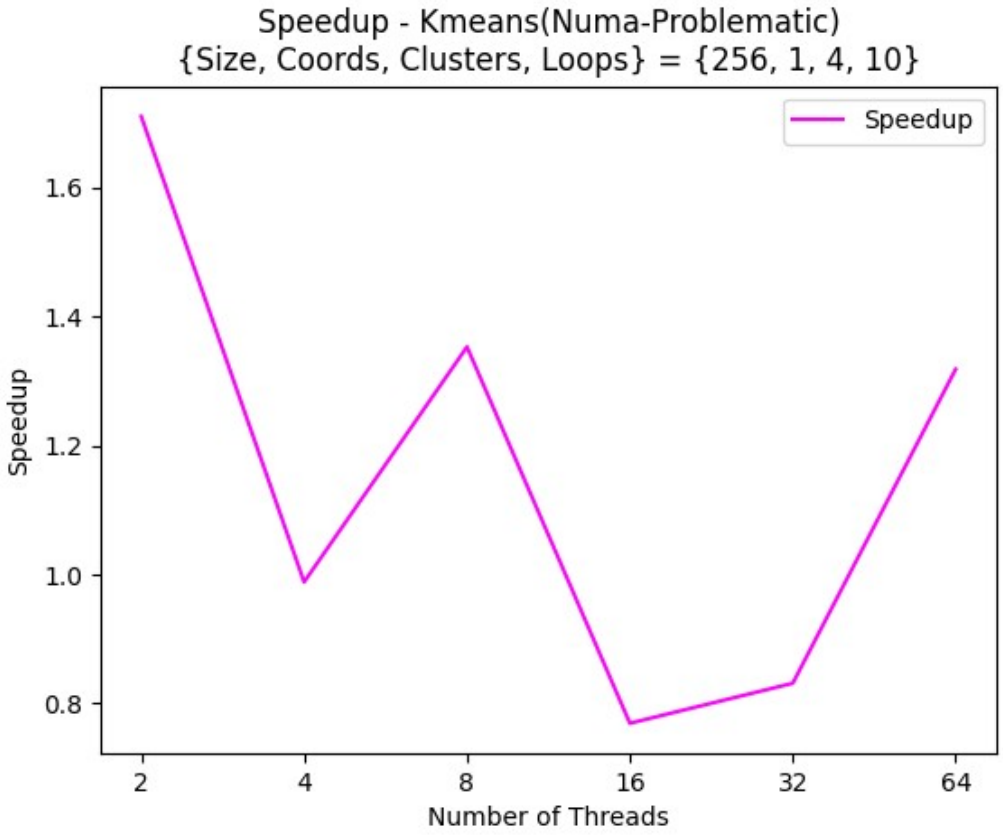
Συγκρίνοντας αυτήν την έκδοση του αλγορίθμου με την προηγούμενη φαίνεται ότι η χρήση ατομικών εντολών για λόγους συγχρονισμού έχει σημαντικό κόστος από άποψη χρόνου. Μάλιστα, τουλάχιστον στον συγκεκριμένο αλγόριθμο, ο χρόνος που προσθέτει η χρήση atomics είναι πολλαπλάσιος του ωφέλιμου χρόνου που χρειάζεται ο αλγόριθμος με αποτέλεσμα να υπάρχει τεράστια επίπτωση στην επίδοση αν τα χρησιμοποιήσουμε. Πλέον δεν έχουμε ούτε overheads λόγω του πρωτοκόλλου συνάφειας, ούτε σειριοποίηση παρα μόνο στό τέλος κάθε iteration όπου συσσωρεύουμε τα αποτελέσματα όλων των cores (reduction).

Δοκιμάζοντας τώρα το τελευταίο πρόγραμμα σε configuration: Size, Coords, Clusters, Loops} = {256, 1, 4, 10} η επίδοση πέφτει σημαντικά. Παραθέτουμε τα αποτελέσματα μας για αυτά τα πειράματα:

- Διάγραμμα Χρόνου Εκτέλεσης



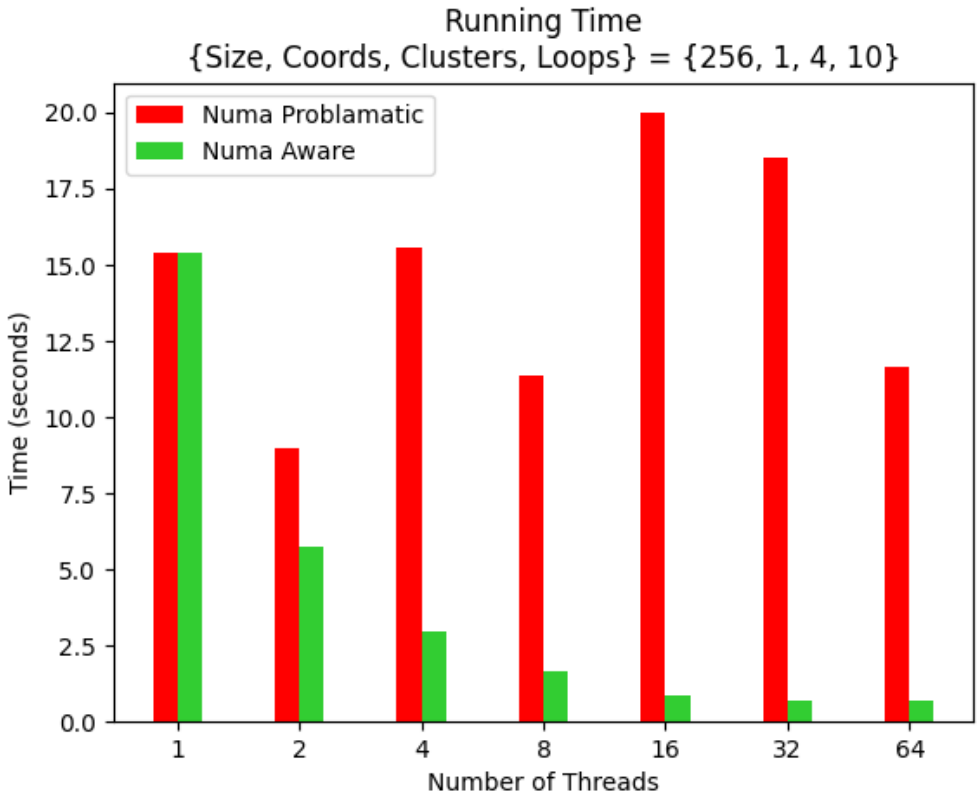
- Διάγραμμα Speedup



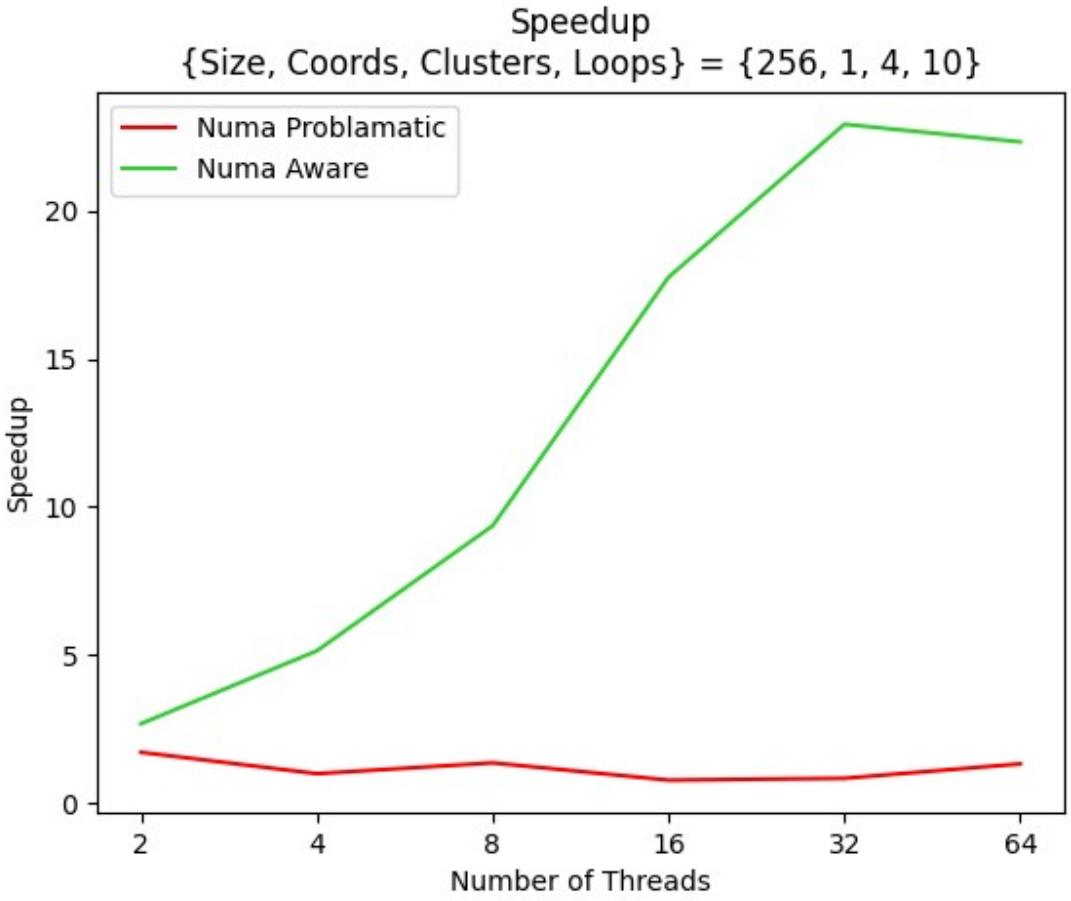
Όπως φαίνεται στα παραπάνω διαγράμματα έχουμε επιτάχυνση κατά τον παραλληλισμό μόνο για τις περιπτώσεις των 2, των 8 και των 64 threads, ενώ η καλύτερη περίπτωση είναι αυτή των 2 Threads (Speedup περί του 1.7). Είναι λοιπόν φανερό ότι χάνεται η κλιμακωσιμότητα που παρατηρήσαμε στο προηγούμενο Configuration. Η χειροτέρευση της συμπεριφοράς για το "μικρότερο" configuration οφείλεται στο φαινόμενο του False Sharing. Συγκεκριμένα, το Thread που κάνει Initialize τους (copied) πίνακες local_newClusters, τους κάνει allocate στην κοντινότερη σε αυτό cache. Αυτό συμβαίνει σύμφωνα με την First-Touch στρατηγική του Linux. Αυτό από μόνο του καθιστά τους πίνακες αυτούς απομακρυσμένους για τα Threads που θα τους χρησιμοποιήσουν ως δικούς τους τοπικούς πίνακες κατά την διάρκεια του αλγορίθμου. Επιπλέον, δεδομένου ότι σε αυτό το Configuration έχουμε μικρούς πίνακες local_newClusters υπάρχει περίπτωση δύο διαφορετικοί πίνακες να συμπίπτουν στο ίδιο block της μνήμης. Αυτό έχει ως αποτέλεσμα το block αυτό να γίνεται συχνά Invalidated στην Cache, καθώς διαφορετικά Threads προσπαθούν να γράψουν τον προσωπικό τους πίνακα local_newClusters, ενώ στην πραγματικότητα τα Threads δεν ενδιαφέρονται για τα ίδια δεδομένα (False Sharing). Τα δύο παραπάνω φαινόμενα που προσθέτουν επιπλέον overhead για τον παραλληλοποιημένο αλγόριθμο και δεν επιτρέπουν την κλιμάκωσή του.

Για να περιορίσουμε αυτά τα φαινόμενα κάνουμε την εξής αλλαγή στον κώδικα: Παραλληλοποιούμε τον βρόχο στον οποίο γίνονται Initilize οι πίνακες local_newClusters ώστε να εξασφαλίσουμε ότι κάθε Thread θα κάνει Initilize των πίνακα τον οποίο και θα χρησιμοποιεί. Έτσι φέρνουμε τους πίνακες πιο κοντά στα Threads που τους χρησιμοποιούν και συγχρόνως περιορίζουμε το False-Sharing. Ακολουθούν τα αποτελέσματα μας μετά την προαναφερθείσα τροποποίηση. Μάλιστα τα παραθέτουμε μαζί με τα προηγούμενα αποτελέσματα για το ίδιο configuration για λόγους σύγκρισης:

- Το διάγραμμα Χρόνου Εκτέλεσης



- Το διάγραμμα Speedup

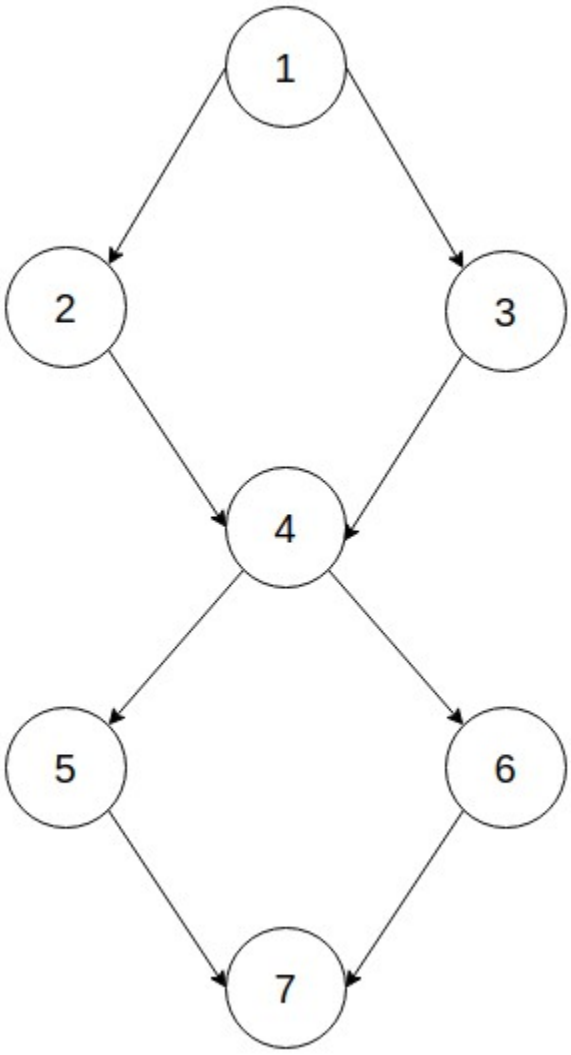


Μετά την τροποποίηση βλέπουμε ότι η συμπεριφορά του αλγορίθμου βελτιώνεται θεαματικά. Η εικόνα των διαγραμμάτων μοιάζει πλέον πολύ με αυτή που είδαμε στο προηγούμενο (και "μεγαλύτερο") configuration ({Size, Coords, Clusters, Loops} = {256, 16, 16, 10}) με την διαφορά ότι εδώ η κλιμάκωση σταματά στα 32 Threads. Ο Καλύτερος χρόνος που πετύχαμε για το configuration {Size, Coords, Clusters, Loops} = {256, 1, 4,10} είναι 0.6702 sec (για 32 Threads).

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Έκδοση 1: Task parallelism of recursive version

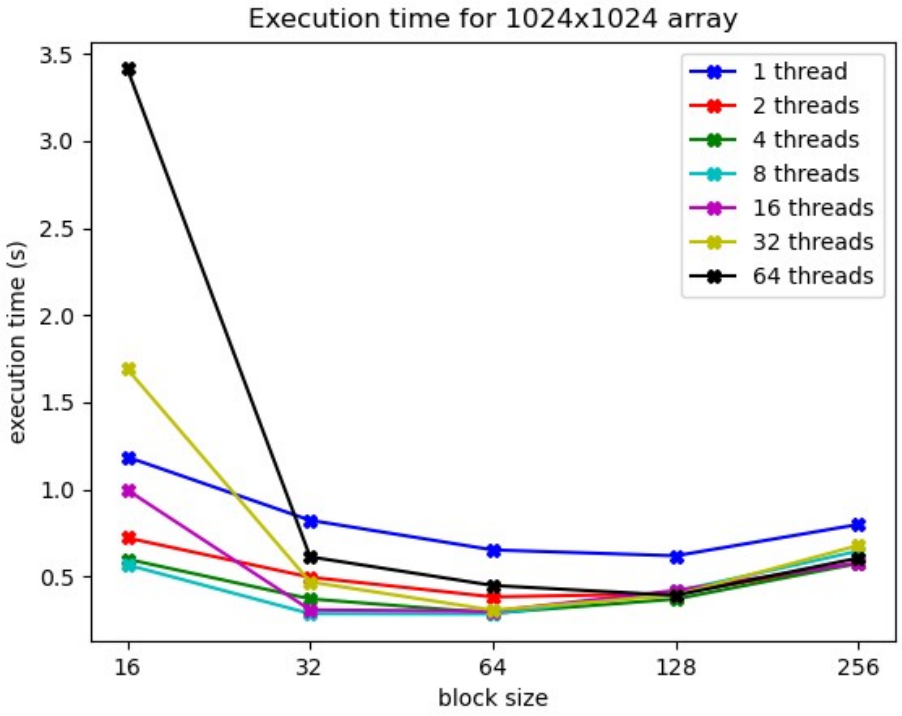
Κατασκευάζοντας τον γράφο των εξαρτήσεων της αναδρομικής έκδοσης του αλγορίθμου FW παρατηρούμε ότι υπάρχει εκμεταλλεύσιμη παραλληλία:

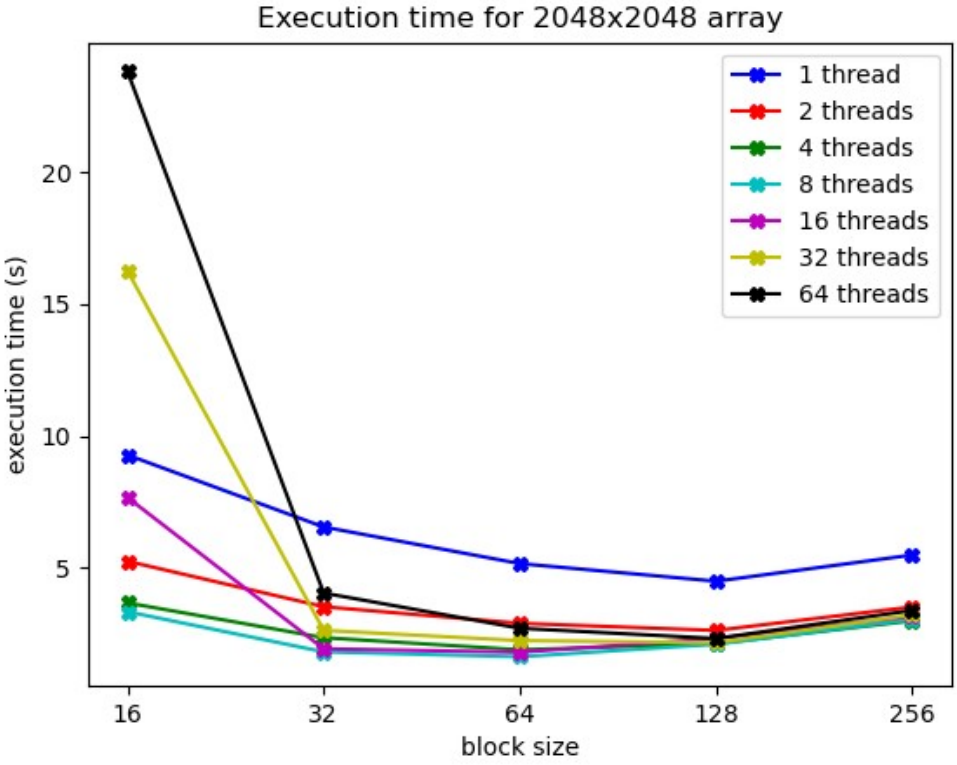


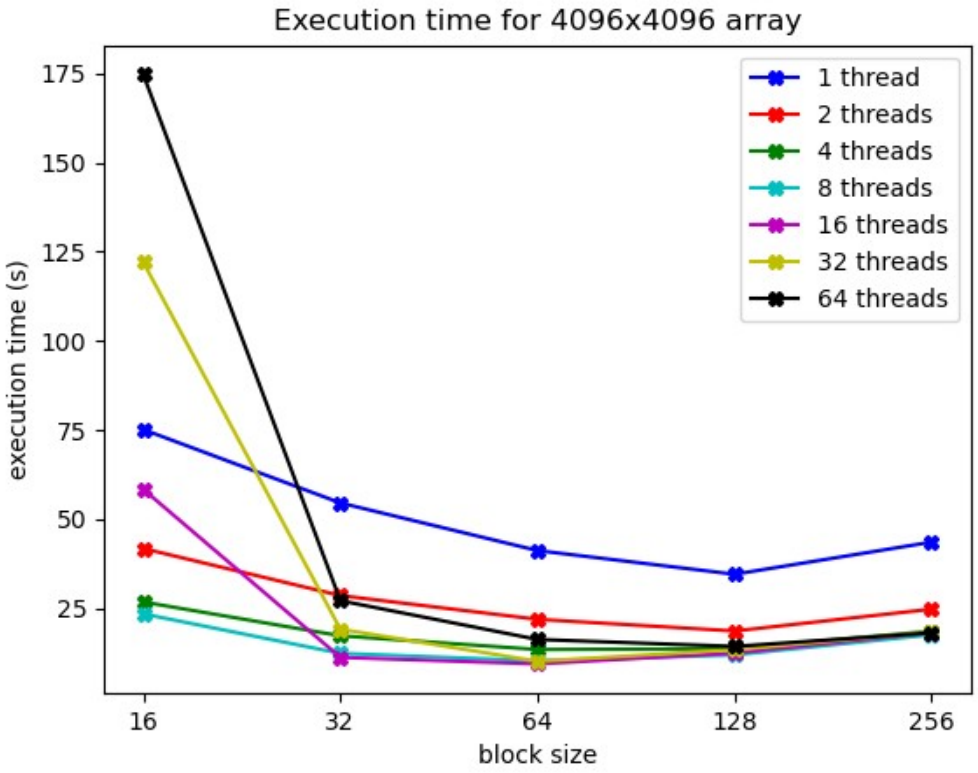
Αριθμούμε τα jobs που εμφανίζονται στον γράφο εξαρτήσεων προκειμένου να διευκολύνουμε την εξήγηση παρακάτω. Είναι φανερό ότι υπάρχει παραλληλία μεταξύ των 2, 3 και 5, 6. Προκειμένου να την εκμεταλλευτούμε αναθετούμε αυτά τα jobs σε 2 tasks (το ένα θα είναι deferred και θα εκτελεστεί από το thread το οποίο θα εκτελέσει και τα σειριακά jobs 1, 2, 4 του συγκεκριμένου βήματος της αναδρομής). Επίσης φροντίζουμε ώστε να εντάξουμε σε parallel region την κλήση στην αναδρομική συνάρτηση και όχι το body της προκειμένου να αποφύγουμε το φαινόμενο όπου το openMP δεν θα αναθέσει σε επιπλέον threads να εκτελέσουν τα tasks που δημιουργούνται από nested parallel regions όσο θα προχωρά η αναδρομή (Εναλλακτικά μπορούμε να θέσουμε σε true τη μεταβλητη περιβάλλοντος OMP_NESTED). Καθώς η συνάρτηση είναι αναδρομική και σε κάθε κλήση της θα δημιουργούνται και άλλα tasks (έως ότου φτάσουμε στο κρίσιμο block size όπου η εκτέλεση θα γίνει σειριακά) βάσει του παραπάνω dependency graph περιμένουμε και ο κάθε κόμβος του γραφήματος να εκτελείται γρηγορότερα από τη σειριακή έκδοση και επομένως speedup μεγαλύτερο του 1.25 που φαίνεται από τον αρχικό γράφο. Εξετάστηκαν τα παρακάτω configurations:
Array size: 1024x1024, 2048x2048, 4096x4096

Block size: 16, 32, 64, 128, 256
Number of threads: 1, 2, 4, 8, 16, 32

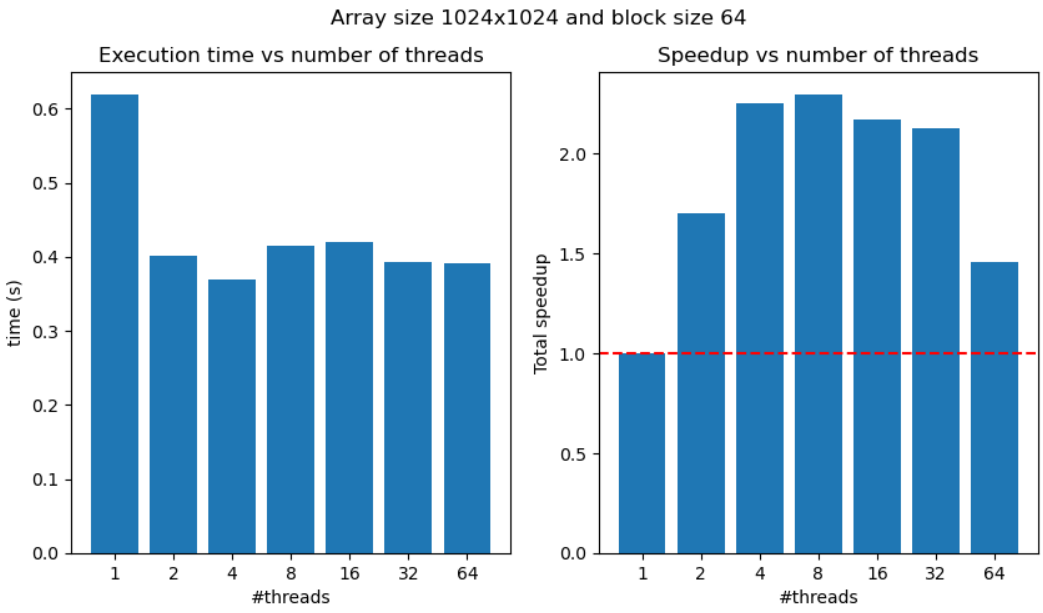
Αρχικά παρουσιάζουμε τα διαγράμματα του χρόνου εκτέλεσης σε σχέση με τα διάφορα block sizes για όλα τα μεγέθη πίνακα και όλους τους δυνατούς αριθμούς threads προκειμένου να αξιολογήσουμε ποιο είναι το καλύτερο block size σε κάθε περίπτωση.

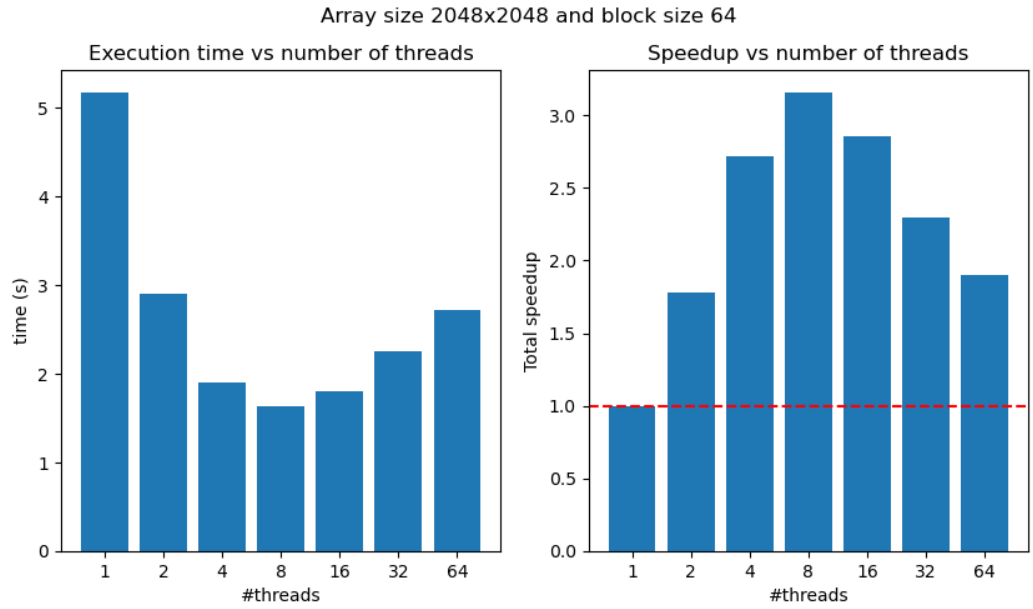


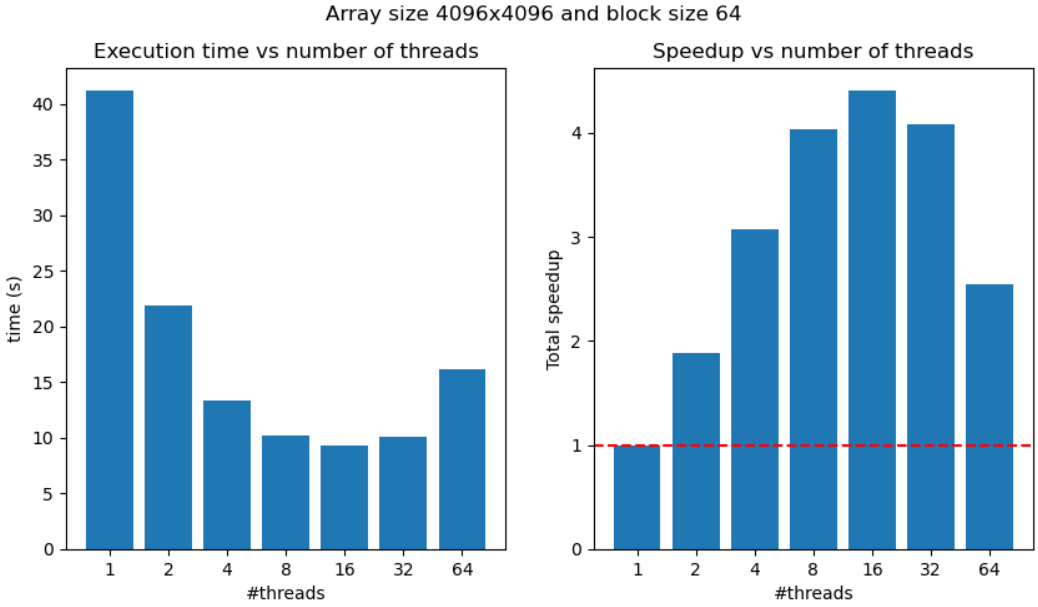

Execution time for 2048x2048 array
1 thread
2 threads
4 threads
8 threads
16 threads
32 threads
64 threads
execution time (s)
20
15
10
5
16
32
64
128
256
block size



Είναι αρχικά εμφανές ότι η ταυτόχρονη χρήση πολλών threads με μικρό block size δυσχεραίνει σημαντικά την επίδοση. Αυτό συμβαίνει διότι ο βαθμός παραλληλίάς και τα overheads κατασκευής των tasks, ανάληψής τους από threads και συγχρονισμού δεν μπορούν να δικαιολογηθούν για τόσο μικρές εργασίες όπως η εκτέλεση του FW για ένα τόσο μικρό block. Ένας άλλος σημαντικός λόγος της μείωσης της επίδοσης με πολλά threads και μικρό μέγεθος block είναι ότι η εκτέλεση δεν επωφελείται καθόλου από το locality καθώς όλα τα threads θα έχουν cache misses αρχικά ώστε να φέρουν το block στο οποίο θα εργαστούν, δημιουργώντας και συμφόρηση στον διάδρομο της μνήμης, ενώ ένα μεγαλύτερο block πιθανόν θα χωρούσε ολόκληρο στη μνήμη ή αν χρησιμοποιούσαμε λιγότερα threads θα είχαμε λιγότερα ταυτόχρονα misses, σε κάθε περίπτωση αποσυμφορώντας το σύστημα μνήμης.
Από τα παραπάνω διαγράμματα είναι εμφανές πως σε κάθε περίπτωση τα βέλτιστα block sizes είναι 32 και 64. Με μικρό margin καλύτερη επίδοση και στα 3 διαφορετικά μεγέθη ταμπλό επιτυγχάνει το block size = 64. Για μικρότερο αριθμό threads (έως 8) καλύτερο φαίνεται να είναι το block size = 128 πιθανότατα επειδή τα threads δεν επαρκούν για να αναλάβουν τα tasks που δημιουργούνται για μικρότερο block size. Ανεβαίνοντας σε αριθμό threads δεν φαίνεται να μπορούμε να επωφεληθούμε από περαιτέρω μείωση του block size (υπό του 64) για τους λόγους που εξηγήθηκαν παραπάνω. Κρατώντας ως sweet spot λοιπόν το block size = 64 θα προχωρήσουμε σε σύγκριση του χρόνου εκτέλεσης και του speedup βάσει των threads για συγκεκριμένο block size και για όλα τα μεγέθη πίνακα.

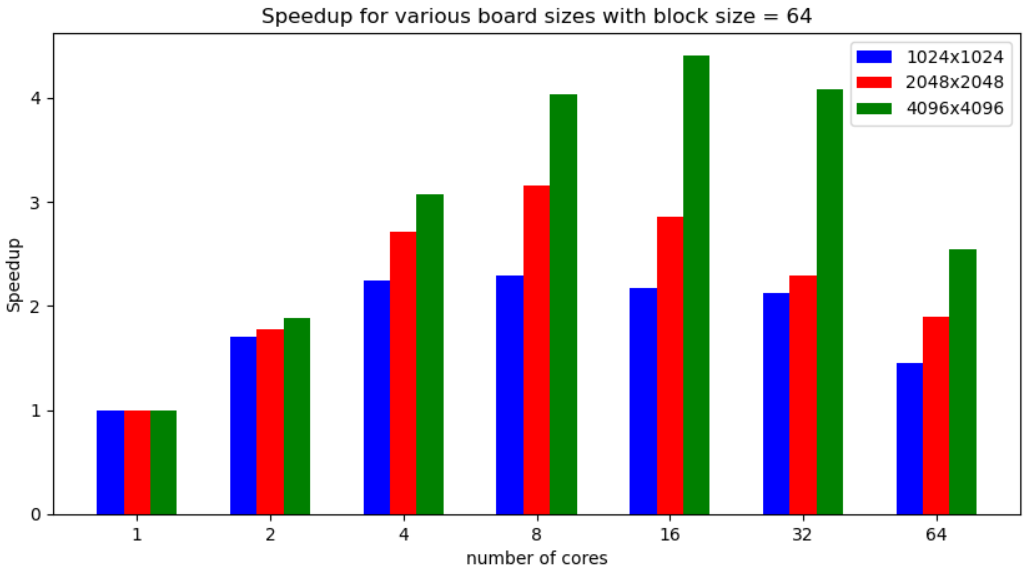

Array size 1024x1024 and block size 64
Execution time vs number of threads
time (s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
2
4
8
16
32
64
#threads
Speedup vs number of threads
Total speedup
0.0
0.5
1.0
1.5
2.0
1
2
4
8
16
32
64
#threads





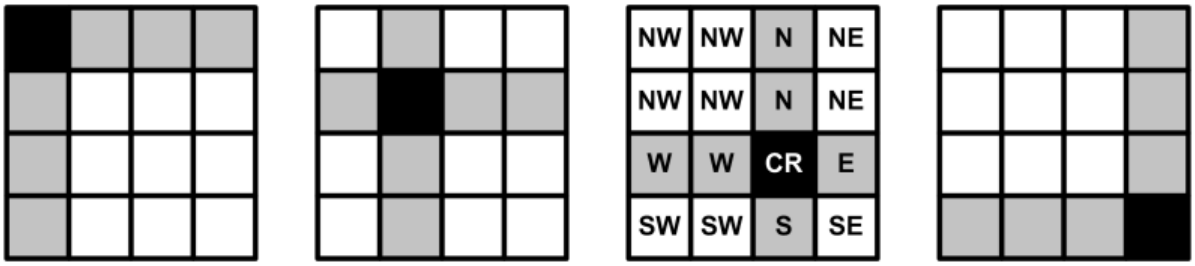
Παρατηρούμε ότι για όλα τα μεγέθη ταμπλό υπάρχει πλαφόν στον αριθμό των threads για τον οποίο συνεχίζουμε να βλέπουμε βελτίωση στην επίδοση. Συγκεκριμένα για τα μεγέθη 1024x1024 και 2048x2048 ο βέλτιστος αριθμός threads είναι 8 και για το μεγαλύτερο ταμπλό 4096x4096 είναι 16. Επίσης η κλιμάκωση δεν είναι ιδανική (ήδη από τα δύο threads έχουμε speedup μικρότερο του 2 η αύξηση του οποίου χειροτερεύει όσο ανεβάζουμε τα threads). Αυτό συμβαίνει τόσο γιατί υπάρχει σειριακό κομμάτι (Amdahl's law) αλλά και γιατί τα tasks δημιουργούν overhead και δεν επωφελούνται από πιθανόν υπάρχον locality. Συνολικά το καλύτερο speedup που πετυχαίνουμε είναι 4.4 για 4096x4096 array, 16 threads, block size = 64, με χρόνο εκτέλεσης 9.3534s.
Ενδιαφέρον επίσης παρουσιάζει το γεγονός ότι όσο μεγαλώνει το μέγεθος του ταμπλό μπορούμε να

πετύχουμε μεγαλύτερο speedup όπως φαίνεται και στο παρακάτω διάγραμμα.



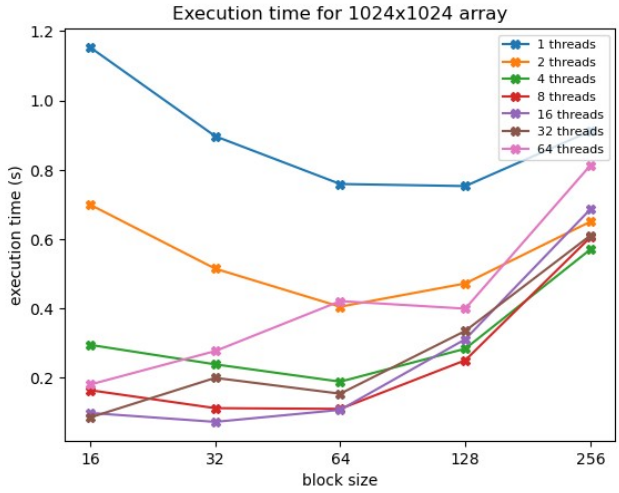
Πρόκειται για σαφή υπόδειξη ότι για τα μικρότερα μεγέθη ταμπλό το τμήμα που δύναται να παραλληλοποιηθεί δεν είναι αρκετά μεγάλο σε σύγκριση με το σειριακό τμήμα. Ωστόσο βρισκόμαστε σε αδιέξοδο καθώς προσπάθεια αύξησης του παράλληλου τμήματος με μείωση του block size οδηγεί σε χειρότερο speed up λόγω ποικίλων overheads και συμφόρηση μνήμης όπως αναφέρθηκε και παραπάνω. Είναι εμφανές ότι αυτή η έκδοση του αλγορίθμου δεν είναι η βέλτιστη προς παραλληλοποίηση και πιθανόν μπορούμε να εκμεταλλευτούμε κάποια άλλη ώστε να πετύχουμε μεγαλύτερο και πιο ομοιόμορφο για τις διάφορες περιπτώσεις εκτέλεσης του αλγορίθμου speedup.

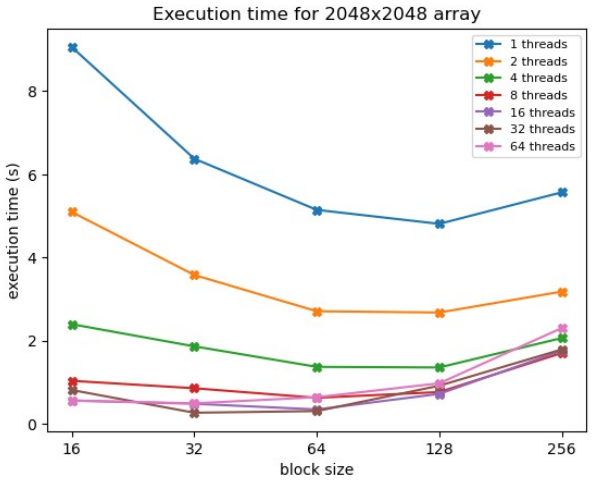
Έκδοση 2: Task parallelism of tiled version

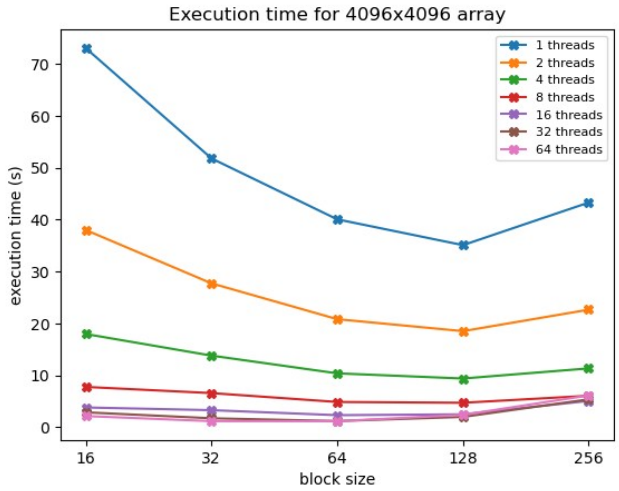


Για την εκτέλεση του αλγόριθμου FW με την tiled version χωρίζουμε τον πίνακα σε τετράγωνα μεγέθους BxB. Έπειτα, κινούμενοι στα τετράγωνα της κυρίας διαγωνίου, από πάνω αριστερά προς τα κάτω δεξιά, δημιουργούμε έναν σταυρό κάθε φορά με κέντρο το τετράγωνο της διαγωνίου και ακμές τις τέσσερις κατευθύνσεις: πάνω, κάτω, δεξιά, αριστερά. Υπολογίζουμε πρώτα το κεντρικό κουτάκι του σταυρού. Έπειτα, πάνω στις 4 ακμές του σταυρού υπολογίζουμε τις τιμές των τετραγώνων του σταυρού. Περιμένουμε να τερματίσουν όλα τα threads και τέλος, υπολογίζουμε όλα τα κουτάκια στα 4 τεταρτημόρια που διαχωρίζονται από τον σταυρό. Αυτή η υλοποίηση μας επιτρέπει να παραλληλοποιήσουμε τον υπολογισμό των τετραγώνων που βρίσκονται πάνω στις ακμές του σταυρού καθώς και των τεσσάρων τεταρτημορίων. Για βελτιστοποίηση των parallel for loops χρησιμοποιούμε το keyword schedule(dynamic), ώστε να κάθε ελεύθερο task να μπαίνει σε ένα queue από το οποίο παίρνουν και εκτελούν όλα τα ελεύθερα threads tasks. Αυτή η βελτιστοποίηση βοηθάει για μικρό αριθμό threads σε

σχέση με τον αριθμό gridsize/blocksize (N/B).
Όπως και στο προηγούμενο ερώτημα, οι συνδυασμοί που εξετάστηκαν ήταν οι εξείς:
Array size: 1024x1024, 2048x2048, 4096x4096
Block size: 16, 32, 64, 128, 256
Number of threads: 1, 2, 4, 8, 16, 32
Αρχικά παρουσιάζουμε τα διαγράμματα του χρόνου εκτέλεσης σε σχέση με τα διάφορα block sizes για όλα τα μεγέθη πίνακα και όλους τους δυνατούς αριθμούς threads προκειμένου να αξιολογήσουμε ποιο είναι το καλύτερο block size σε κάθε περίπτωση.

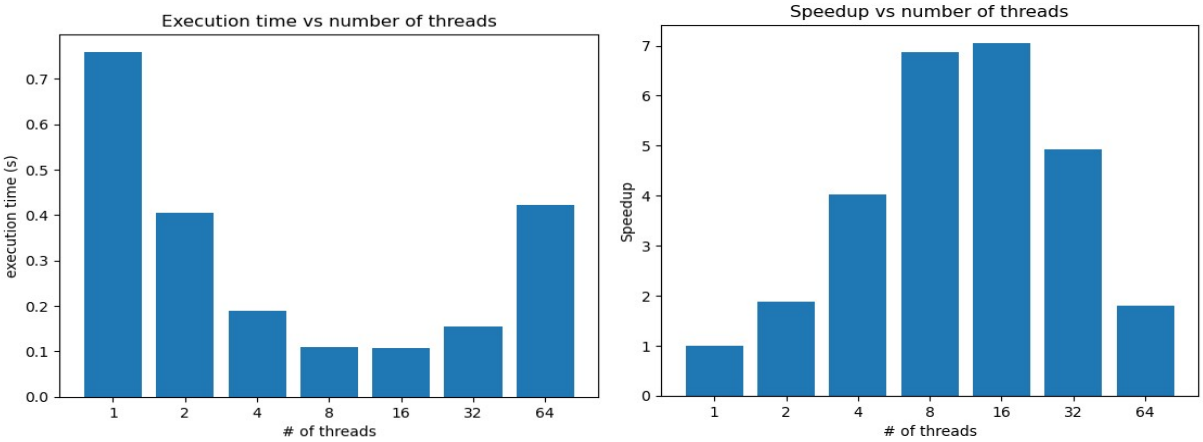




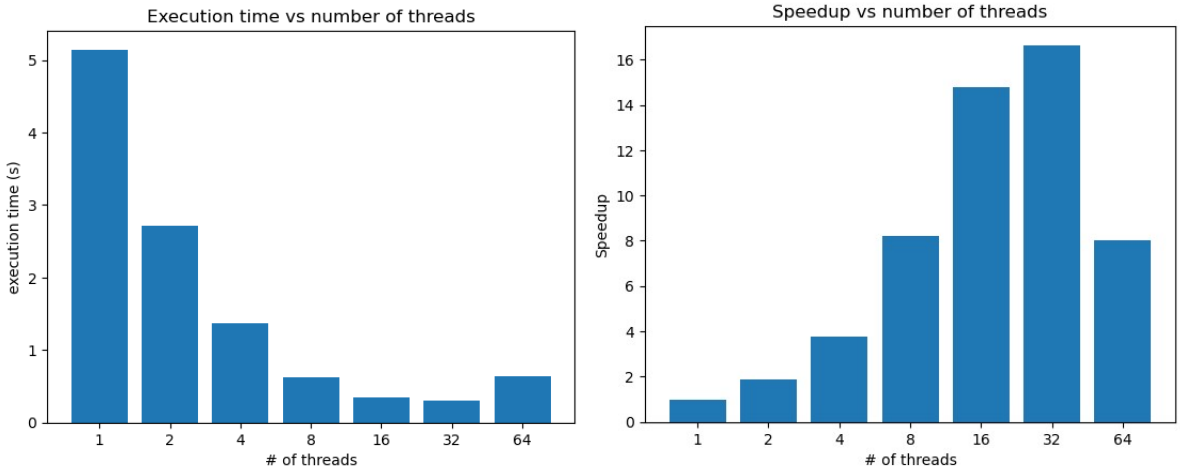


Είναι και πάλι εμφανές ότι η ταυτόχρονη χρήση πολλών threads με μικρό block size δυσχεραίνει σημαντικά την επίδοση. Και σε αυτή την περίπτωση θα χρησιμοποιήσουμε τα αποτελέσματα των Block size = 64 καθώς επωφελούνται τα περισσότερα threads από αυτό.

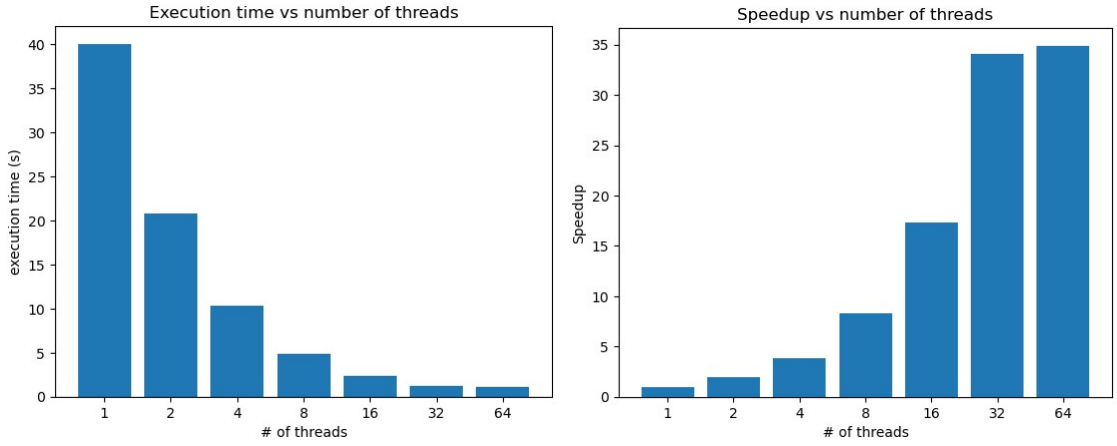
Array size 1024x1024 block size 64



Array size 2048x2048 block size 64



Array size 4096x4096 block size 64

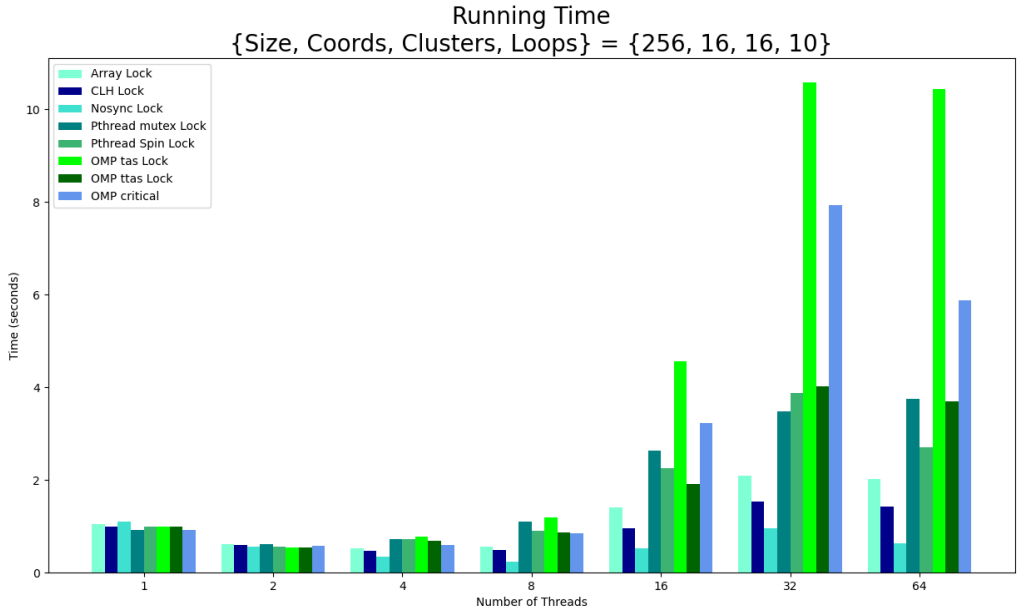


Παρατηρούμε πολύ μεγαλύτερο speedup σε σχέση με τον αναδρομικό FW (της τάξης του 5 για μικρές τιμές πίνακα και μέχρι 35 για μεγαλύτερες). Αυτό οφείλεται στο γεγονός ότι παραλληλοποιούμε μεγαλύτερο μέρος των διεργασιών που πρέπει να υπολογίσουμε. Πριν παραλληλοποιούσαμε μονο τις δύο διαγώνιες διαδικασίες ενώ τώρα μπορούμε να παραλληλοποιήσουμε όλο τον σταυρό και κάθε τεταρτημόριο.

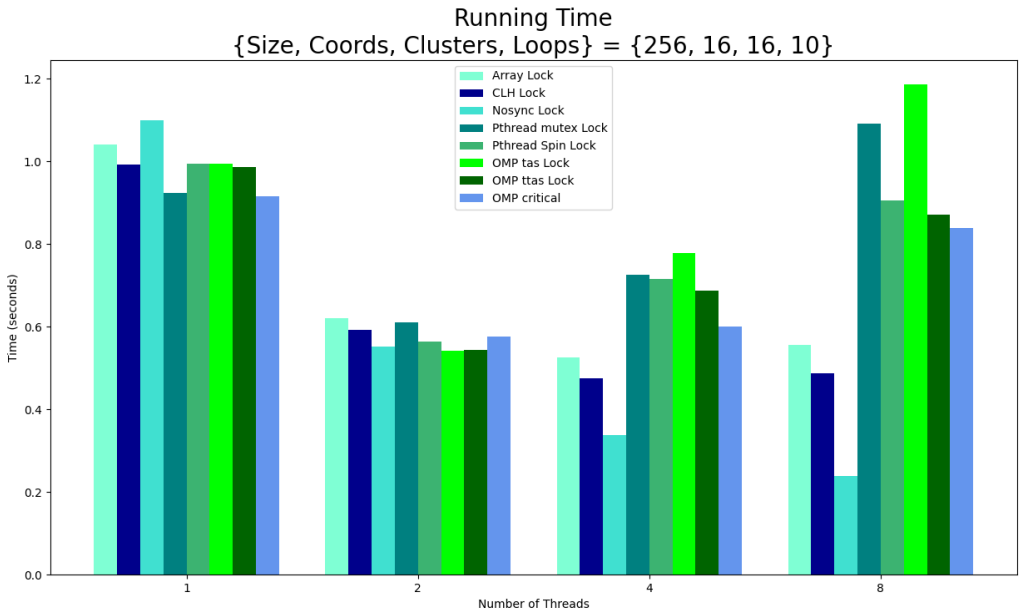
Σε μικρότερους πίνακες παρατηρούμε μια μείωση για τον μεγαλύτερο αριθμό των threads. Αυτό συμβαίνει τόσο γιατί υπάρχει σειριακό κομμάτι (Amdahl's law) αλλά και γιατί τα tasks δημιουργούν overhead και δεν επωφελούνται από πιθανόν υπάρχον locality. Σε αυτό οφείλεται και η περίεργη συνάρτηση των διεργασιών με περισσότερα threads (16 και άνω) οι οποίες καταναλώνουν μεγαλύτερο χρόνο
Μέγιστο speedup είχαμε για 64 threads στον πίνακα 4096x4096 με χρόνο 1.148 seconds.

Αμοιβαίος Αποκλεισμός - Κλειδώματα στον K-means:

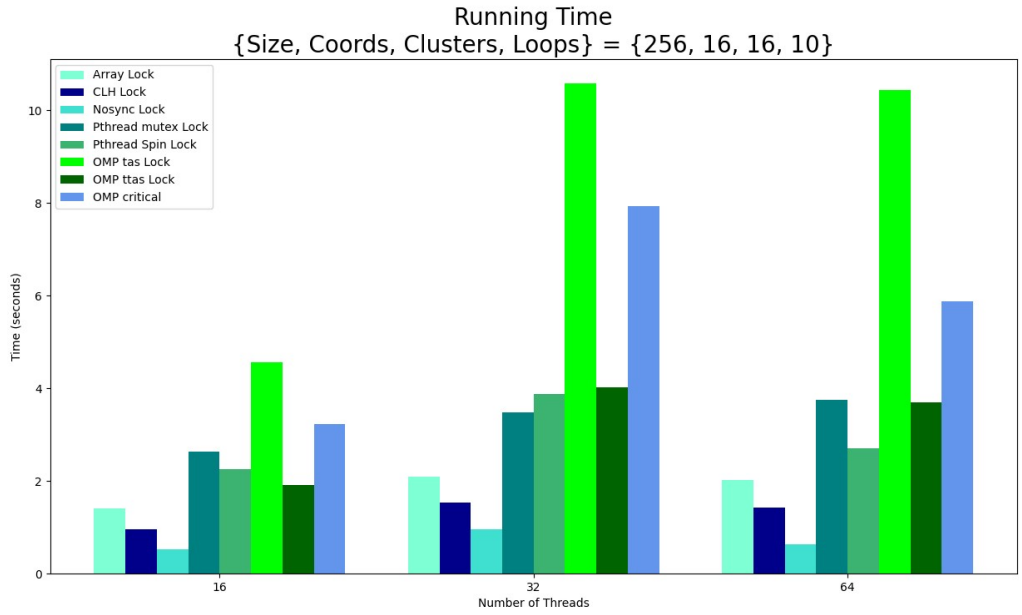
- Χρόνος εκτέλεσης για όλες τις περιπτώσεις πλήθους νημάτων:



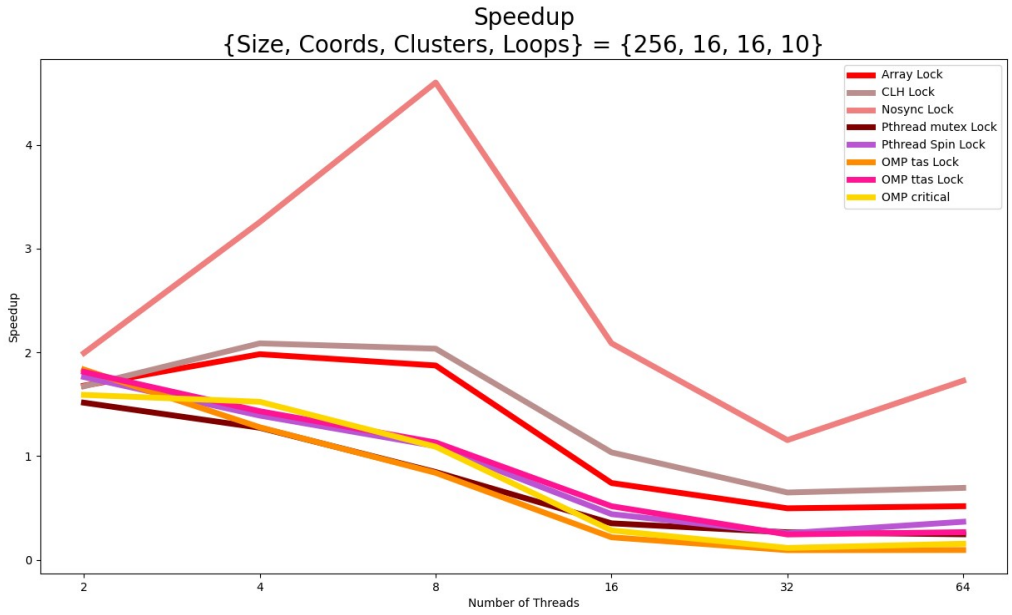
- Χρόνος εκτέλεσης για threads={1,2,4,8}:



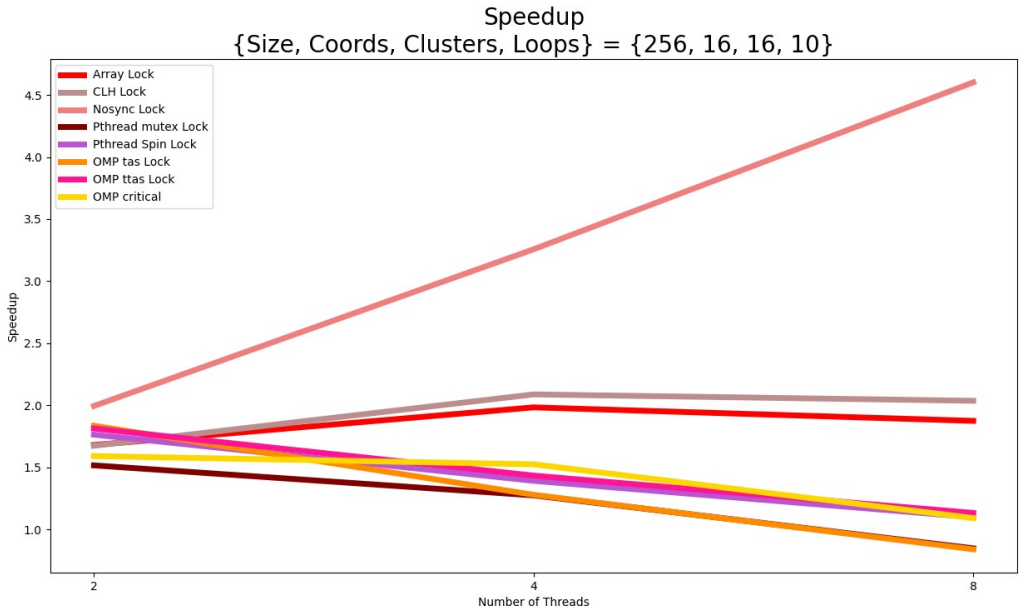
- Χρόνος εκτέλεσης για τις threads = {16,32,64}:



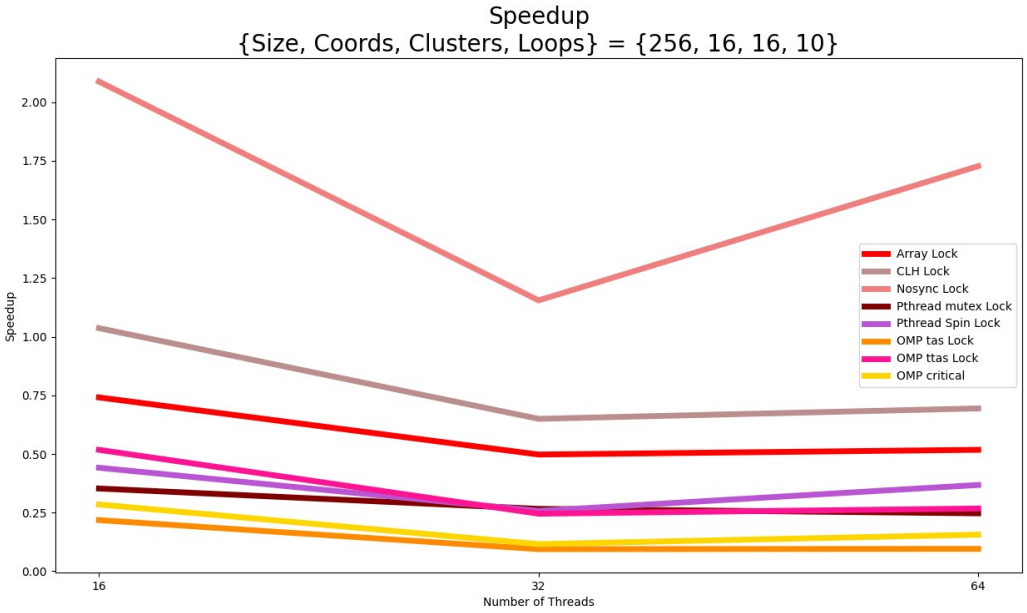
- Speedup για όλες τις περιπτώσεις πλήθους νημάτων:



- Speedup για threads = {2,4,8}



- Speedup για threads={16, 32, 64}



Αρχικά, παρατηρούμε πως η επίδοση του **nosync_lock** είναι σαφώς καλύτερη των άλλων. Λο
Add Headings (Format > Paragraph styles) and they will appear in your table of contents.
γικό, καθώς η έλλειψη συγχρονισμού στο lock επιτρέπει την αύξηση της ταχύτητας των εφαρμογών εις βάρος, όμως, της ορθότητας των αποτελεσμάτων.

Έπειτα, το **Array Lock** δημιουργεί μία λίστα μεγέθους ίσου με τον αριθμό των threads. Έχουμε έναν counter στο τέλος ο οποίος αναδεικνύει το τέλος της λίστας και αρχικά έχει την τιμή μηδέν. Επίσης, το στοιχείο lock[0] έχει την τιμή true ώστε η πρώτη διαδικασία που θα θελήσει να εκτελέσει να μπορέσει. Κάθε φορά που ένα thread θέλει να εκτελέσει παίρνει την τιμή του counter αυτού και τον κάνει increment. Περιμένει έως ότου το στοιχείο του (έστω i) του πίνακα lock γίνει lock[i] = true. Όταν γίνει true, εκτελεί το κρίσιμο τμήμα. Όταν τελειώσει με το κρίσιμό του τμήμα θέτει lock[i] = false, και lock[(i+1)%size] = true ώστε να μπορέσει να εκτελέσει η επόμενη διαδικασία. Προφανώς, το array κάνει wrap around, δηλαδή όταν το ο counter φτάσει στην τιμή των threads παίρνει την τιμή 0. Έτσι, το Array λειτουργεί ως queue στο οποίο παίρνουμε από μπροστά στοιχεία και προσθέτουμε από πίσω. Αυτό βοηθά στην εξοικονόμηση overhead και συνεπώς, ο αλγόριθμος είναι αρκετά γρήγορος. Όπως φαίνεται και στο σχήμα, είναι ο δεύτερος γρηγορότερος από αυτούς που επιστρέφουν σωστά αποτελέσματα.

Το **pthread_mutex_lock** είναι υλοποίηση του lock με mutexes. Το αντικείμενο mutex στο οποίο δείχνει ο pointer lock καλείται από τη συνάρτηση. Εάν ο mutex είναι ήδη locked περιμένει να γίνει unlocked, το παίρνει, είναι δηλαδή owner του lock και το κάνει lock.Όταν ένα thread προσπαθεί να διεκδικήσει το lock, εάν είναι κατειλημμένο, “κοιμίζει” τον εαυτό του και περιμένει έως ότου ελευθερωθεί για να το διεκδικήσει εκ νέου. Το πρόβλημα με αυτήν τη λειτουργία είναι ότι η διαδικασία ώστε να μεταβεί ένα thread σε κατάσταση ύπνου ή να ξυπνήσει είναι αργή και ειδικά όταν η εργασία που έχει να εκτελέσει το

κάθε thread είναι μικρή, όπως εδώ, δεν μπορεί να δικαιολογηθεί το overhead.

Σε αυτήν την περίπτωση καλύτερα αποδίδει το **spinlock**. Σε αυτήν την υλοποίηση όταν ένα thread δεν βρει διαθέσιμο το κλειδί θα spin-άρει πάνω από αυτό έως ότου το βρει ελεύθερο και καταφέρει να το πάρει καταναλώνοντας μεν χρόνο CPU τον οποίο θα μπορούσε να εκμεταλλευτεί κάποια άλλη εφαρμογή, όμως κάνοντας τη διαδικασία μετάβασης του κλειδιού και επομένως τη συνέχιση της εργασίας πολύ γρηγορότερη και αυξάνοντας έτσι το throughput.

Το καλύτερο είδος κλειδώματος για το συγκεκριμένο πρόβλημα που δοκιμάστηκε είναι το **CLH Lock**. Βασίζεται σε μία συνδεδεμένη λίστα όπου κάθε thread κατέχει έναν κόμβο με ένα boolean πεδίο locked το οποίο υποδεικνύει ότι το thread είτε κατέχει το lock είτε επιθυμεί να το πάρει. Όταν ένα thread θελήσει να πάρει το lock θα θέσει το πεδίο του σε true και θα βάλει τον κόμβο του στο τέλος της λίστας, διατηρώντας ένα link στον προηγούμενο κόμβο. Εκεί θα spin-άρει έως ότου δει ότι ο προκάτοχός του έχει θέσει το πεδίο του σε false και επομένως πλέον έχει αυτός το κλειδί. Με αυτόν τον τρόπο εξασφαλίζεται αφενός ο περιορισμός των cache invalidations καθώς όταν ένα thread ελευθερώνει το κλειδί αλλάζοντας το πεδίο του κόμβου του σε false γίνεται invalidate μόνο η cache του επόμενου στη λίστα thread. Αφετέρου, εξασφαλίζεται και μία First-Come-First-Serve μορφή δικαιοσύνης η οποία καθιστά βέβαιο πως δε θα υπάρξει λιμοκτονία.

Στην περίπτωση του **Test-and-Set** lock **(**TAS**)**, ο αμοιβαίος αποκλεισμός γίνεται μέσω της ομώνυμης ατομικής εντολής (Test-and-set). Η εντολή επιτρέπει στο thread που την εκτελεί να διαβάσει την προηγούμενη τιμή μίας μεταβλητής (ενός lock) και στη συνέχεια να την κάνει set (ίση με 1), ενώ αυτή η σειρά από ενέργειες εξασφαλίζεται ότι εκτελείται ατομικά. Αν η προηγούμενη τιμή της μεταβλητής ήταν 0 (το lock ήταν ελεύθερο), το thread συνεχίζει, μπαίνοντας στην κρίσιμη περιοχή. Αλλιώς συνεχίζει να "σπινάρει" επανεκτελώντας την Test-and-Set μέχρι να διαβάσει 0. Το πρόβλημα με αυτή την διαδικασία είναι ότι όλα τα threads που έχουν μπλοκαριστεί περιμένοντας να μπουν στην κρίσιμη περιοχή γράφουν συνέχεια την κοινή μεταβλητή (το lock), αφού κάθε φορά που εκτελούν Test-and-Set κάνουν το lock ίσο με 1. Το αποτέλεσμα είναι το κάθε thread να κάνει εναλλάξ invalidate τα αντίγραφα της μεταβλητής lock στις caches των υπόλοιπων threads. Έτσι δημιουργείται περιττή κίνηση από την επικοινωνία μεταξύ των threads για το invalidation της μεταβλητής και για να πάρει κάθε φορά το thread που γράφει το lock τον exclusive έλεγχο (χρησιμοποιώντας ορολογία MESI πρωτοκόλλου). Η επικοινωνία αυτή δημιουργεί στην περίπτωσή μας τόσο overhead ώστε η υλοποίηση του K-means που χρησιμοποιεί TAS δίνει την χειρότερη επίδοση για όλα τα πλήθη Threads, από 4 και πάνω. Μάλιστα για τα 32 και 64 threads η πτώση της επίδοσης είναι θεαματική σε σχέση με τις υπόλοιπες υλοποιήσεις.

Μια βελτίωση της παραπάνω περίπτωσης lock είναι το **Test-and-Test-and-Set** lock (TTAS). Αυτό χρησιμοποιεί μια τροποποίηση της ατομικής εντολής Test-and-Set, την Test-and-Test-and-Set. Η TTAS εντολή εξασφαλίζει ότι σε πρώτη φάση το thread διαβάζει την τιμή του lock και την ελέγχει. Αν αυτή είναι 0 (ελεύθερο lock) προχωράει για να κάνει κλασικό Test-and-Set όπου πάλι ενδέχεται να υπάρξει ανταγωνισμός καθώς πολλά threads είδαν μαζί το ελεύθερο lock.. Αλλιώς (αν το lock είναι 1) "σπινάρει" μέχρι να το δει να γίνεται 0. Με αυτόν τον τρόπο αποφεύγουμε την συνεχή εγγραφή του lock, ενώ αυτό είναι ήδη set από Threads που περιμένουν να το πάρουν. Έτσι γλιτώνουμε άσκοπα I/Os και πολύ overhead από το πρωτόκολλο συνάφειας μνήμης. Είναι φανερό και από την περίπτωση του αλγορίθμου μας ότι το TTAS lock βελτιώνει θεαματικά την επίδοση σε σχέση με το TAS lock, ειδικά στις

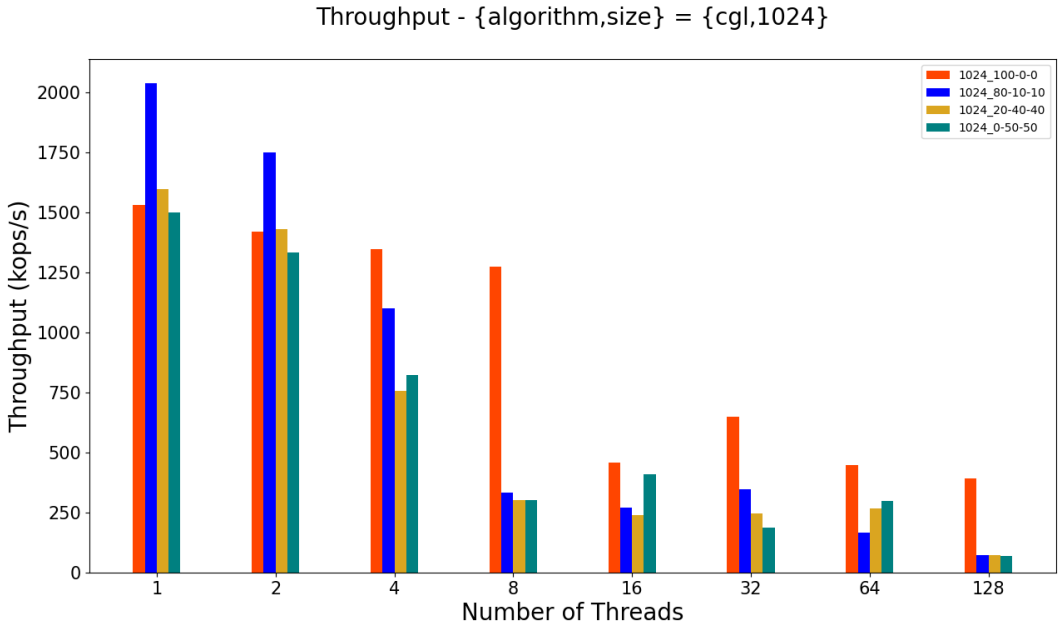
περιπτώσεις πολλών threads. Η επίδοση του TTAS lock είναι για τις περισσότερες περιπτώσεις που εξετάσαμε κοντινή με αυτή του Spin Lock και του Mutex Lock.

Ο αλγόριθμος που τρέχει με **OMP critical** ορίζει ένα σημείο του κώδικα το οποίο πρέπει να τρέξει ένα μοναδικό thread κάθε φορά. Είναι μια πολύ γενική δομή η οποία ορίζει ότι αν ένα thread θέλει να μπει στην κρίσιμη περιοχή, περιμένει πρώτα να τελειώσει την εκτέλεση της κρίσιμης περιοχής κάποιο thread που βρίσκεται ήδη σε αυτήν, εκτός και αν η κρίσιμη περιοχή δεν είναι "κατηλημμένη". Αυτό προκαλεί μεγάλο overhead κατά την έξοδο και είσοδο του thread στο critical section. Έτσι, εξηγείται η μεγάλη χρονική καθυστέρηση που φαίνεται να προκύπτει για μεγάλο αριθμό threads.

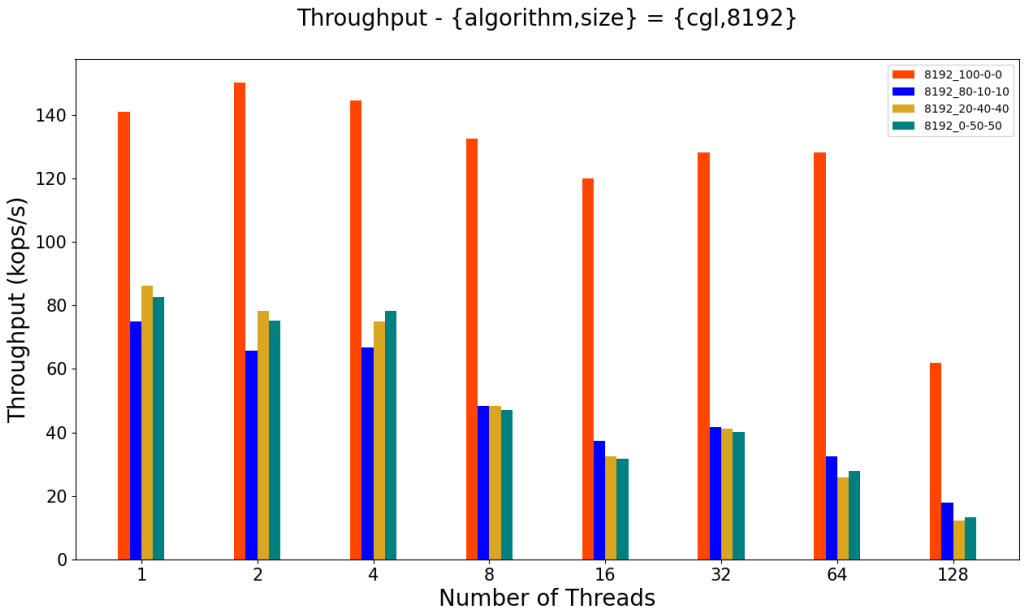
2.3 Ταυτόχρονες Δομές Δεδομένων

Ακολουθούν τα διαγράμματα Throughput για όλες τις περιπτώσεις:

- Coarse-Grain Locking (size=1024)

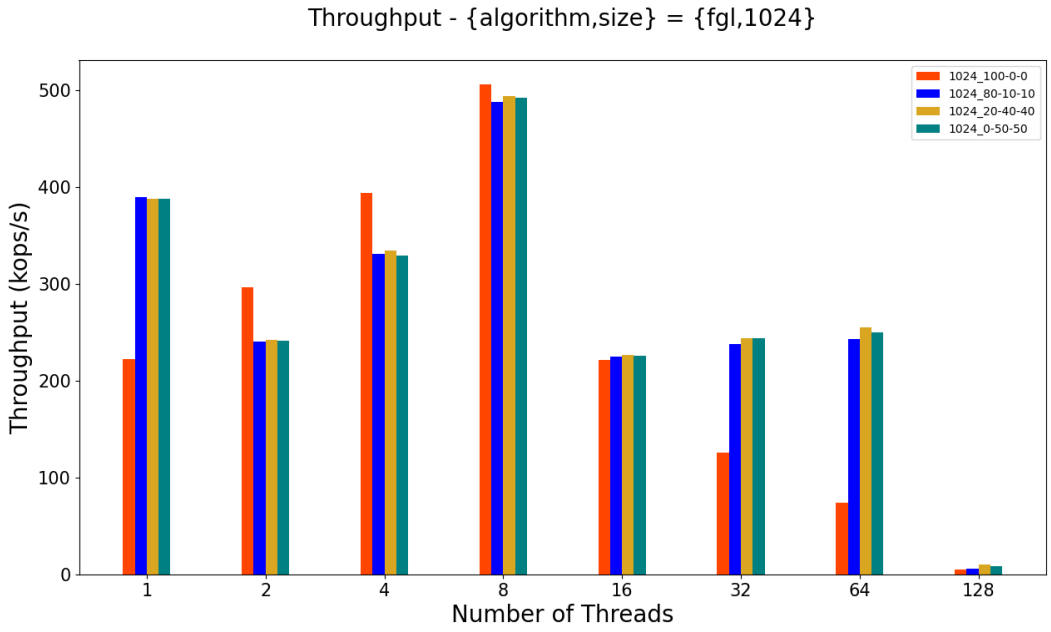


- Coarse-Grain Locking (size=8192)

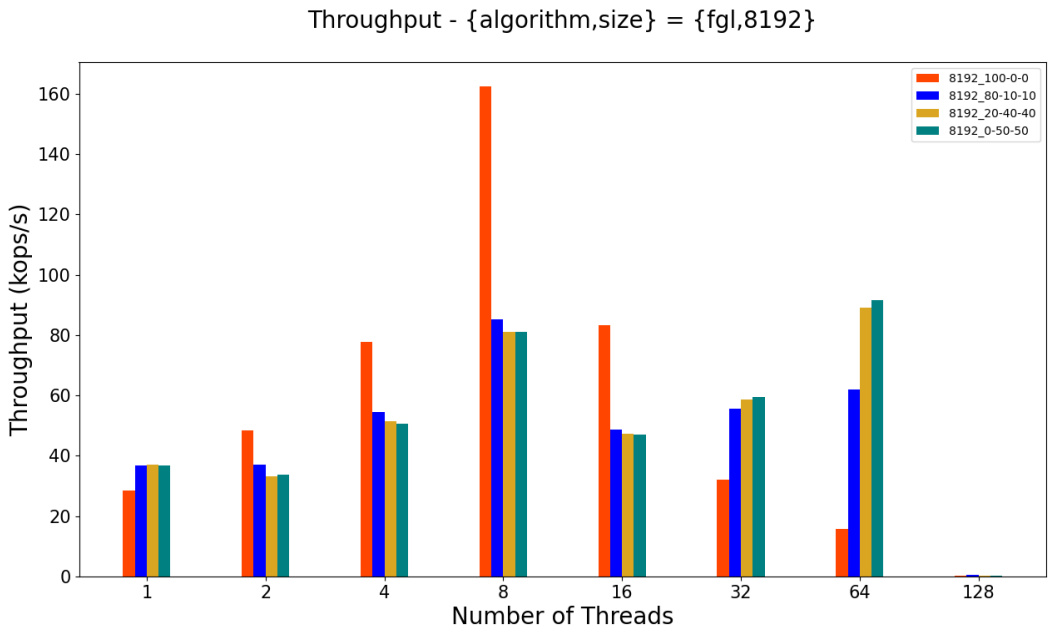


Στην περίπτωση του coarse-grain locking όπου ολόκληρη η δομή κλειδώνεται σε κάθε operation είναι αναμενόμενο να έχουμε τουλάχιστον χειρότερο performance από τη single threaded περίπτωση αφού το πολύ ένα thread κάθε φορά μπορεί να έχει πρόσβαση σε αυτήν. Προφανώς το overhead που εισάγει η διαχείριση των threads και των locks δυσχεραίνει την απόδοση όσο αυξάνουμε τα threads.

- Fine-Grain Locking (size=1024)

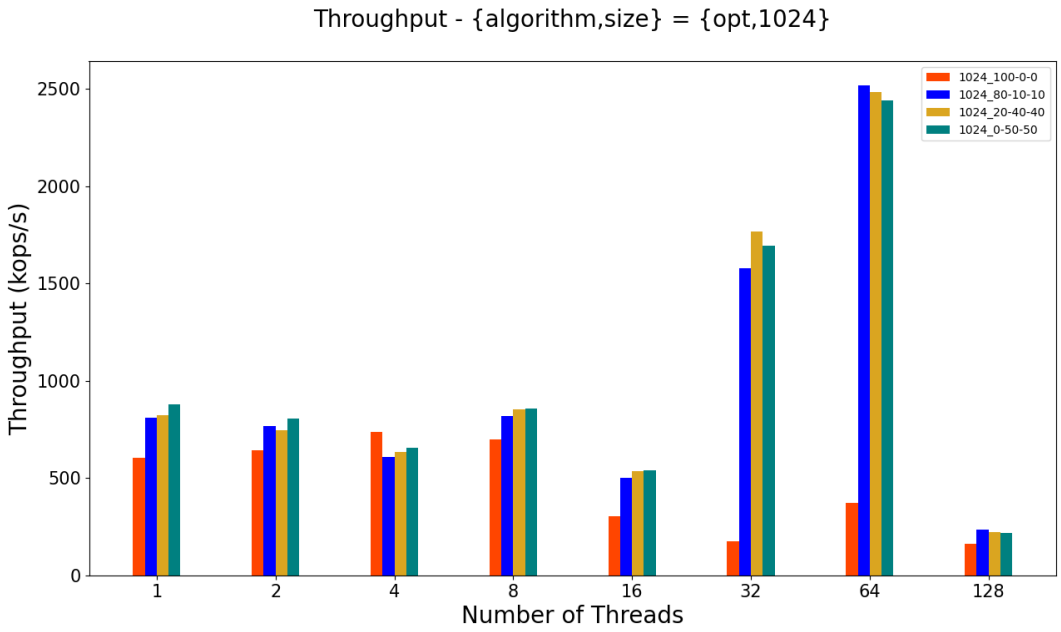


- Fine-Grain Locking (size=8192)



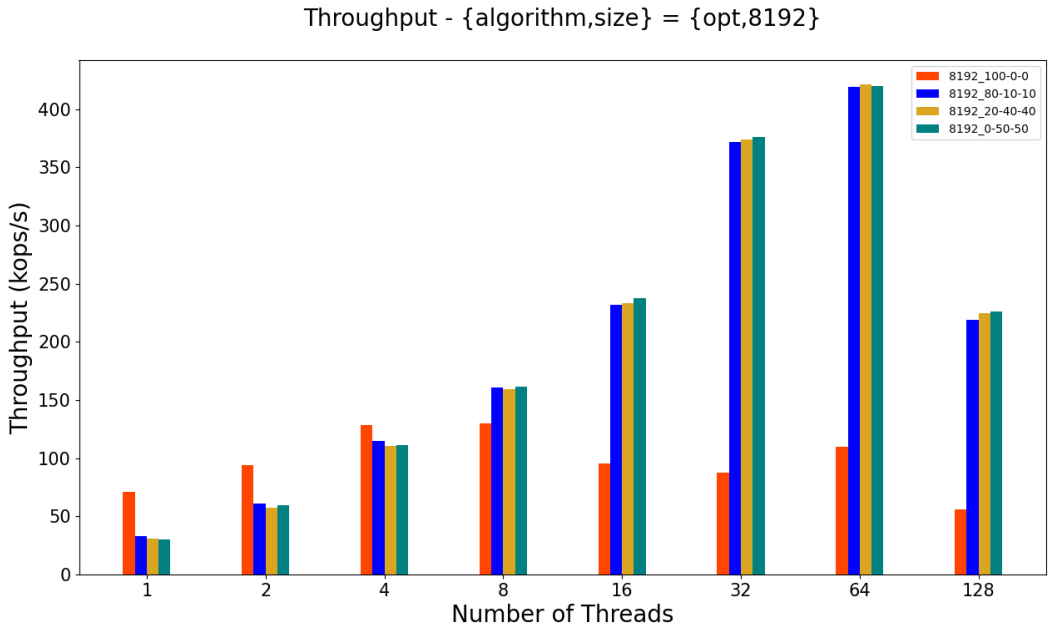
Στην περίπτωση του Fine-Grain Locking παρατηρούμε speedup και μάλιστα αρκετά μεγάλο με χρήση πολλαπλών threads. Βέβαια δεν πλησιάζει ούτε κατ' ελάχιστον το βέλτιστο και καταρρέει για πολύ μεγάλο αριθμό threads. Αυτό συμβαίνει καθώς, για να εκμεταλλευτεί την παραλληλία, δεν κλειδώνει ολόκληρη τη δομή αλλά μόνο τους κόμβους στους οποίους δουλεύει, ωστόσο χρησιμοποιεί hand over hand locking δημιουργώντας αδιαπέραστα "φράγματα" στη δομή για threads που προσπαθούν να προσπεράσουν το προπορευόμενο thread. Αυτο όπως φαίνεται και στο διάγραμμα δημιουργεί ακόμα μεγαλύτερο πρόβλημα για την αναζήτηση (contains) καθώς όλος ο χρόνος του operation αποτελείται από τη διάσχιση της λίστας (δεν υπάρχει κάποια δυνητικά χρονοβόρα πράξη να εκτελεστεί). Όταν λοιπόν ένας μεγάλος αριθμός από threads κληθεί να διασχίσει ταυτόχρονα τη λίστα και το ένα thread κολλά πίσω από το άλλο περιμένοντας τις συναλλαγές στα locks, ο χρόνος που δαπανάται ωφέλιμα θα είναι μικρός σε σχέση με αυτόν που δαπανάται στα lock transactions και το throughput θα μειώνεται σημαντικά. Ενώ το πρόβλημα του hand over hand locking εξακολουθεί να υφίσταται και στα άλλα δύο operations (add, remove) λόγω του ότι αυτά εκτελούν κάποια πράξη μόλις βρουν το επιθυμητό στοιχείο θα έχουν καλύτερη αναλογία actual workload, overhead.

- Optimistic Synchronization (size=1024)



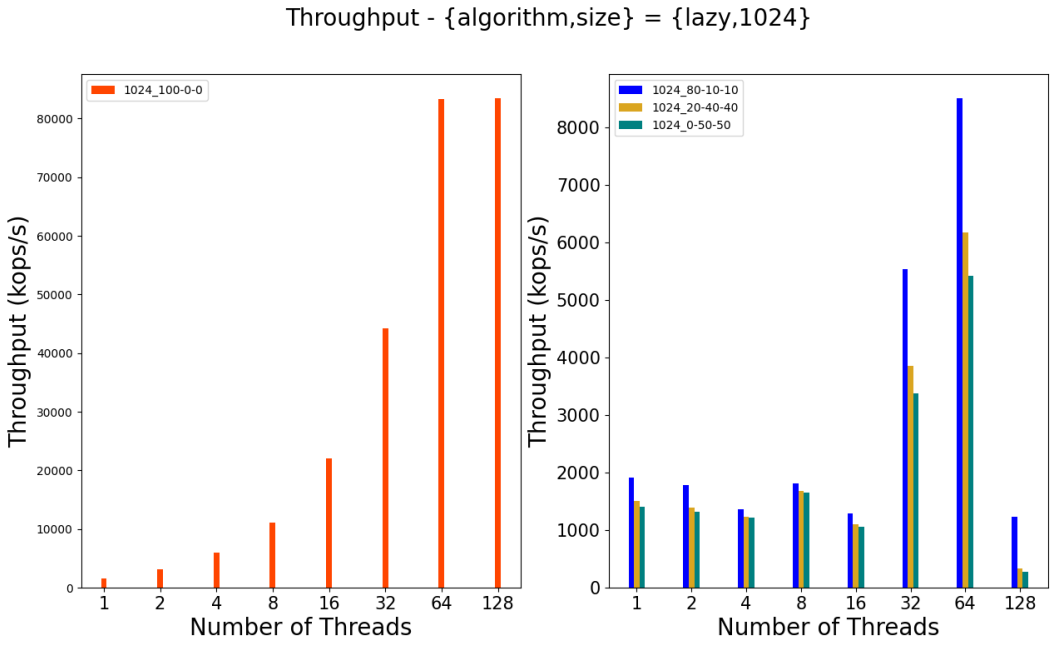
- Optimistic Synchronization (size=8192)

Η περίπτωση του Optimistic Synchronization είναι η πρώτη όπου παρατηρούμε καλό scaling στο Throughput, για πολλά Threads. Συγκεκριμένα η επίδοση βελτιώνεται μέχρι και τα 64 threads (που ισοδυναμούν και στο πλήθος φυσικών πυρήνων), ενώ για τα 128 Threads έχουμε βύθιση. Η βελτίωση στην επίδοση, βέβαια αφορα μίξεις operations που περιλαμβάνουν έστω λίγες εισαγωγές / διαγραφές. Αυτές οι πράξεις είναι εκ φύσεως πιο χρονοβόρες και έτσι μπορούν, αφενός να δικαιολογήσουν το overhead από την χρήση κλειδωμάτων (το overhead είναι σημαντικά μικρότερο από τον απαιτούμενο χρόνο των operations). Αφετέρου, αποδεικνύεται ότι για την περίπτωση μας η “Optimistic” υπόθεση αυτής της τεχνικής βγαίνει αληθινή. Δηλαδή δεν είναι τόσο συχνό το φαινόμενο ένα insert/delete operation να συμπέσει με ένα άλλο και έτσι δεν χαλάει η επίδοση από την επιλογή της τεχνικής να επιτρέπει την αποτυχία και την επανέναρξη ενός operation εξαιτίας ενός άλλου. Αντίθετα, στην περίπτωση που έχουμε αποκλειστικά αναζητήσεις έχουμε ελάχιστη βελτίωση με την αύξηση των Threads και αυτό για πλήθος Threads μέχρι το 8. Αυτή η διαφορά οφείλεται στο γεγονός ότι οι πράξη της

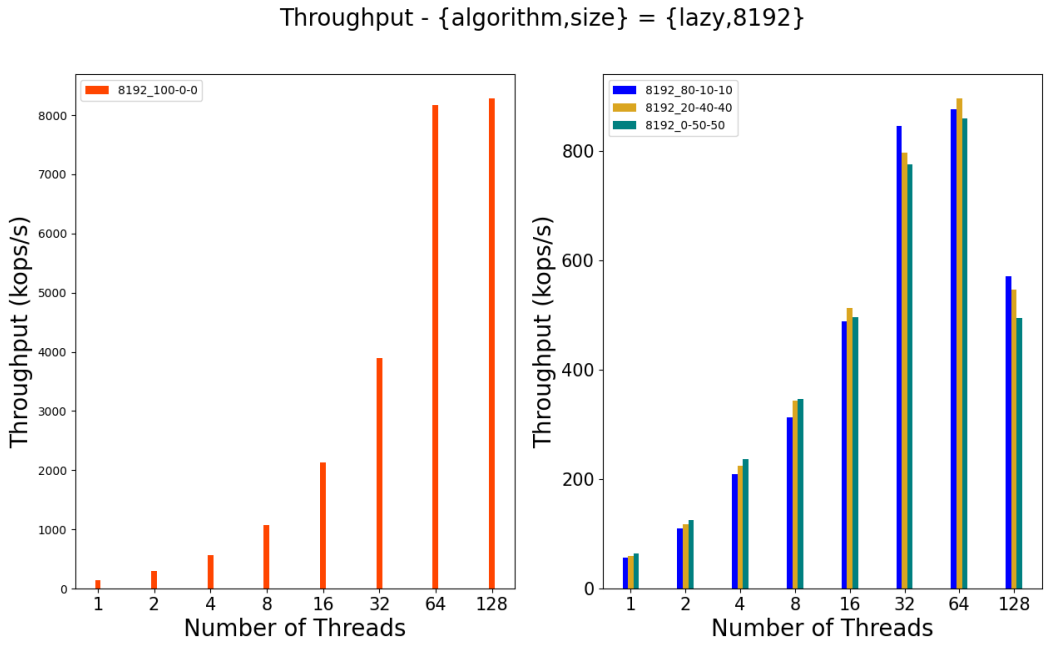


αναζήτησης δεν είναι χρονοβόρα (οσο η εισαγωγή και η διαγραφή) με αποτέλεσμα το overhead του κλειδώματος να γίνεται συγκρίσιμο με τον χρόνο της πράξης, ειδικά μάλιστα για τις περιπτώσεις με πολλά Threads, όπου οι συγκρούσεις / συνθήκες ανταγωνισμού είναι συχνότερο φαινόμενο.

- Lazy Synchronization(size=1024). Εδώ χωρίσαμε τα διαγράμματα, λόγω μεγάλης διαφοράς στην κλίμακα. Το ίδιο ισχύει και για τα παρακάτω.



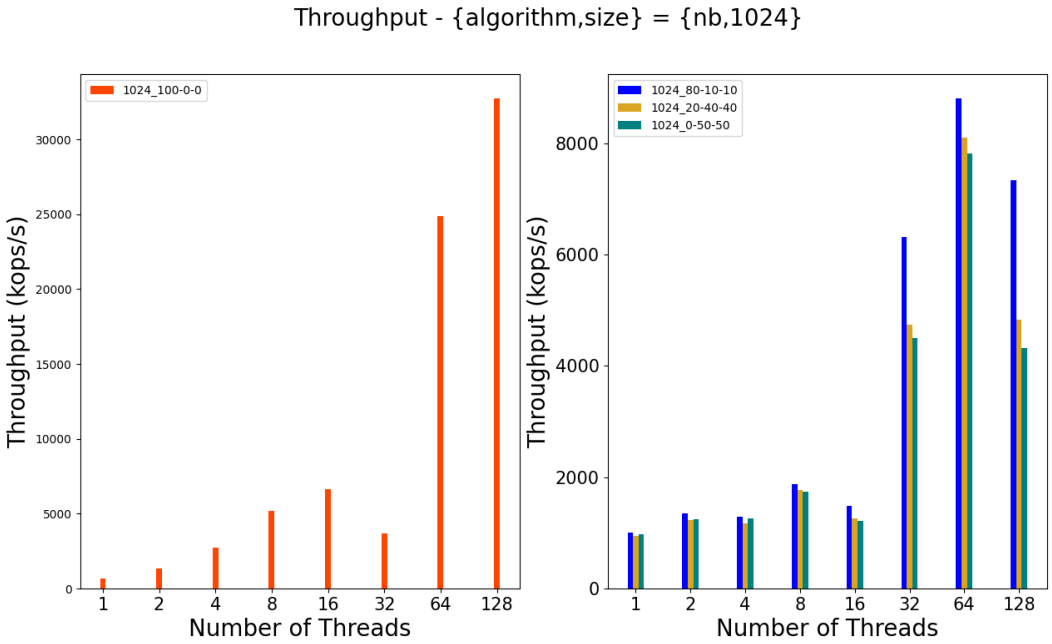
- Lazy Synchronization(size=8192)



Η μέθοδος Lazy-Synchronization μπορεί να εξεταστεί σαν εξέλιξη της Optimistic, αφού διορθώνει τις αδυναμίες της προηγούμενης. Όσον αφορά την περίπτωση που κάνουμε μόνο Αναζητήσεις, παρατηρούμε

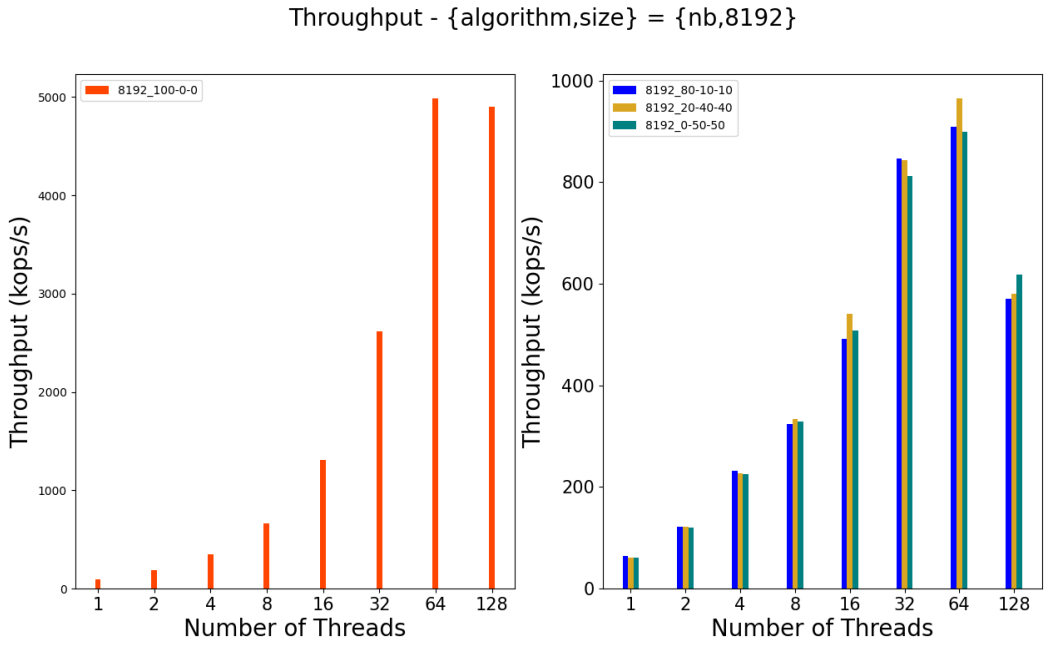
ότι έχουμε θεαματική αύξηση του Throughput σε σχέση με τις προηγούμενες τεχνικές, ενώ πετυχαίνουμε Scaling για όλο το εύρος από πλήθη Threads που εξετάσαμε. Αυτό οφείλεται στην πλήρη εγκατάλειψη των κλειδωμάτων για την λειτουργία αναζήτησης. Από την άλλη, η περίπτωση που αφορά μόνο εισαγωγές / διαγραφές παρουσιάζει επίσης βελτίωση σε σχέση με τις προηγούμενες τεχνικές. Αυτό οφείλεται στην προσθήκη Boolean μεταβλητών στους κόμβους που δείχνουν αν ένας κόμβος είναι (λογικά) μέρος της λίστας. Αυτές οι μεταβλητές μας γλιτώνουν από την επαναδιάσχιση της λίστας κατά το validation μετά την απόκτηση των locks και πριν την εισαγωγή ή διαγραφή. Όπως είναι αναμενόμενο, οι περιπτώσεις που έχουμε μίξη operations παρουσιάζουν επίσης βελτιωμένο throughput.

- Non-Blocking Synchronization(size=1024)



- Non-Blocking Synchronization(size=8192)

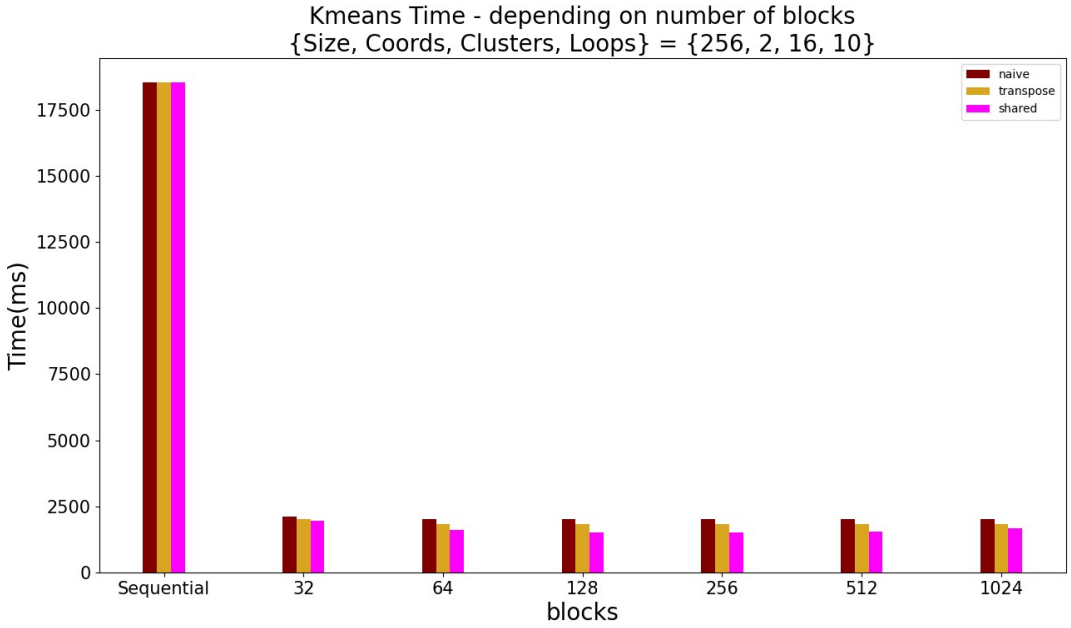
Τέλος, η μέθοδος non-blocking synchronization αποτελεί τη λύση στην χρήση lock της lazy

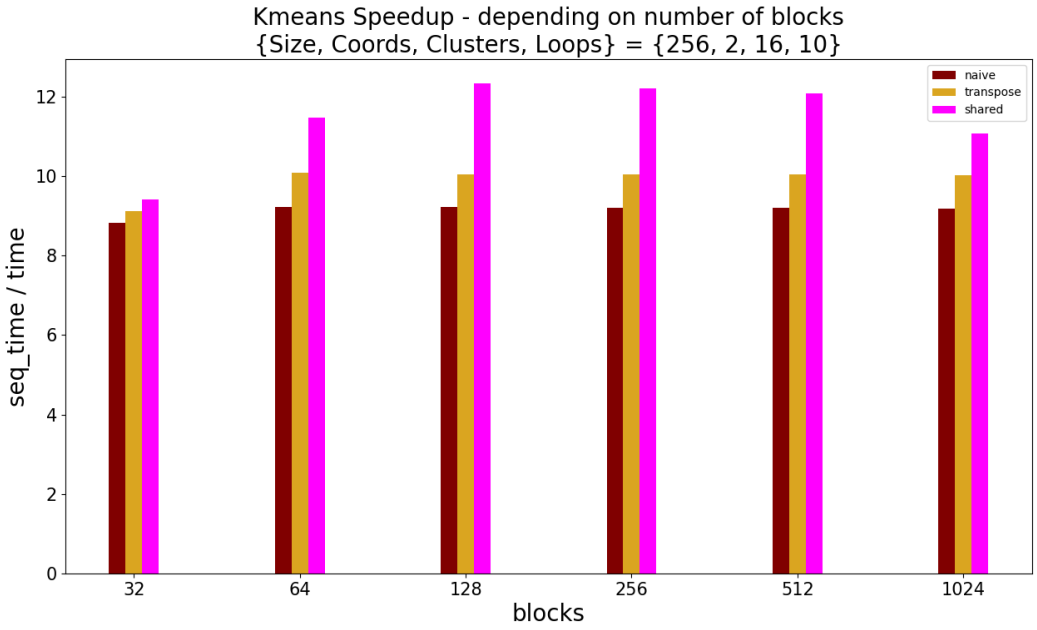


synchronization στα insert και delete tasks. Το overhead των locks απαλείφεται και αντικαθιστάται με reruns στην περίπτωση που δεν ισχύει η τωρινή κατάσταση κάποιου εκ των εξεταζόμενων κόμβων. Το μεγαλύτερο μειονέκτημα αυτής της στρατηγικής είναι η επιβάρυνση της find. Αυτή, κατά τη διάσχιση της λίστας, κάνει έναν έλεγχο σε κάθε κόμβο, ώστε να δει αν είναι marked. Εάν είναι, τότε τον αφαιρεί και γίνεται κανονικά η φυσική αφαίρεση, αλλιώς συνεχίζει κανονικά. Η επιβάρυνση, αν και μικρή, φαίνεται από την χαμηλότερη επίδοση της γραφικής παράστασης στα αριστερά και στις δύο περιπτώσεις. Αυτή η επιβάρυνση όμως είναι μικρότερη του κέρδους που προσφέρει η απαλοιφή των locks και γι αυτό, στο δεξί διάγραμμα κάθε φορά, βλέπουμε αρκετή βελτίωση για τα προγράμματα με μεγαλύτερο αριθμό threads στον 1024, μικρή στο 8192 και καθόλου για μικρό αριθμό threads (< 16).

3.1 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε Επεξεργαστές Γραφικών

- **Διαγράμματα χρόνου εκτέλεσης και Speedup για configuration: {Size, Coords, Clusters, Loops} = {256, 2, 16, 10}**





Με configuration {Size, Coords, Clusters, Loops} = {256, 2, 16, 10} τρέξαμε τα τρία προγράμματα που δημιουργήσαμε και παρατηρούμε τα εξής:

Naive version: Κάνουμε μία πρώτη απόπειρα να αναθέσουμε το υπολογιστικά βαρύ κομμάτι του αλγορίθμου στη GPU προκειμένου να το παραλληλοποιήσουμε. Προς αυτό, αναθέτουμε σε κάθε thread της GPU ένα object, ώστε να υπολογίσει την απόστασή του από τα clusters και να βρει το κοντινότερό του, στο οποίο θα ανήκει στο επόμενο iteration. Αυτό γίνεται δημιουργώντας ένα grid με διαστάσεις `(numObjs + numThreadsPerClusterBlock -1) / numThreadsPerClusterBlock` αφού κάθε block θα περιέχει συγκεκριμένο αριθμό threads και θέλουμε συνολικά να έχουμε τόσα threads όσα και objects (προφανώς το υπογραμμισμένο τμήμα αντιστοιχεί σε padding). Επίσης για την μεταβλητή delta, η οποία δείχνει τον βαθμό των αλλαγών όσον αφόρα σε ποιο centroid ανήκει κάθε object στο συγκεκριμένο iteration απαιτούνται ατομικά increments κάθε φορά που γίνεται μία αλλαγή. Επειδή θα ήταν πολύ αργά αυτά τα increments αν γίνονταν με τα atomics που προσφέρει η CUDA για κάθε object, εφαρμόζουμε reduction. Συγκεκριμένα χρησιμοποιούμε εντός των blocks τη shared μνήμη κάθε SM ώστε να κατασκευάσουμε έναν πίνακα από integers με μέγεθος όσο το μέγεθος του block, ώστε κάθε thread να αποθηκεύει στη θέση που του αντιστοιχεί έναν άσσο εφόσον το object του άλλαξε centroid. Έπειτα αφού τελειώσουν οι υπολογισμοί για όλα τα threads του block τα μισά threads αθροίζουν ανά δύο τις θέσεις του πίνακα και αποθηκεύουν τα αποτελέσμα στον μισό πίνακα. Η διαδικασία συνεχίζεται έως όλα τα delta του block να έχουν αθροιστεί στην πρώτη του θέση και μόνο τότε αυτό το αποτέλεσμα αθροίζεται ατομικά στη global μεταβλητή για το delta. Με αυτόν τον τρόπο επιταχύνουμε σημαντικά ένα κομμάτι που κατά τα άλλα θα ήταν bottleneck χωρίς να παραβιάζουμε κάποιο race condition. Επίσης σε κάθε περίπτωση το μέγεθος της shared μνήμης ανά block φτάνει στη χειρότερη περίπτωση τα 4K για μέγεθος block 1024 το οποίο δε θα μειώσει το occupancy (περισσότερα για το occupancy στη συνέχεια). Ο υπολογισμός των νέων cluster centers εξακολουθεί να γίνεται στη CPU αφού έχουμε αντιγράψει εκεί το που ανήκει κάθε object, και έπειτα αντιγράφουμε τα νέα centers στη GPU για το επόμενο iteration.
Η επίδοση της naive version είναι πολύ καλύτερη από αυτή της σειριακής (έως και ~9 speedup). Ωστόσο το speedup είναι μάλλον μικρό αν αναλογιστούμε τον βαθμό παραλληλίας που εισάγει η GPU.
Το block size δε φαίνεται να παίζει ιδιαίτερο ρόλο για αυτήν την έκδοση. Για να έχουμε καλύτερη εικόνα για το πως επηρεάζει το block size οφείλουμε να γνωρίζουμε παραπάνω πληροφορίες για το hardware στο οποίο τρέχει η εφαρμογή. Χρησιμοποιώντας τη συνάρτηση cudaGetDeviceProperties του API της CUDA μπορούμε να εξάγουμε τις παρακάτω πληροφορίες:

```
CUDA version:   v11040
CUDA Devices:

0: Tesla K40c: 3.5
  Global memory:   11441mb
  Shared memory:   48kb
  Constant memory: 64kb
  Block registers: 65536

  Max threads per Multiprocessor: 2048
  Max threads per Block: 1024
  Number of Multiprocessors: 15

  Warp size:         32
  Threads per block: 1024
  Max block dimensions: [ 1024, 1024, 64 ]
  Max grid dimensions:  [ 2147483647, 65535, 65535 ]
```

Βλέπουμε ότι πρόκειται για το μοντέλο Tesla K40c (compute capability 3.5). Γνωρίζοντας αυτό μπορούμε να αναζητήσουμε ακόμα πιο αναλυτικές πληροφορίες. Βλέπουμε ότι υποστηρίζονται μέχρι 16 active blocks/SM οπότε η χρήση πολύ μικρών blocks (32 threads/block) οδηγεί σε 512 active threads/block δηλαδή μόλις 25% occupancy για αυτό έχει και χειρότερο performance από τα υπόλοιπα μεγέθη. Για 64 έχουμε 50% occupancy ενώ για 128 και πάνω 100%. Ωστόσο, το speedup δε φαίνεται να αυξάνεται από block_size 64 σε 128. Αυτό, πιθανόν, γίνεται γιατί η εφαρμογή μας είναι memory bound (και μάλιστα σε αυτήν την έκδοση οι προσβάσεις στη μνήμη δεν είναι coalesced) οπότε δεν υπάρχουν αρκετές πράξεις για να εκτελεστούν προκειμένου να κρύψουν τα latencies στη μνήμη και ακόμα και αν πετύχουμε 100% occupancy η μνήμη θα μας καθυστερεί αρκετά ώστε το performance να μη βελτιώνεται.

Transpose version: Στην έκδοση αυτή απλώς αντιστρέφουμε το indexing των πινάκων. Αυτό υπολογιστικά δεν αλλάζει κάτι στα δεδομένα μας, βοηθάει όμως την GPU στην λήψη αυτών από τη DRAM της. Η GPU πρέπει να πάρει στη σειρά τα δεδομένα κάθε γραμμής, τα οποία θα επεξεργαστούν από γειτονικά threads προκειμένου να τα ομαδοποιήσει σε ένα μεγάλο memory access πληρώνοντας μία φορά το latency και αξιοποιώντας το μεγάλο throughput της μνήμης. Τα γειτονικά threads κάνουν προσβάσεις στα ίδια coordinates διαφορετικών objects (αντίστοιχα και των clusters). Τα coordinates αυτά, χάρη στην transpose συνάρτηση, βρίσκονται σε γειτονικές θέσεις μνήμης (στη naive βρίσκονται σε γειτονικές θέσεις τα διαφορετικά coordinates του ίδιου object) και μπορεί να τα πάρει μαζικά και να τα διαμοιράσει στα threads που είναι υπεύθυνα για τον υπολογισμό. Έτσι, προκύπτει το speedup ως προς την naive version το οποίο είναι περίπου 10% (από το 9 στο 10). Καθώς χρησιμοποιούμε κάρτα με compute capability 3.x, αυτή διαθέτει L1 cache γεγονός που σώζει κάποια memory accesses στη naive έκδοση καθώς cacheαρίζονται κάποια δεδομένα που θα χρησιμοποιηθούν στη συνέχεια, τα οποία φέρνουν στην cache τα πολλά αχρείαστα accesses των non coalesced προσβάσεων. Για αυτό η διαφορά στο speedup είναι τόσο μικρή, καθώς η naive έκδοση δεν υποφέρει όσο θα ήταν αναμενόμενο αν δεν διέθεται cache.

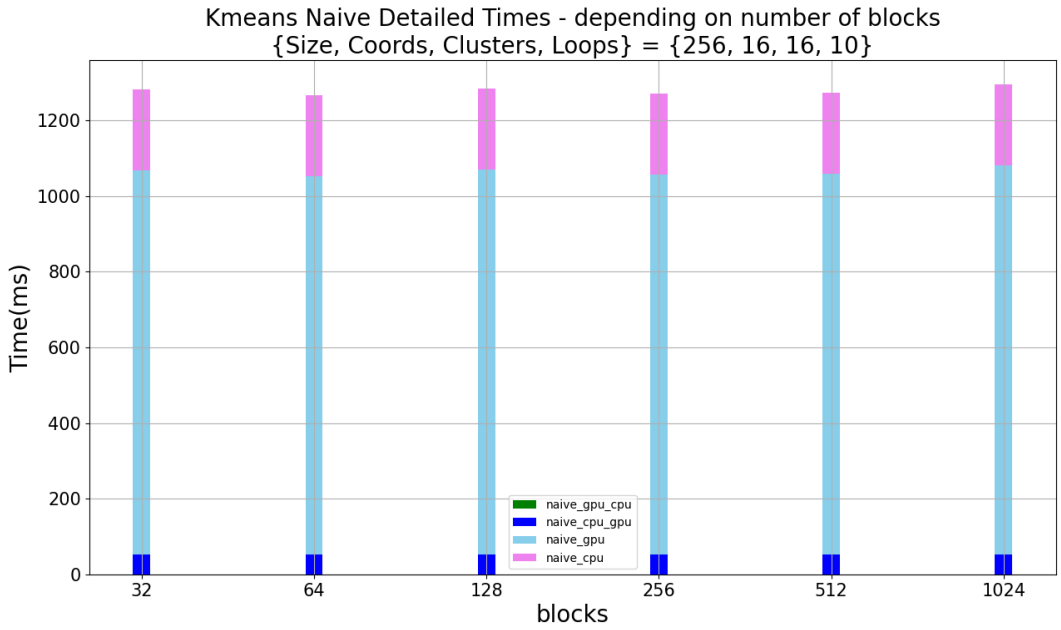
Όσον αφορά το block size παρατηρούμε εντελώς αντίστοιχη συμπεριφορά με τη naive έκδοση.
Shared version: Η τρίτη και γρηγορότερη έκδοση είναι η Shared version, η οποία αποθηκεύει τα clusters

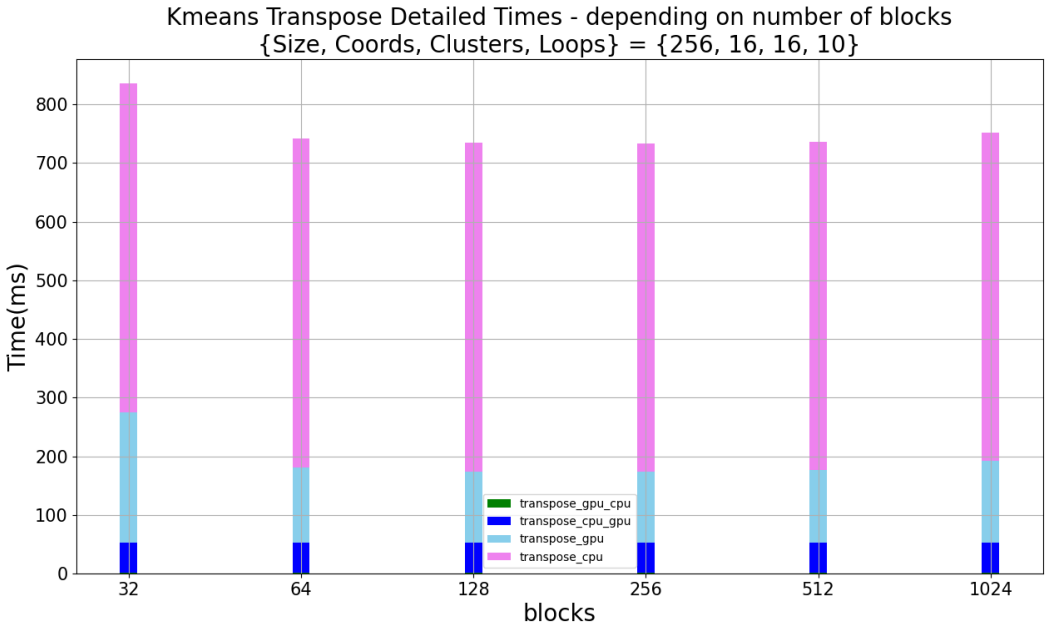
στην κοινή μνήμη της GPU. Με αυτόν τον τρόπο αποφεύγονται τα accesses στην κύρια μνήμη για τον υπολογισμό της απόστασης του κάθε object από όλα τα clusters, τα οποία στοιχίζουν χρόνο λόγω του latency. Αντί αυτού, κάθε block χρησιμοποιεί τα threads του συνεργατικά, ώστε να αντιγράψει στην αρχή όλα τα clusters στη shared μνήμη, με χρήση του local_tid προκειμένου κάθε thread να φέρει τα σωστά δεδομένα, και έπειτα διαβάζει από εκεί τις επιθυμητές τιμές. Χρειαζόμαστε
`numClusters*numCoords*sizeof(float)`
επιπλέον shared μνήμη.

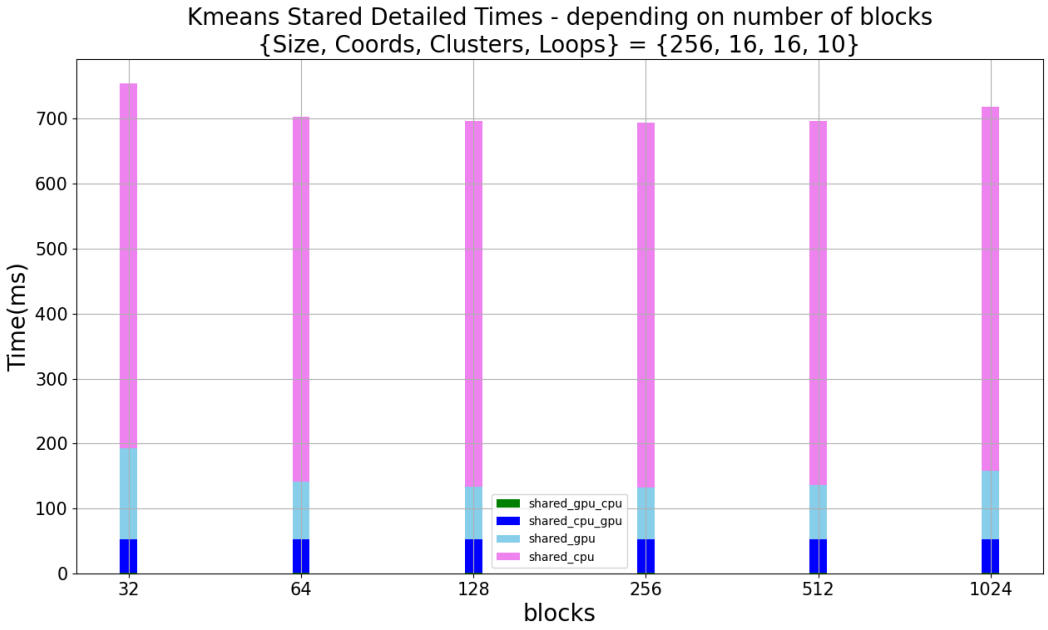
Το speedup αυξάνεται σημαντικά σε σχέση με την naive και την transpose εκδοση. Αυτό συμβαίνει διότι κάθε thread εκτελεί προσβάσεις σε όλα τα clusters, επομένως έχοντας τα στην κοινή μνήμη στην οποία τα φέρνουμε μία φορά όλα μαζί χρησιμοποιώντας πολλά threads γλιτώνουμε πολλά χρονοβόρα accesses.

Το block size τώρα παίζει διαφορετικό ρόλο. Για πολύ μικρό block size αντιγράφουμε πολλές φορές τα clusters, καθώς κάθε block διαθέτει τη δική του shared μνήμη, επομένως όταν έχουμε πολλά μικρά blocks θα εκτελούνται πολλές προσβάσεις στον πίνακα των clusters σε κάθε loop από τα διάφορα blocks προκειμένου να αντιγραφούν στη shared μνήμη τους. Αντιθέτως όταν έχουμε πολύ μεγάλα blocks πολλά threads χρησιμοποιούν την ίδια κοινή μνήμη, οδηγώντας σε shared memory bank conflicts με τα requests προς αυτήν να μην μπορούν να εξυπηρετηθούν παράλληλα, γεγονός που δυσχεραίνει την επίδοση. Το μέγεθος της shared μνήμης που χρησιμοποιείται ανά SM για τα deltas και τα clusters είναι αρκετά μικρό ώστε σε καμία περίπτωση να μη δημιουργεί θέμα με το occupancy. Με βάση τα παραπάνω και τα πειραματικά αποτελέσματά μας, το sweetspot φαίνεται να βρίσκεται στο block_size=128 με speedup μεγαλύτερο του 12.

- **Αναλυτική Μέτρηση των Χρόνων για τις διάφορες εκδόσεις του αλγορίθμου**



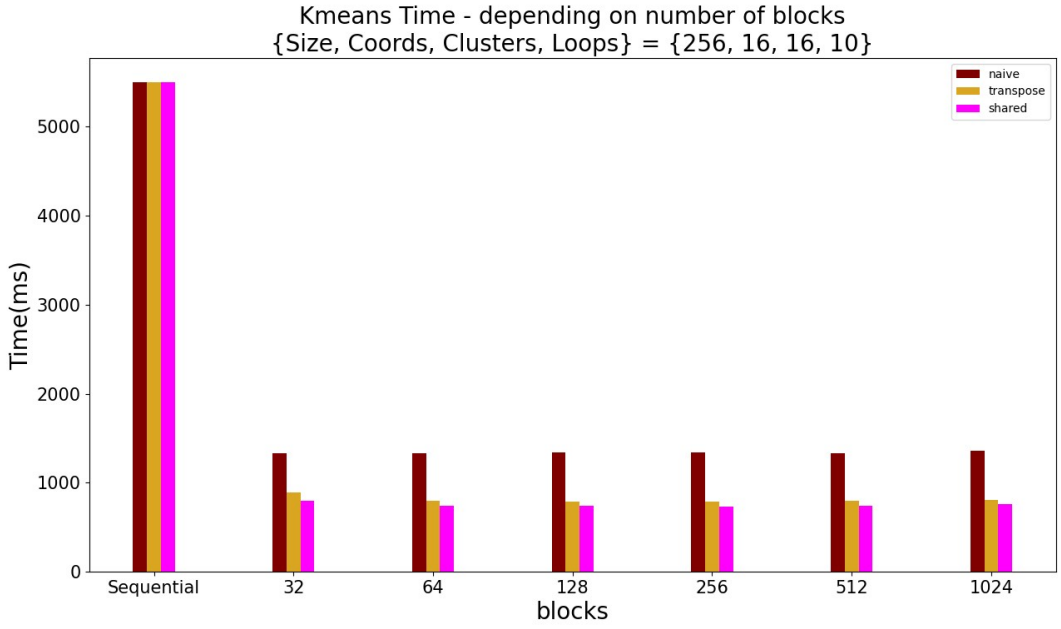




Με χρήση wtime() χρονομετρήσαμε τα διάφορα σημεία του while loop. Ο naive αλγόριθμος έχει πολύ

μεγαλύτερο GPU κομμάτι σε σχέση με τους άλλους δύο. Αυτό, όπως αναλύσαμε στα προηγούμενα ερωτήματα, συμβαίνει επειδή η GPU δυσκολεύεται να φέρει τα δεδομένα στα threads. Στις άλλες δύο υλοποιήσεις μοιάζει αυτό το πρόβλημα να εξαλείφεται και να μεταφέρεται στην CPU. Αφού αντιστρέφουμε τις διαστάσεις των πινάκων, καταστρέφεται το locality, που με τη σειρά του χαλάει την απόδοση της CPU που αξιοποιούσε αυτό το locality με τις caches της και προκαλεί αυτό το bottleneck. Είναι εμφανές ότι οι βελτιστοποιημένες υλοποιήσεις μας υποφέρουν στον χρόνο της CPU και όχι σε αυτόν που δαπανάται στους kernels ή στις μεταφορές τόσο πολύ, επομένως περαιτέρω βελτιστοποιήσεις πρέπει να εστιάσουν εκεί.

- **Διαγράμματα χρόνου εκτέλεσης και Speedup για configuration: {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}**

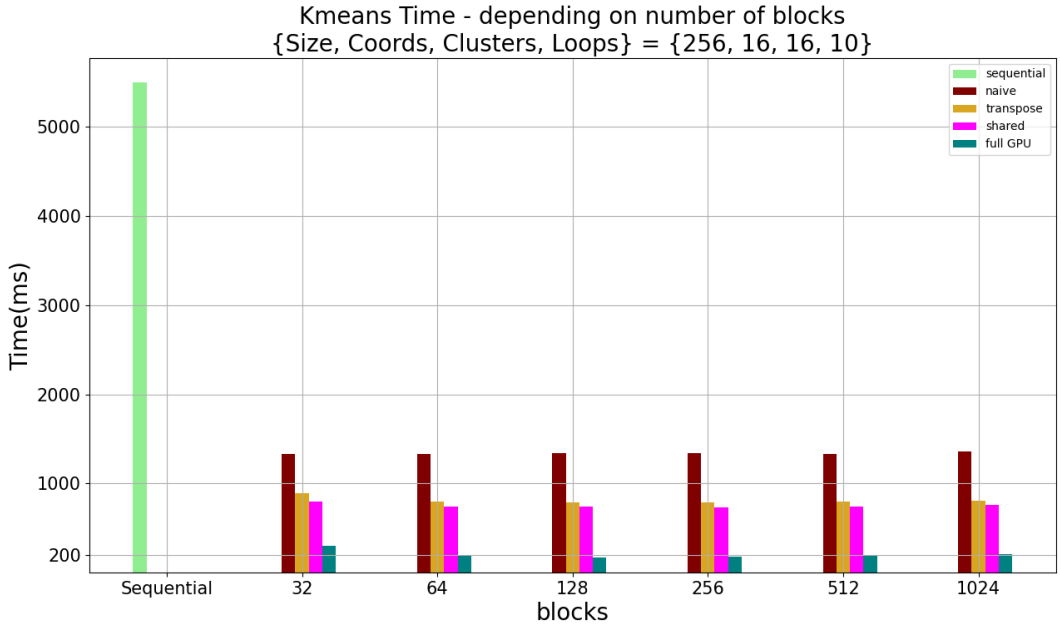


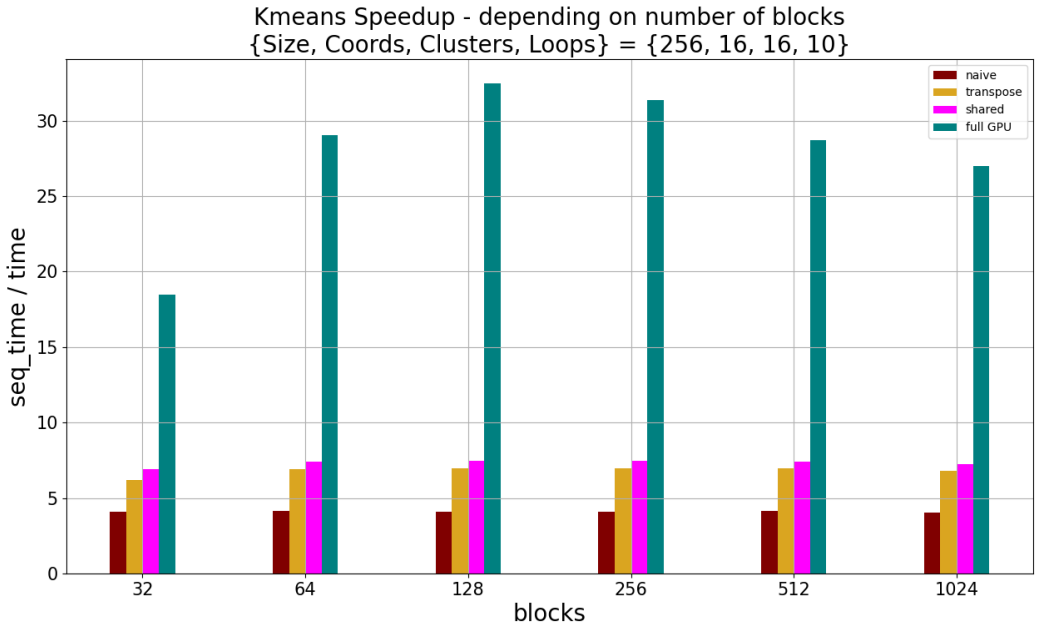
Με την αύξηση των διαστάσεων από 2 σε 16 παρατηρούμε μείωση του sequential time όπως και του speedup που κερδίζουμε από την GPU. Αφού αυξήσαμε τον αριθμό των διαστάσεων και μένει σταθερό το size του dataset στα 256MB, έχουμε μείωση του αριθμού των αντικειμένων και συνεπώς μείωση του sequential time και των υπολογισμών που πρέπει να γίνουν. Επομένως, το κομμάτι των υπολογισμών της GPU μειώνεται ακόμα παραπάνω σε σχέση με τις περιπτώσεις των 2 dimensions.

Επίσης εδώ παρατηρούμε μεγαλύτερη αύξηση του speedup μεταβαίνοντας από τη naive στην transpose έκδοση, γεγονός λογικό αφού τώρα τα συγκεκριμένα coordinates διαφορετικών objects βρίσκονται ακόμα πιο μακριά στη μνήμη (λόγω της αύξησης από 2 σε 16 coordinates) στη naive έκδοση. Αντιθέτως η αύξηση είναι μικρή από την transpose στη shared έκδοση. Αυτό πιθανότατα συμβαίνει γιατί πλέον ο πίνακας των clusters είναι πολύ μεγαλύτερος (1KB) και μαζί με τα deltas κάθε block καταλαμβάνει σημαντικό μερίδιο της κοινής μνήμης του SM. Παρόλο που ακόμα δεν έχουμε σε καμία περίπτωση θέμα με το occupancy λόγω υπερβολικής κατανάλωσης shared μνήμης, πιθανότατα έχουμε πολύ περισσότερα

conflicts ακόμα και μεταξύ διαφορετικών blocks για μικρά block_size γεγονός που δεν επιτρέπει να αξιοποιήσουμε πλήρως τα πλεονεκτήματα αυτής της υλοποίησης. Είναι εμφανές ότι η παρούσα shared υλοποίηση δεν είναι κατάλληλη για επίλυση του kmeans για arbitrary configurations καθώς με αύξηση των coordinates ή των clusters μπορεί να αυξηθεί σημαντικά η κατανάλωση shared μνήμης, που μπορεί να οδηγήσει σε δυσμενείς καταστάσεις όπως πάρα πολύ χαμηλό occupancy, διότι τα 48KB ενός SM μπορεί να επαρκούν πιθανόν μόνο για ένα block σε ακραίες περιπτώσεις, ενώ σε πιο ακραίες μπορεί να μην επαρκούν καν και ο kernel να μην μπορεί να τρέξει.

- **Bonus Ερώτημα: Διαγράμματα Χρόνου Εκτέλεσης και Speedup για την All GPU έκδοση του αλγορίθμου (config : {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}):**



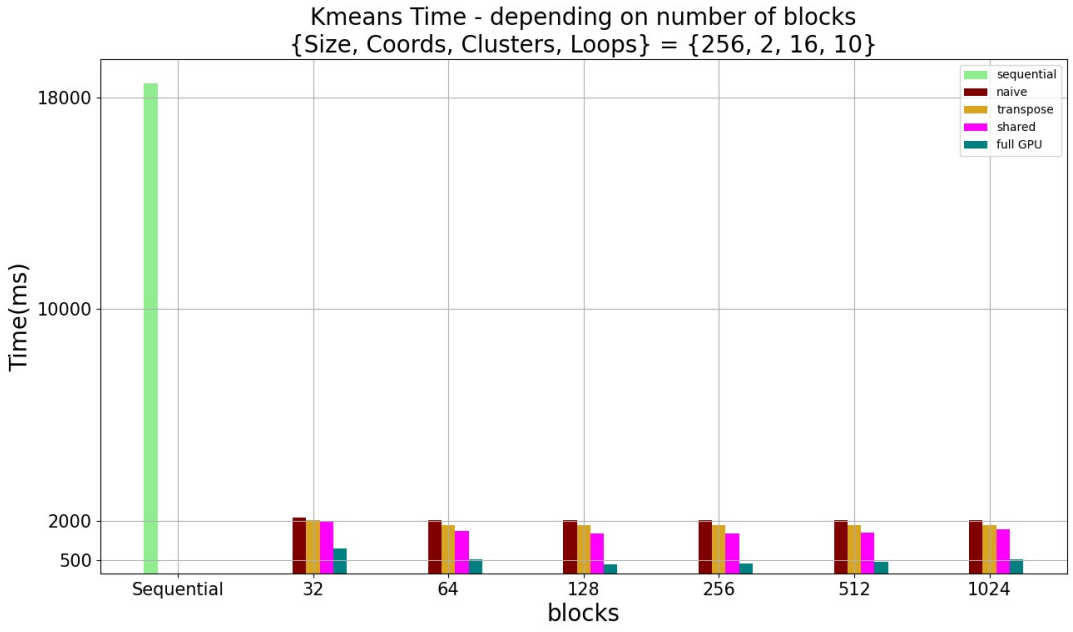


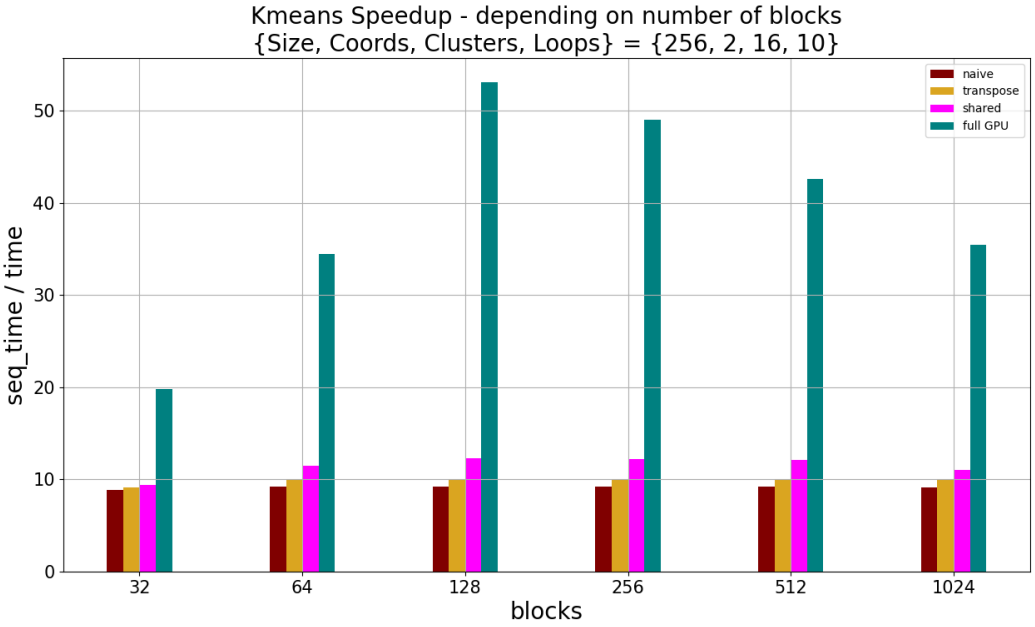
Όπως βλέπουμε, το bottleneck του προηγούμενου ερωτήματος αντιμετωπίζεται. Η νέα υλοποίηση προσφέρει τεράστια επιτάχυνση σε σχέση με το προηγούμενο ερώτημα, όπως φαίνεται από τα παραπάνω και παρακάτω διαγράμματα. Αναθέτοντας τους υπολογισμούς της CPU στην GPU εξοικονομούμε πολύ χρόνο στις πράξεις.

Για να το πετύχουμε αυτό χρειάστηκαν αρκετές τροποποιήσεις τόσο στον ήδη υπάρχοντα kernel, όσο και δημιουργία νέου. Πρέπει τώρα ο kernel που υπολόγιζε σε ποιο cluster ανήκει κάθε object, επιπλέον να αθροίζει σε αυτό το cluster τα coordinates του object που του ανήκει και να αυξάνει το μέγεθός του κατά 1 (αυτές οι τιμές θα χρησιμοποιηθούν από άλλο kernel για τον άμεσο υπολογισμό των νέων clusters). Το πρόβλημα με αυτό είναι ότι τα accesses στους πίνακες newClusterSize και newClusters πρέπει να είναι ατομικά και επιπλέον είναι τυχαία, αφού κάθε object μπορεί να ανήκει σε οποιοδήποτε cluster. Αυτά τα χαρακτηριστικά είναι αντίθετα με τον τρόπο που λειτουργεί η GPU η οποία θέλει υψηλό επίπεδο παραλληλισμού και coalesced memory accesses. Είναι λοιπόν ασφαλές να πούμε πως το κομμάτι update_centroids δεν είναι κατάλληλο για GPU. Προκειμένου να μετριάσουμε αυτό το πρόβλημα τοποθετούμε και αυτούς του πίνακες στην κοινή μνήμη. Με αυτόν τον τρόπο και γλιτώνουμε το penalty των μη coalesced random accesses το οποίο δεν υφίσταται με την κοινή μνήμη, καθώς και πολλές συγκρούσεις στα atomic writes καθώς στη shared μνήμη προσπαθούν να γράψουν στη χειρότερη όλα τα threads ενός block ταυτόχρονα, ενώ στη global όλα τα threads γενικά που είναι δραματικά περισσότερα. Μόλις τελειώσουν όλοι οι υπολογισμοί του block αναθέτουμε σε όσο το δυνατόν περισσότερα threads του να γράψουν ατομικά τις τιμές αυτών των πινάκων που υπολόγισαν στους global πίνακες. Η επιπλέον χρήση `numClusters*numCoords*sizeof(float)+ numClusters*sizeof(int)` bytes κοινής μνήμης δε δημιουργεί προβλήματα με το occupancy για φυσιολογικά block sizes και ενώ μπορεί να είναι μεγάλη και να αμβλύνει προβλήματα που αναφέρθηκαν στη shared έκδοση για arbitrary configurations, εκ του αποτελέσματος δικαιολογείται πλήρως καθώς είναι πολύ σημαντικότερη η

αφαίρεση του CPU bottleneck. Εδώ είναι ακόμα πιο σημαντικό να μην έχουμε ούτε πολύ μικρό block size (κακό occupancy, πολλά blocks, πολλά writes στη global μνήμη) αλλά ούτε πολύ μεγάλο (επιπλέον πολλές συγκρούσεις στα atomic writes εντός του block).
Εξετάστηκε σε θεωρητικό επίπεδο και το ενδεχόμενο να είχαμε έναν πίνακα για κάθε thread ώστε εντός του block να μπορούμε να εφαρμόσουμε reduction αντί για atomics ωστόσο το μέγεθος της κοινής μνήμης που θα απαιτούνταν ήταν πραγματικά απαγορευτικό και θα οδηγούσε σε άθλιο occupancy.
Έχοντας αυτά τα αποτελέσματα δημιουργούμε έναν επιπλέον kernel ο οποίος θα τα χρησιμοποιεί για να υπολογίσει τα νέα κέντρα. Αυτός ο kernel θα έχει χρήσιμα threads όσα ο αριθμός των clusters επί τον αριθμό των coordinates ώστε κάθε thread να υπολογίσει μία θέση του πίνακα. Δηλαδή Grid Size: `(numCoords*numClusters + update_centroids_block_sz - 1)/ update_centroids_block_sz`
Αυτός ο kernel είναι λιγότερο ευαίσθητος όσον αφορά το block size καθώς είναι πολύ μικρός και συνήθως δεν μπορεί να σπάσει σε πολλά blocks.
Αυτή η υλοποίηση έχει επιπλέον το πλεονέκτημα ότι εξαλείφονται πλήρως οι καθυστερήσεις που αφορούν μεταφορές δεδομένων από τη CPU στη GPU και το αντίστροφο. Οπότε καταφέραμε με την αλλαγή αυτή να εξαλείψουμε και το bottleneck των υπολογισμών της CPU που πλέον γίνονται παράλληλα (ενώ στη CPU γίνονταν σειριακά και με κακό locality όπως αναφέρθηκε και σε προηγούμενο ερώτημα) αλλά και ανεπιθύμητους χρόνους που ξοδεύονταν σε μεταφορές δεδομένων σε κάθε loop αφού πλέον μέχρι να τελειώσει το πρόγραμμα όλα τα χρήσιμα δεδομένα βρίσκονται στη GPU, εξού και η δραματική βελτίωση στο speedup.

- **Τα αντίστοιχα διαγράμματα για config : {Size, Coords, Clusters, Loops} = {256, 2 16, 10}**

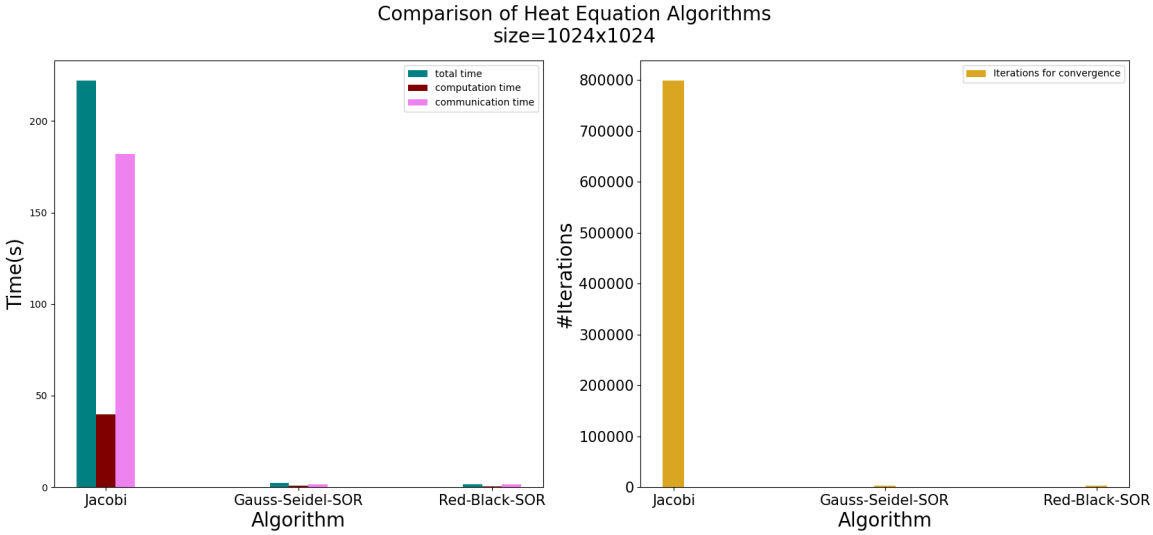




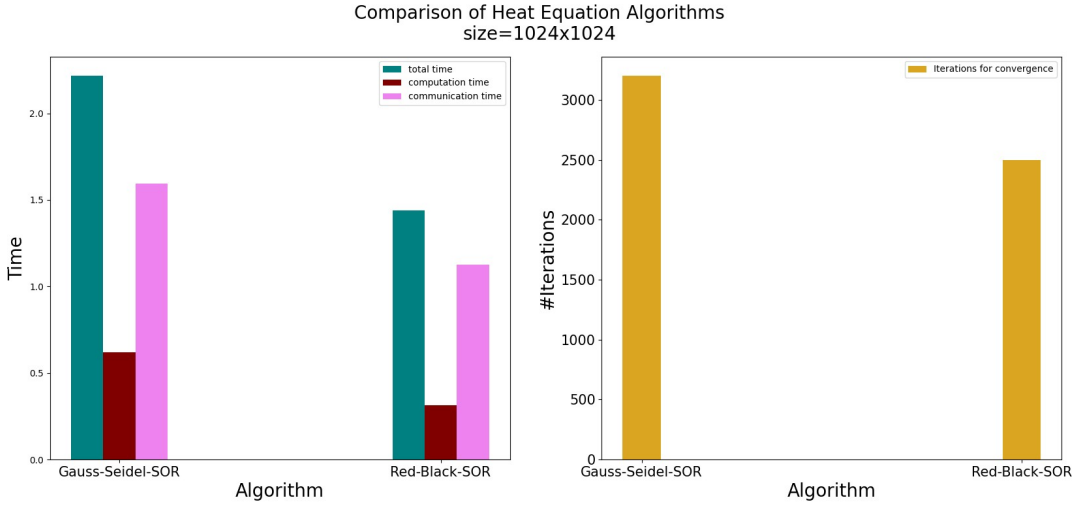
Μεταξύ των 2 configurations παρατηρούμε αντίστοιχες διαφορές με αυτές των προηγούμενων ερωτημάτων, όπως μικρότερη ευαισθησία του configuration με τα 16 coordinates σε αλλαγές του block size πιθανότατα επειδή η υψηλή χρήση shared memory (τώρα ακόμα ψηλότερη) το περιορίζει. Επίσης λόγω του μικρότερου sequential time της έκδοσης με τα 16 coordinates λόγω λιγότερων objects βλέπουμε και μικρότερο speedup.

4.Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

- **Μετρήσεις σε grid=1024x1024 με έλεγχο σύγκλισης**

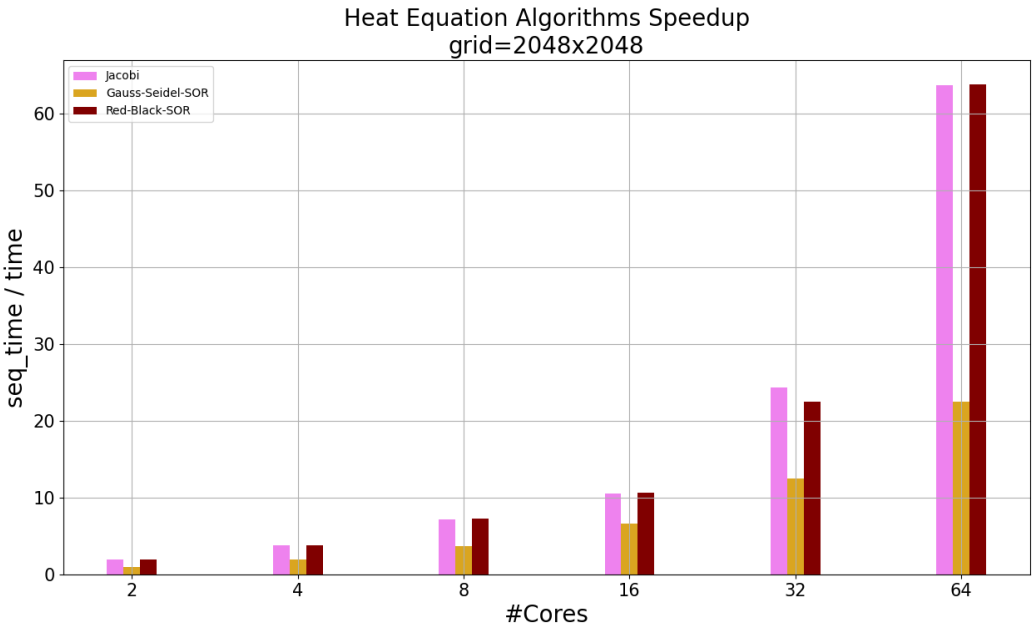


Παρακάτω παραθέτουμε τα συγκριτικά αποτελέσματα για τους τρεις αλγορίθμους: Δεδομένου ότι η διαφορά κλίμακας κρύβει τις λεπτομέρειες για τους δύο ταχύτερους αλγορίθμους δίνουμε εδώ ξεχωριστά τα δικά τους αποτελέσματα:



Προκύπτει ότι οι μέθοδοι Gauss-Seidel-SOR και Red-Black-SOR είναι σημαντικά ταχύτερες σε σχέση με την Jacobi. Η βελτίωση που παρέχουν οι δύο ταχύτερες μέθοδοι κινείται, μάλιστα γύρω στις δύο τάξεις μεγέθους, σύμφωνα με τα αποτελέσματά μας. Αυτό, από προγραμματιστική άποψη δεν είναι αναμενόμενο, δεδομένου ότι οι τρεις αλγόριθμοι δεν έχουν σημαντικές διαφορές από άποψη υπολογιστικής πολυπλοκότητας (εκτελούν παρόμοιου τύπου πράξεις και

επικοινωνίες). Η καλύτερη συμπεριφορά των δύο ταχύτερων αλγορίθμων εξηγείται όμως μαθηματικά, καθώς επιταχύνουν σημαντικά των ρυθμό σύγκλισης. Αυτό αντικατοπτρίζεται στον αριθμό των iterations που εκτελεί κάθε αλγόριθμος μέχρι να επιτύχει σύγκλιση. Πράγματι, ο Jacobi απαιτεί περί των 800.000 iterations για να συγκλίνει την στιγμή που ο Gauss-Seidel και ο Red-Black απαιτούν γύρω στα 3000 και 2500 αντίστοιχα. Συνεπώς, αν και προγραμματιστικά, η δουλειά που απαιτεί κάθε αλγόριθμος ανά iteration είναι παρόμοιου μεγέθους, η σημαντικά ταχύτερη σύγκλιση των δύο καλύτερων αλγορίθμων είναι που τους καθιστά ταχύτερους. Μεταξύ του Gauss-Seidel και του Red-Black υπερισχύει ο Red-Black, αν και με σημαντικά μικρότερης τάξης διαφορά. Η καλύτερη συμπεριφορά του Red-Black οφείλεται και εδώ στο γεγονός ότι εκτελεί λιγότερα iterations για να πετύχει σύγκλιση. Αξίζει μάλιστα να παρατηρήσουμε ότι ο Red-Black έχει μεγαλύτερο κέρδος στο computation time απ' ότι στο communication time σε σχέση με τον Gauss-Seidel. Αυτό βέβαια εξηγείται, αφού ο Red-Black απαιτεί κάποια επιπλέον επικοινωνία. Τελικά η η ταχύτερη μέθοδος, την οποία και θα επιλέγαμε είναι η Red-Black-SOR.



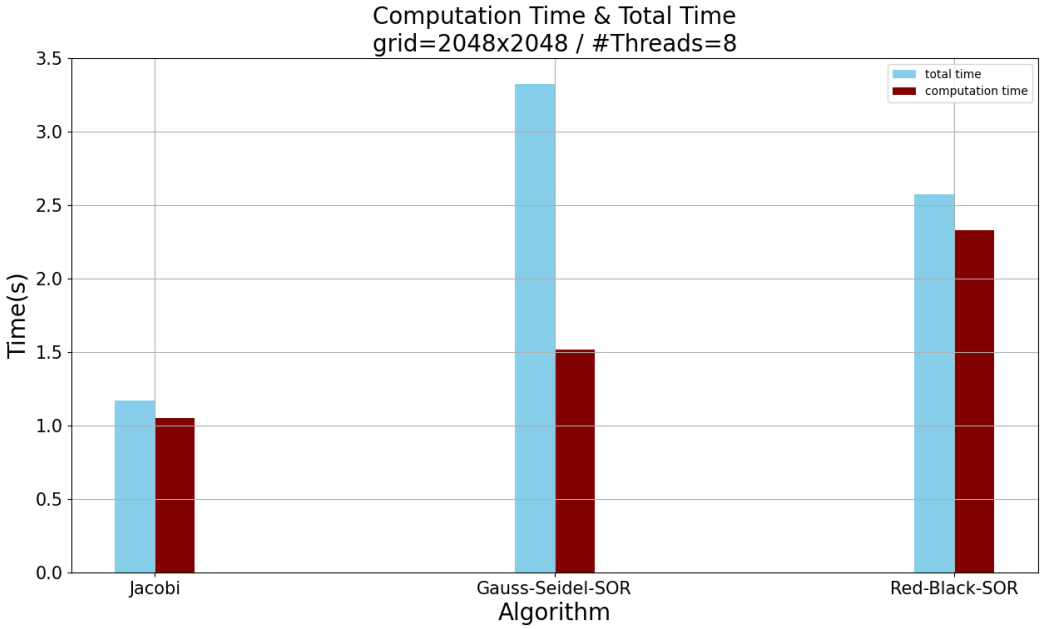
- **Μετρήσεις χωρίς έλεγχο σύγκλισης για grid=2048x2048**
  Στο παραπάνω διάγραμμα Speedup παρατηρούμε αρχικά ότι όλες οι μέθοδοι επιταχύνονται καθώς αυξάνουμε τον αριθμό των cores (κάνουν scale). Συνεπώς, σε όλες τις περιπτώσεις το όφελος από τον παραλληλισμό είναι μεγαλύτερο από το κόστος της επικοινωνίας. Παρατηρούμε επίσης ότι οι μέθοδοι Jacobi και Red-Black κάνουν πολύ καλύτερο scaling σε σχέση με την Gauss-Seidel, στην οποία έχουμε σημαντικά πιο αργή επιτάχυνση, καθώς προσθέτουμε cores. Αυτό θα μπορούσαμε να το αποδόσουμε στο γεγονός ότι στις μεθόδους Jacobi και Red-Black οι διεργασίες επιτελούν λειτουργίες σε κάθε iteration με πιο "συγχρονισμένο" τρόπο σε σχέση με την Gauss-Seidel.

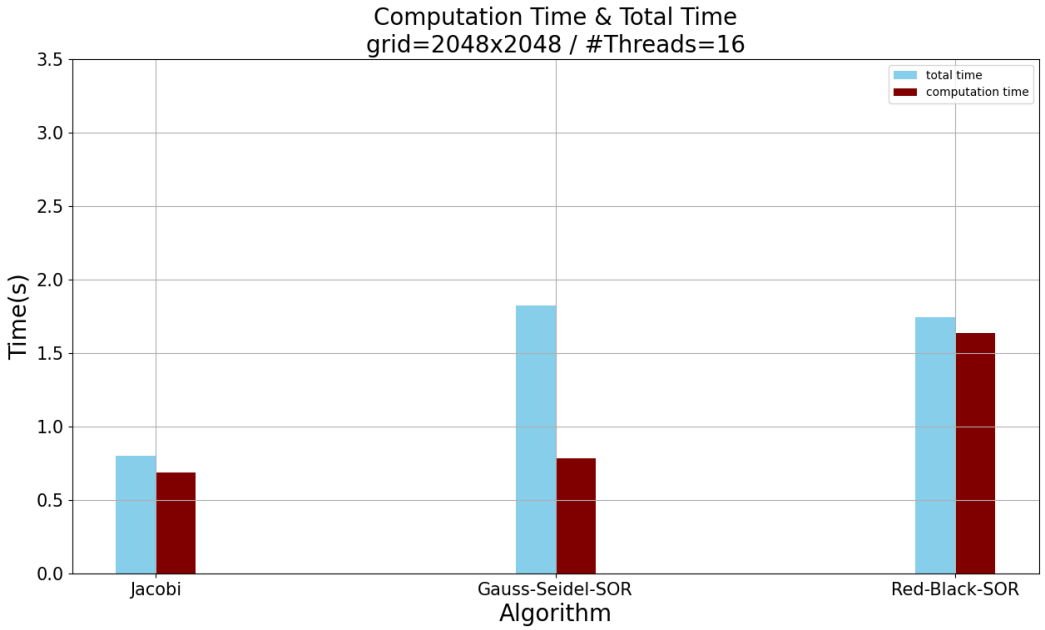
Συγκεκριμένα στην μέθοδο Jacobi σε κάθε iteration όλες οι διεργασίες περνάνε από μία φάση που ανταλλάσσουν δεδομένα με τους γείτονες τους και στη συνέχεια όλες μπαίνουν σε μια φάση υπολογισμών. Όλες οι διεργασίες είναι συμμετρικές ως προς τον χρόνο που μπαίνουν σε αυτές τις φάσεις και συνεπώς λειτουργούν συγχρονισμένα.

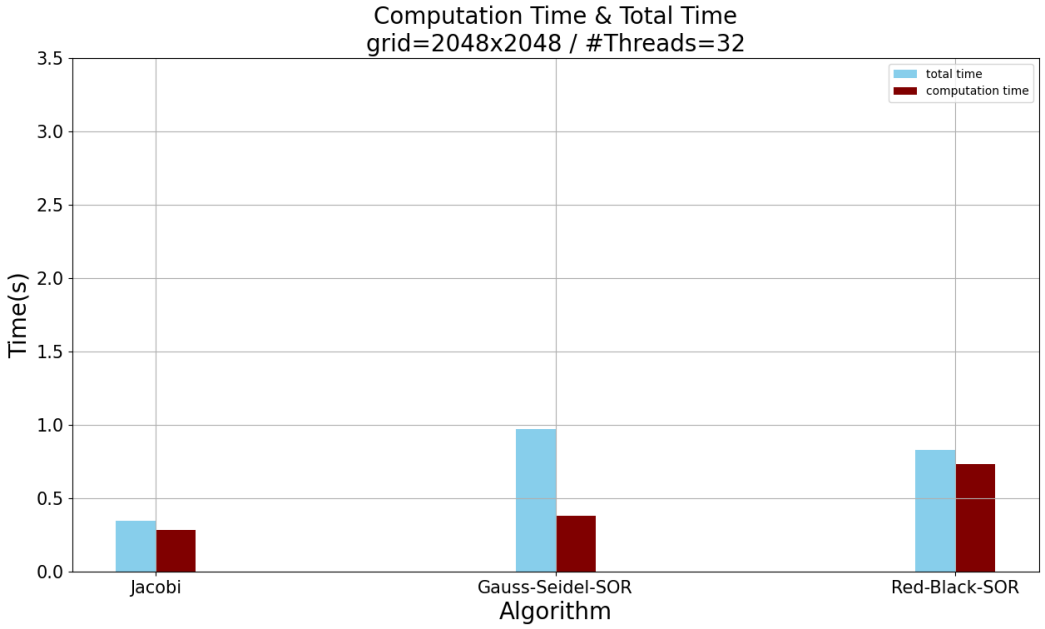
Αντίστοιχα στην μέθοδο Red-Black, αρχικά όλες οι διεργασίες ανταλλάζουν δεδομένα (του προηγούμενου time step) με τους γείτονές τους. Στη συνέχεια κάνουν όλες τους "κόκκινους" υπολογισμούς. Μετά ανταλλάσσουν όλες δεδομένα του τρέχοντος time step και τέλος μπαίνουν όλες στην φάση των "μαύρων" υπολογισμών. Υπάρχει και πάλι χρονική συμμετρία στο πώς μπαίνουν οι διεργασίες σε αυτές τις φάσεις.

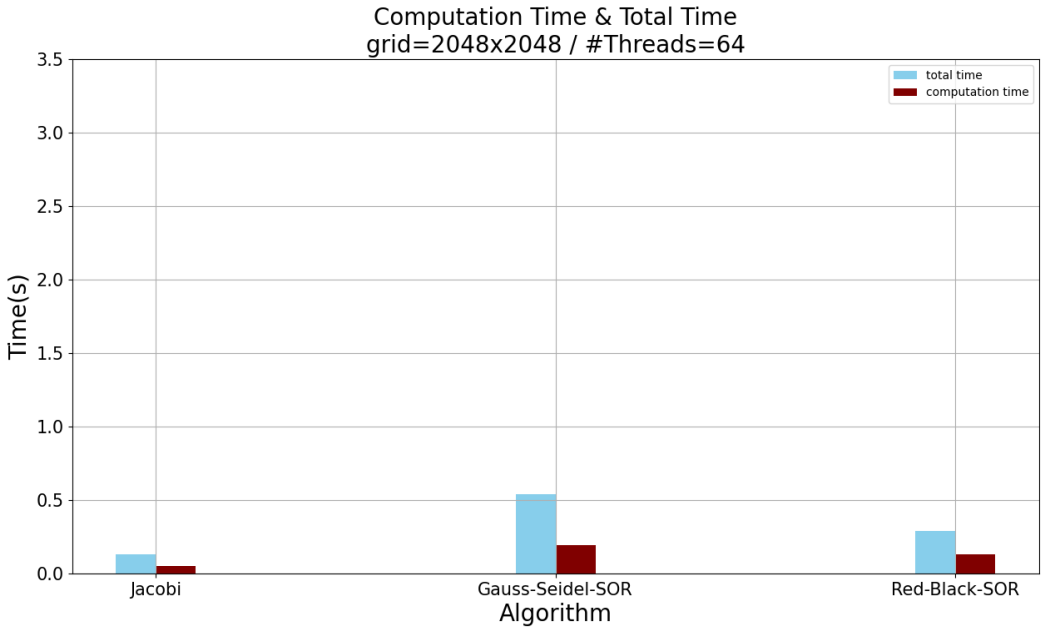
Από την άλλη, για την μέθοδο Gauss-Seidel, λόγω των εξαρτήσεων κάθε κελιού του πίνακα από τα αριστερότερο και το από πάνω, στο ίδιο χρονικό βήμα δεν υπάρχει συμμετρία στην επικοινωνία σε κάθε iteration. Συγκεκριμένα, πρέπει πρώτα η πάνω αριστερά διεργασία να ολοκληρώσει τους υπολογισμούς της και να διαδώσει τα δεδομένα της προς τα δεξια και προς τα κάτω. Το ίδιο πρέπει να κάνει στη συνέχεια και κάθε διεργασία (να διαδώσει τα αποτελέσματά της δεξιά και κάτω) αφού ολοκληρώσει τους υπολογισμούς της. Ωστόσο μια διεργασία δεν μπορεί να ξεκινήσει πριν λάβει τους υπολογισμούς τους τρέχοντος timestep από την αριστερή της και την από πάνω της. Λόγω αυτών των αλυσιδωτών εξαρτήσεων μεταξύ εργασιών, η πιο κάτω-δεξιά διεργασία καταλήγει να περιμένει πολύ για δεδομένα και συνολικά το κάθε iteration κρατάει περισσότερο απ' ότι στις άλλες μεθόδους, που οι διεργασίες επικοινωνούν και υπολογίζουν με συγχρονισμένο τρόπο.

Ακολουθούν τα διαγράμματα χρόνου για τους 3 αλγορίθμους:



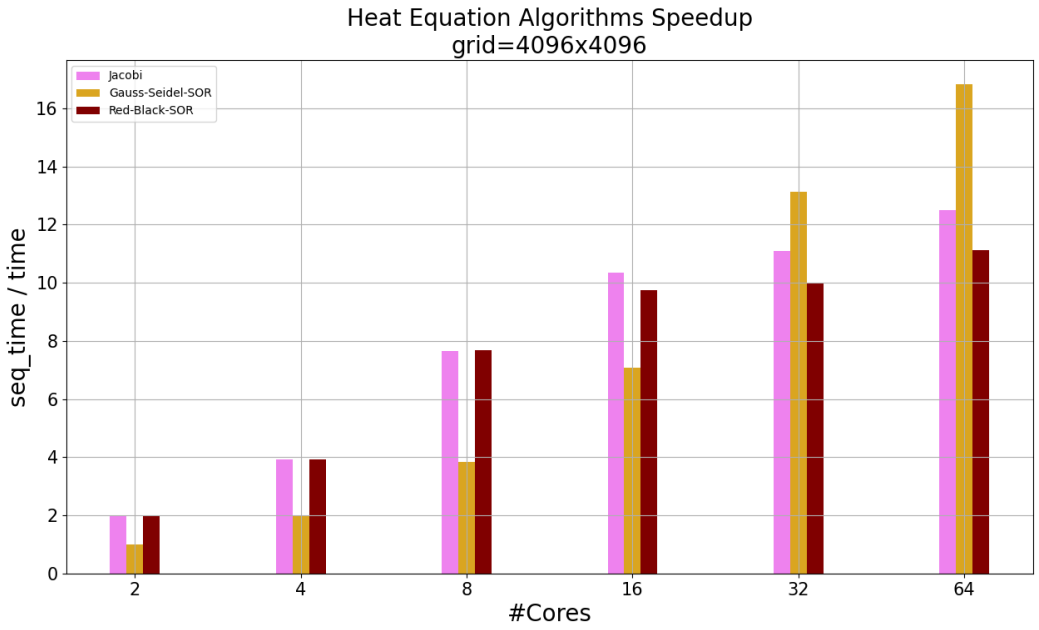

Computation Time & Total Time
grid=2048x2048 / #Threads=16
total time
computation time
Time(s)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm


Computation Time & Total Time
grid=2048x2048 / #Threads=32
total time
computation time
Time(s)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm

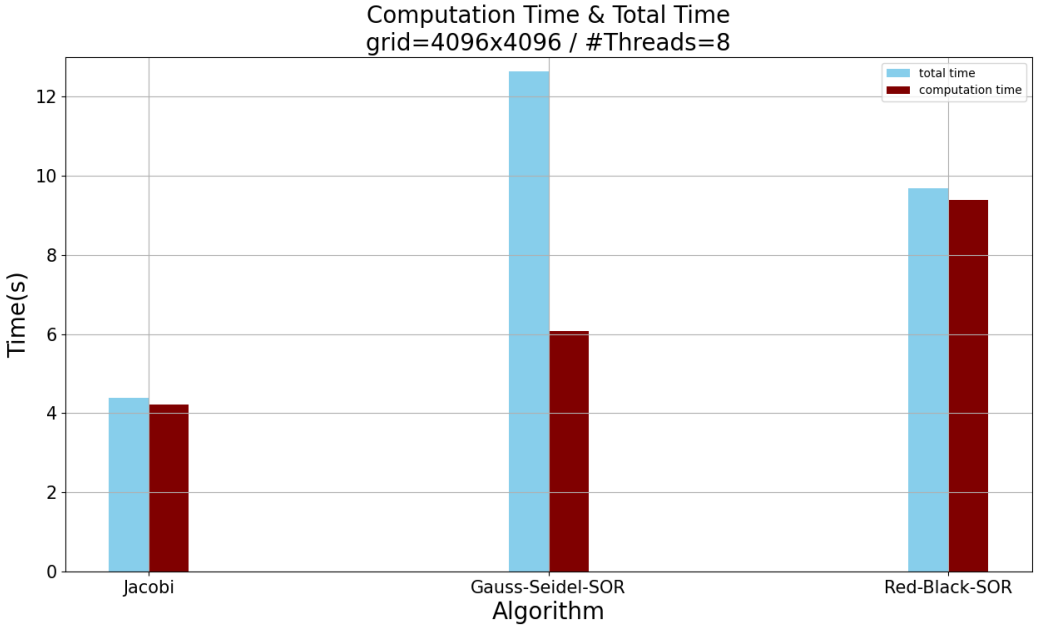


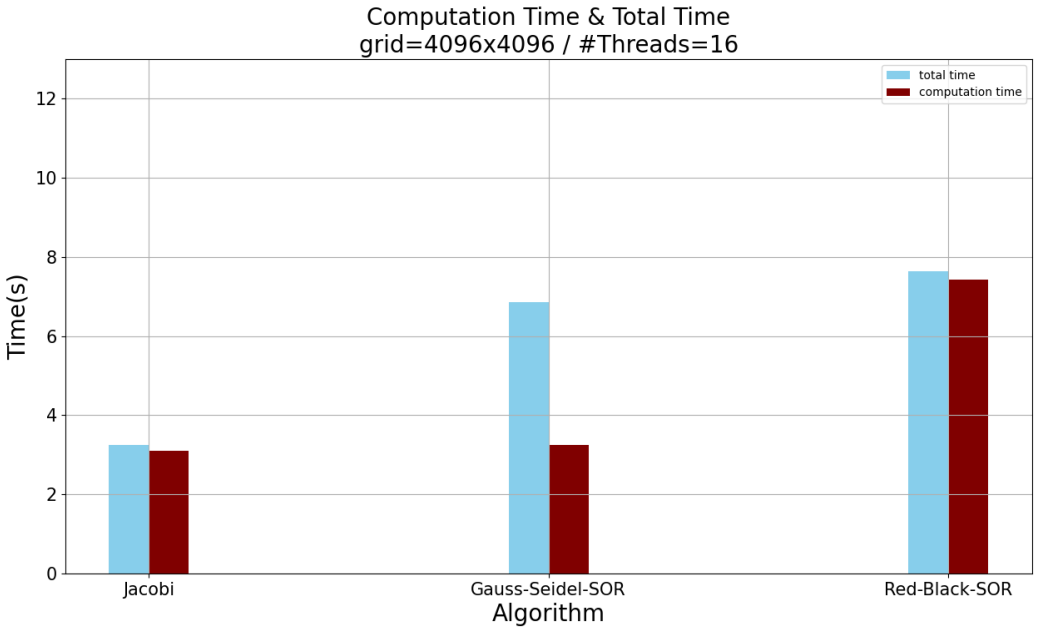
Αρχικά, παρατηρούμε ότι ο χρόνος επικοινωνίας (διαφορά total από computation time) για τον Gauss-Seidel είναι δυσανάλογα μεγάλος σε όλες τις περιπτώσεις. Αυτό επαληθεύει την παρατήρηση που κάναμε νωρίτερα σχετικά με την μη συμμετρική επικοινωνία αυτού του αλγορίθμου που κάνει τις "κάτω-δεξιές" διεργασίες να περιμένουν πολύ. Παρατηρούμε επίσης (κυρίως για τον Jacobi και τον Red-Black) ότι καθώς αυξάνουμε τα cores το computation time γίνεται όλο και μικρότερο ποσοστό του συνολικού χρόνου, το οποίο είναι απολύτως λογικό, αφού αυξάνουμε την επικοινωνία και αυξάνουμε τον παραλληλισμό στους υπολογισμούς. Τέλος, αξίζει να τονίσουμε και εδώ ότι για σταθερό αριθμό επαναλήψεων ο Jacobi είναι ταχύτερος από όλους τους αλγορίθμους, έχει δηλαδή τον μικρότερο χρόνο ανά επανάληψη. Η υπεροχή των άλλων αλγορίθμων, όπως ξαναείπαμε οφείλεται στον σημαντικά ταχύτερο ρυθμό σύγκλισης.

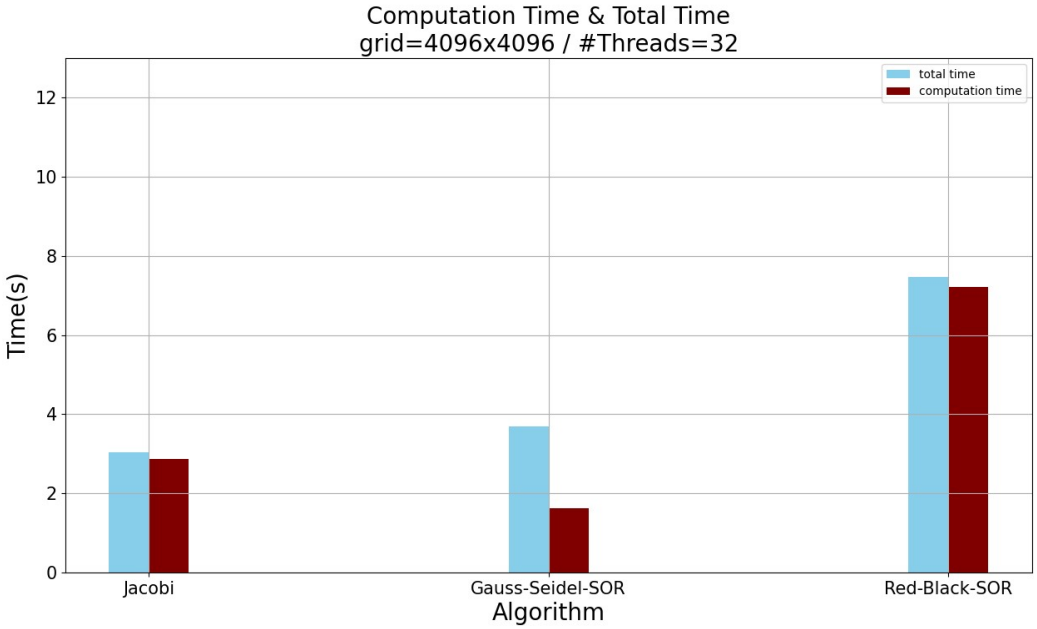
- **Μετρήσεις χωρίς έλεγχο σύγκλισης για grid=4096x4096**

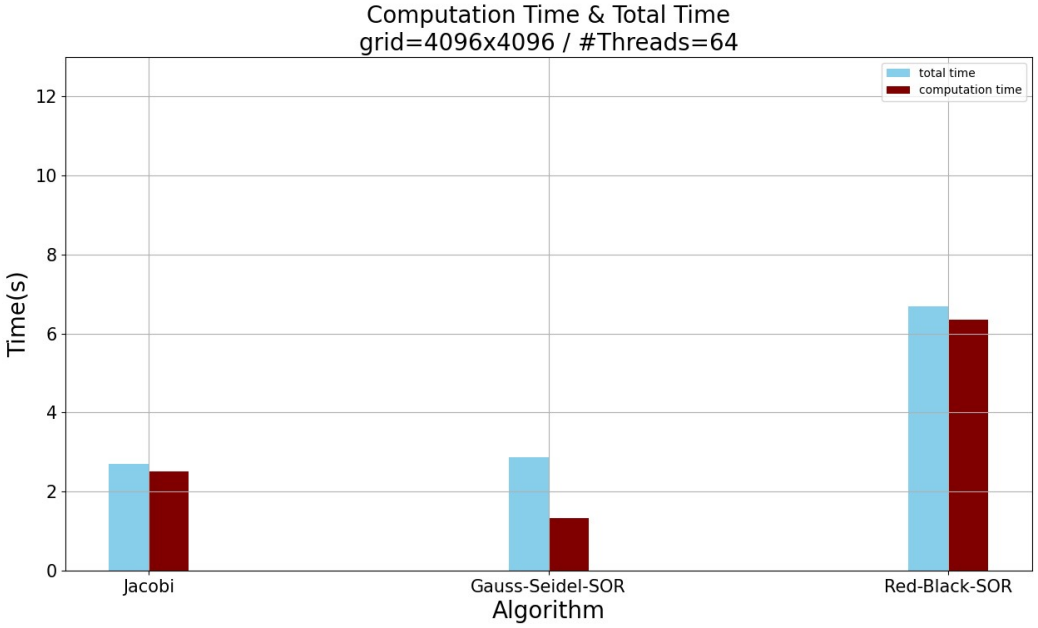


Στο 4096x4096 grid παρατηρούμε αρκετά διαφορετική συμπεριφορά σε σχέση με το προηγούμενο. Εδώ οι μέθοδοι Jacobi και Red-Black πρακτικά σταματούν να κάνουν scale στα 16 threads (ενώ πριν κάνανε μέχρι τα 64). Η μέθοδος Gauss-Seidel κάνει scale όπως και στην προηγούμενη περίπτωση. Υποθέτουμε ότι η διαφορετική συμπεριφορά οφείλεται σε επιβράδυνση, λόγω των λειτουργιών πρόσβασης στη μνήμη. Συγκεκριμένα, για τις μεθόδους Red-Black και Jacobi, το scaling σταματά στα 16 threads. Δεδομένου ότι η ουρά parlab έχει 8 nodes με 2 τετραπύρηνες CPUs στον καθένα, αυτός είναι και ο μεγαλύτερος αριθμός threads, για τον οποίο έχουμε ένα thread σε κάθε CPU. Για 32 και 64 threads χρησιμοποιούμε πολλαπλά cores σε κάθε CPU. Συνεπώς είναι λογικό να υποθέσουμε ότι, τώρα που έχουμε μεγαλύτερο πίνακα και μικρότερο μέρος του χωράει στην Cache κάθε CPU, η εκτέλεση πολλαπλών threads σε κάθε CPU προκαλεί ανταγωνισμό για την χρήση της cache και δεν επιτρέπει το scaling για παραπάνω από 16 threads. Από την άλλη στην μέθοδο Gauss-Seidel τα διαφορετικά threads δεν εκτελούν υπολογισμούς εντελώς συγχρονισμένα, όπως περιγράψαμε νωρίτερα. Συνεπώς δεν δημιουργούνται οι συνθήκες ανταγωνισμού για την Cache, με αποτέλεσμα να διατηρείται η ίδια δυνατότητα scaling και για τον μεγαλύτερο πίνακα.


Computation Time & Total Time
grid=4096x4096 / #Threads=8
total time
computation time
Time(s)
0
2
4
6
8
10
12
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm

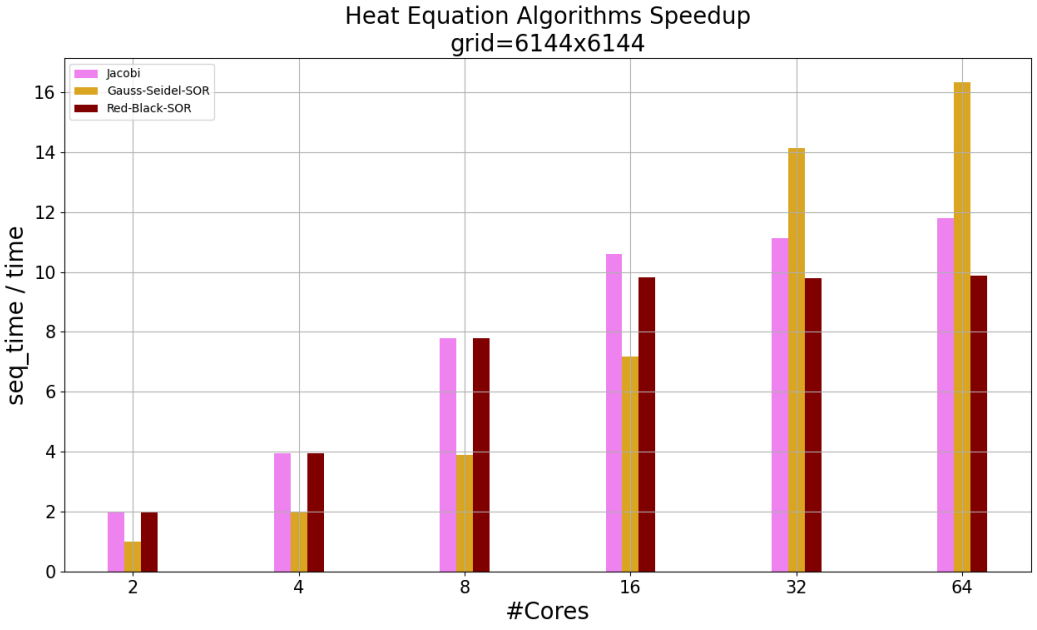

Computation Time & Total Time
grid=4096x4096 / #Threads=16
total time
computation time
Time(s)
0
2
4
6
8
10
12
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm

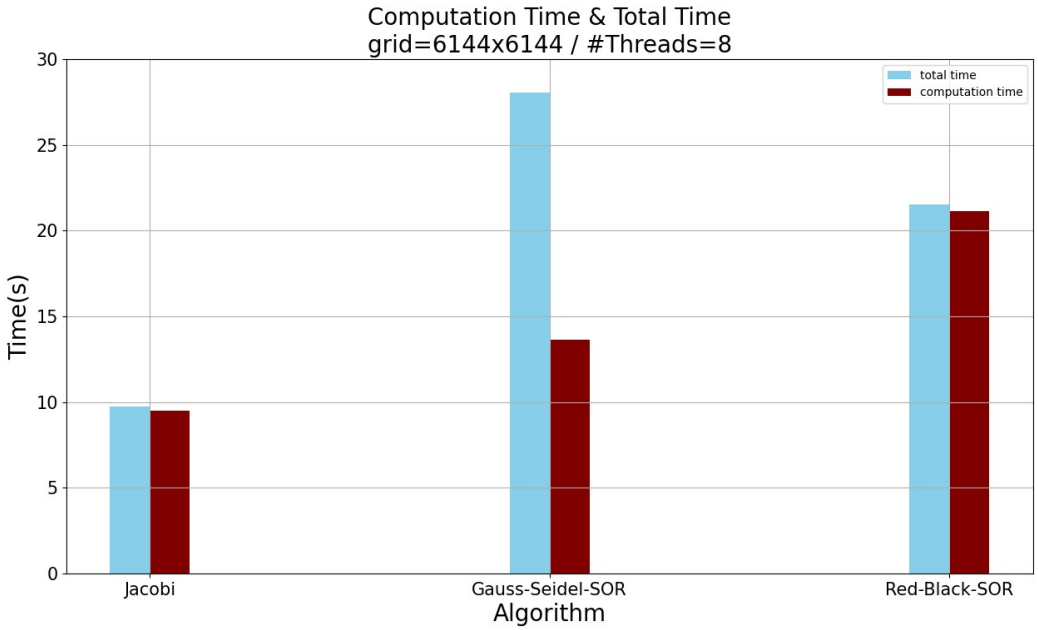

Computation Time & Total Time
grid=4096x4096 / #Threads=32
total time
computation time
Time(s)
0
2
4
6
8
10
12
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm

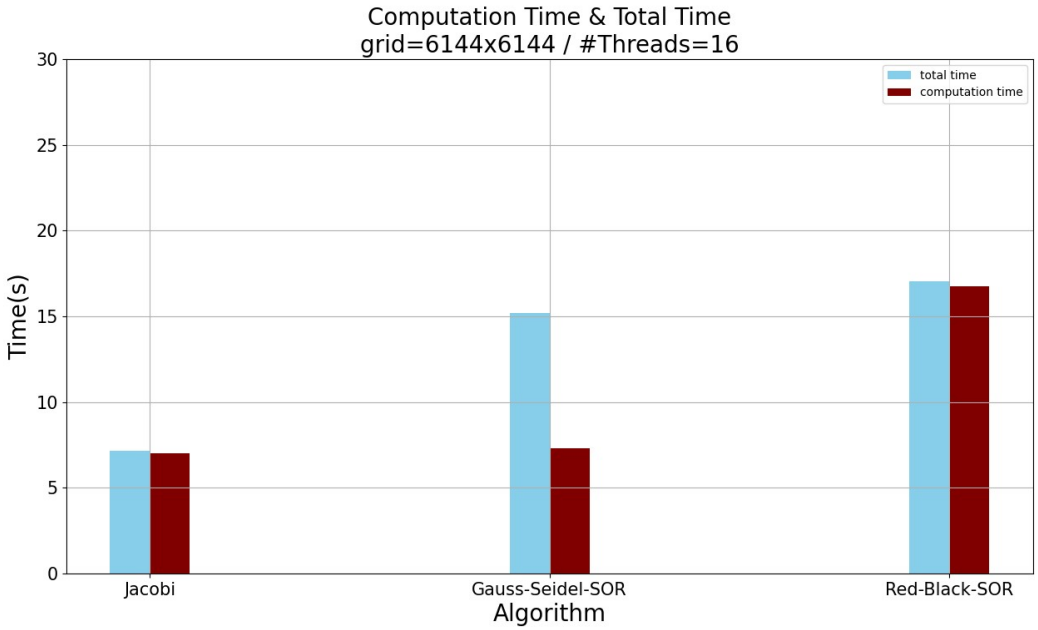

Computation Time & Total Time
grid=4096x4096 / #Threads=64
total time
computation time
Time(s)
0
2
4
6
8
10
12
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm

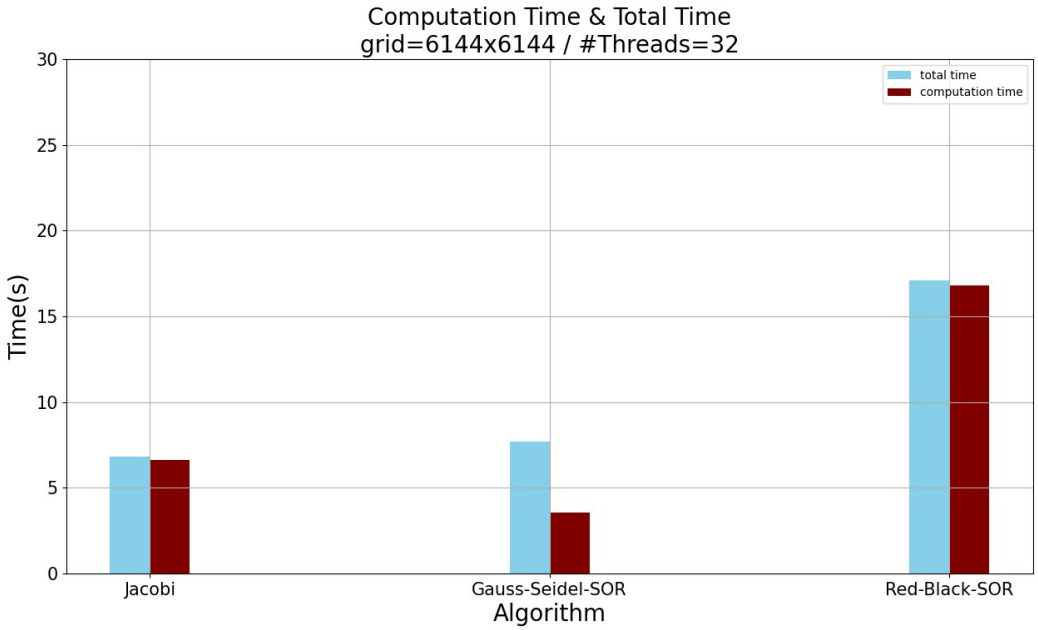
Παρατηρώντας τα διαγράμματα χρόνου μπορούμε να επιβεβαιώσουμε την υπόθεση που κάναμε πιο πριν. Βλέπουμε ότι το “μπλοκάρισμα” στο Speedup για τον Jacobi και τον Red-Black οφείλεται στο στον χρόνο υπολογισμών και όχι στον χρόνο επικοινωνίας (ο χρόνος επικοινωνίας για τις δύο αυτές μεθόδους είναι πολύ μικρό ποσοστό του συνολικού χρόνου εκτέλεσης). Συνεπώς είναι λογική η σκέψη που κάναμε για την καθυστέρηση εξαιτίας των συγκρούσεων στην Cache. Από την άλλη, ο Gauss-Seidel, στις περιπτώσεις των 32 και 64 threads, καταφέρνει να κατεβάσει τον χρόνο υπολογισμών του σε χαμηλότερα επίπεδα από τις άλλες δύο μεθόδους, ενώ έχει σημαντικά μεγαλύτερη καθυστέρηση, εξαιτίας της επικοινωνίας (αναμονής κάποιων threads για τα δεδομένα των προηγούμενων). Πάντως έχει ενδιαφέρον, ότι εδώ ο Gauss-Seidel πετυχαίνει πρακτικά ίδιο χρόνο ανά επανάληψη με τον Jacobi, ο οποίος ήταν με διαφορετικά ο ταχύτερος (ανά επανάληψη) για τον πίνακα 2048x204.

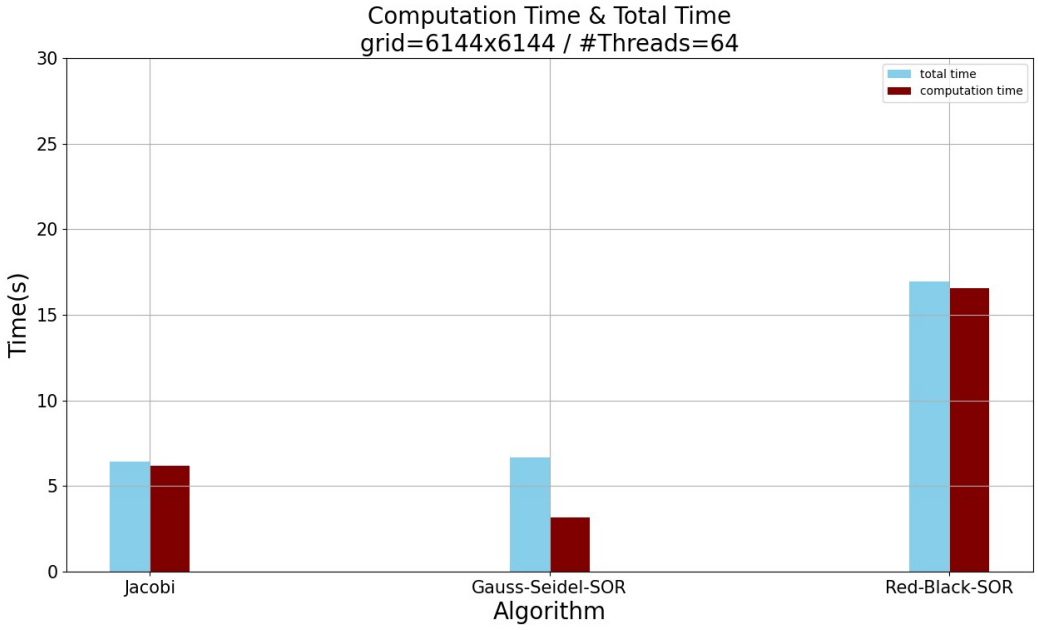
- **Μετρήσεις χωρίς έλεγχο σύγκλισης για grid=6144x6144**




Computation Time & Total Time
grid=6144x6144 / #Threads=8
total time
computation time
Time(s)
0
5
10
15
20
25
30
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm


Computation Time & Total Time
grid=6144x6144 / #Threads=16
total time
computation time
Time(s)
0
5
10
15
20
25
30
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm


Computation Time & Total Time
grid=6144x6144 / #Threads=32
total time
computation time
Time(s)
0
5
10
15
20
25
30
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm


Computation Time & Total Time
grid=6144x6144 / #Threads=64
total time
computation time
Time(s)
0
5
10
15
20
25
30
Jacobi
Gauss-Seidel-SOR
Red-Black-SOR
Algorithm

Το διάγραμμα Speedup είναι ακριβώς ίδιο με αυτό της περίπτωσης 4096x4096, ενώ τα διαγράμματα χρόνου εκτέλεσης έχουν ίδια μορφή (με διαφορά βέβαια στους χρόνους). Οι αλγόριθμοι δηλαδή έχουν ακριβώς ίδια συμπεριφορά με το προηγούμενο μέγεθος grid. Αυτό συνάδει με τις υποθέσεις που κάναμε πριν για το μπλοκάρισμα του Speedup που προκαλεί η Cache στις μεθόδους Jacobi και Red-Black, κάτι που δεν εμφανίζεται στην μέθοδο Gauss-Seidel. Είναι αναμενόμενο να συναντάμε αυτό το φαινόμενο και εδώ, δεδομένου ότι ο πίνακας είναι ακόμα μεγαλύτερος.