



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστηριακές Ασκήσεις

OpenMP, MPI, CUDA

Συστήματα Παράλληλης Επεξεργασίας

Ανδρέας Στάμος

Αριθμός μητρώου: 03120***

Διεύθυνση ηλεκτρονικού ταχυδρομείου: stamos.aa@gmail.com

Περιεχόμενα

Περιεχόμενα	1
1 1η Άσκηση – Game Of Life – OpenMP σε multicore CPUs	3
1.1 Υλοποίηση	3
1.2 Μετρήσεις επίδοσης	3
1.3 Σχολιασμός	6
1.4 Ενδιαφέρουσες Αρχικοποιήσεις (Patterns) στο Game of Life – προαιρετικό	7
2 2η Άσκηση – Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης	9
2.1 Περιγραφή αρχιτεκτονικής υπολογιστικού συστήματος εκτέλεσης	9
2.2 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means	9
2.2.1 Αφελής παραλληλοποίηση	9
2.2.1.1 Υλοποίηση	9
2.2.1.2 Μετρήσεις επίδοσης – Παρατηρήσεις	9
2.2.1.3 Εκτέλεση με σταθερή απεικόνιση OpenMP νημάτων σε νήματα του υλικού	11
2.2.1.4 Σχολιασμοί – Συμπεράσματα	13
2.2.2 Reduction παραλληλοποίηση	15
2.2.2.1 Αρχική υλοποίηση	15
2.2.2.2 Μετρήσεις επίδοσεις – παρατηρήσεις – πρώτα σχόλια	15
2.2.2.3 Μετρήσεις επίδοσης σε διαφορετικό configuration	17
2.2.2.4 Σχολιασμός	18
2.2.2.5 Διορθωμένη υλοποίηση	19
2.2.2.6 Μετρήσεις επίδοσης – παρατηρήσεις στην διορθωμένη υλοποίηση	19
2.2.2.7 Σχολιασμός στην διορθωμένη υλοποίηση	21
2.2.3 Υλοποίηση με NUMA-aware allocation	24
2.2.3.1 Σκεπτικό	24
2.2.3.2 Υλοποίηση	24
2.2.3.3 Μετρήσεις επίδοσης – παρατηρήσεις	25
2.2.3.4 Σχολιασμός	28
Υπόθεση bottleneck στο εύρος ζώνης διαύλου μνήμης – μάλλον καταρρίπτεται	28
Υπόθεση bottleneck στο σειριακό τμήμα – καταρρίπτεται	28
Υπόθεση bottleneck στον υπολογισμό από την CPU – μάλλον καταρρίπτεται	29
2.2.4 Reduction παραλληλοποίηση αυτόματα με OpenMP Reduction Directive	30
2.2.4.1 Υλοποίηση	30
2.2.4.2 Απομακρυσμένη μεταγλώττιση από τοπικό υπολογιστή για την υποδομή του Εργαστηρίου	30
2.3 Αξιολόγηση διαφορετικών υλοποιήσεων Κλειδωμάτων στον αλγόριθμο K-means	31
2.3.1 Περιγραφή – εξήγηση λειτουργίας κλειδωμάτων	31
2.3.2 Μετρήσεις	32
2.3.3 Παρατηρήσεις – συμπεράσματα	35
2.4 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall	36
2.4.1 Μετρήσεις για διάφορα block sizes στην σειριακή εκδοχή – παρατηρήσεις	36
2.4.2 Υλοποίηση παραλληλοποίησης	38
2.4.3 Μετρήσεις για διάφορα πλήθη πυρήνων στην παράλληλη εκδοχή	38
2.4.4 Μετρήσεις για διάφορα block sizes στην παράλληλη εκδοχή	41
2.4.5 Παρατηρήσεις – συμπεράσματα	43
2.4.6 Εξήγηση για την έλλειψη κλιμάκωσης	43
2.4.7 Tiled Αλγόριθμος: Υλοποίηση, Μετρήσεις, Σχόλια	47
2.5 Ταυτόχρονες Δομές Δεδομένων	49
2.5.1 Περιγραφή	49
2.5.2 Μετρήσεις – συμπεράσματα	49
3 3η Άσκηση – Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών – Αλγόριθμος KMeans	58
3.1 Naive υλοποίηση	58
3.1.1 Περιγραφή της υλοποίησης	58
3.1.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1o)	58
3.1.3 Παρατηρήσεις – σχόλια (Ερώτημα 2o – α' μέρος)	59

3.1.4	Παρατηρήσεις – σχόλια (Ερώτημα 2ο – β' μέρος)	59
3.1.5	Σχολιασμός του block size (Ερώτημα 3ο)	60
3.2	Transpose υλοποίηση	60
3.2.1	Περιγραφή της υλοποίησης	60
3.2.2	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1ο – α' μέρος)	61
3.2.3	Παρατηρήσεις – σχόλια (Ερώτημα 1ο – β' μέρος)	62
3.2.4	Ερμηνεία βελτίωσης επίδοσης έναντι της παίνε υλοποίησης (Ερώτημα 2ο)	63
3.3	Shared Memory υλοποίηση	63
3.3.1	Περιγραφή της υλοποίησης	63
3.3.2	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση	63
3.3.3	Παρατηρήσεις – σχόλια	65
3.3.4	Απεικόνιση των επιδόσεων όλων των υλοποιήσεων (Ερώτημα 1ο – α' μέρος)	65
3.3.5	Επίδραση του block size (Ερώτημα 1ο – β' μέρος)	67
3.4	Σύγκριση υλοποίησεων – Bottleneck Analysis	67
3.4.1	Bottleneck στο iterative μέρος του αλγορίθμου (Ερώτημα 1ο)	67
3.4.2	Μετρήσεις Επίδοσης και Διαγραμματική Απεικόνιση για 2ο Configuration για όλες τις υλοποιήσεις (Ερώτημα 2ο – α' μέρος)	68
3.4.3	Παρατηρήσεις – σχόλια μεταξύ των δύο Configurations (Ερώτημα 2ο – β' μέρος)	73
3.4.4	Καταλληλότητα shared memory GPU υλοποίησης για arbitrary Configurations (Ερώτημα 2ο – γ' μέρος)	73
3.4.5	Επιλογή block size με την συνάρτηση cudaOccupancyMaxPotentialBlockSize (Ερώτημα 3ο – BONUS 1)	73
3.5	All-GPU Υλοποίηση	74
3.5.1	Περιγραφή της υλοποίησης	74
3.5.2	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1ο – α' μέρος)	74
3.5.3	Παρατηρήσεις – σχόλια (Ερώτημα 1ο – β' μέρος)	77
3.5.4	Επίδραση του block size (Ερώτημα 2ο)	77
3.5.5	Καταλληλότητα του τμήματος update_centroids για GPU (Ερώτημα 3ο)	77
3.5.6	Διαφορά επίδοσης μεταξύ των δύο Configurations (Ερώτημα 4ο)	78
3.6	All-GPU Delta Reduction Υλοποίηση (BONUS 2)	79
3.6.1	Περιγραφή της υλοποίησης	79
3.6.2	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1ο – α' μέρος)	79
3.6.3	Παρατηρήσεις – Σχόλια (Ερώτημα 1ο – β' μέρος)	82
3.6.4	Επίδραση Block Size (Ερώτημα 2ο)	82
4	4η Άσκηση – Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης – MPI	83
4.1	Αλγόριθμος Kmeans	83
4.1.1	Περιγραφή της υλοποίησης	83
4.1.2	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση	83
4.1.3	Παρατηρήσεις – Σχόλια	85
4.1.4	Σύγκριση με την 2η Άσκηση (υλοποίηση σε αρχιτεκτονική κοινής μνήμης) – BONUS	85
4.2	Αριθμητική Επίλυση της Εξίσωσης Θερμότητας σε 2Δ ορθογώνιο χωρίο (με 3 μεθόδους)	85
4.2.1	Περιγραφή της υλοποίησης των 3 αριθμητικών μεθόδων	85
4.2.2	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (για σταθερό πλήθος επαναλήψεων – χωρίς ελέγχο σύγκλισης)	86
4.2.3	Παρατηρήσεις – σχόλια	95
4.2.4	Μετρήσεις επίδοσης και διαγραμματική απεικόνιση για πλήθος επαναλήψεων μέχρι την σύγκλιση	97
4.2.5	Παρατηρήσεις – σχόλια – δεδομένης και της ταχύτητας σύγκλισης	98
Κατάλογος σχημάτων		99

1 1η Άσκηση – Game Of Life – OpenMP σε multicore CPUs

1.1 Υλοποίηση

Η παραλληλοποίηση έγινε στο τμήματα που υπολογίζεται το νέο grid σε κάθε χρονικό βήμα. Κάθε επεξεργαστής αναλαμβάνει να υπολογίσει ένα μέρος των γραμμών του πίνακα. Προσέχουμε να δώσουμε συνεχές τμήμα γραμμών σε κάθε επεξεργαστή (αντί π.χ. να κάνουμε interleaving), ώστε να αυξήσουμε την χωρική τοπικότητα των προσβάσεων αλλά και επιπλέον ώστε να μειώσουμε το false sharing (sharing γίνεται μόνο στην πρώτη και τελευταία γραμμή του τμήματος κάθε επεξεργαστή). Το `schedule(static)`, όπου δεν δίνεται αριθμός, εξασφαλίζει αυτό ακριβώς.

Προκειμένου, να υλοποιηθεί αυτό, έγιναν τα παρακάτω:

1. Προστέθηκε ακριβώς πριν από τον βρόχο για την γραμμές του πίνακα (πάνω από το `for` με την επαγγική μεταβλητή `i`) το OpenMP directive:
`#pragma omp parallel for schedule(static)`
2. Οι επαγγικές μεταβλητές `i`, `j` καθώς και η μεταβλητή `nbrs` δηλώθηκαν εντός του scope του `for`, καθώς τότε το OpenMP τις θεωρεί ως `private`. (από προεπιλογή, οι μεταβλητές θεωρούνται `shared`, οπότε οι πίνακες αυτόματα θεωρούνται `shared`.)

1.2 Μετρήσεις επίδοσης

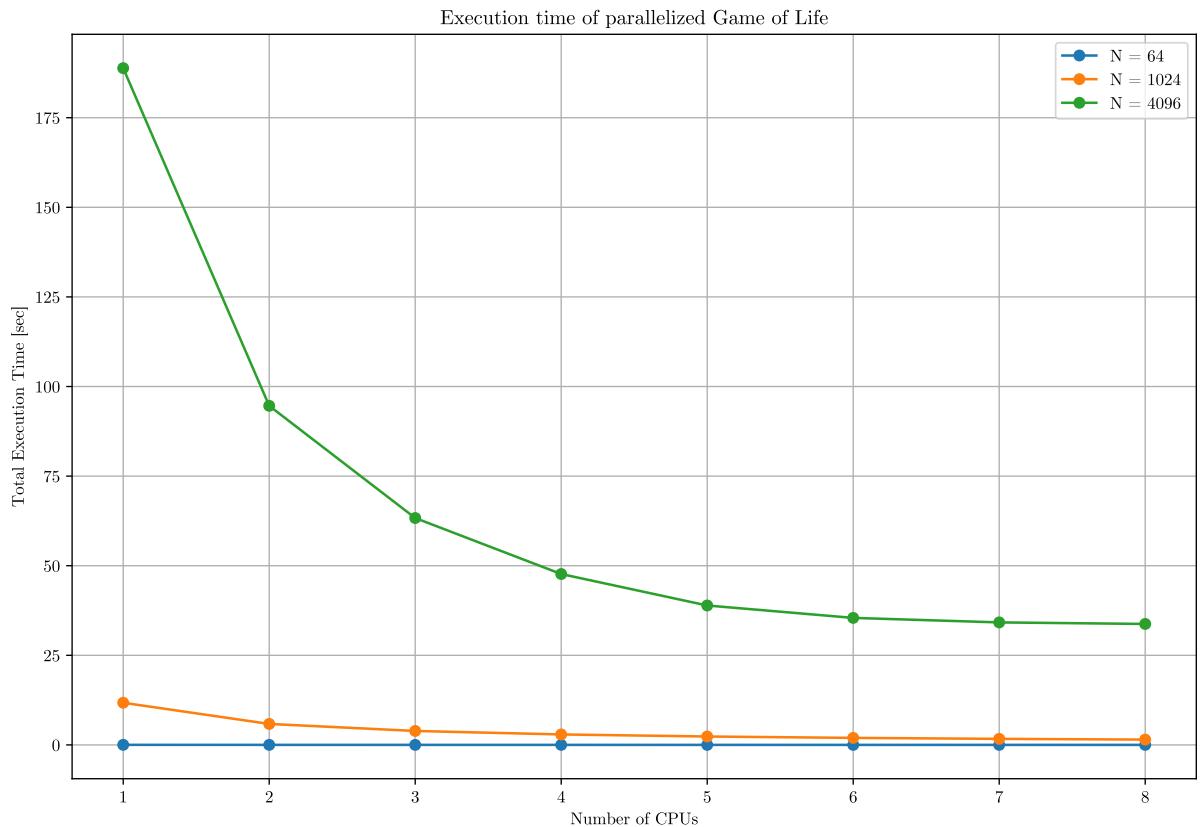
Στην συνέχεια έγιναν μετρήσεις χρόνου εκτέλεσης για τους συνδυασμούς:

$$(\text{πλήθος CPUs}, \text{διάσταση } N \text{ πίνακα}) = \{1, 2, 3, 4, 5, 6, 7, 8\} \times \{64, 1024, 4096\}$$

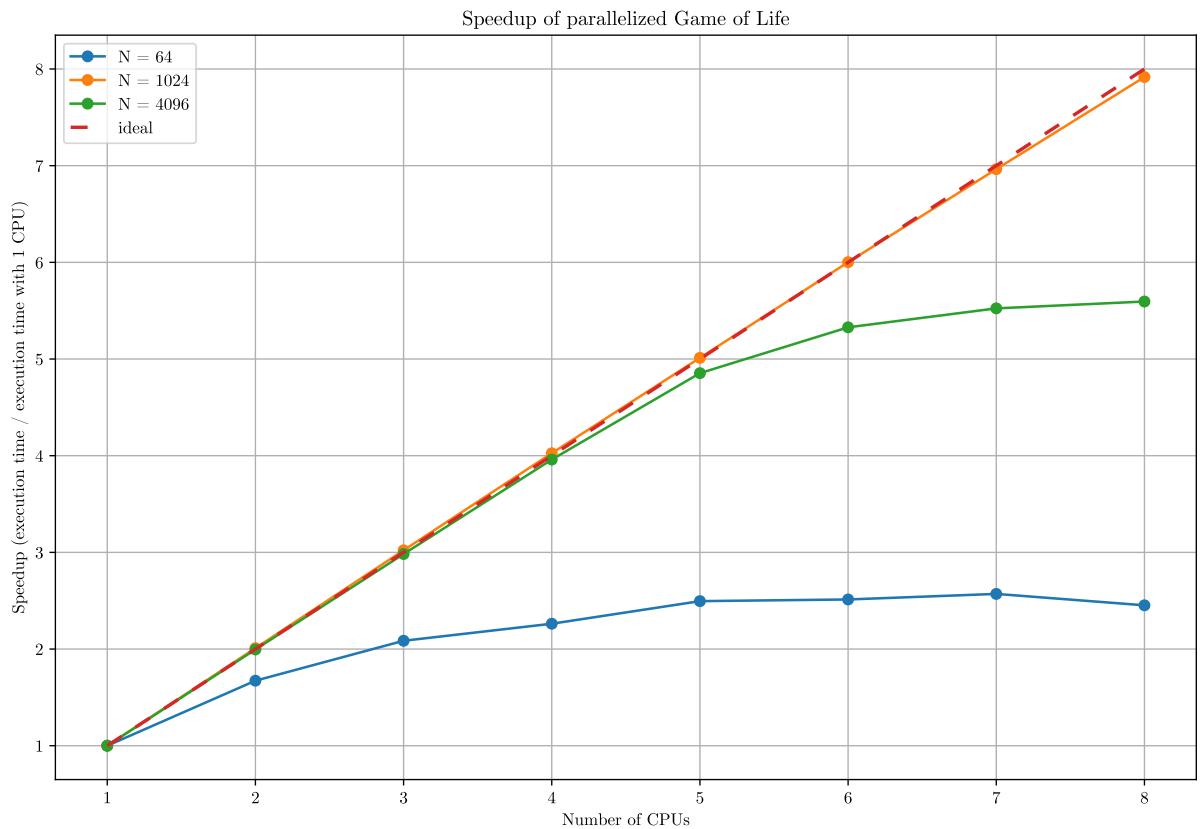
Τα αποτελέσματα για τον ολικό χρόνο εκτέλεσης παρουσιάζονται στο σχήμα 1.

Επιπρόσθετα παρουσιάζουμε τον συντελεστή επιτάχυνσης στο σχήμα 2.

Οι μετρήσεις επίσης φαίνονται συνολικά και στον πίνακα 1.



Σχήμα 1: Μετρήσεις ολικού χρόνου εκτέλεσης για το παραλληλοποιημένο Game of Life



Σχήμα 2: Μετρήσεις επιτάχυνσης για το Game of Life για το παραλληλοποιημένο Game of Life

N	#(CPUs)	Ολικός χρόνος εκτέλεσης [sec]	Επιτάχυνση
64	1	0.02	1.00
64	2	0.01	1.67
64	3	0.01	2.09
64	4	0.01	2.26
64	5	0.01	2.50
64	6	0.01	2.51
64	7	0.01	2.57
64	8	0.01	2.45
1024	1	11.77	1.00
1024	2	5.86	2.01
1024	3	3.89	3.02
1024	4	2.92	4.03
1024	5	2.35	5.01
1024	6	1.96	6.00
1024	7	1.69	6.96
1024	8	1.49	7.92
4096	1	188.83	1.00
4096	2	94.61	2.00
4096	3	63.31	2.98
4096	4	47.68	3.96
4096	5	38.90	4.85
4096	6	35.44	5.33
4096	7	34.18	5.52
4096	8	33.75	5.59

Πίνακας 1: Ολικός χρόνος εκτέλεσης και επιτάχυνση στο παραλληλοποιημένο Game of Life

1.3 Σχολιασμός

Θα σχολιάσουμε την κλιμακωσιμότητα της παραλληλοποίησης, βασιζόμενοι στις μετρήσεις της επιτάχυνσης, όπως φαίνονται στην εικόνα 2.

Θεωρητικά η επιτάχυνση πρέπει να είναι ίση με το πλήθος των CPUs. Όταν η επιτάχυνση είναι μικρότερη, η παραλληλοποίηση δεν κλιμακώνει καλά.

Παρατηρούμε πως για τις διάφορες τιμές της διάστασης του πίνακα N , αλλάζει η κλιμακωσιμότητα. Πιο συγκεκριμένα:

Για $N=64$ Η επιτάχυνση δεν κλιμακώνει καλά, λόγω του ανταγωνισμού στην κρυφή μνήμη (cache contention).

Πιο συγκεκριμένα, ανταγωνισμός, συμβαίνει μόνο στην πρώτη και τελευταία γραμμή του μέρος του πίνακα που αναλαμβάνει κάθε επεξεργαστής. Στην περίπτωση που η διάσταση είναι η μικρή, σε κάθε χρονικό βήμα, γίνονται προσβάσεις σε λιγότερες γραμμές. Συνεπώς, για μικρότερη διάσταση, είναι μεγαλύτερο το ποσοστό των προσβάσεων μνήμης που έχουν ανταγωνισμό, οπότε η επίδοση υποφέρει. Το αντίστροφο συμβαίνει για μεγάλη διάσταση.

Για $N=1024$ Η επιτάχυνση κλιμακώνει κατά το θεωρητικό μέγιστο.

Για $N=4096$ Η επιτάχυνση δεν κλιμακώνει καλά σε αντίθεση με το $N = 1024$. Εξηγούμε τον λόγο και έπειτα τεκμηριώνουμε. Ο λόγος είναι πως το μέρος του πίνακα που επεξεργάζεται κάθε επεξεργαστής σταματά να χωρά στην κρυφή μνήμη, οπότε σε κάθε χρονικό βήμα διαβάζεται από την κύρια μνήμη. Πίνακας διάστασης N έχει N^2 στοιχεία, που στην δοθείσα υλοποίηση είναι τύπου int, δηλαδή 4 bytes καθένα. Συνεπώς σε κάθε επεξεργαστή αντιστοιχούν $\frac{4N^2}{\#(CPUs)}$ bytes.

Από το αρχείο /proc/cpuinfo βλέπουμε ότι κάθε επεξεργαστής έχει κρυφή μνήμη 4096 KiB.

Για $N = 4096$, ο πίνακας έχει μέγεθος 64MiB, οπότε ακόμα και για $\#(CPUs) = 8$, σε κάθε επεξεργαστή αντιστοιχούν 8 MiB του πίνακα, που είναι παραπάνω από την κρυφή του μνήμη.

Αντίθετα για $N = 1024$, ο πίνακας έχει μέγεθος 4 MiB, οπότε χωράει ούτως ή άλλως ολόκληρος στην κρυφή μνήμη κάθε επεξεργαστή.

Συνεπώς, επιχειρηματολογούμε πως για $N = 4096$ το εύρος ζώνης του κοινού για όλους τους επεξεργαστές δίσταλο κύριας μνήμης είναι το bottleneck.

Εκτελέσαμε το **STREAM** benchmark¹ και βρήκαμε εύρος ζώνης κύριας μνήμης περίπου 4500 MB/s.

Με χρήση του **likwid-perfctr** βρήκαμε ότι για $N = 4096$ διαβάζονται από την μνήμη περίπου 152 GB δεδομένων².

Με βάση όσα είπαμε πριν, αφού το τμήμα πίνακα που αντιστοιχεί στον επεξεργαστή δεν χωράει στην κρυφή μνήμη, προβλέπεται ότι όλα αυτά τα δεδομένα θα διαβαστούν από την κύρια μνήμη. Με άλλα λόγια, στον δίσταλο της κύριας μνήμης θα μεταφερθούν 152 GB. Για να γίνει αυτό απαιτούνται $\frac{152GB}{4500MB/s} = 33.77$ sec.

Παρατηρούμε, ότι ο χρόνος εκτέλεσης του προγράμματος για $N = 4096$ και 8 CPUs μετρήθηκε 33.75 sec, που αποτελεί, μάλιστα, και μια πρόβλεψη με τεράστια ακρίβεια! Με άλλα λόγια, όλοι οι υπολογισμοί σειριακούς πάνω στον δίσταλο κύριας μνήμης και στην πραγματικότητα δεν υπάρχει παραλληλία!

Αν και φαινομενικά, αυτό φαίνεται να οδηγεί στο παράξενο γεγονός, ότι δεν απαιτείται καθόλου χρόνος για να γίνουν υπολογισμοί, αυτό που πραγματικά συμβαίνει, είναι πως λόγω του out-of-order execution, όσο αναμένονται νέα δεδομένα, γίνονται και οι υπολογισμοί για τα δεδομένα που έχουν ήδη έρθει. Βέβαια οι υπολογισμοί αυτοί, τελειώνουν πριν η μνήμη παραδώσει νέα δεδομένα, οπότε ο επεξεργαστής συνεχίζει να περιμένει την μνήμη.

Συμπληρωματικά σχόλια

Προς επιβεβαίωση του επιχειρήματος, τροποποιήσαμε τον κώδικα σε δύο εκδοχές:

- Αντικαταστήσαμε τον πίνακα με στοιχεία τύπου int (μεγέθους 4 bytes) σε τύπου char (μεγέθους 1 byte). Έτσι ο πίνακας έχει μέγεθος 16 MiB, οπότε σε κάθε πυρήνα, για 8 CPUs, αντιστοιχούν 4 MiB, που χωράνε στην κρυφή μνήμη.
- Αντικαταστήσαμε τον πίνακα int [] [] με vector< vector<bool> >, όπου κάθε στοιχείο καταλαμβάνει αριθμός 1 bit. Για να γίνει αυτό, χρησιμοποιήσαμε, φυσικά, C++, αντί για C.

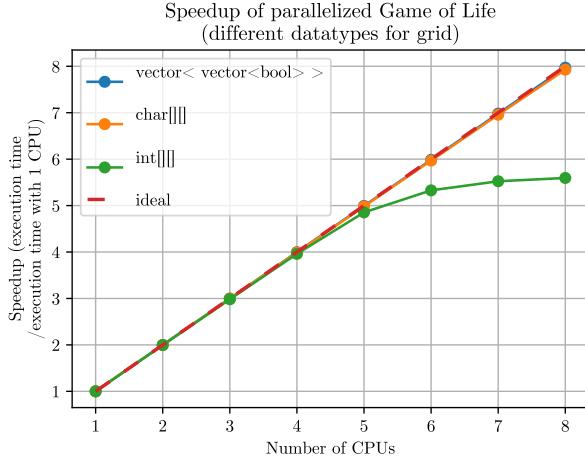
¹<https://www.cs.virginia.edu/stream/>

²το profiling αυτό έγινε σε τοπικό υπολογιστή, αντί της υποδομής του CSlab, καθώς το perf απαιτεί αυξημένα capabilites, όμως η ποσότητα δεδομένων που διαβάζεται από την μνήμη (όσχετα αν καταλήξει στην cache ή όχι) δεν εξαρτάται από την αρχιτεκτονική του συστήματος, αλλά από την σημασιολογία του προγράμματος.

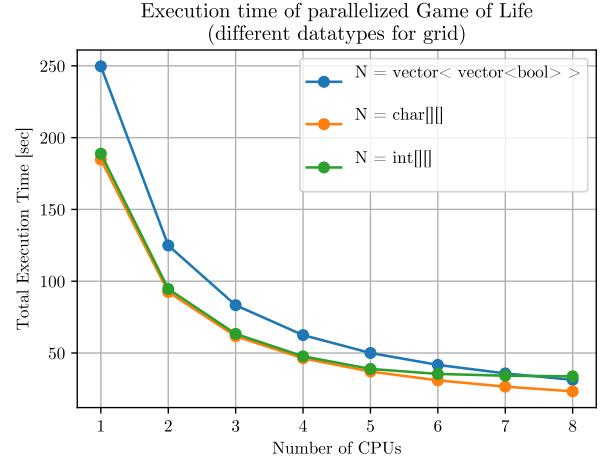
Έγιναν μετρήσεις επίδοσης για $N = 4096$. Τα αποτελέσματα για την επιτάχυνση, φαίνονται στην εικόνα 3α', και για τον συνολικό χρόνο εκτέλεσης στην εικόνα 3β'.

Παρατηρούμε πως, όπως ήταν θεωρητικά αναμενόμενο, οι εκδόχες με `char[] []` και `vector < vector<bool> >` είναι κλιμακώσιμες ως προς τον παραλληλοποίηση, σε αντίθεση με την εκδοχή του `int[] []`.

Αξίζει, επίσης, να παρατηρηθεί, πως το `vector< vector<bool> >` είναι κατά πολλαπλασιαστική σταθερά πιο αργό στον ολικό χρόνο εκτέλεσης από το `char[] []`, αν και στον ίδιο βαθμό κλιμακώσιμο. Αυτό συμβαίνει, διότι απαιτούνται επιπρόσθετοι υπολογισμοί με ANDs, ORs και shifts για να γίνει ανάγνωση και εγγραφή ενός bool. Ωστόσο, το `vector< vector<bool> >` είναι επίσης πιο αργό και για λίγους πυρήνες και από το `int[] []`. Παρόλα αυτά, επειδή είναι πιο κλιμακώσιμο, τελικά, στους πολλούς πυρήνες, καταλήγει να είναι πιο γρήγορο στον ολικό χρόνο εκτέλεσης. Με άλλα λόγια, η κλιμακωσιμότητα είναι σημαντική, ακόμα και αν για λίγους πυρήνες είναι πιο αργή.



(α') Επιτάχυνση



(β') Ολικός χρόνος εκτέλεσης

Σχήμα 3: Μετρήσεις επίδοσης στο παραλληλοποιημένο Game of Life για $N = 4096$ και για διάφορους τύπους δεδομένων πίνακα

1.4 Ενδιαφέρουσες Αρχικοποιήσεις (Patterns) στο Game of Life – προαιρετικό

- Glider (Ολισθητής)** Αποτελείται από ένα μικρό σύνολο “ζωντανών” κελιών που διατάσσονται συνήθως σε ένα σχήμα το οποίο μοιάζει με λοξή γραμμή τριών κελιών, δίπλα σε ένα ακόμα κελί. Το Glider έχει την ιδιότητα να ταξιδεύει διαγώνια στο άπειρο εφόσον δεν συναντήσει άλλα κύτταρα ή τοίχωμα (δηλαδή εάν το ταμπλό είναι άπειρο ή τυλιγμένο τοπολογικά). Σε κάθε γενιά, η μορφή του Glider αλλάζει ελαφρώς, με αποτέλεσμα να φαίνεται σαν να κινείται.
- Blinker (Αναβοσβήσιμο)** Πρόκειται για μια γραμμή τριών διαδοχικών ζωντανών κελιών. Παρατηρείται ταλαντωση (oscillation) μεταξύ δύο καταστάσεων. Στη μία κατάσταση, τα κελιά είναι τοποθετημένα οριζόντια, ενώ στην επόμενη κατάσταση, τα ίδια αυτά κελιά γίνονται κάθετα. Εναλλάσσεται η ευθυγράμμισή τους διαρκώς, με περίοδο ίση με 2 γενιές. Πολλές φορές εμφανίζεται αυθόρυμη ως υπο-δομή σε άλλα πολύπλοκα μοτίβα.
- Toad (Βάτραχος)** Αποτελείται από δύο γραμμές από τρία ζωντανά κελιά η καθεμία, τοποθετημένες με τρόπο ώστε το ένα σύνολο να “εφάπτεται” σχεδόν του άλλου. Όπως ο Blinker, το Toad είναι ταλαντωτής με περίοδο 2 γενιές. Σε κάθε εναλλαγή, τα κελιά περνούν από μία “αγκαλιασμένη” διάταξη σε μία πιο αραιή.
- Beacon (Φάρος)** Αποτελείται από δύο τετράγωνα (2×2) ζωντανών κελιών που αγγίζουν σε μία μόνο γωνία. Έχει περίοδο 2. Στη μία φάση, τα δύο τετράγωνα “απομακρύνονται”, αφήνοντας μια κενή περιοχή ανάμεσά τους, ενώ στην επόμενη φάση “επεκτείνονται” πάλι σε γειτονικές θέσεις.
- Pulsar (Παλλόμενος Αστέρας)** Είναι αρκετά μεγαλύτερος από τους προηγούμενους ταλαντωτές. Αποτελείται από έναν “κεντρικό” πυρήνα κελιών και έξι “βραχίονες” που εκτείνονται περιφερειακά σε απόσταση. Έχει περίοδο 3 γενιές. Ολόκληρη η διάταξη υφίσταται μια περιοδική μεταβολή, όπου ζωντανά κελιά εξαφανίζονται και επανεμφανίζονται σε συγκεκριμένες θέσεις. Πρόκειται για ένα εντυπωσιακό οπτικά μοτίβο, επειδή τα “βραχίονά” του εμφανίζονται και εξαφανίζονται διαδοχικά.
- Gosper Glider Gun (Το Κανόνι των Gliders)** Αποτελείται από μια περίτεχνη διάταξη ζωντανών κελιών σε αρκετές σειρές, με συγκεκριμένη δομή “δωματίων” ή “θαλάμων”. Κατά τακτά διαστήματα, εκλύει (παράγει) Gliders που κινούνται προς μία συγκεκριμένη κατεύθυνση. Η περίοδος εκπομπής νέου Glider είναι συνήθως 30 γενιές (ανάλογα με την ακριβή δομή).

7. **Spaceships (Διαστημόπλοια)** πέρα από το Glider Υπάρχουν μεγαλύτερα “κινούμενα” μοτίβα με διαφορετικές ταχύτητες, όπως το “Lightweight Spaceship (LWSS)”, “Middleweight Spaceship (MWSS)”, και “Heavyweight Spaceship (HWSS)”).
8. **Oscillators υψηλότερης περιόδου** Πέρα από τους ταλαντωτές με περίοδο 2 ή 3, έχουν βρεθεί πολύπλοκοι ταλαντωτές με μεγάλες περιόδους, που χρειάζονται αρκετές γενιές για να επανέλθουν στην αρχική τους μορφή.
9. **Methuselahs (Μεθουσάλες)** Μικρές αρχικοποιήσεις που “ζωντανεύουν” τον χώρο για μεγάλο αριθμό γενιών πριν τελικά σταθεροποιηθούν ή επαναληφθούν. Το πιο διάσημο παράδειγμα είναι το “R-pentomino”.

2 2η Άσκηση – Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

2.1 Περιγραφή αρχιτεκτονικής υπολογιστικού συστήματος εκτέλεσης

Το υπολογιστικό σύστημα που χρησιμοποιούμε αποτελείται από 4 επεξεργαστές Intel(R) Xeon(R) CPU E5-4620 @ 2.20GHz.

Ο επεξεργαστής αυτός έχει 8 φυσικούς πυρήνες και υποστηρίζει Simultaneous Multithreading με 2 νήματα ανά φυσικό πυρήνα, γνωστό με την εμπορική ονομασία, *Hyperthreading*.

Το υπολογιστικό σύστημα είναι μηχάνημα τύπου NUMA με συνάφεια κρυφής μνήμης (ccNUMA).

Πιο συγκεκριμένα, κάθε επεξεργαστής έχει την δική του φυσική μνήμη με χωριστό δίστημα. Ωστόσο ο χώρος διευθύνσεων όλων των μνημών είναι κοινός και ένας επεξεργαστής μπορεί να προσπελάσει την φυσική μνήμη ενός άλλου επεξεργαστή, απλά κάνοντας προσβάσεις στις αντίστοιχες μνήμες, με αντίστοιχες σημασιολογικές εγγυήσεις με αυτές για τις προσβάσεις σε κοινή μνήμη από δυό χωριστούς πυρήνες.

Συνοψίζοντας, έχουμε 4 NUMA κόμβους με δική τους φυσική μνήμη, καθένας έχει 8 φυσικούς πυρήνες, και καθένας από τους φυσικούς πυρήνες έχει 2 νήματα. Συνολικά, δηλαδή, μπορούμε να υποστηριχθούν από το υλικό, 64 νήματα.

2.2 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

2.2.1 Αφελής παραλληλοποίηση

2.2.1.1 Υλοποίηση

Σε πρώτο επίπεδο δοκιμάσαμε να παραλληλοποίησουμε τον υπολογισμό του κοντινότερου cluster centre κάθε σημείου, δηλαδή πρακτικά ον βρόχο for (`int i=0; i<numObjs; i++`). Προκειμένου να μην απαιτηθούν περιττές δηλώσεις στο OpenMP, αλλά και λόγους απλότητας, μεταφέραμε όλες τις επαγγελματικές μεταβλητές πάνω στο for statement. Επίσης η μεταβλητή index μεταφέρθηκε στο εσωτερικό του βρόχου scope, ώστε να θεωρηθεί αυτόματα private.

Ωστόσο οι επαναλήψεις του βρόχου δυστυχώς επηρεάζουν κοινά για όλες τις επαναλήψεις δεδομένα και πιο συγκεκριμένα τους πίνακες newClusters, newClusterSize και την μεταβλητή delta.

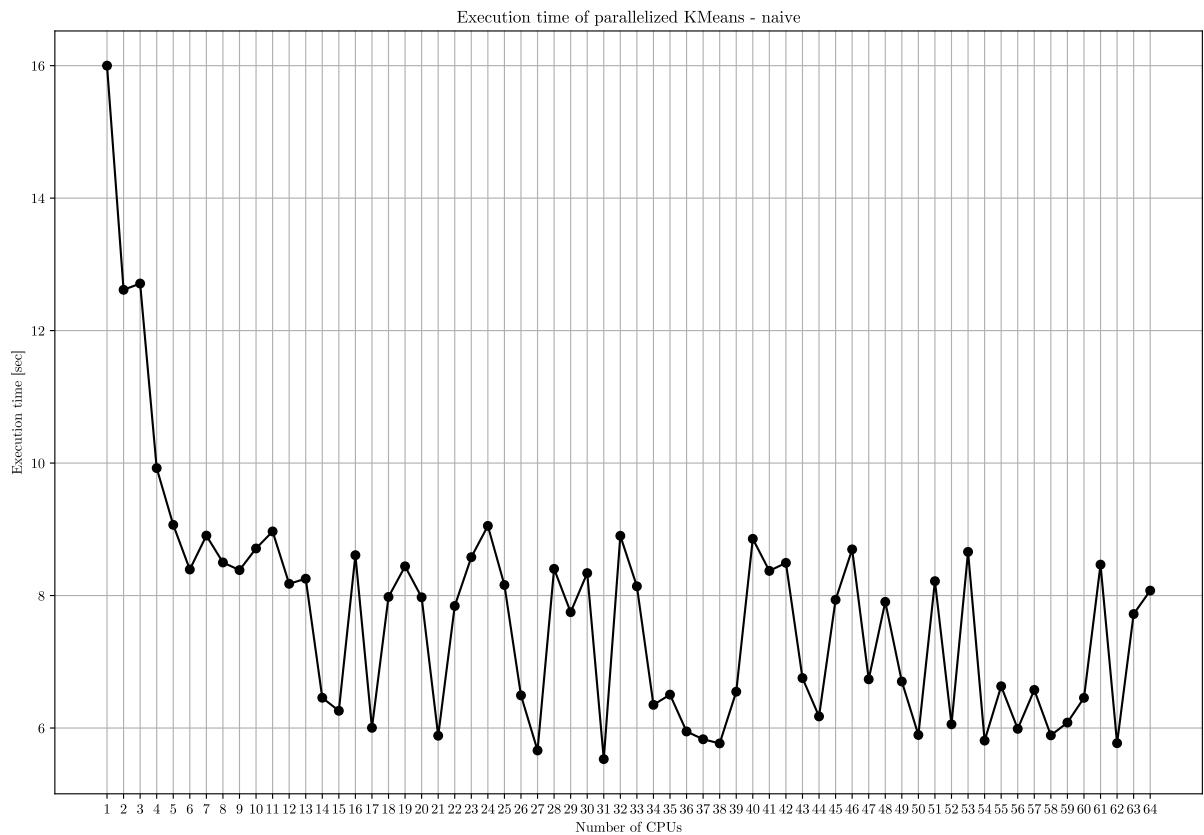
Με δεδομένη την αντιμεταθετικότητα και την προσεταιριστικότητα της πρόσθεσης, αν απλά εξασφαλίσουμε την ατομικότητα της εντολής `+=` οι προσβάσεις θα είναι σωστές.

Βέβαια, αν και η προσεταιριστικότητα ισχύει για ακέραιους αριθμούς, δεν ισχύει για αριθμούς κινητής υποδιαστολής, οπότε το τελικό αποτέλεσμα στην μεταβλητή delta μπορεί να μην είναι ακριβώς ίσο με την sequential εκδοχή. Παρόλα αυτά, εδώ θα θεωρήσουμε ότι ισχύει η προσεταιριστικότητα και θα αποδεχθούμε ένα ίσως ελάχιστα προσεγγιστικό αποτέλεσμα στην delta. Εξάλλου, το delta θα μπορούσε να υπολογίζεται ως ακέραιος αριθμός και απλά στο τέλος να γίνεται η μετατροπή σε αριθμό κινητής υποδιαστολής και αυτό θα ήταν και ταχύτερο, διότι θα κάνουμε μοναδιαίες αυξήσεις ακέραιου (πολύ γρήγορο), αντί για προσθέσεις αριθμών κινητής υποδιαστολής διπλής ακριβείας.

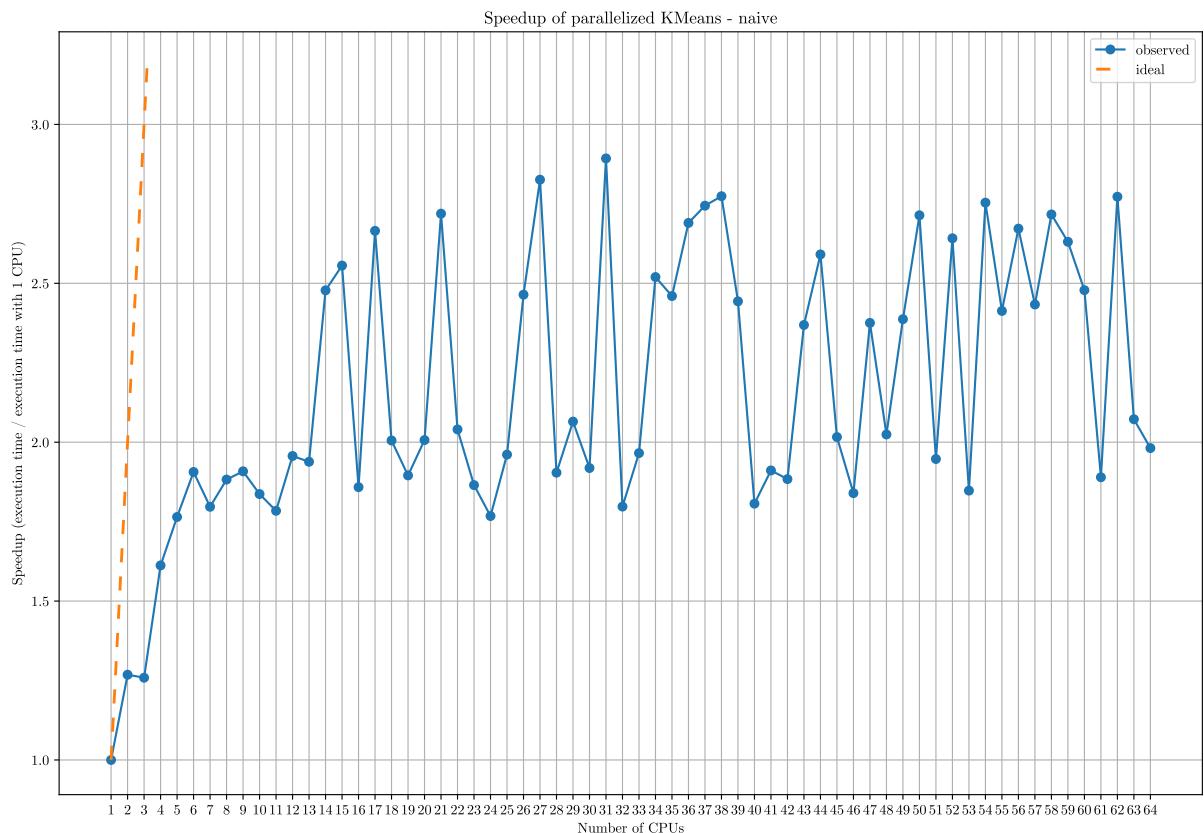
Κατ' επέκταση προσθέτουμε την οδηγία: `#pragma omp atomic` πριν τις προσβάσεις στις αντίστοιχες μεταβλητές που προαναφέραμε. Η οδηγία αυτή υποχρύπτει την χρήση αντίστοιχων ατομικών instructions που προσφέρει το ISA της CPU.

2.2.1.2 Μετρήσεις επίδοσης – Παρατηρήσεις

Μετρήθηκε ο χρόνος εκτέλεσης για πλήθων επεξεργαστών 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 4 και η επιτάχυνση στο σχήμα 5.



Σχήμα 4: Χρόνος εκτέλεσης για naive παραλληλοποιημένο KMeans



Σχήμα 5: Επιτάχυνση για naive παραλληλοποιημένο KMeans

Παρατηρούμε ότι για μέχρι 5 επεξεργαστές υπάρχει κάποια επιτάχυνση, αυτή είναι γραμμική, αλλά σιγούρα αρκετά πιο αργή από την ιδανική. Για περισσότερους από 5 επεξεργαστές, η επίδοση δεν βελτιώνεται, μέχρι τους 16 επεξεργαστές, όπου μπαίνει ο 2ος κόμβος NUMA. Η απόδοση δεν βελτιώνεται για περισσότερους επεξεργαστές.

Επιτάχυνση κοντά στην ιδανική επιτυγχάνεται μόνο για 2 επεξεγαστές, ενώ για παραπάνω απέχει σημαντικά από αυτή.

Η μέγιστη επιτάχυνση που επιτυγχάνεται είναι περίπου 2.5x παρόλο που έχουμε βάλει 10αδες επεξεργαστές να δουλέψουν παράλληλα.

Προτού προβούμε σε πρόωρα συμπεράσματα, επιχειρούμε την εκτέλεση με σταθερή, στατική ανάθεση νημάτων εκτέλεσης σε νήματα υλικού και προχωράμε σε συμπεράσματα στην συνέχεια.

2.2.1.3 Εκτέλεση με σταθερή απεικόνιση OpenMP νημάτων σε νήματα του υλικού

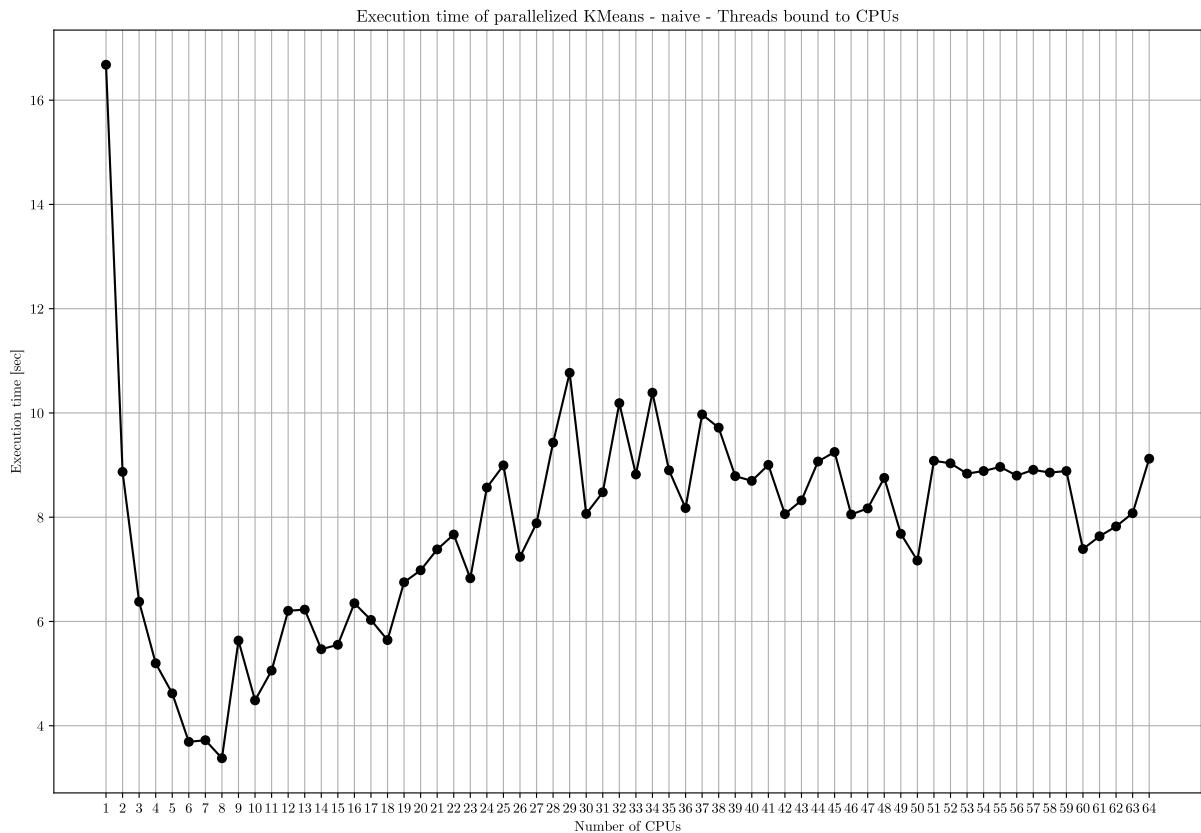
Επιχειρήσαμε να εκτελέσουμε την ίδια υλοποίηση με πριν, αλλά βάζοντας κάθε νήμα εκτέλεσης να απεικονίζεται σε ένα σταθερό νήμα του υλικού.

Για να το επιτυχούμε αυτό, για εκτέλεση σε πλήθος NCPUS θέσαμε την μεταβλητή περιβάλλοντος (environmental variable) GOMP_CPU_AFFINITY="0-NCPUS".

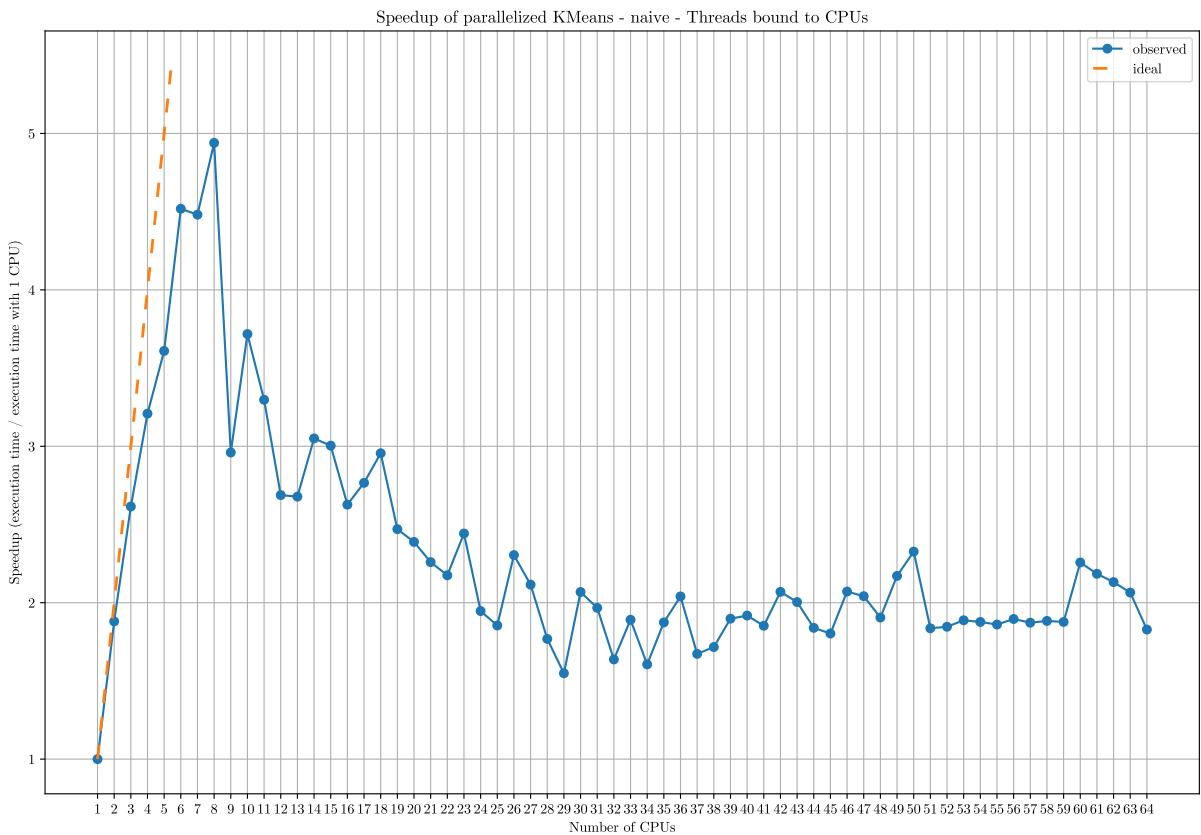
Στο σημείο αυτό, αξίζει να σημειώσουμε ότι ο επεξεργαστής που χρησιμοποιείται, υποστηρίζει Simultaneous Multithreading (SMT), γνωστό με την εμπορική ονομασία Hyperthreading. Κάθε φυσικός πυρήνας υποστηρίζει 2 νήματα εκτέλεσης. Στο Linux αυτό κωδικοποιείται στους identifier, θέτοντας τους πρώτους μισούς να αντιστοιχούν στο ένα από τα δύο νήματα όλων των πύρηνων, και έπειτα τους άλλους μισούς να αντιστοιχούν στο άλλο νήμα όλων των πύρηνων. Πιο συγκεκριμένα για 32 φυσικούς πύρηνες, δηλαδή 64 νήματα, οι αριθμοί 0 – 31 αναφεόνται στο ένα νήμα των 32 φυσικών πυρήνων, και οι αριθμοί 32 – 63 στο άλλο νήμα.

Έτσι με την εκτέλεση αυτή για μέχρι 32 πυρηνές, χρησιμοποιούμε τον κάθε πυρήνα μόνο με ένα νήμα, και μόνο για ≥ 33 πυρήνες, χρησιμοποιούμε και 2o νήμα. Όπως θα δούμε αργότερα σε άλλη υλοποίηση, αυτό οδηγεί σε κάποιες σημαντικές επίπτωσεις που ίσως αρχικά να μην είναι φανερές.

Μετρήθηκε εκ νέου ο χρόνος εκτέλεσης για πλήθων πυρήνων 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 6 και η επιτάχυνση στο σχήμα 7.



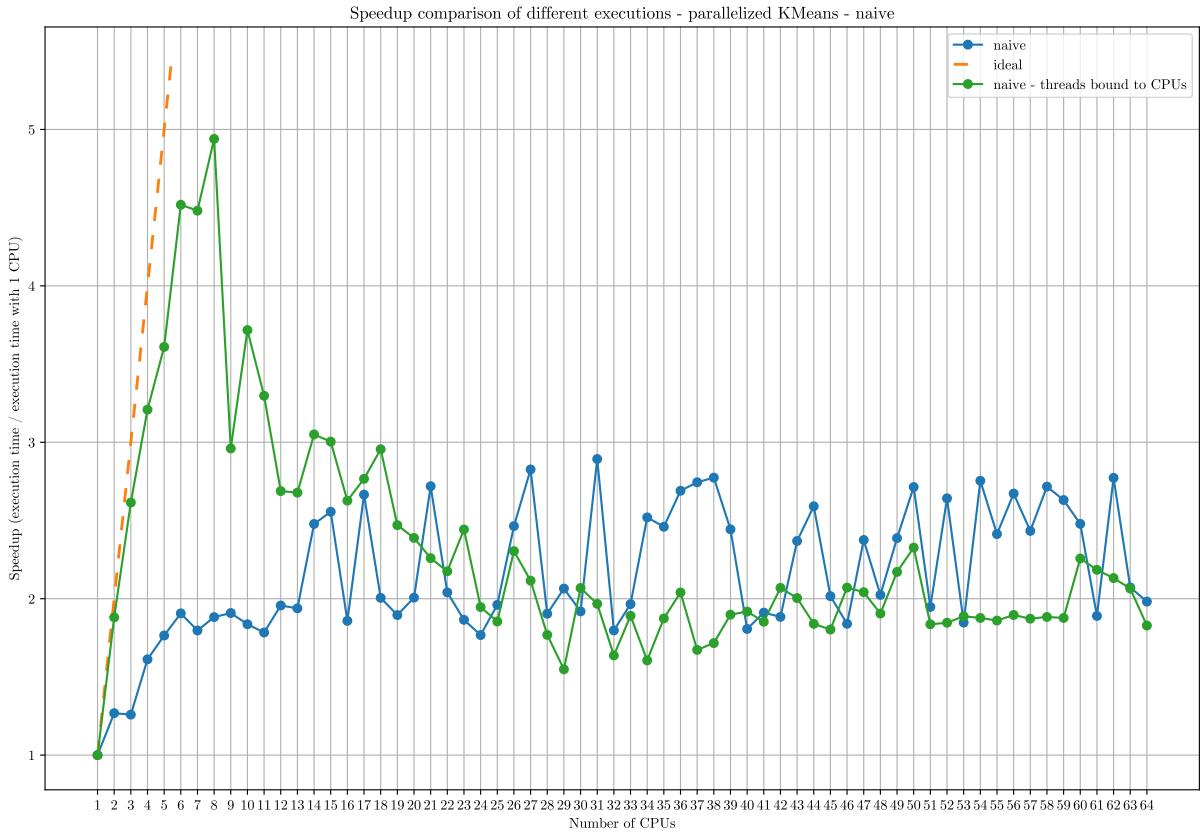
Σχήμα 6: Χρόνος εκτέλεσης για naive παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού



Σχήμα 7: Επιτάχυνση για naive παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού

Παρατηρούμε ότι μέχρι περίπου 5 πυρήνες, η επιτάχυνση είναι γραμμική και κοντά στην ιδανική. Η επιτάχυνση δίνει μέγιστο 5x για 8 πυρήνες και έπειτα μειώνεται.

Για να αντιληφθούμε καλύτερα τα πράγματα η επιτάχυνση και για τις δύο εκτελέσεις φαίνεται στο σχήμα 8.



Σχήμα 8: Επιτάχυνση για naive παραλληλοποιημένο KMeans – σύγκριση εκτελέσεων

Παρατηρούμε πως μέχρι τους 16 πυρήνες η στατική ανάθεση πυρήνων είναι καλύτερη, ενώ μετά η δυναμική τα πάει καλύτερα.

2.2.1.4 Σχολιασμοί – Συμπεράσματα

Η υλοποίηση που έχουμε κάνει έχει το αρνητικό ότι έχει πάρα πολύ επικοινωνία.

Με βάση τα προηγούμενα, εικάζεται πως ο δρομολογητής (scheduler) μάλλον το αντιλαμβάνεται αυτό και προσπαθεί κάπως να το βελτιώσει κάνοντας migrations, μάλλον για να φέρει πιο κοντά αυτούς που του φαίνεται ότι επικοινωνούν περισσότερο. Όμως αυτό που βλέπει στιγμαία ο δρομολογητής ως συχνότερη επικοινωνία μεταξύ νήματων είναι απλά τυχαίο, αφού τα νήματα κάνουν τυχαίες προσβάσεις στους πίνακες, και επικοινωνούν όλα μεταξύ τους με τρόπο που τείνει προς τον ομοιόμορφο. Εκτός αν υπάρχει κάποια πολύ ισχυρή ανωμαλία στα συγκεκριμένα δεδομένα κάθε φοράς, η ποσότητα επικοινωνίας μεταξύ όλων των επεξεργαστών είναι περίπου ίση. Εδώ, μάλιστα, διαλέξαμε τα δεδομένα τυχαία, οπότε αυτό με καλή πιθανότητα θα ισχύει.

Με άλλα λόγια, ο δρομολογητής φάχνει για patterns επικοινωνίας, που εδώ απλά δεν υπάρχουν, και ερμηνεύει λανθασμένα επειδή κάποιες φορές τυχαίνει κάποιοι επεξεργαστές να επικοινωνούν περισσότερο, ότι υπάρχει και pattern συχνής επικοινωνίας. Η λάθος αυτή ερμηνεία, τον οδηγεί σε περιττά migrations, που όμως επιφέρουν κόστος, τόσο λόγω του context switch, όσο και λόγο των compulsory cache misses που επιφέρει.

Συνεπώς στην συνέχεια σχολιάζουμε μόνο την εκδοχή που έχει γινει σταθερή στατική ανάθεση των νημάτων εκτέλεσης σε φυσικά νήματα υλικού.

Παρατηρούμε ότι όσο χρησιμοποιείται μόνο ένας κόμβος NUMA, οι επιδόσεις είναι σχετικά καλές, αν και όταν ξεπεράσουμε τους 4 – 5 πυρήνες, η κλιμακωσμότητα μειώνεται, όμως η επιτάχυνση συνεχίζει να αυξάνεται. Μόλις μπει για $n \geq 9$ ο δεύτερος κόμβος NUMA, οι επιδόσεις μειώνονται πολύ άμεσα, και όσο μπαίνουν περισσότεροι πυρήνες και κόμβοι, η επίδοση διαρκώς χειροτερεύει, παρά βελτιώνεται.

Αρχικά, όσο είμαστε στον ένα μόνο κόμβο, η κλιμακωσμότητα αρχίζει να μειώνεται όσο αυξάνονται οι πυρήνες, διότι η επικοινωνία για να γίνει απαιτεί να τρέχει ένα πρωτόκολλο επικοινωνίας (π.χ. MESI), που επιφέρει καθυστερήσεις. Σε κάποιο πιο περιορισμένο βέβαια βαθμό συμβαίνει και αυτό που θα δούμε για τους πολλούς κόμβους.

Όταν μπει ο δεύτερος κόμβος, για να εκτελεστούν οι ατομικές εντολές που έχουμε βάλει αυτό που γίνεται είναι πως διαβάζει ένας πυρήνας μια τιμή και μέχρι να κάνει ότι υπολογισμό θέλει και να την ξαναγράψει, κανείς άλλος δεν μπορεί να ξεκινήσει ανάγνωση και υπολογισμό ή αν ξεκινήσει θα πρέπει στην συνέχεια να ακυρώσει ότι έκανε. Ο τρόπος υλοποιήσης της ατομικότητας δεν έχει σημασία, καθώς αυτή η καθυστέρηση επιβάλλεται από την ίδια την σημασιολογία της ατομικής μεταβολής.

Αυτό που συμβαίνει όταν μπει ο 2ος NUMA κόμβος, είναι πως εκτελούνται ατομικές μεταβολές σε απομακρυσμένη μνήμη, δηλαδή πυρήνας ενός κόμβου διαβάζει μια μεταβλητή από την μνήμη άλλου κόμβου, εκτελεί έναν υπολογισμό, την ξαναγράψει, και μέχρι να ολοκληρωθεί αυτή η επανεγγραφή κανείς άλλος πυρήνας δεν μπορεί να εκτελέσει άλλες μεταβολές στην μεταβλητή αυτή. Το πρόβλημα είναι, πως η επικοινωνία μεταξύ των NUMA κόμβων έχει αρκετή καθυστέρηση, οπότε ο χρόνος που απαιτείται για να έρθει η παλιά τιμή και να επιστρέψει η νέα, είναι αρκετός ώστε άλλοι πυρήνες να έχουν τελειώσει την υπόλοιπη δουλειά τους και απλά να περιμένουν να έρθει η σειρά τους να κάνουν την δική τους ατομική μεταβολή. Τα πράγματα χειροτερεύουν όσο μπαίνουν παραπάνω κόμβοι, διότι πολλοί πυρήνες πλεόν έχουν τελειώσει την υπόλοιπη δουλειά τους, και περιμένουν διαδοχικά να έρθει η σειρά τους για να κάνουν την ατομική μεταβολή, γεγονός που εισάγει επιπρόσθετη καθυστέρηση, οπότε ακόμα περισσότεροι τελειώνουν την υπόλοιπη δουλειά τους. Με άλλα λόγια, **ο παράλληλος υπολογισμός έχει πρακτικά σειριοποιηθεί στις ατομικές μεταβολές**, οπότε δεν κερδίζουμε σε επίδοση. Όσο είμαστε εντός του ίδιου NUMA κόμβου, αυτό δεν γίνεται τόσο αισθητό επειδή η επικοινωνία είναι πιο γρήγορη, οπότε υπάρχουν χρονικά κενά ανάμεσα στις ατομικές μεταβολές από διαφορετικούς πυρηνές, οπότε η εκτέλεση εκτός σειράς (out-of-order execution) μάλλον καταφέρνει να κρύψει τις αναμονές που προκύπτουν όταν η μεταβλητή είναι “κατειλλημένη”.

2.2.2 Reduction παραλληλοποίηση

2.2.2.1 Αρχική υλοποίηση

Με βάση τα προηγούμενα, προκειμένου να μειώσουμε την επικοινωνία και να αφήσουμε κάθε επεξεργαστή να δουλέψει όσο το δυνατόν πιο αυτονόμα, παρατηρήσαμε πως ακριβώς λόγω της προσεταιριστικότητας της πρόσθεσης και του γεγονότος πως η πρόσθεση έχει ουδέτερο στοιχείο, το μηδέν, μπορούμε να αφήσουμε κάθε πυρήνα να κάνει χωριστά τις πράξεις, και στο τέλος να αθροίσουμε τα αποτελέσματα που υπολόγισαν.

Για την μεταβλητή `delta` χρησιμοποιήσαμε το reduction clause του OpenMP, βάζοντας στην οδηγία του OpenMP για το παράλληλο for το clause `reduction(+:delta)`.

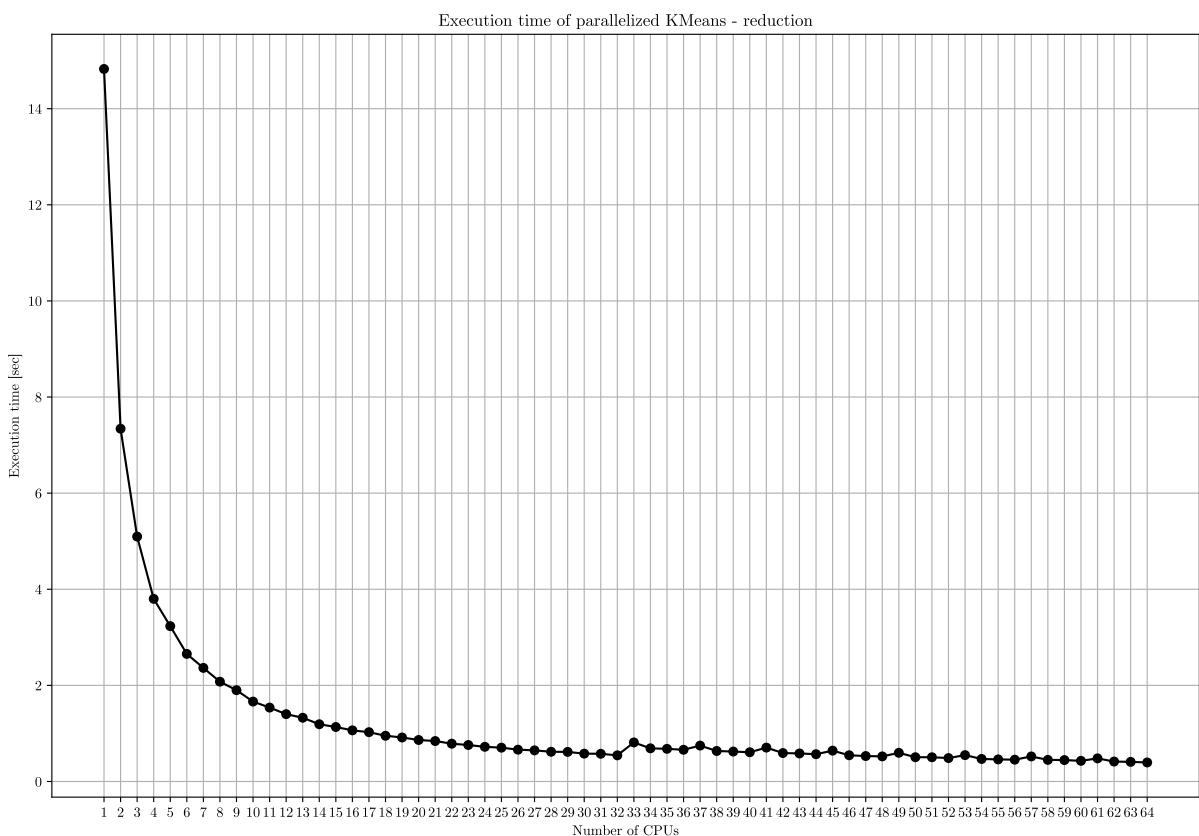
Για τους πίνακες, δημιουργούμε τους πίνακες `int * local_newClusterSize[nthreads]` και `double * local_newClusters[nthreads]`, που έχουν καθένας k υποπίνακες, καθένας από τους οποίους είναι ίδιας μορφής με τους `newClusters` και `newClusterSize` αντίστοιχα.

Κάθε νήμα μέσω της συνάρτησης `omp_get_thread_num()` βρίσκει τον αύξοντα αριθμό του, και τον χρησιμοποιεί για να επιλέξει τον σωστό υποπίνακα και έπειτα κάνει μεταβολές μόνο σε αυτό.

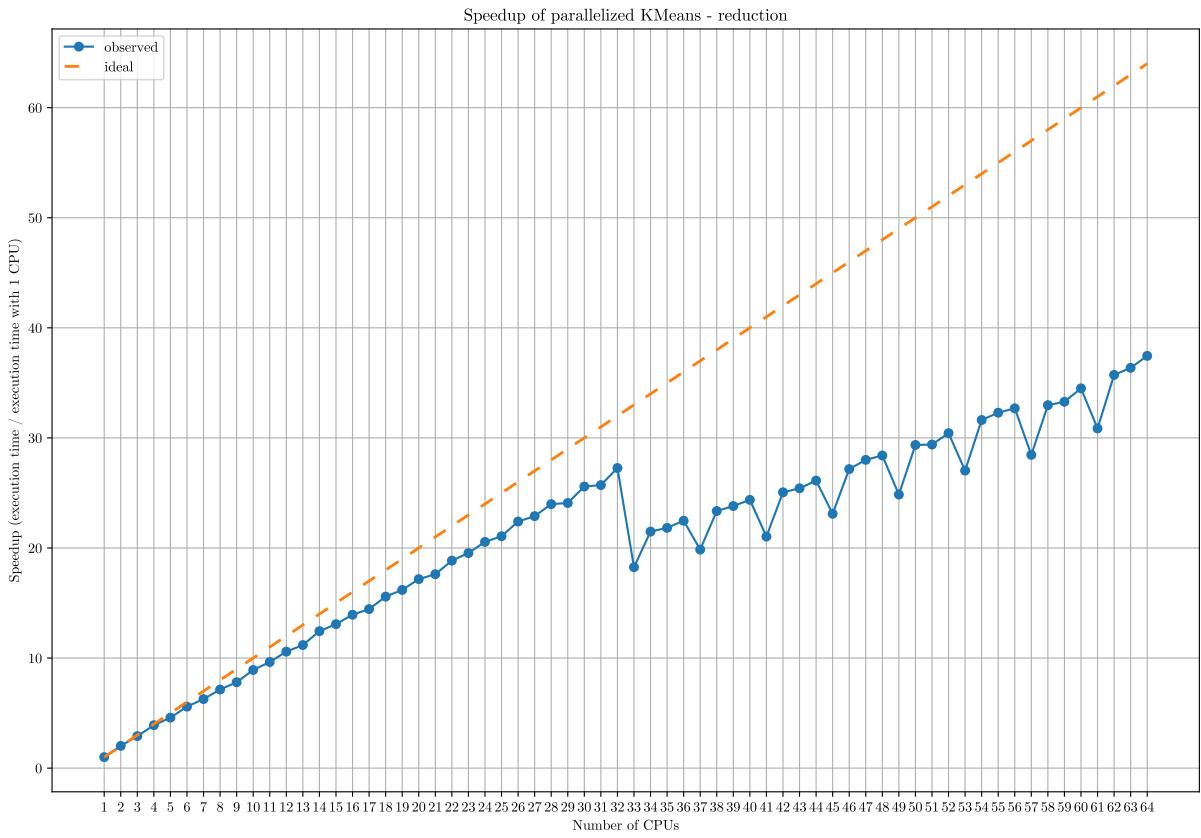
Αφού ολοκληρωθεί το for (το OpenMP βάζει barrier στο τέλος), το master νήμα αθροίζει τα αποτελέσματα των υποπίνακων και τοποθετεί το αποτέλεσμα στους πίνακες `newClusters` και `newClusterSize`. Οι προσθέσεις αυτές είναι αρκετά λίγες, οπότε δεν αξίζει να τις παραλληλοποιήσουμε.

2.2.2.2 Μετρήσεις επίδοσεις – παρατηρήσεις – πρώτα σχόλια

Μετρήθηκε ο χρόνος εκτέλεσης για πλήθων επεξεργαστών 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 9 και η επιτάχυνση στο σχήμα 10.



Σχήμα 9: Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού



Σχήμα 10: Επιτάχυνση για reduction παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού

Οι επιδόσεις είναι ασύγκριτα καλύτερες σε σχέση με την αφελή παραλληλοποίηση.

Μέχρι 32 πυρήνες, η κλιμακωσιμότητα είναι γραμμική. Ο συντελεστής κλιμάκωσης της επιτάχυνσης είναι λίγο μικρότερος από τον ιδιαίτερο 1, με τιμή περίπου 0.85³, όμως είναι σημαντικό πως έχει επιτευχθεί γραμμική κλιμάκωση. Για 32 πυρηνές, λαμβάνουμε περίπου 27x επιτάχυνση. Η επίδοση αυτή χαρακτηρίζεται σε γενικές γραμμές σχετικά καλή.

Για πάνω από 32 πυρηνές, ξεκινώντας από τους 33 πυρηνές, παρατηρείται μια πολύ αποτόμη μείωση της επιδόσης, που φαίνεται σε πρώτο επίπεδο αρκετά παράξενη.

Θυμόμαστε, ωστόσο, όσα αναφέραμε παραπόνω στην παράγραφο 2.2.1.3. Ο 33ο πυρήνας, στην πραγματικότητα δεν είναι φυσικός πυρήνας, αλλά το δεύτερο νήμα του 1ου φυσικού πυρήνα. Έτσι όταν μπει ο 33ος πυρήνας, έχουμε έναν πυρήνα που δουλεύει με δύο νήματα και όλους τους άλλους να δουλεύουν με μόνο ένα. Στο Simultaneous Multithreading, κερδίζουμε φυσικά σε επιδόση, ως προς το εμβαδόν του chip (ισοδύναμα ως προς το κόστος κατασκευής) και ως προς την κατανάλωση ηλεκτρικής ισχύος, όμως η επίδοση των δύο νημάτων είναι μικρότερη από την επίδοση που θα είχαν δύο χωριστοί πυρήνες.

Θυμόμαστε επίσης πως έχουμε διαλέξει στατική κατανομή των επαναλήψεων του βρόχου στους πυρηνές, η οποία μάλιστα είναι και ίση σε κάθε πυρήνα (αυτό λόγω του OpenMP clause `schedule(static)`). Η επιλογήγ αυτή υποκρύπτει την υπόθεση, πως όλοι οι πυρήνες έχουν την ίδια υπολογιστική δύναμη, όποτε αν τους βάλουμε ίση ποσότητα δουλειάς, θα τους πάρει περίπου ίσο χρόνο να την κάνουν, όποτε θα τελειώσουν περίπου ταυτόχρονα, οπότε δεν θα υπάρχει αδρανής χρόνος και θα χρησιμοποιούμε αποδοτικά τους πυρηνές.

Ωστόσο, τα νήματα που τρέχουν μαζί με κάποιο άλλο νήμα σε κάποιον φυσικό πυρήνα, είναι πιο αργά από τα νήματα που τρέχουν μόνα τους σε έναν φυσικό πυρήνα. Συνεπώς, αυτό που συμβαίνει, είναι πως τα νήματα που είναι μαζί σε πυρήνα αργούν να τελειώσουν σε σχέση με τα νήματα που είναι μόνα τους, οπότε τα τελευταία αδρανούν. Με άλλα λόγια, έχουμε κακή κατανομή της δουλειάς στα νήματα, διότι δεν έχουμε λάβει υπόψη την περιορισμένη υπολογιστική δύναμη των νήματων όταν τρέχουν μαζί σε έναν πυρήνα.

Πράγματι, το δυναμικό scheduling σε κάποιο βάθμο φάνηκε να περιορίζει το πρόβλημα αυτό, όχι όμως και να το εξαλείφει πλήρως. Για λόγους συντομίας, δεν παραθέτουμε τα πρόσθετα αυτά διαγράμματα.

³ Ωηλαδή η επιτάχυνση είναι 0.85 φορές το πλήθος των CPUs

Θα μπορούσαμε να μετρήσουμε πόσο πιο άργη είναι η εκτέλεση ενός νήματος όταν τρέχει μαζί με κάποιο άλλο σε έναν φυσικό πυρήνα, και από αυτό να φτιάξουμε ένα καλύτερο στατικό schedule όπου δεν θα ισοκατανέμουμε την δουλειά απλά, αλλά θα ισοκατανέμουμε τον χρόνο εκτέλεσης. Με άλλα λόγια αν υποθέσουμε ότι έχουμε n_1 πυρήνες που τρέχουν ένα νήμα εκτέλεσης και n_2 πυρήνες που τρέχουν δύο νήματα εκτέλεσης και μετρήσουμε ότι στους δεύτερους απαιτείται a φορές ο χρόνος που απαιτείται στους πρώτους για την ίδια δουλειά, τότε αναθέτουμε ποσοστό δουλειάς $P_1\%$ και $P_2\%$ αντίστοιχα και στους πρώτους και δεύτερους ώστε:

$$\begin{cases} P_1 = P_2 \cdot a \\ P_1 \cdot n_1 + P_2 \cdot n_2 = 1 \end{cases} \iff \begin{cases} P_1 = \frac{a}{an_1+n_2} \\ P_2 = \frac{1}{an_1+n_2} \end{cases} \quad (1)$$

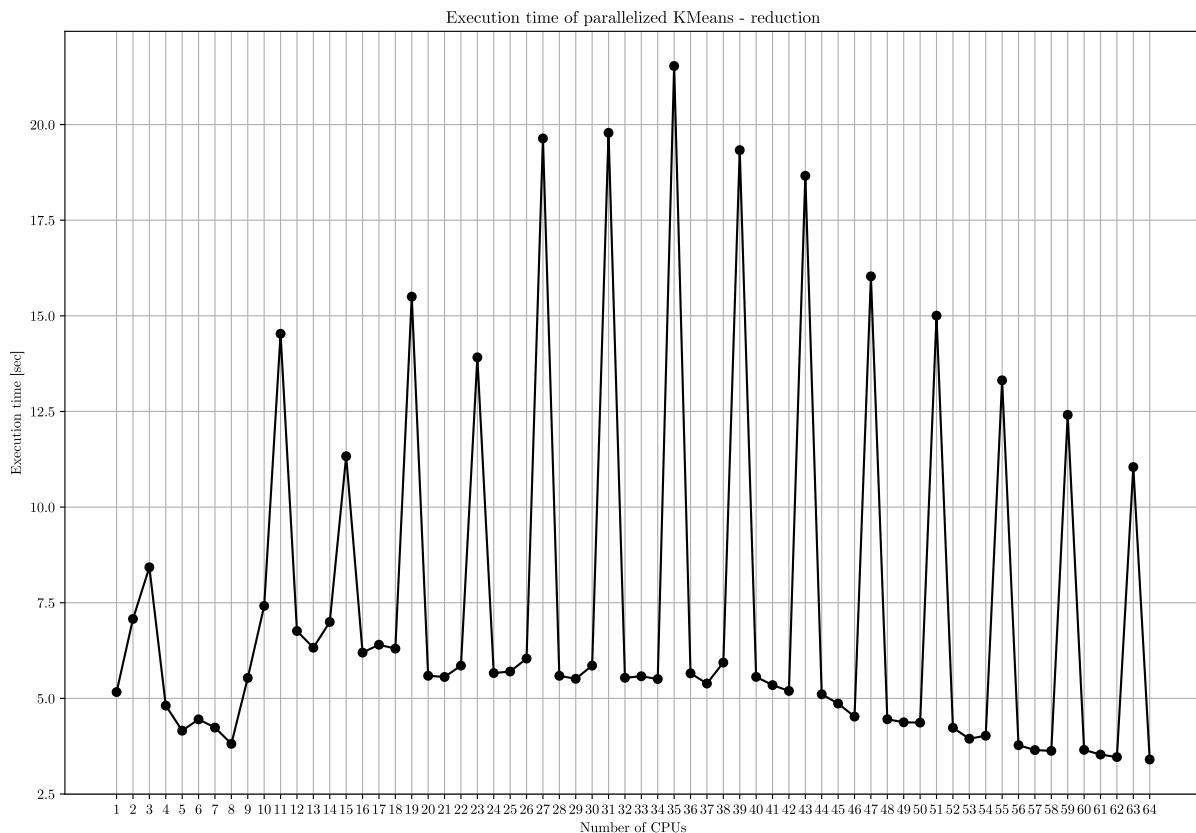
όπου η πρώτη σχέση εξασφαλίζει τον ίσο χρόνο εκτέλεσης.

To profiling αυτό και ο προγραμματισμός ενός τέτοιου custom static schedule (το OpenMP δεν το υποστηρίζει εγγενώς) φαίνεται πέραν των στόχων αυτής της άσκησης, οπότε στην συνέχεια θα λαμβάνουμε υπόψη τις επιδόσεις για μέχρι 32 πυρηνές και έπειτα ξανά για 64, όπου και πάλι η ισοκατανομή της δουλείας σημαίνει και ισοκατανομή χρόνου αφού και πάλι όλα τα νήματα έχουν ίση υπολογιστική δύναμη.

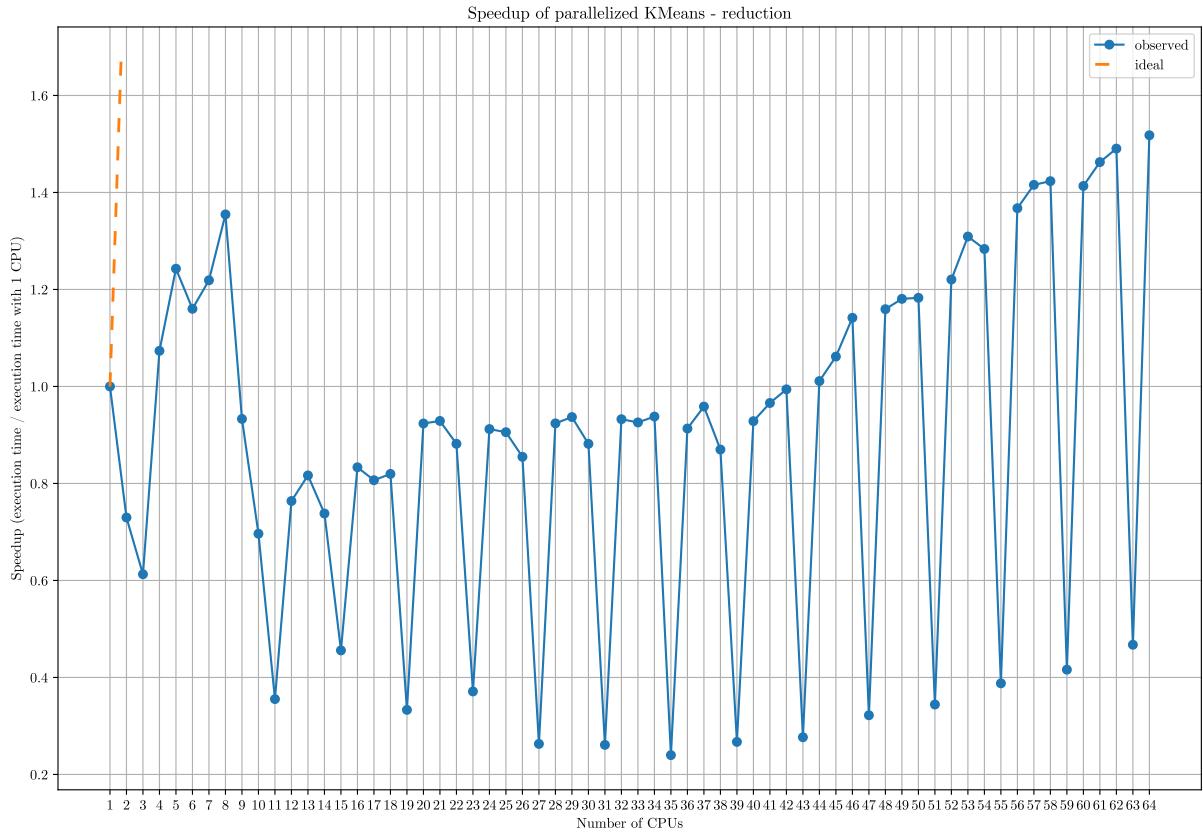
2.2.2.3 Μετρήσεις επίδοσης σε διαφορετικό configuration

Δοκιμάσαμε ένα δεύτερο configuration με την ίδια υλοποίηση. Ειδικότερα, δοκιμάσαμε για 1 συντεταγμένη και 4 clusters, αντί για 16 συντεταγμένες και 32 clusters που χρησιμοποιήσουμε πριν.

Μετρήθηκε ο χρόνος εκτέλεσης για πλήθων επεξεργαστών 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 11 και η επιτάχυνση στο σχήμα 12.



Σχήμα 11: Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – 2o configuration



Σχήμα 12: Επιτάχυνση για reduction παραλληλοποιημένο KMeans – 2o configuration

Παρατηρούμε ότι για την ίδια υλοποίηση με πριν, οι επιδόσεις είναι σημαντικά χειρότερες. Μάλιστα, η παραλληλοποίηση φαίνεται να κάνει τις επιδόσεις χειρότερες από την σειριακή εκδοχή! Παίρνουμε για 64 πυρήνες μέγιστο speedup 1.5x που χαρακτηρίζεται ως αρκετά κακή επίδοση.

2.2.2.4 Σχολιασμός

Ανακύπτει το ερώτημα τι συνέβη και οι επιδόσεις χειροτερέψαν τόσο. Η απάντηση βρίσκεται στον τρόπο εκχώρησης της μνήμης για τους τοπικούς πίνακες κάθε νήματος.

Ξεκινάμε αρχικά με κάποιες παρατηρήσεις για τον τρόπο που γίνεται η εκχώρηση μνήμης από την `malloc`. Το master thread ζητά μέσω της `malloc` να του δοθεί κάποιο πλήθος από bytes μνήμης. Η `malloc` αποκτά αυτή την μνήμη ζητώντας αντίστοιχα μνήμη από το Linux. Το Linux, όμως, παρέχει την μνήμη σε διαχριτά “χομμάτια”, τις σελίδες, που έχουν συγκεκριμένο μέγεθος, συνήθως 4KB = 4096 bytes. Η `malloc` για να διαχειριστεί αιτήματα μικρότερα από 4096 bytes, διατηρεί μια διακή δεξαμένη από μνήμη που έχει διαθέσιμη, και την εκχωρεί στον προγραμματιστή όταν ζητηθεί. Με άλλα λόγια, αν ζητήσω 128 bytes μνήμη, θα ζητήσει από το Linux μία σελίδα μνήμης και θα μου δώσει 128 bytes από αυτή. Αν στην συνέχεια ζητήσω ακόμα 128 bytes μνήμης, θα μου δώσει 128 bytes από την ίδια σελίδα με πριν (είχαν μείνει 4096 – 128 = 3968 bytes αχρησιμοποίητα, θα ήταν μεγάλη σπατάλη μνήμης, να ζητά συνεχώς καινούριες σελίδες για μικρά αιτήματα).

Συνεχίζουμε με μερικές παρατηρήσεις για τον τρόπο που το Linux εκχωρεί φυσική μνήμη όταν του ζητηθεί. Το Linux, όταν εκχωρεί μνήμη, καταγράφει στον Χάρτη Μνήμη της αντίστοιχης διεργασίας, ότι εκχώρησε αντίστοιχο πλήθος από σελίδες μνήμης και δεν κάνει εκείνη την στιγμή καμία πραγματική εκχώρηση φυσικής μνήμης. Την πρώτη φόρα που θα γίνει πρόσβαση (ανάγνωση ή εγγραφή) σε εικονική διεύθυνση μιας σελίδας που εκχωρήθηκε, η CPU θα προκαλέσει Page Fault στο Linux, και τότε μόνο το Linux θα εκχωρήσει φυσική μνήμη στην σελίδα αυτή. Στην περίπτωση ενός NUMA μηχανήματος, την στιγμή αυτή, δηλαδή στην πρώτη πρόσβαση στην σελίδα, το Linux θα δει ποιος κόμβος NUMA έκανε την πρόσβαση, και θα εκχωρήσει την σελίδα στην φυσική μνήμη του κόμβου αυτού. (αυτό είναι γνωστό ως *first-touch policy*.)

Επιστρέφοντας στην υλοποίηση που έχουμε δώσει, ο τρόπος που θέλουμε να λειτουργήσει ο κώδικας, είναι η `malloc` που εκτελεί το master thread να λάβει κάποια μνήμη, η οποία ακόμα δεν θα αντιστοιχίζει σε φυσική μνήμη. Όταν τρέξει το πρώτο παραλληλο loop, οι πυρήνες θα είναι οι πρώτοι που θα κάνουν πρόσβαση στους πίνακες που

τους αντιστοιχούν, οπότε λέμε ότι το Linux θα εκχωρήσει τους πίνακες στην μνήμη του κόμβου όπου βρίσκεται ο αντίστοιχος πύρηνας.⁴

Αυτό όμως που υποκρύπτεται σε αυτή την υπόθεση, είναι ότι κάθε local πίνακας, θα πάει στην δική του σελίδα. Αυτό, όπως είδαμε πριν, θα ισχύει μόνο αν η ποσότητα μνήμης που ζητηθεί από την malloc είναι μεγαλύτερη από το μέγεθος της σελίδας.

Πράγματι, για 16 συντεταγμένες και 32 clusters, ο local πίνακας κάθε πυρήνα (με στοιχεία αριθμούς κινητής υποδιαστολής διπλής ακριβείας μεγέθους 8 bytes) έχει μέγεθος $16 \cdot 32 \cdot 8 = 4096$ bytes! Έτσι, κάθε local πίνακας τοποθετείται σε χωριστή σελίδα, και το παραπάνω σκεπτικό λειτουργεί.

Αντίθετα, για 1 συντεταγμένη και 4 clusters, ο local πίνακας έχει μέγεθος $1 \cdot 4 \cdot 8 = 32$ bytes. Ακόμα και για 64 πυρηνές, λαμβάνουμε συνολικά $32 \cdot 64 = 2048$ bytes, οπότε όλοι οι υποπίνακες τοποθετούνται από την malloc στην ίδια σελίδα, οπότε θα εκχωρηθούν στην φυσική μνήμη ενός συγκεκριμένου NUMA κόμβου, που προφανώς, είναι ο αντίθετο από αυτό που θέλαμε, και φυσικά, η λύση αυτή αντιμετωπίζει τα ίδια ακριβώς προβλήματα με πριν.

2.2.2.5 Διορθωμένη υλοποίηση

Σκεφτόμαστε, μήπως η malloc έχει thread-safe υλοποίηση και αυτή η υλοποίηση φροντίζει για την εκχώρηση μνήμης από χωριστές δεξαμενές για κάθε κόμβο NUMA. Πιο συγκεκριμένα, σκεφτόμαστε αν η malloc ελέγχει ποιος πυρήνας την κάλεσε, και εκχωρεί μνήμη από την φυσική μνήμη του κόμβου NUMA που ανήκει αυτός ο πυρήνας. Αναφερούμε, πως εκ πρώτης όψεως, η υλοποίηση της malloc, μπορεί να μην είναι καν thread-safe, οπότε αν ταυτόχρονα την καλέσουμε από διαφορετικούς πυρηνές, να προκαλέσουμε race conditions.

Για να το εξετάσουμε, επισκεφτήκαμε το documentation της glibc, στην ιστοσελίδα https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html που με την σειρά της μας έστειλε στην <https://sourceware.org/glibc/wiki/MallocInternals>.

Πράγματι η malloc έχει thread-safe υλοποίηση και πράγματι διατηρεί χωριστές δεξαμενές μνήμης, η οποία ονομάζονται arenas. Ωστόσο, δεν υπάρχουν εγγυήσεις για το ποια arena χρησιμοποιεί ένα νήμα. Όπως χαρακτηριστικά αναφέρεται:

“While this malloc is aware of multiple threads, that’s pretty much the extent of its awareness - it knows there are multiple threads. There is no code in this malloc to optimize it for NUMA architectures, coordinate thread locality, sort threads by core, etc. It is assumed that the kernel will handle those issues sufficiently well. Each thread has a thread-local variable that remembers which arena it last used. If that arena is in use when a thread needs to use it the thread will block to wait for the arena to become free. If the thread has never used an arena before then it may try to reuse an unused one, or pick the next one on the global list.”

Αυτή η υλοποίηση έχει νόημα αν τα νήματα εκτέλεσης δεν είναι στατικά ανατεθειμένα σε πυρήνες του υλικού, που βρίσκονται κατά στατικό τρόπο σε συγκεκριμένους NUMA κόμβους, δηλαδή αν επιτρέπονται migrations.

Θα θέλαμε, όμως, να υπάρχουν εγγυήσεις ότι η μνήμη που ζητάμε θα εκχωρηθεί στην φυσική μνήμη του NUMA κόμβου στον οποίο ανήκει ο πυρήνας που την ζήτησε.

Αναζητώντας, εντοπίσαμε την libnuma, που έχει την συνάρτηση void *numa_alloc_local(size_t size), η οποία υλοποιεί ακριβώς την λειτουργία που θέλουμε. Πιο συγκεκριμένα το σχετικό manpage αναφέρει:

“numa_alloc_local() allocates size bytes of memory on the local node. The size argument will be rounded up to a multiple of the system page size. This function is relatively slow compared to the malloc(3) family of functions. The memory must be freed with numa_free(). On errors NULL is returned.”

Μετρήσεις χρόνου (θα παρατεθούν και σχολιαστούν στην συνέχεια), έδειξαν ότι η χρήση της numa_alloc_local() οδηγεί σε ίσους χρόνους με την χρήση της calloc(), οπότε μάλλον στην συγκεκριμένη περίπτωση, η calloc() ήταν εκχώρησε μνήμη στον τοπικό NUMA κόμβο. Ωστόσο, θα μπορούσε αυτό να μην είχε γίνει, η σημασιολογία της calloc δεν το επιβάλλει. Συνεπώς στην συνέχεια χρησιμοποιούμε την numa_alloc_local().

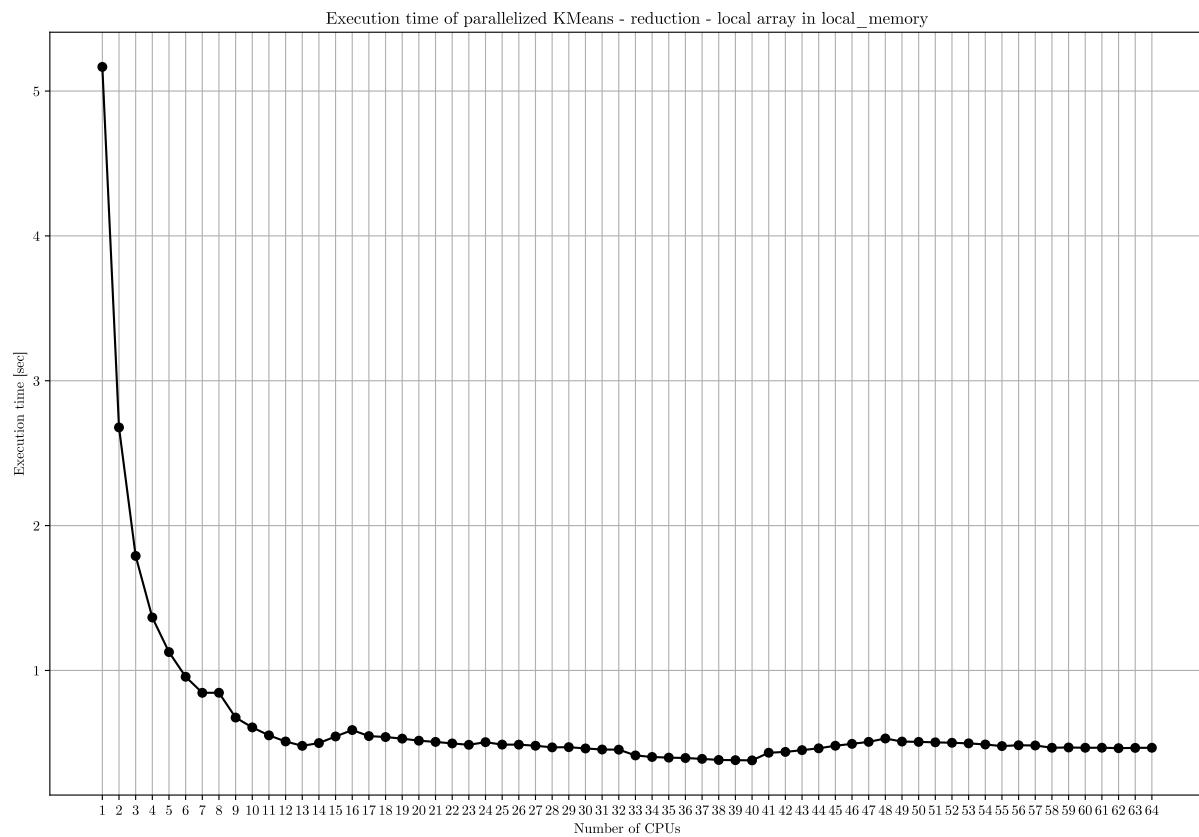
Φροντίζουμε φυσικά για την χρήση numa_free() αντί free().

2.2.2.6 Μετρήσεις επίδοσης – παρατηρήσεις στην διορθωμένη υλοποίηση

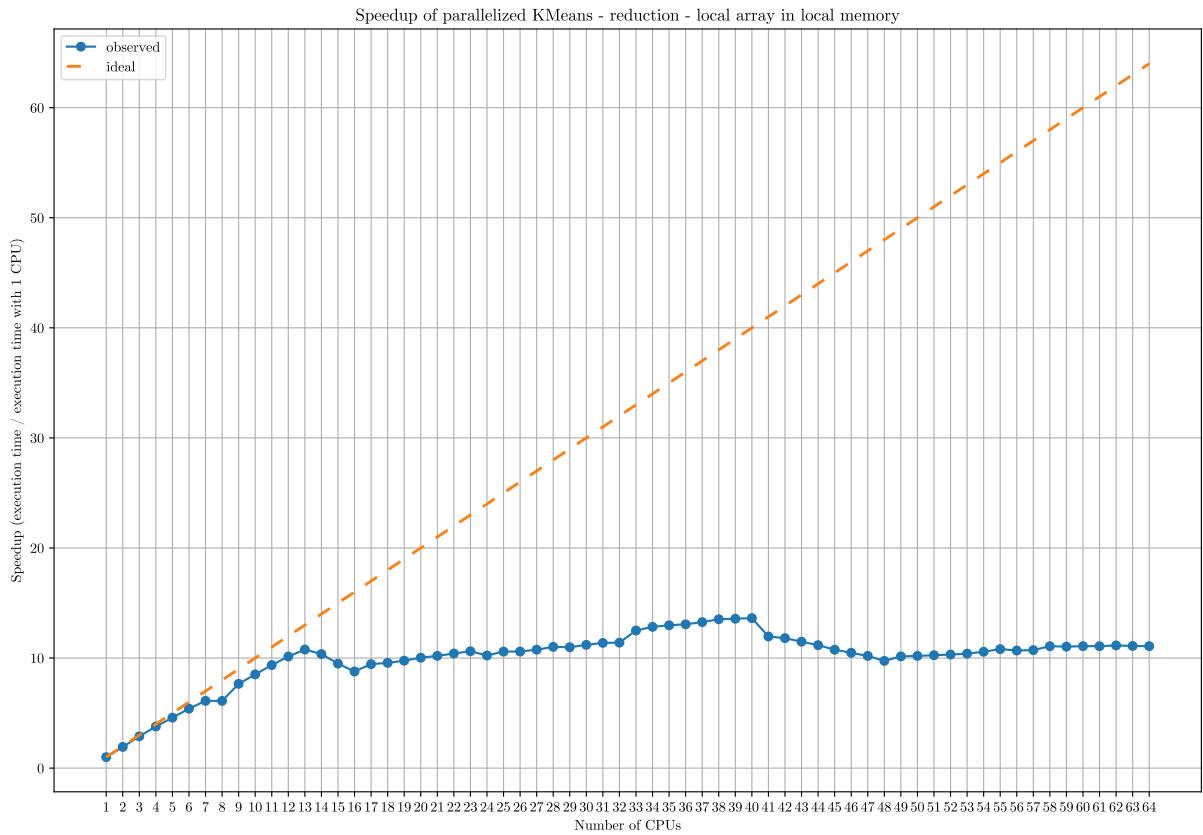
Εκτελέστηκε το προηγούμενο προβληματικό configuration με 1 συντεταγμένη και 4 clusters.

⁴επειδή το dataset είναι τυχαίο και αρκετά μεγάλο, με αρκετά υψηλή πιθανότητα θα γίνουν προσβάσει σε όλα τα στοιχεία των local πινάκων μέσα στο παράλληλο for.

Μετρήθηκε ο χρόνος εκτέλεσης για πλήθη επεξεργαστών 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 13 και η επιτάχυνση στο σχήμα 14.



Σχήμα 13: Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – διορθωμένη υλοποίηση – 2o configuration



Σχήμα 14: Επιτάχυνση για reduction παραλληλοποιημένο KMeans – διορθωμένη υλοποίηση – 2o configuration

Παρατηρούμε ότι μέχρι περίπου 13 πυρήνες η κλιμάκωση είναι γραμμική και σχετικά κοντά στην ιδανική. Πιο συγκεκριμένα για 13 πυρήνες επιτυγχάνεται επιτάχυνση 10.8x που αντιστοιχεί σε συντελεστή γραμμικής κλιμάκωσης περίπου 0.83 (έναντι του ιδανικού 1).

Παρόλα αυτά, για περισσότερους από 13 πυρήνες η επίδοση όχι απλά δεν κλιμακώνει, αλλά γίνεται και χειρότερη. Ελάχιστος χρόνος, κατ' απόλυτο νούμερο, επιτυγχάνεται για 40 πυρήνες και είναι 380 milliseconds.

2.2.2.7 Σχολιασμός στην διορθωμένη υλοποίηση

Παρόλο που διορθώθηκε το πρόβλημα που περιγράψαμε πριν (που περισσότερο μάλλον μπορεί να χαρακτηριστεί ως “bug”, αφού εξαρχής τον ίδιο στόχο είχαμε, απλά τον υλοποιήσαμε λάθος), το δεύτερο configuration εξακολουθεί να μην κλιμακώνει για πάνω από 13 πυρήνες, ενώ το πρώτο configuration κλιμακώνει. Ανακύπτει το ερώτημα γιατί να συμβαίνει αυτό.

Για να μπουμε καλύτερα στο κλίμα, υπενθυμίζουμε ότι και τα δύο configuration έχουν δεδομένα συνολικού μεγέθους $N = 256$ MiB, και εκτελούν 10 επαναλήψεις της βασικής επανάληψης του KMeans.

Η 1η διαφορά είναι πως στο 1o configuration τα σημεία έχουν 16 συντεταγμένες, ενώ στο 2o έχουν 1 συντεταγμένη.

Η 2η διαφορά είναι πως στο 1o configuration υπάρχουν 32 clusters, ενώ στο δεύτερο μόλις 4.

Σκοπός μας είναι να συγκρίνουμε την αριθμητική ένταση που έχουν οι εκτελέσεις για τα δύο configurations, ως προς τα δεδομένα εισόδου. Τελικός στόχος είναι να δείξουμε πως το πρώτο configuration χρειάζεται περισσότερο χρόνο υπολογισμού ανάμεσα στις αναγνώσεις από τα δεδομένα εισόδου, οπότε στον χρόνο αυτό προλαβαίνουν όλοι οι πυρήνες να εξυπηρετηθούν, ενώ το δεύτερο configuration, δεν έχει τόσους υπολογισμούς να κάνει, οπότε στον χρόνο που το πρώτο configuration θα έχανε χρήσιμους υπολογισμούς, απλά περιμένει να έρθει η σειρά του για πρόσβαση στους πίνακες.

Έστω c το πλήθος των clusters και d το πλήθος συντεταγμένων.

Για κάθε σημείο που διαβάζεται από την μνήμη, αρχικά εκτελείται η `find_nearest_cluster` που εκτελεί $O(c \cdot d)$ υπολογισμούς (με γραμμική αναζήτηση ψάχνει το κοντινότερο cluster και κάθε υπολογισμός απόστασης κοστίζει $O(d)$), οι οποίοι, μάλιστα, είναι και πολλαπλασιασμοί αριθμών κινητής υποδιαστολής διπλής ακριβείας, που είναι από

τις σχετικά χρονοβόρες εντολές. Έπειτα γίνονται ενημερώσεις στους τοπικούς πίνακες κόστους $O(d)$ (προσθέσεις διπλής ακριβείας).

Συνολικά, μπορούμε να θεωρήσουμε ότι για κάθε σημείο των δεδομένων εισόδου που διαβάζεται από την μνήμη, γίνονται $O(c \cdot d)$ υπολογισμοί. Μάλιστα, ο χρόνος αυτός αφιερώνεται σχεδόν αποκλειστικά στους $c \cdot d$ πλήθους πολλαπλασιασμούς αριθμών διπλής ακριβείας, που είναι και οι πιο χρονοβόροι.

To 1^o configuration δίνει $c_1 \cdot d_1 = 16 \cdot 32 = 512$, ενώ το δεύτερο configuration δίνει $c_2 \cdot d_2 = 4 \cdot 1 = 4$, οπότε:

$$c_1 d_1 = 128 \cdot c_2 d_2 \quad (2)$$

Με άλλα λόγια, στους πρώτο configuration ανάμεσα στις προσβάσεις στα δεδομένα εισόδου μεσολαβεί 128 φορές παραπάνω χρόνος από ότι στο δεύτερο configuration.

Βέβαια πρέπει να σημειωθεί ότι κάθε σημείο απαιτεί $O(d)$ αναγνώσεις, αφού έχει d συντεταγμένες. Συνεπώς τελικά η αριθμητική ένταση ή *operational intensity* πρόκυπτει να είναι $O(c)$.

Ισχυριζόμαστε, συνεπώς, πως στο δεύτερο configuration, οι προσβάσεις μνήμης έρχονται τόσο κοντά που η μνήμη δεν έχει προλάβει να στείλει στους άλλους πυρήνες τα δεδομένα που έχουν ζητήσει, οπότε ο πυρήνας που ζήτησε τα δεδομένα αναγκάζεται να περιμένει την σειρά του!

Τυποστηρίζουμε τον ισχυρισμό μας με αριθμητικά στοιχεία.

Θα μετρήσουμε πόσο χρόνο θέλει ένας πολλαπλασιασμός αριθμών διπλής ακριβείας.

Γράψαμε το εξής απλό benchmark σε C++:

```

1 #include<iostream>
2 #include<chrono>
3
4 using namespace std;
5
6 #define N 1000000000 //0
7
8 int main() {
9     double pi      = 3.141592653589793;
10    double epsilon = 2.718281828459045;
11
12    auto start = std::chrono::high_resolution_clock::now();
13    for (long long int i=0; i<N; ++i) {
14        double foo = pi * epsilon;
15    }
16    auto end = std::chrono::high_resolution_clock::now();
17
18    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end -
19    start).count();
20    cout << static_cast<double>(duration) / N << endl;
21    cout << duration << endl;
22 }
```

Κάνοντας emit την assembly επιβεβαιώσαμε ότι γίνεται αυτό που θέλουμε. Θέσαμε αριθμούς που να μην έχουν ίδια τα ψηφία τους, ώστε να μην πέσουμε σε κάποια παθολογική περίπτωση, όπου η Floating Point Unit κάνει κάποια έξυπνη βελτιστοποίηση και βρίσκει γρήγορα το αποτέλεσμα.

Βρέθηκε ότι κάθε πολλαπλασιασμός απαιτεί περίπου 2.6 nanoseconds. Άρα στο πρώτο configuration απαιτούνται περίπου $2.7 \cdot c_1 d_1 = 1382$ ns ανάμεσα σε δύο προσβάσεις σημείων, ενώ στο δεύτερο configuration απαιτούνται περίπου $2.6 \cdot c_2 d_2 = 10$ ns ανάμεσα σε δύο προσβάσεις σημείων.

Με το STREAM benchmark, μετρήθηκε το εύρος ζώνης μνήμης ενός NUMA κόμβου σε 10 GB/s = 10 bytes/nanosecond.

Ένα σημείο αποτελείται από d αριθμούς κινητής υποδιαστολής διπλής ακριβείας, μεγέθους 8 bytes καθενας, οπότε η μετάδοση κάθε σημείου καταλαμβάνει τον δίαυλο για $\frac{8d}{10}$ nanoseconds.

Στο πρώτο configuration, συνεπώς, η μετάδοση ενός σημείου καταλαμβάνει τον δίαυλο για 26 ns, ενώ στο δεύτερο για 0.8 ns.

Αυτό σημαίνει ότι ανάμεσα σε δύο αναγνώσεις σημείων από έναν πυρήνα, στο πρώτο configuration χωράνε το πολύ $\frac{1382}{26} = 53$ μεταδόσεις, ενώ στο δεύτερο configuration χωράνε το πολύ $\frac{10}{0.8} = 13$ μετάδοσεις.

Συνεπώς, για να συντηρηθεί ένας ρυθμός που όλοι οι πυρήνες αφιερώνουν τον χρόνο τους σε χρήσιμο έργο, μπορούν στο πρώτο configuration να ζητάνε ταυτόχρονα το πολύ $53 + 1 = 54$ πυρήνες, ενώ στο δεύτερο configuration το πολύ $13 + 1 = 14$ πυρήνες. Αν χρησιμοποιηθούν περισσότεροι, οι πυρήνες θα περιμένουν την μνήμη να τους δώσει δεδομένα χωρίς να κάνουν χρήσιμο έργο, οπότε για μια ακόμη φορά θα έχουμε σειριοποίηση της παράλληλης εκτέλεσης, πάνω στον δίαυλο της μνήμης.

Πράγματι, οι πειραματικές μετρήσεις που έγιναν προηγούμενως, έδειξαν, ότι στο πρώτο configuration μέχρι 32 φυσικούς πυρήνες που έγιναν μετρήσεις, η κλιμακωσιμότητα παρέμεινε γραμμική.⁵ Αντίθετα, στο δεύτερο configuration, πράγματι η κλιμακωσιμότητα είναι γραμμική ως τους 13 πυρήνες και μετά ο χρόνος εκτέλεσης σταθεροποιείται, καθώς η μνήμη δεν μπορεί να δώσει μεγαλύτερο εύρος ζώνης οπώς απαιτούν οι επεξεργαστές.

Η πρόβλεψη μας για 14 πυρήνες αντί 13 είναι αρκετά κοντινή και η διαφορά αποδίδεται σε σφάλματα προσέγγισης, καθώς έχουμε θεωρήσει ότι για κάθε ανάγνωση σημείου, γίνονται μόνο $c \cdot d$ πολλαπλασιασμοί αριθμών διπλής ακριβείας, που πράγματι είναι οι πιο χρονοβόροι, όμως υπάρχουν και κάποιες ακόμα εντολές (οι προθέσεις για την ενημέρωση των clusters καθώς και εντολές ακεραίων για τις επαγγελματικές μεταβλητές, ελέγχου, κ.λπ.), που και αυτές απαιτούν όλες μαζί κάποιον χρόνο, επιτρέποντας σε αυτόν τον χρόνο λίγους ακόμα πυρήνες να ζητήσουν δεδομένα από την μνήμη.

⁵δεν αναφερόμαστε σε ≥ 33 διότι υπάρχει το πρόβλημα με το κακό schedule που έχει σχολιαστεί παραπάνω

2.2.3 Υλοποίηση με NUMA-aware allocation

2.2.3.1 Σκεπτικό

Για να αυξήσουμε την κλιμακωσιμότητα πέραν των 13 πυρήνων, θα πρέπει κάπως να αλλάξουμε τους συσχετισμούς που βρήκαμε προηγουμένως.

Σκεπτόμενοι σε λίγο πιο high-level, η έλλειψη κλιμάκωσης έχει προκύψει επειδή έχουμε παραλληλοποιήσει τους υπολογισμούς, όμως έχουμε αφήσει σειριακό τον δίσυλο μνήμης, αντιμετωπίζοντας κατ' αυτόν τον τρόπο, τις συνέπειες του Νόμου του Amdahl.

Συνεπώς, το λογικό επόμενο, είναι να παραλληλοποιήσουμε και τον δίσυλο μνήμης!

Η λογική NUMA μας προσφέρει ακριβώς αυτή την δυνατότητα. Η μνήμη έχει χωριστεί σε χωριστές μνήμες, και καθένα από αυτά τα τμήματα έχει ανατεθεί σε διαφορετικό σύνολο επεξεργαστών.

Αν οι επεξεργαστές ζητάνε δεδομένα εισόδου μόνο από την μνήμη του NUMA κόμβου που βρίσκονται, τότε η θεωρητική ανάλυση που περιγράφηκε προηγουμένως ισχύει χωριστά για κάθε κόμβο NUMA, οπότε ισχύουν και τα αντίστοιχα όρια πυρήνων που υπολογίστηκαν χωριστά για κάθε κόμβο NUMA.

Έτσι έχοντας 4 κόμβους NUMA, αναμένουμε να τετραπλασιάσουμε τα όρια, δηλαδή στο δεύτερο, και προβληματικό ως τώρα configuration, να μπορούν να μπουν ως 9 επεξεργαστές ανά κόμβο NUMA, οπότε με 4 κόμβους να πάμε σε μέχρι 36 επεξεργαστές που θα είναι αρκετά καλό, με δεδομένο ότι κάθε κόμβος NUMA έχει 8 επεξεργαστές στο υπολογιστικό μας σύστημα.

2.2.3.2 Υλοποίηση

Παρατηρούμε ότι με τον τρόπο που έχει γίνει η παραλληλοποίηση, ένας πυρήνας εκτελεί υπολογισμούς για ένα συγκεκριμένο σύνολο σημείων και κανείς άλλος πυρήνας δεν εκτελεί υπολογισμούς σε αυτά.

Συνεπώς, η κατανομή των δεδομένων εισόδου στις χωριστές μνήμες των NUMA κόμβων, που θα κάνουν τους αντίστοιχους υπολογισμούς, φαίνεται μια εφικτή και καλά υποσχομένη ιδέα.

Θα θέλαμε τα δεδομένα που αντιστοιχούν σε χωριστούς NUMA κόμβους να είναι τοποθετημένα στις αντίστοιχες μνήμες, όμως, τα δεδομένα που αφορούν ίδιους πυρήνες ενός NUMA κόμβου να είναι κοντά στην μνήμη, ώστε να εκμεταλλευτούμε τις κοινές χρυφές μνήμες που έχουν νήματα ίδιου NUMA κόμβου.

Σκεφτήκαμε τον ίδιο πίνακα να τον κατανείμουμε σε κομμάτια σε διαφορετικές φυσικές μνήμες (η εικονική μνήμη επιτρέπει μια τέτοια λύση). Για να λειτουργήσει αυτό, θα πρέπει το κομμάτι που αναθέτουμε σε κάθε πυρήνα να έχει μέγεθος πολλαπλάσιο του μεγέθους σελίδας. Το στατικό schedule που χρησιμοποιούμε, και που θέλουμε να συνεχίστουμε να χρησιμοποιούμε για να έχουμε ισόχρονη ποσότητα δουλειάς σε κάθε πυρήνα, αναθέτει πιθανώς πλήθος objects σε κάθε πυρήνα, που δεν δίνουν μέγεθος ακέραιου πολλαπλάσιου μεγέθους σελίδας. Η απάντηση σε αυτό, θα ήταν να προσθέσουμε λίγο padding ώστε να συμπληρωθεί το μέγεθος σελίδας. Μια τέτοια λύση, όμως, θα επηρέαζε τον τρόπο διεύθυνσιο δότησης του πίνακα, και γενικά η λύση θα γινόταν σημαντικά πιο σύνθετη.

Ο πίνακας objects που δημιουργείται έχει μέγεθος $N \text{ MiB} = N \cdot 2^{20} \text{ bytes}$. Συνεπώς, αν το πλήθος των πυρήνων είναι (ακέραια) δύναμη του 2, τότε το τμήμα του πίνακα που αντιστοιχεί σε κάθε πυρήνα είναι ακέραιο πολλαπλάσιο του 4096, που είναι το μέγεθος σελίδας! Επίσης ο πίνακας έχει μέγεθος τουλάχιστον 1 MB (για $N = 1$), που είναι μεγαλύτερο από το MMAP_THRESHOLD = 128 KB (από προεπιλογή), οπότε η malloc θα οδηγήσει σε κλήση της mmap, οπότε, τελικά, ο πίνακας που θα λάβουμε θα είναι aligned στο μέγεθος της σελίδας. Συνεπώς, πρόβλημα θα δημιουργηθεί μόνο αν το πλήθος πυρήνων δεν είναι δύναμη του 2 και αυτό θα είναι για σχετικά λίγα στοιχεία. Πιο συγκεκριμένα, για $N = 256$ σε κάθε NUMA node αντιστοιχούν (ας πούμε για max 4 NUMA nodes) τουλάχιστον $\frac{256}{4} = 64 \text{ MiB}$ μνήμης = 16384 σελίδες με τα προβληματικά στοιχεία που βρίσκονται σε απομακρυσμένη μνήμη να ανήκουν το πολύ σε μία σελίδα (την πρώτη), δηλαδή το πολύ 0.006% των στοιχείων, που ακόμα και αν αργήσουν να υπολογιστούν, μάλλον δεν μπορούν να οδηγήσουν σε αυξημένο χρόνο υπολογισμού για το συνολικό πρόβλημα. Όπως θα δούμε εξάλλου πράγματι δεν επηρεάζουν.

Με βάση όλα τα προηγούμενα, στο αρχείο file_io.c απλά εκτελέσαμε την αρχικοποίηση του πίνακα παράλληλα, προσθέτοντας δηλαδή την οδηγία OpenMP πριν τον βρόχο υπολογισμού:

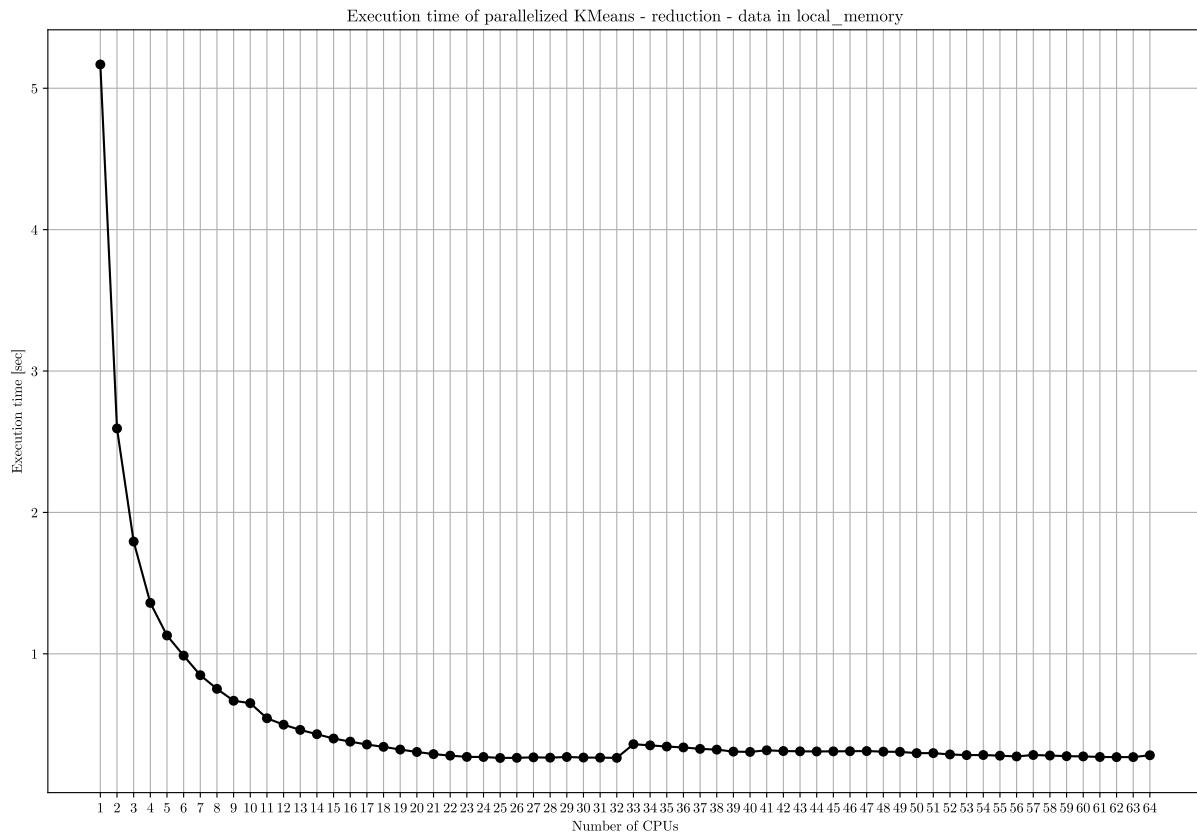
```
#pragma omp parallel for schedule(static)
```

Με βάση την πολιτική first-touch και με βάση όλα τα σχόλια που προηγήθηκαν, η εκχώρηση μνήμης θα γίνει όπως περιγράφηκε.

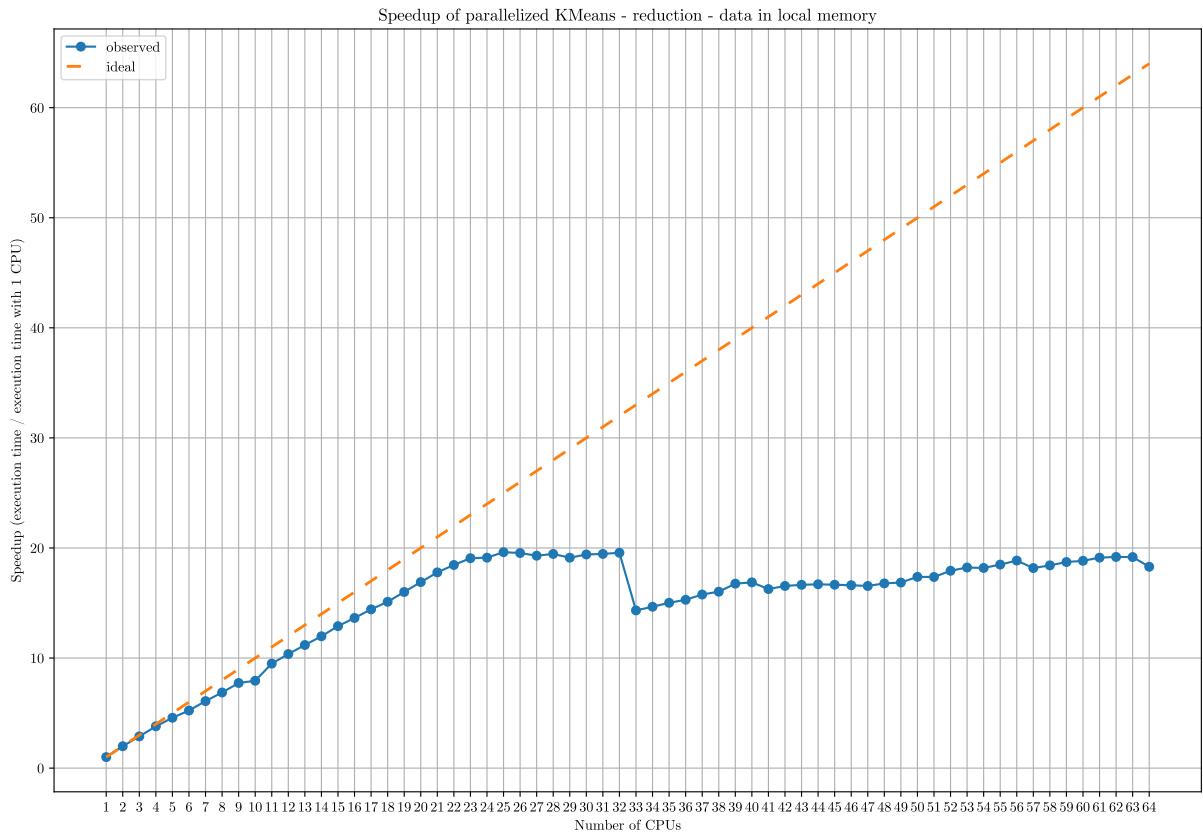
2.2.3.3 Μετρήσεις επίδοσης – παρατηρήσεις

Εκτελέστηκε το προηγούμενο προβληματικό configuration με 1 συντεταγμένη και 4 clusters.

Μετρήθηκε ο χρόνος εκτέλεσης για πλήθη επεξεργαστών 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 15 και η επιτάχυνση στο σχήμα 16.



Σχήμα 15: Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation



Σχήμα 16: Επιτάχυνση για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation

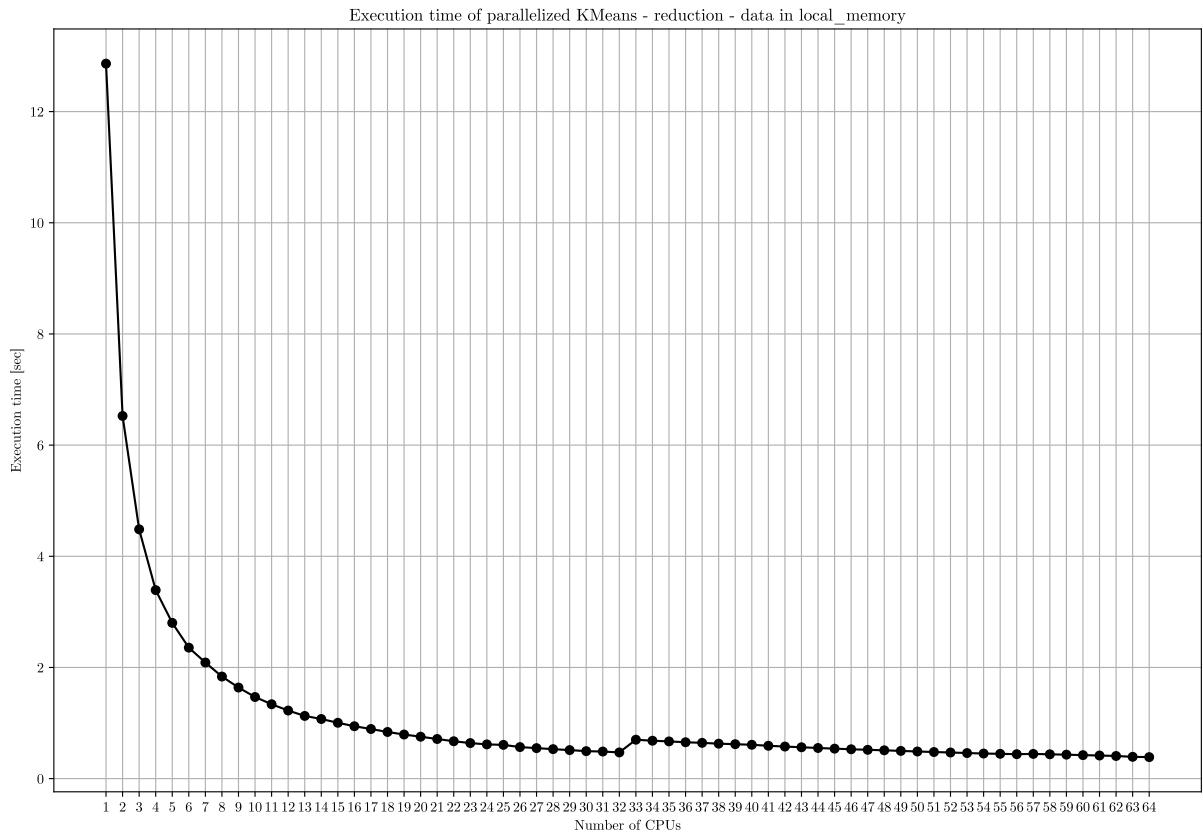
Παρατηρούμε ότι μέχρι 23 πυρήνες επιτυγχάνεται γραμμική κλιμάκωση, σχετικά κοντά στην ιδανική. Για 23 πυρήνες επιτυγχάνεται επιτάχυνση 19x, που αντιστοιχεί σε συντελεστή γραμμικής κλιμάκωσης 0.83 (σε σχέση με τον ιδανικό 1).

Για περισσότερους από 23 πυρήνες, η κλιμακωσιμότητα σταματά και ο χρόνος εκτέλεσης σταθεροποιείται ανεξάρτητα του πλήθους πυρήνων. Αυτό γίνεται μέχρι 32 πυρήνες.

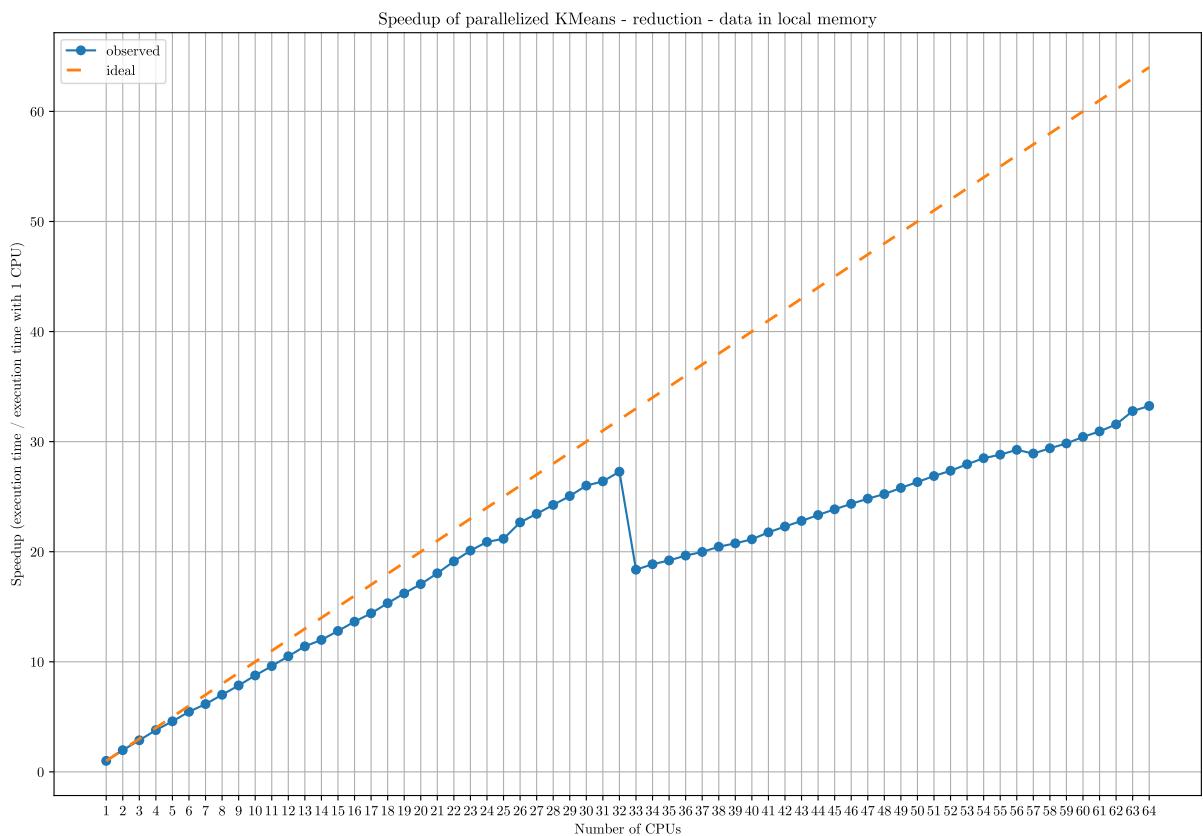
Για περισσότερους από 32 πυρήνες, η απόδοση μειώνεται απότομα, όμως αυτό οφείλεται στο κακό scheduling επειδή ξεκινά η χρήση του Simultaneous Multithreading (SMT), όπως είχαμε σχολιάσει νωρίτερα.

Εκτελέστηκε επίσης το configuration με 16 συντεταγμένες και 32 clusters.

Μετρήθηκε ο χρόνος εκτέλεσης για πλήθη επεξεργαστών 1 ως 64. Ο χρόνος εκτέλεσης φαίνεται στο σχήμα 17 και η επιτάχυνση στο σχήμα 18.



Σχήμα 17: Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation – 1o configuration



Σχήμα 18: Επιτάχυνση για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation – 1o configuration

Παρατηρούμε ότι μέχρι και 32 φυσικούς πυρήνες, η κλιμάκωση είναι γραμμική, και μάλιστα, η κλιμακωσιμότητα της επιτάχυνσης είναι ίδια με προηγουμένως, όπου δεν γινόταν η NUMA-aware allocation. Ωστόσο, σε απόλυτο χρόνο, οι εκτέλεσεις είναι όλες περίπου 13% ταχύτερες! (παρόλο που ο χρόνος εκτέλεσης κλιμακώνει με τον ίδιο τρόπο)

2.2.3.4 Σχολιασμός

Ανακύπτει το ερώτημα τι συμβαίνει και στους 23 πυρήνες ο χρόνος εκτέλεσης σταθεροποιείται, γεγόνος που όπως εξηγήσαμε, δεν αναμενόταν πριν τους 36 επεξεργαστές, και που δηλαδή, δεν θα το βλέπαμε, καθώς έχουμε 32 φυσικούς πυρήνες, και δεν μελετάμε τα νήματα του Simultaneous Multithreading λόγω των προβλημάτων με το scheduling.

Προκύπτει επίσης το ερώτημα γιατί συγκεκριμένα το 2o configuration να μην κλιμακώνει, ενώ το 1o configuration να διατηρεί καλή κλιμακωσιμότητα.

Τυπόθεση bottleneck στο εύρος ζώνης διαύλου μνήμης – μάλλον καταρρίπτεται

Η ανάλυση που παραθέσαμε παραπάνω, προβλέπει πως ο εύρος ζώνης διαύλου μνήμης δεν θα έπρεπε να περιορίσει τον χρόνο εκτέλεσης και την κλιμακωσιμότητα.

Για λόγους επιβεβαίωσης, σκεφτόμαστε και εκτελούμε έναν δεύτερο διαφορετικό ελέγχο μήπως ευθύνεται ο διάυλος μνήμης, και κάτι χάσαμε στην ανάλυσή μας, δηλαδή μήπως όντως έχουμε φτάσει στο μέγιστο εύρος ζώνης που μπορεί να μας δώσει η μνήμη κάθε κόμβου.

Όπως και στην 1η Άσκηση, χρησιμοποιούμε το likwid-perfctr⁶ στην σειριακή εκδοχή, για να μετρήσουμε πόση μεταφορά δεδομένων από και προς την μνήμη επιβάλλει η ίδια η σημασιολογία του αλγόριθμου.

Αναφερόμενοι στο προβληματικό configuration, μετρήθηκε πως από την μνήμη πρέπει να διαβαστούν συνολικά 4.46 GB και να γραφτούν 1.83 GB μνήμης, με τα συνολικά δεδομένα που μεταφέρονται να είναι 6.29 GB.

Με το STREAM benchmark, μετρήθηκε το εύρος ζώνης μνήμης ενός NUMA κόμβου σε 10 GB/s.

Αν:

1. Αγνοηθεί η cache. Αυτό είναι σχετικά έγκυρο, διότι οι πίνακες διαβάζονται σε κάθε επανάληψη του KMeans από την μνήμη, αφού ο πίνακας των objects είναι μεγαλύτερος από την cache.
2. Ληφθούν υπόψη και οι εγγραφές. Αυτό είναι σχετικά έγκυρο, καθώς οι ουρές εγγραφών στις caches θα έκρυβαν τις καθυστερήσεις λόγω εγγραφών, αν ο μέσος ρυθμός εγγραφών είναι μικρότερος από το εύρος ζώνης της μνήμης, που εδώ ίσως δεν ισχύει διότι γίνονται πολλές μεταφορές στην μνήμη.
3. Θεωρήσουμε 4πλάσιο εύρος ζώνης λόγω ύπαρξης 4 διαφορετικών NUMA nodes. Επιβεβαιώθηκε μέσω του παραλληλοποιημένου STREAM, πως αν όλοι διαβάζουν ταυτόχρονα από τις μνήμες τους, πράγματι επιτυγχάνεται 4πλάσιο εύρος ζώνης από την μνήμη συνολικά. Εξάλλου είναι και θεωρητικά σωστό, αφού κάθε κόμβος NUMA έχει χωριστό διάυλο μνήμης.

προκύπτει ελάχιστος χρόνος μόνο λόγω των προσβάσεων μνήμης $\frac{6.29}{4 \cdot 10} = 0.16$ sec. Ο χρόνος εκτέλεσης έχει σταθεροποιηθεί στα 0.26 – 0.27 sec.

Τα στοιχεία αυτά υποδηλώνουν ότι το bottleneck, τουλάχιστον αποκλειστικά, μάλλον δεν είναι το εύρος ζώνης του διαύλου μνήμης, διότι ο μέσος ρυθμός μεταφοράς δεδομένων από και προς την μνήμη ανά κόμβο NUMA προκύπτει 5.8 GB/s, που είναι σημαντικά μικρότερο των 10 που μας έδωσε το STREAM.

Τυπόθεση bottleneck στο σειριακό τμήμα – καταρρίπτεται

Σκεφτόμαστε μήπως ο χρόνος που απομένει (τα 0.27 – 0.16 = 0.11 sec) αφορά σε χρόνο που χρειάζεται το master thread για να εκτελέσει το σειριακό μέρος του αλγορίθμου. Με άλλα λόγια, σκεφτόμαστε μήπως αντιμετωπίζουμε τις πρώτες συνέπειες του Νόμου του Amdahl. Εκτιμάται πως δεν συμβαίνει και θα μετρήσουμε πως δεν συμβαίνει.

Επιχειρηματολογούμε τον ισχυρισμό μας με αριθμητικά στοιχεία. Θα υπολογίσουμε τον χρόνο που χρειάζεται το master thread. Ο αλγόριθμος KMeans, όπως έχει υλοποιηθεί, δίνει χρόνο εκτέλεσης γραμμικό ως προς το πλήθος σημείων, δηλαδή ισοδύναμα γραμμικό ως προς την παράμετρο N .

⁶likwid-perfctr -g MEM ./kmeans_seq -s 256 -n 1 -c 4 -l 10

Άρα ο χρόνος εκτέλεσης είναι της μορφής:

$$T(n) = a \cdot n + b \quad (3)$$

Παρατηρούμε το εξής ενδιαφέρον. Το μέρος του κώδικα που έχει παραλληλοποιηθεί δίνει χρόνο εκτέλεση ευθέως ανάλογο (προσοχή ανάλογο, όχι γραμμικό) του n , αφού ο παραλληλοποιημένος βρόχος έχει επαναλήψεις ανάλογες του n . Αντίθετα, το μέρος του κώδικα που δεν έχει παραλληλοποιηθεί, δεν εξαρτάται καθόλου από το n , αφού γίνονται αθροίσεις, υπολογισμοί, κ.λπ. για κάθε cluster και δεν εμπέκονται τα ίδια τα σημεία. Με άλλα λόγια, ο όρος $a \cdot n$, αφορά το τμήμα που παραλληλοποιήθηκε, και το b αφορά το τμήμα που θα εκτελέσει το master thread.

Συνεπώς, μπορούμε να υπολογίσουμε τον χρόνο που απαιτεί το master thread ως την σταθερά b !

Για τον σκοπό αυτό, εκτελέσαμε για $N = 128$ MiB και $N = 256$ MiB και μετρήσαμε χρόνους εκτέλεσης, στην σειριακή εκδοχή, αντίστοιχα, 2.578 sec και 5.156 sec. Επιλύοντας το σύστημα εξισώσεων που δίνουν οι 3 βρίσκουμε $b \approx 0$ sec, και όχι 0.11 sec.

Ο χρόνος b που εκτελεί το master thread το σειριακό τμήμα δεν αναμένουμε να επηρεαστεί σημαντικά είτε είναι παραλληλοποιήμενο είτε σειριακό το άλλο τμήμα. Για λόγους επιβεβαίωσης, εκτελέσαμε το προηγούμενο και στην παραλληλοποιημένη εκδοχή για 32 πυρήνες. Για $N = 128$ MiB και $N = 256$ MiB, μετρήθηκαν χρόνοι εκτέλεσης, αντίστοιχα, 0.272 sec και 0.531 sec. Επιλύοντας το σύστημα εξισώσεων που δίνουν οι 3 βρίσκουμε $b \approx 0.01$ sec, που και πάλι, απέχει από το 0.11 sec.

Συνεπώς δεν ευθύνεται το σειριακό τμήμα για τα 0.11 sec που δεν εξηγεί το εύρος ζώνης διαύλου μνήμης, οπότε δεν βρισκόμαστε αντιμέτωποι με τις συνέπειες του Νόμου του Amdahl.

Τυπόθεση bottleneck στον υπολογισμό από την CPU – μάλλον καταρρίπτεται

Θα μπορούσαμε να ισχυριστούμε πως πρόκειται για χρόνο που αναλώνεται σε υπολογισμούς εσωτερικά στην CPU, όμως ούτε αυτό φαίνεται λογική υπόθεση, καθώς τότε ο χρόνος θα έπρεπε να κλιμακώνει με την αύξηση των πυρήνων που δεν συμβαίνει. (η out-of-order execution συνδυαστικά με το prefetching θα επέτρεπαν να γίνει αυτή η κλιμάκωση αφού δεν έχουμε φτάσει το όριο εύρους ζώνης του διαύλου μνήμης).

Σε τελική ανάλυση, δεν προκύπτει σαφές συμπέρασμα γιατί το δεύτερο configuration δεν κλιμακώνει και κυρίως γιατί το δεύτερο δεν κλιμακώνει όμως το πρώτο κλιμακώνει. Πάντως, η σταθεροποίηση του χρόνου έκτελεσης σε σταθερή τιμή, υποδεικνύει μάλλον συνέπεια του Νόμου του Amdahl, οπότε, μάλλον, τα αίτια πρέπει να αναζητηθούν σε κάποιο σειριακό τμήμα (π.χ. μοναδικό κοινό πόρο όπως η μνήμη) που το παράλληλο τμήμα χρησιμοποιεί από κοινού.

2.2.4 Reduction παραλληλοποίηση αυτόματα με OpenMP Reduction Directive

2.2.4.1 Υλοποίηση

Οι σύγχρονες εκδόσεις του OpenMP, από το OpenMP 5.0, υποστηρίζουν να γίνεται reduction και σε πίνακες, όχι μόνο σε βαθμώτες μεταβλητές, θεωρώντας πως η πρόσθεση στους πίνακες γίνεται element-wise, όπως, δηλαδή, στα διανύσματα.

Αυτό διευκολύνει σημαντικά την υλοποίηση, καθώς ο προγραμματιστής δεν χρειάζεται να γράψει χωριστούς πίνακες, ούτε να σκεφτεί πως θα τους κάνει αποδοτικά allocate.

Πιο συγκεκριμένα από την naïve παραλληλοποίηση μεταβαίνουμε στην reduction εκδοχή απλά και μόνο αλλάζοντας την οδηγία που δίνουμε στο OpenMP για την παραλληλοποίηση του βρόχου σε:

```
#pragma omp parallel for schedule(static) reduction(+:delta)
reduction(+:newClusters[:numClusters*numCoords]) reduction(+:newClusterSize[:numClusters])
```

Για λόγους επιβεβαίωσης, έγιναν μετρήσεις επίδοσης και βρέθηκε ότι η κλιμάκωση χρόνου εκτέλεσης είναι ίδια με την εκδοχή που γράψαμε μόνοι μας.

2.2.4.2 Απομακρυσμένη μεταγλώττιση από τοπικό υπολογιστή για την υποδομή του Εργαστηρίου

Δυστυχώς, ο μεταγλωττιστής που είναι εγκατεστημένος στην υποδομή όπου γίνονται οι εκτελέσεις, δεν υποστηρίζει αρκετά σύγχρονη έκδοση του OpenMP. Αυτό το βρίσκουμε εκτυπώνοντας την τιμή της macro _OPENMP (ενδεικτικά με την εντολή gcc -fopenmp -dM -E - | grep OPENMP), που είναι 201307, η οποία αντιστοιχεί σε OpenMP 4.0.

Οπτόσο, δεν πτοούμαστε. Μεταγλωττίζουμε τοπικά, και μάλιστα, βελτιστοποιημένα για το μηχάνημα που θα γίνει η εκτέλεση, και μεταφέρουμε το εκτέλεσιμο.

Τρέχοντας στο μηχάνημα που γίνεται η μεταγλώττιση (χανονικά μέσω του συστήματος Torque) την εντολή: gcc -march=native -Q --help=target | grep march βρίσκουμε για ποια αρχιτεκτονική κάνει βελτιστοποίηση ο GCC.

Βρέθηκε πως γίνεται για την αρχιτεκτονική που ο GCC ονομάζει sandybridge. (όχι τυχαία, αφού ο επεξεργαστής γνωρίζουμε πως είναι της γενιάς Sandy Bridge.)

Με βάση αυτό, προσθέτουμε στα CFLAGS το option: -march=sandybridge ώστε να εκτελέσουμε μεταγλώττιση βελτιστοποιημένη για το μηχάνημα που θα γίνει η εκτέλεση.

Επιπρόσθετα, επειδή υπάρχουν ζητήματα με βιβλιοθήκες του συστήματος που είναι διαφορετικές, κάνουμε το linking να είναι στατικό, ώστε να περιέχει ό,τι απαιτείται, προσθέτοντας στα CFLAGS το option: -static.

Εκτελούμε την μεταγλώττιση, με βάση τα προηγούμενα, στον τοπικό υπολογιστή, μεταφέρουμε το εκτέλεσιμο στο μηχάνημα που γίνεται η εκτέλεση και το τρέχουμε χανονικά. Η εκτέλεση επιτυγχάνει και η κλιμάκωση χρόνου εκτέλεσης είναι πανομοιότυπη. Μάλιστα, επειδή χρησιμοποιούμε νεότερη έκδοση του GCC στον τοπικό μας υπολογιστή, που εκτελεί καλύτερες βελτιστοποιήσεις, βρίσκουμε πως ο χρόνος εκτέλεσης όχι απλά δεν είναι χειρότερος, αλλά είναι περίπου 28% μειωμένος για τον ίδιο κώδικα C!

Με βάση αυτά εκτελέστηκε επιτυχώς στο μηχάνημα sandman, όπως αναφέραμε παραπάνω με τα αντίστοιχα αποτελέσματα που αναφέρθηκαν, το πρόγραμμα με το αυτόματο reduction σε πίνακες, με την νέα έκδοση OpenMP.

2.3 Αξιολόγηση διαφορετικών υλοποιήσεων Κλειδωμάτων στον αλγόριθμο K-means

2.3.1 Περιγραφή – εξήγηση λειτουργίας κλειδωμάτων

Σκοπός είναι η αξιολόγηση διάφορων υλοποιήσεων Κλειδωμάτων. Για τον σκοπό αυτό, χρησιμοποιούμε την προηγούμενη υλοποίηση του αλγορίθμου KMeans, που χρησιμοποιεί κοινό πίνακα για τα clusters και τα μεγέθη των clusters. Οι ενημερώσεις στους πίνακες απαιτείται να γίνονται ατομικά, και αυτό μπορεί να επιτευχθεί με την χρήση κλειδωμάτων.

Στην συνέχεια δοκιμάζουμε τα εξής κλειδώματα, για τα οποία παρέχουμε μια επιγραμματική περιγραφή της λειτουργίας τους:

TAS (Test-and-Set) Χρησιμοποιείται η ατομική εντολή test-and-set(variable, value) σε μια μεταβλητή tύπου boolean, που παιρνεί δύο τιμές LOCKED και UNLOCKED.

Είναι: `lock(): while (test-and-set(lock, LOCKED) == LOCKED);`

και: `unlock(): lock := UNLOCKED.`

Η ατομικότητα του test-and-set εξασφαλίζει ότι δεν πρόκειται ποτέ δύο επεξεργαστές να λάβουν και οι δύο την τιμή UNLOCKED.

TTAS (Test-Test-and-Set) Πρόκειται για προέκταση του προηγούμενου κλειδώματος Test-and-Set. Το προηγούμενο κλειδωμα δημιουργεί συνεχώς εγγραφές σε μια θέση μνήμης, με αποτέλεσμα, στο πρωτόκολλο συνάρφειας μνήμης (cache coherence – π.χ. MESI), αν υπάρχουν δύο κόμβοι που ανταγωνίζονται για το ίδιο κλειδωμα, να προκαλούνται συνεχώς ακυρώσεις στα cache blocks και έτσι η απόδοση να είναι μειωμένη.

Αντί αυτού, στο παρόν κλειδωμα, επεκτείνουμε το Test-and-Set εξετάζοντας πριν την εντολή test-and-set απλά με reads αν το lock έχει τιμή UNLOCKED και μόνο όταν συμβεί αυτό, προκαλούμε τον ανταγωνισμό. Ετσι τον περισσότερο χρόνο, δηλαδή όσο το κλειδωμα χρατίεται από κάποιον, η απόδοση είναι καλύτερη. Η ορθότητα είναι ισοδύναμη με του Test-and-Set, αφού στο τέλος, η πράξη του ίδιου κλειδώματος γίνεται με τον ίδιο τρόπο. Είναι: `lock(): while(1) while (lock == LOCKED); if (test-and-set(lock, LOCKED) == UNLOCKED) break; και: unlock(): lock := UNLOCKED`

Pthread Mutex Lock Πρόκειται για το κλειδωμα pthread_mutex_lock της βιβλιοθήκης pthreads (POSIX threads). Το κλειδωμα αυτό προκαλεί κοίμισμα του αντίστοιχου νήματος από το Λειτουργικό Σύστημα (με αλλαγή περιβάλλοντος στον φυσικό επεξεργαστή – context switch – σε άλλη διεργασία/νήμα), μέχρι το κλειδωμα να απελευθερωθεί οπότε το Λειτουργικό Σύστημα αναλαμβάνει να ξυπνήσει το νήμα.

Ενδεικνύται για μεγάλης διάρκειας αναμονές.

Είναι το μόνο κλειδωμα που μελετάμε που περιλαμβάνει κοίμισμα, καθώς όλα τα υπόλοιπα είναι ενεργού αναμονής (busy waiting).

Pthread Spinlock Πρόκειται για το κλειδωμα pthread_spinlock_t της βιβλιοθήκης pthreads (POSIX threads). Το κλειδωμα αυτό είναι κλειδωμα ενεργού αναμονής (busy waiting) και είναι implementation-defined το πώς υλοποιείται.

Array Lock Υπάρχει ένας circular buffer τόσων θέσεων όσοι οι επεξεργαστές του συστήματος. Όλες οι θέσεις έχουν τιμή LOCKED εκτός μίας που έχει τιμή UNLOCKED. Υπάρχει μια μεταβλητή (πρακτικά ένας δείκτης), η tail, που δείχνει την τελευταία ελεύθερη θέση του πίνακα.

Για το lock() ο επεξεργαστής λαμβάνει την τρέχουσα τιμή της tail, την προχωρά (κυκλικά) κατά 1 και περιμένει μέχρι το αντίστοιχο στοιχείο του πίνακα να πάρει την τιμή UNLOCKED.

Πρέπει κανείς άλλος επεξεργαστής να μην λάβει την ίδια τιμή της tail. Για τον λόγο αυτό όταν λαμβάνεται η τιμή της μεταβλητής χρησιμοποιείται η ατομική εντολή atomic-fetch-and-add που ατομικά λαμβάνει την παλιά τιμή και αυξάνει την μεταβλητή. Για να μην γίνει ατομικά υπολογισμός και του κυκλικού wrapping, αφήνουμε την τιμή να μεγαλώνει, και υπολογίζουμε κάθε φορά σε κάθε νήμα τοπικά το υπόλοιπο ως προς το μέγεθος του πίνακα.

Για το unlock() ο επεξεργαστής θέτει στοην θέση tail που είχε λάβει αρχικά, το LOCKED, ώστε οι επόμενοι να χρειαστεί να περιμένουν να τους δοθεί η σκυτάλη του κλειδώματος, και επίσης θέτει την τιμή UNLOCKED στην κυκλικά επόμενη θέση του κυκλικού πίνακα, πρακτικά δίνοντας στον επόμενο κόμβο κυκλικά την σκυτάλη του κλειδώματος.

Το κλειδωμα αυτό πλεονεκτεί αν προβλέπεται να ανταγωνιστούν πολλοί επεξεργαστές για το κλειδωμα, καθώς κάθε κόμβος περιμένει σε μια δική του τιμή και δεν υπάρχει επικοινωνία κατά την αναμονή. (φροντίζουμε με

κατάλληλο padding, να υπάρχει αρκετό padding ανάμεσα στις τιμές του κυκλικού buffer ώστε κάθε τιμή να είναι σε διαφορετικό cache block).

Το μεγαλύτερο όμως, ίσως, πλεονέκτημα αυτό του κλειδώματος είναι ότι δεν θα συμβεί starvation, δηλαδή αν κάποιος ζητήσει το κλειδωμα, θα το πάρει σίγουρα σε πεπερασμένο χρόνο. Μάλιστα, η απονομή του κλειδώματος είναι και “δίκαιη”, καθώς με την ίδια χρονική σειρά που οι επεξεργαστές ζητάνε το κλειδωμα, με την ίδια θα το πάρουν.

Craig-Hagersten-Landin (CLH) Lock Το κλειδωμα αυτό αποτελεί προέκταση του προηγούμενου κλειδώματος. Αντί να υπάρχει ένας πίνακας και ανατίθεται κυκλικά θέσεις στους επεξεργαστές που ζητάνε το κλειδωμα, δημιουργείται μια ουρά, στην πράξη μια απλά συνδεδεμένη λίστα. Διατηρούμε έναν δείκτη στο τέλος της λίστας. Η λίστα αποτελείται από κόμβους που έχουν μια μεταβλητή LOCKED/UNLOCKED, και φυσικά εναν δείκτη στον επόμενο κόμβο. Για το `lock()` ο επεξεργαστής δημιουργεί έναν κόμβο με την τιμή LOCKED και τον προσθέτει στο τέλος της λίστας. Ο κόμβος αυτό θα χρησιμοποιηθεί αργότερα στο `unlock()` για να δώσει το κλειδωμα στον επόμενο κόμβο. Προτού προσθέσει τον δικό του κόμβο, κρατά τον παλιό τελευταίο κόμβο, που είναι ακριβώς προηγούμενος κόμβος από αυτόν που έχει ζητήσει το κλειδωμα. Περιμένει μέχρι αυτός κόμβος να αποκτήσει τιμή UNLOCKED, δηλαδή μέχρι ο προηγούμενος κόμβος να του παραδώσει την σκυτάλη του κλειδώματος. Αν κανείς δεν έχει το κλειδωμα, η λίστα είναι κενή. Στην περίπτωση αυτή ο επεξεργαστής δημιουργεί τον κόμβο τον βάζει στην λίστα και θεωρεί ότι πήρε το κλειδωμα.

Δεν απαιτείται να διατηρούμε explicitly τους δείκτες στους επόμενους κόμβους, καθώς κάθε επεξεργαστής μπορεί να κρατήσει στην τοπική του μνήμη απλά κάθε φορά το τέλος της λίστας.

Τέλος, σημειώνεται πως η λήψη του τέλους της λίστας και η αλλαγή της τιμής του, πρέπει να γίνουν ατομικά. Για τον λόγο αυτό χρηματισμοποιείται η εντολή `test-and-set`.

Το κλειδωμα αυτό, υλοποιεί στην πραγματικότητα το Array Lock που είδαμε προηγουμένως, με λίγο διαφορετικό allocation στην μνήμη. Από άποψη ορθότητας και δικαιοσύνης είναι ισοδύναμα.

Το θετικό αυτού του κλειδώματος είναι ότι δεν απαιτείται να είναι γνωστό εκ των προτέρων το πλήθος των νημάτων που μπορούν να ζητήσουν κλειδωμα, και δεν απαιτείται να δεσμευτεί μνήμη για το μέγιστο πλήθος επεξεργαστών ενώ ταυτόχρονα για το κλειδωμα μπορεί σε μια χρονική στιγμή να ανταγωνίζονται μόνο λιγότερη.

Επίσης το κλειδωμα αυτό έχει μεγάλο πλεονέκτημα σε συστήματα NUMA χωρίς cache, καθώς οι προσβάσεις γίνονται στην τοπική μνήμη αντί σε απομακρυσμένη (αν υπάρχει cache θα γίνει απλά μια απομακρυσμένη πρόσβαση μια φορά, και οι υπόλοιπες θα γίνονται στην cache).

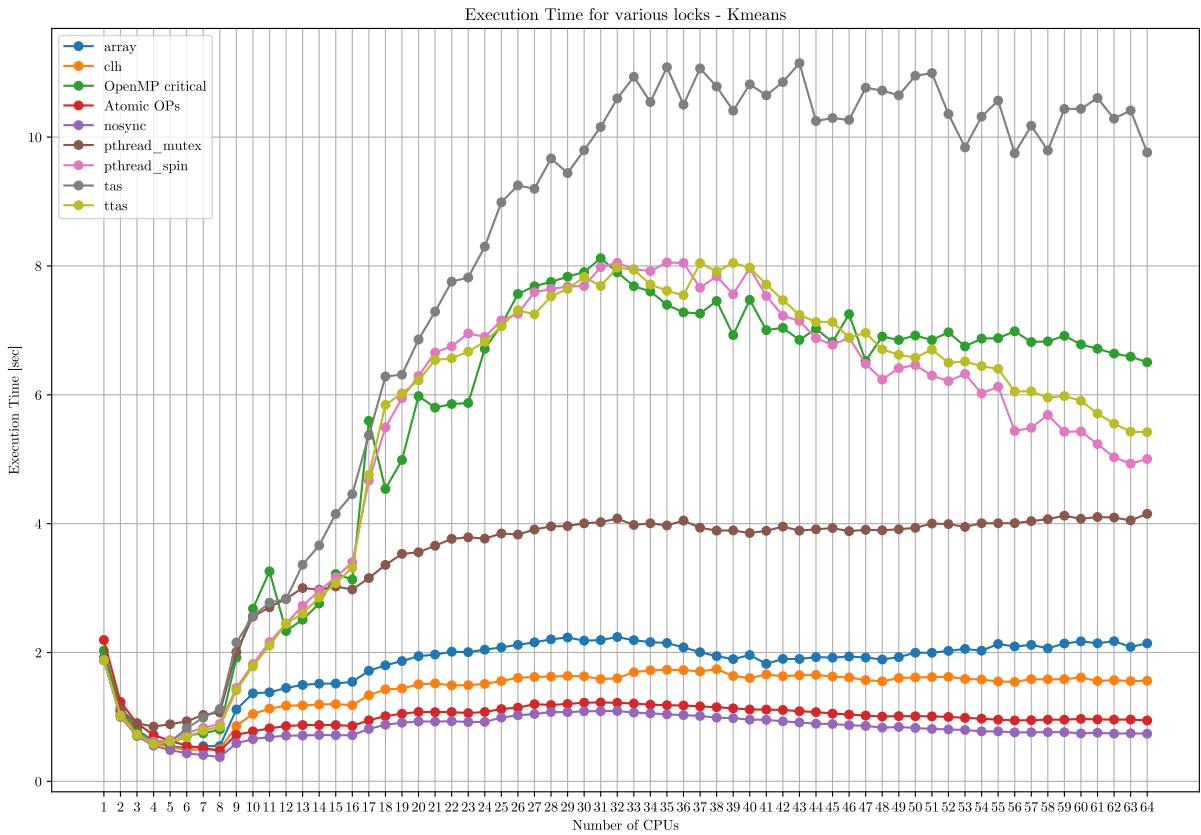
Εκτός των παραπάνω κλειδώματων, δοκιμάζουμε επίσης την χρήση της OpenMP οδηγίας `critical` (υλοποιεί κάποιο κλειδωμα της υλοποίησης του OpenMP) και την χρήση ατομικών εντολών αντί κλειδωμάτων.

Επιπρόσθετα, μετράμε τον χρόνο εκτέλεσης χωρίς καμία μορφή συγχρονισμού, που αν και δίνει λάθος αποτελέσματα, είναι ένα άνω φράγμα του χρόνου εκτέλεσης ως προς την επιλογή του κλειδώματος.

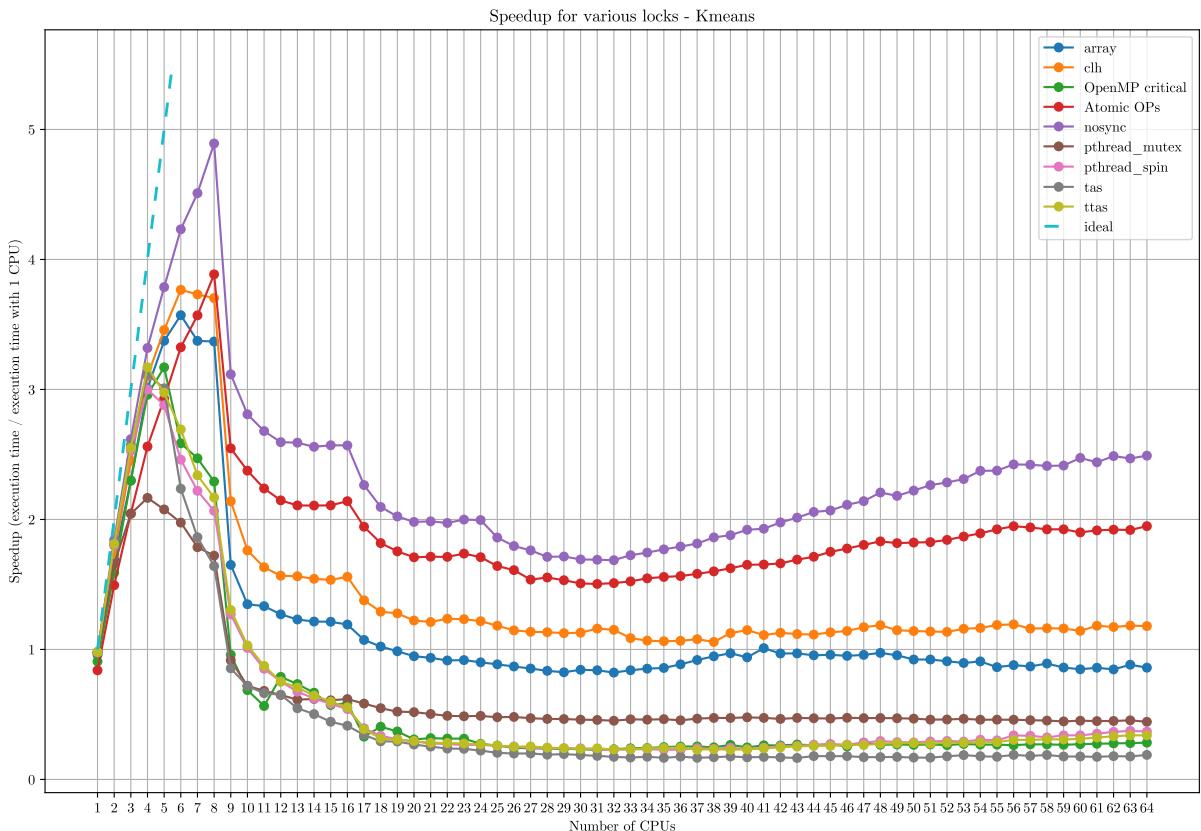
2.3.2 Μετρήσεις

Μετρήθηκε ο χρόνος εκτέλεσης για τις περιπτώσεις που περιγράφηκαν. Οι χρόνοι εκτέλεσης φαίνονται συνολικά στο σχήμα 19 και οι επιταχύνσεις στο σχήμα 20.

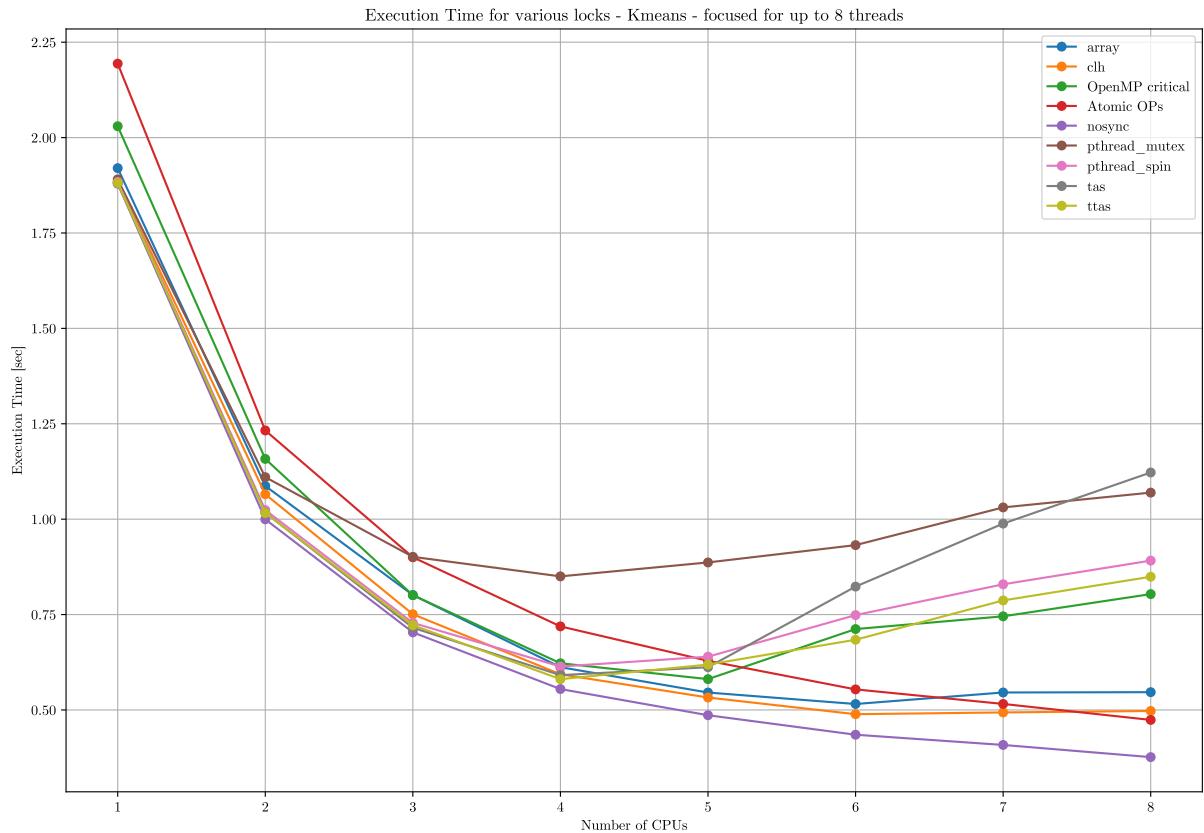
Επειδή για περισσότερους από 8 πυρήνες μπαίνουν περισσότεροι NUMA κόμβοι, απεικονίζονται οι χρόνοι εκτέλεσης και οι επιταχύνσεις εστιασμένοι για ως 8 πυρηνές στα σχήματα αντίστοιχα 21 και 22.



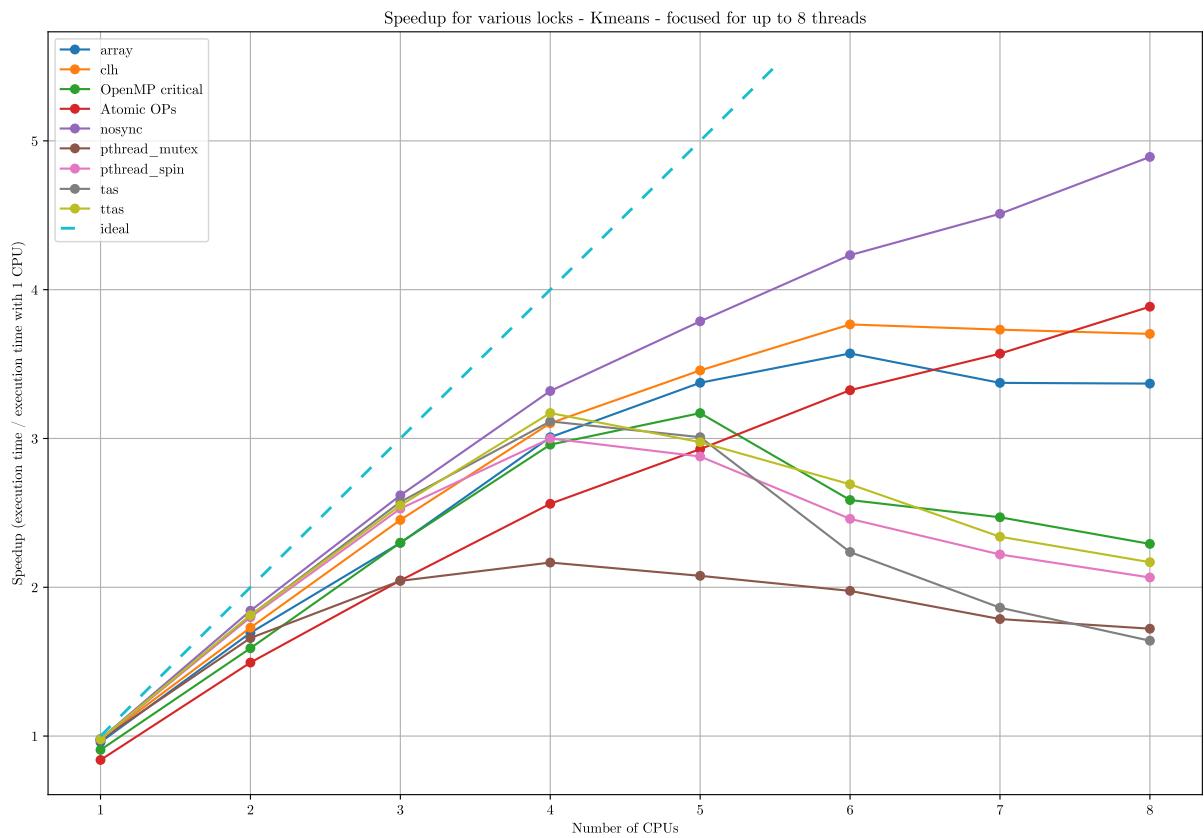
Σχήμα 19: Χρόνοι εκτέλεσης KMeans για διάφορες υλοποιήσεις αλειδωμάτων



Σχήμα 20: Επιτάχυνση KMeans για διάφορες υλοποιήσεις αλειδωμάτων



Σχήμα 21: Χρόνοι εκτέλεσης KMeans για διάφορες υλοποιήσεις κλειδωμάτων – εστίαση ως 8 πυρήνες



Σχήμα 22: Επιτάχυνση KMeans για διάφορες υλοποιήσεις κλειδωμάτων – εστίαση ως 8 πυρήνες

2.3.3 Παρατηρήσεις – συμπεράσματα

Παρατηρούμε πως για περισσότερους από 8 πυρήνες, το ταχύτερο είναι οι ατομικές εντολές, ενώ για λιγότερους από 8 πυρήνες οι ατομικές εντολές είναι πιο αργές από τα κλειδώματα.

Από τα κλειδώματα, σταθερά ταχύτερο είναι το κλειδωμα Craig-Hagersten-Landin (CLH) και ακολουθεί κατά μια σχεδόν σταθερή πολλαπλασιαστική σταθερά πιο αργό, το Array Lock.

Για λιγότερους από 8 πυρήνες, βλέπουμε πως αυτά τα δύο κλειδώματα είναι που κρατάνε κλιμακούμενη την επίδοση, ενώ με τα άλλα κλειδώματα η επίδοση για περισσότερους από 4 πυρήνες χειροτερεύει. Αυτό είναι αναμενόμενο, καθώς αυτά τα κλειδώματα έχουν διαχωρίσει τις αναμονές σε χωριστά cache blocks, ώστε να μην προκύπτει ανταγωνισμός στον δίαυλο μνήμης μεταξύ των επεξεργαστών.

Τα άλλα κλειδώματα (που δεν είναι Queue-based) είναι πιο αργά. Το Test-Test-and-Set, ειδικά καθώς αυξάνονται οι πυρήνες, γίνεται ταχύτερο από Test-and-Set, που είναι αναμενόμενο καθώς προκαλεί σημαντικά λιγότερες ακυρώσεις cache blocks λόγω εγγραφών, που καθώς αυξάνει ο αριθμός επεξεργαστών θα ήταν αυξανόμενες.

Τέλος σχετικά με τα έτοιμα κλειδώματα, για ως 8 επεξεργαστές, το OpenMP critical δίνει την καλύτερη επιτάχυνση για λιγότερους από 8 επεξεργαστές, ενώ για περισσότερους από 8 επεξεργατές το Pthread MUTEX, γεγονός που δικαιολογείται καθώς το κοίμισμα προκαλεί μείωση του ανταγωνισμού στο κλειδωμα λόγω ενεργού αναμονής.

Παρατηρήθηκε επίσης ότι το CLH lock είναι σταθερά πιο γρήγορο από το Array lock. Μέσω εκτελέσεων σε ένα πιο απλό πρόγραμμα (τα νήματα κάνουν απλά lock και μετά unlock, πολλές φορές) βρέθηκε ότι η μεγάλη διαφορά στον χρόνο εξηγείται από τον υπολογισμό της πράξης mod με το μέγεθος του πίνακα στο Array Loc, που γενικά πράγματι είναι μια κάπως χρονοβόρα πράξη. Θεωρώντας ότι το πλήθος νημάτων είναι δύναμη του 2, το μέγεθος του πίνακα μπορεί να γίνει και αυτό δύναμη του 2, και έτσι το mod να αντικασταθεί από bitwise-and που είναι μια πολύ γρήγορη πράξη. Πράγματι, αν κάνουμε αυτή την βελτιστοποίηση, το CLH lock και το Array lock αποκτούν παραπλήσιους χρόνους εκτέλεσης.

2.4 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

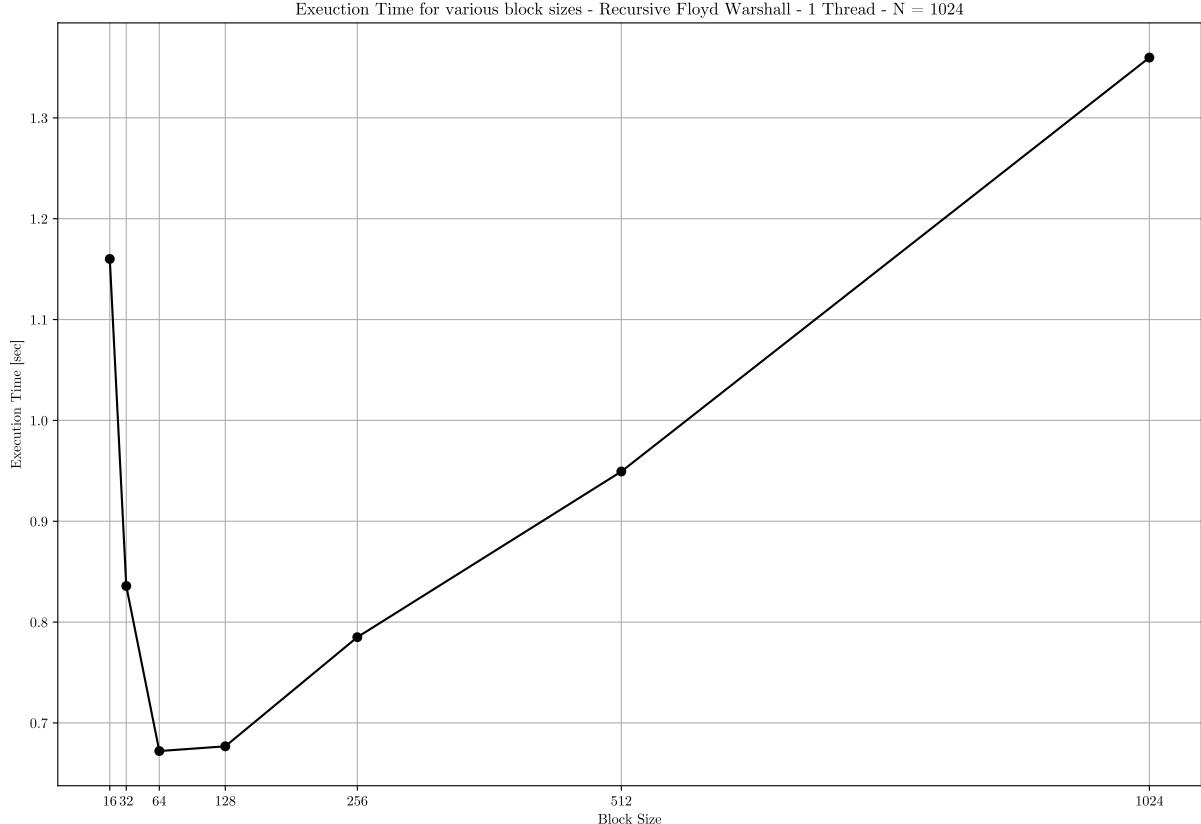
2.4.1 Μετρήσεις για διάφορα block sizes στην σειριακή εκδοχή – παρατηρήσεις

Προτού ασχοληθούμε με την παράλληλη υλοποίηση, χρίναμε σκόπιμο να μελετήσουμε την επίδραση της παράμετρου B , η οποία δεν επηρεάζει το αποτέλεσμα, όμως προβλέπεται να επηρεάζει τον χρόνο εκτέλεσης.

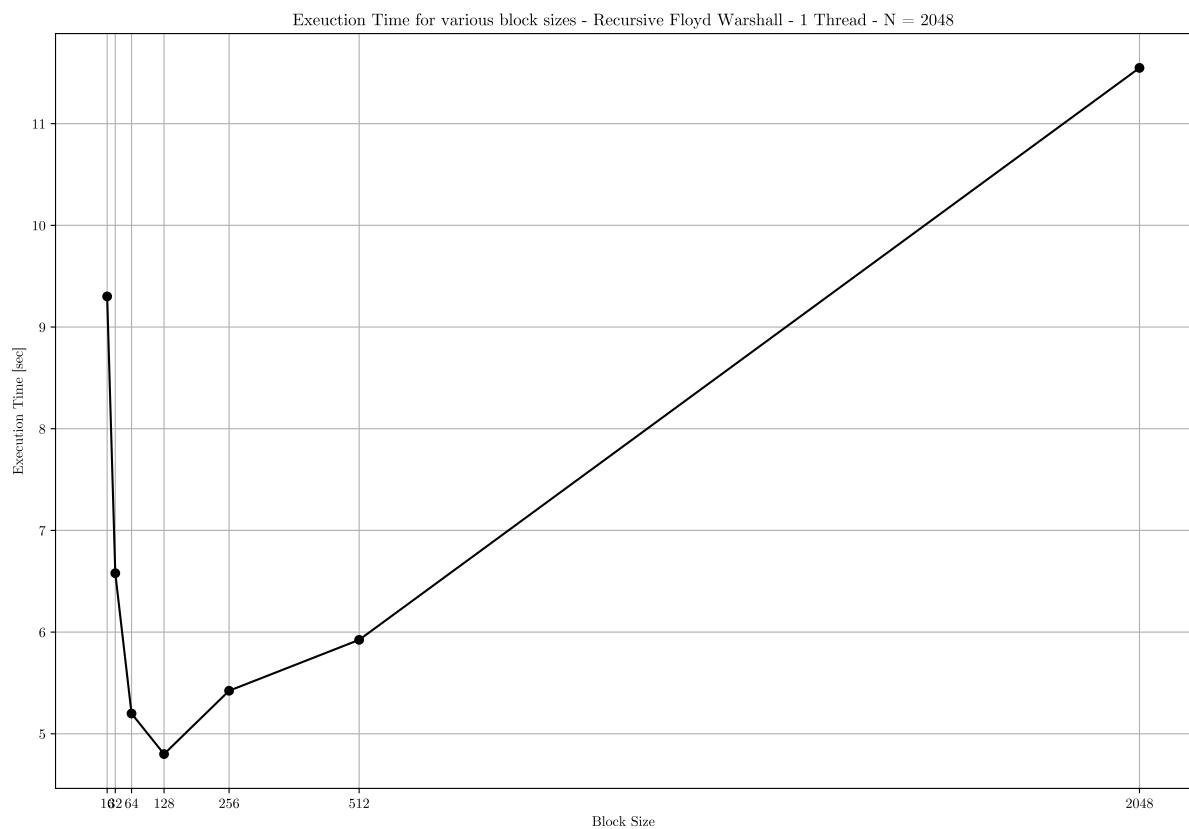
Με ελάχιστο το 16, εκτέλασμα τον αλγόριθμο για όλες τις δύναμεις του 2 ως την διάσταση του πίνακα. Εκτελέσαμε την διαδικασία αυτή για τα τρία μεγέθη πινάκων: 1024×1024 , 2048×2048 , 4096×4096 .

Τα αντίστοιχα γραφήματα για τους χρόνους εκτέλεσης φαίνονται στα σχήματα 23, 24, 25.

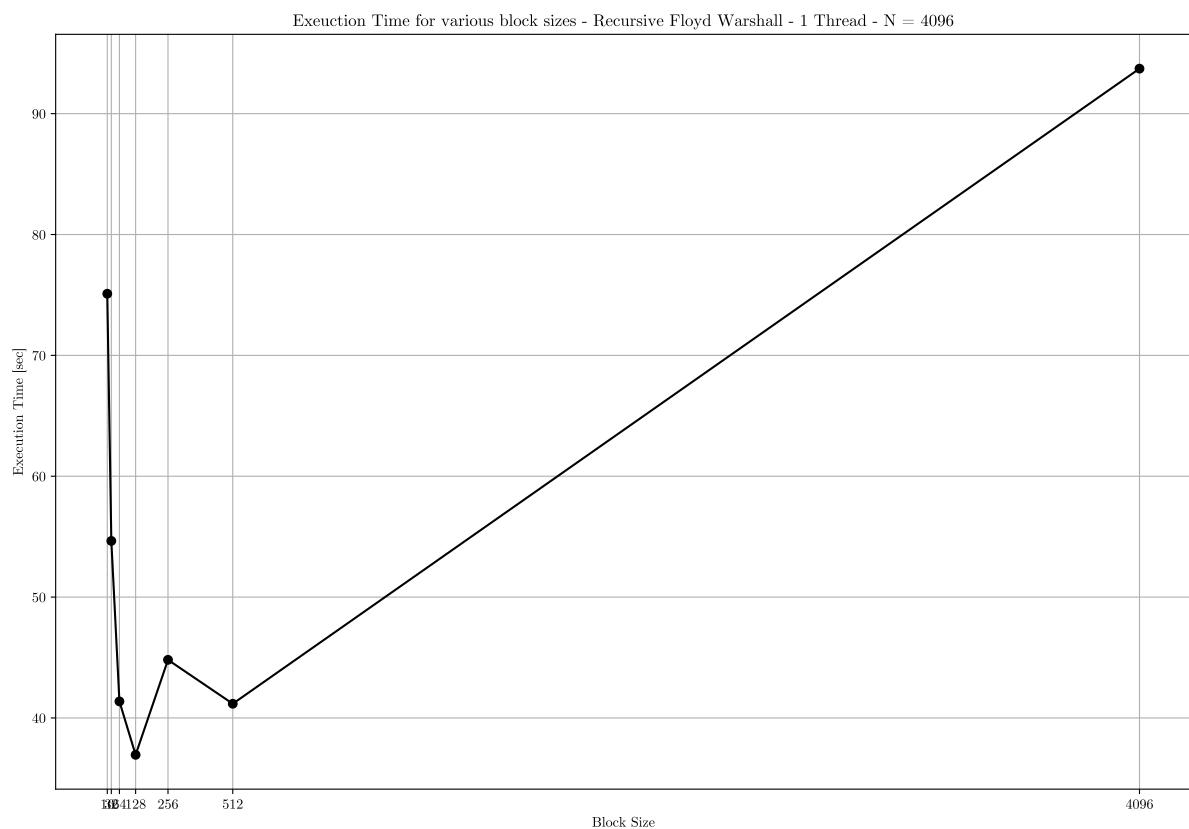
Τα συμπέρασματα βρίσκονται στην παράγραφο 2.4.5.



Σχήμα 23: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – σειριακή εκδοχή – $N = 1024$



Σχήμα 24: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – σειριακή εκδοχή – $N = 2048$



Σχήμα 25: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – σειριακή εκδοχή – $N = 4096$

2.4.2 Υλοποίηση παραλληλοποίησης

Λόγω της αναδρομικής φύσης του αλγορίθμου, η παραλληλοποίηση έγινε με χρήση των OpenMP tasks.

Πιο συγκεκριμένα, οι εξαρτήσεις δεδομένων (Read after Write στους υποπίνακες A_{ij}) επιτρέπουν η 2η και 3η αναδρομική κλήση να γίνουν παράλληλα καθώς και η 6η και η 7η. Όλες οι υπόλοιπες αναδρομικές κλήσεις για να διατηρηθούν οι εξαρτήσεις δεδομένων, πρέπει να γίνουν σειριακά.

Έτσι προστέθηκε πριν την 2η και την 6η αναδρομική κλήση η οδηγία OpenMP:

```
#pragma omp task
```

και πριν την 3η και την 7η αναδρομική κλήση η οδηγία OpenMP:

```
#pragma omp task if(0)
```

Αντίστοιχα προστέθηκε μετά την 3η και 7η αναδρομική κλήση η οδηγία OpenMP:

```
#pragma omp taskwait
```

Τέλος, στην αρχική κλήση από την main, προστέθηκαν οι οδηγίες προκειμένου να εισέλθουμε σε παράλληλο τμήμα του OpenMP:

```
1 #pragma omp parallel
2 {
3     #pragma omp single
4     FW_SR(A,0,0, A,0,0,A,0,0,N,B);
5 }
```

To clause `if(0)`, που σημαίνει ότι το thread που εκτελεί το γονικό task, θα συνεχίσει να εκτελεί το task-παιδί, εξυπηρετεί τον σκοπό να μην γίνουν αχρείαστα πολλά βήματα scheduling που θα είχαν κάποιο πρόσθετο overhead.

Φαινομενικά, θα έλεγε κάποιος ότι θα μπορούσαμε να είχαμε παραλείψει την δημουργία task στην δεύτερη αναδρομική παράλληλη κλήση. Ωστόσο, τότε θα συνέβαινε το εξής αποχές γεγονός: Αρχικά το γονικό task θα έφτιαχνε ένα task-παιδί για την 1η αναδρομική κλήση. Έπειτα, το γονικό task θα συνέχιζε στην 2η αναδρομική κλήση, θα κατέβαινε βαθιά στην ανάδρομη και κάποια στιγμή ενώ είχε κατέβει την ανάδρομη, θα πετύχαινε ένα taskwait. Αυτό που θα θέλαμε, είναι εκείνο το taskwait να περιμένει μόνο την 1η κλήση αυτής της βαθιάς αναδρομικής κλήσης, που φυσικά θα είναι για πολύ μικρότερο n από την έξω-έξω 1η αναδρομική κλήση της πολύ πιο εξωτερικής κλήσης. Ωστόσο, βρισκόμαστε ακόμα στο ίδιο task. Οπότε εκείνο το taskwait, εκτός της 1ης κλήσης στην βαθιά ανάδρομη, θα περιμένει να τελειώσει η πολύ πιο εξωτερική 1η αναδρομική κλήση. Με άλλα λόγια, ενώ πιστεύουμε ότι στην έξω κλήση οι 2 αναδρομικές κλήσεις εκτελούνται παράλληλα, ένα μικρό κομμάτι της 2ης περιμένει να τελειώσει όλη η 1η, προτού συνεχίσει στην υπόλοιπη 2η, οπότε στην πραγματικότητα η εκτέλεση έχει γίνει σειριακή.

Στην πραγματικότητα, το πρόβλημα αυτό δημιουργείται επειδή τα tasks είναι σαν να έχουν δυναμική εμβέλεια (dynamic scoping), αντί για στατική εμβέλεια (static scoping), που είναι η εμβέλεια που ισχύει στις συνηθείς γλώσσες προγραμματισμού.⁷

Σημειώνεται, ότι στο τέλος της παραγράφου 2.4.6, προτείνουμε μια υλοποίηση με ένα πιο προκαθορισμένο schedule, αλλά φαίνεται πως εν τέλει προκύπτουν οι ίδιοι χρόνοι εκτέλεσης.

2.4.3 Μετρήσεις για διάφορα πλήθη πυρήνων στην παράλληλη εκδοχή

Εκτελέσαμε τον αλγόριθμο για διάφορα πλήθη πυρήνων και όλα τα αποδεκτά μεγέθη block > 16.

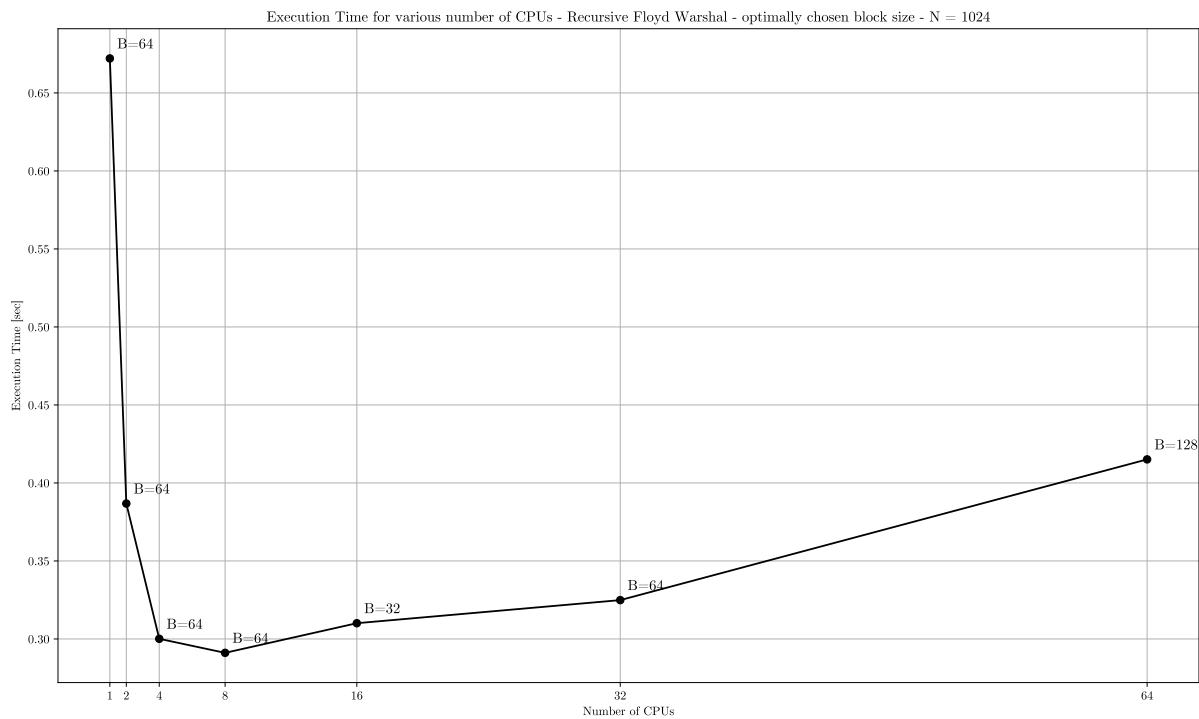
Επιλέξαμε το μέγεθος block που δίνει ελάχιστο χρόνο (στα γραφήματα φαίνεται με επισημένωση δίπλα στα σημεία των μετρήσεων).

Η διαδικασία επαναλήφθηκε για μεγέθη πινάκων 1024×1024 , 2048×2048 , 4096×4096 .

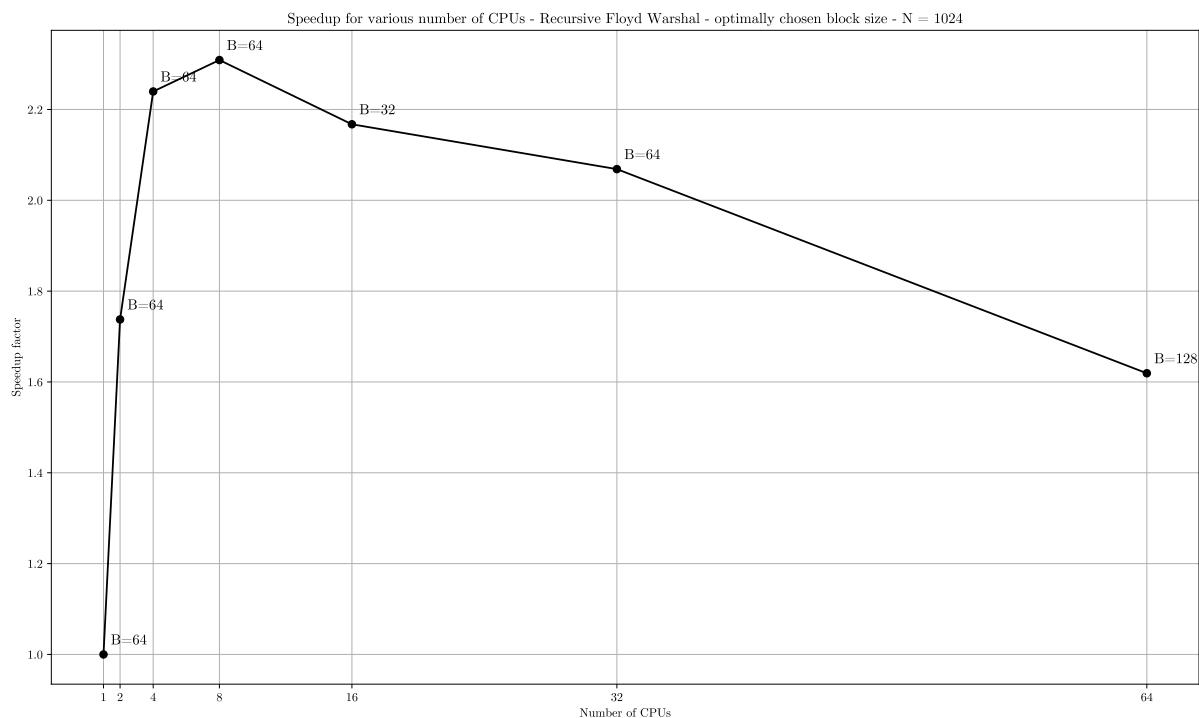
Τα αντίστοιχα γραφήματα για τους χρόνους εκτέλεσης φαίνονται στα σχήματα 26, 28, 30.

Επίσης, τα αντίστοιχα γραφήματα για τις επιταχύνσεις φαίνονται στα σχήματα 27, 29, 31.

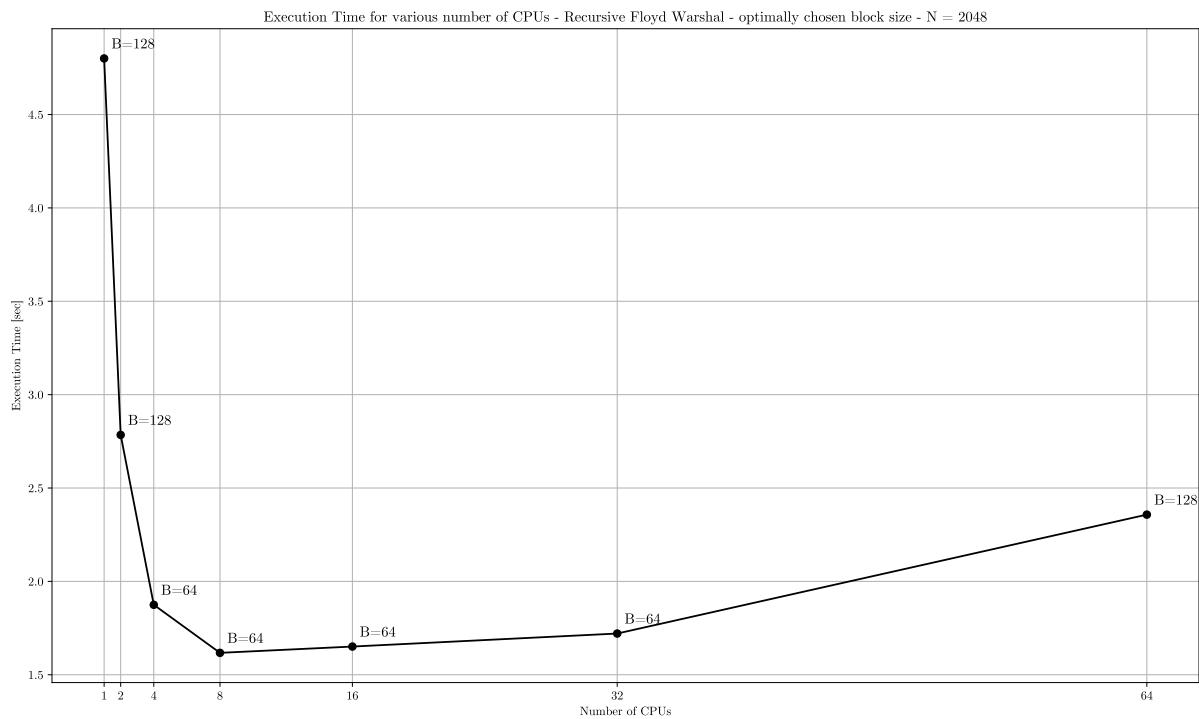
⁷Δυναμική Εμβέλεια σημαίνει πως το ποιες μεταβλητές βρίσκονται στο περιβάλλον εκτέλεσης, εξαρτάται από τον τρόπο που έγινε η εκτέλεση και όχι από τον τρόπο που είναι γραμμένο το πρόγραμμα. Με άλλα λόγια, αν δηλώσουμε μια μεταβλητή x και έπειτα καλέσουμε μια συνάρτηση f , που δεν έχει όρισμα με όνομα x , τότε η f μπορεί να χρησιμοποιήσει την x . Ωστόσο, αν η f διάστημα συνάρτηση, κληθεί από ένα σημείο που η x δεν έχει δηλωθεί, και εντός της f χρησιμοποιηθεί το x θα προκύψει σφάλμα χρόνου εκτέλεσης. Με άλλα λόγια, επφέται στον προγραμματιστή να εξασφαλίσει ότι θα γίνουν σωστά οι εκτελέσεις.



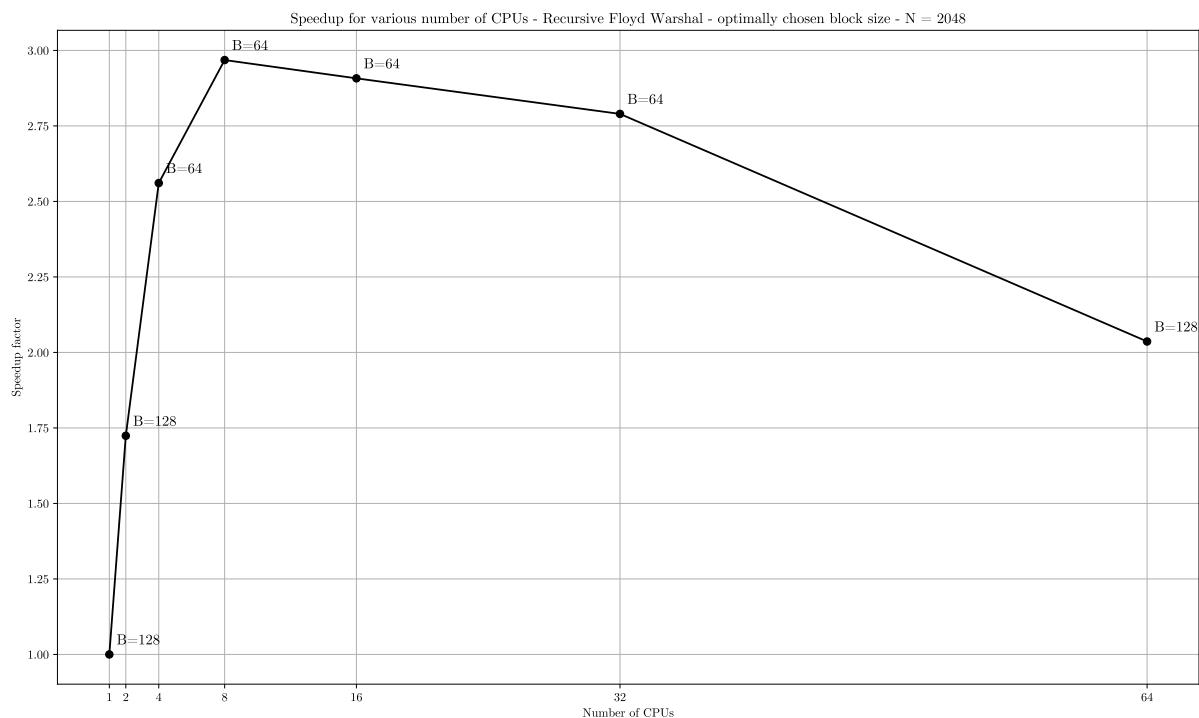
Σχήμα 26: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 1024$



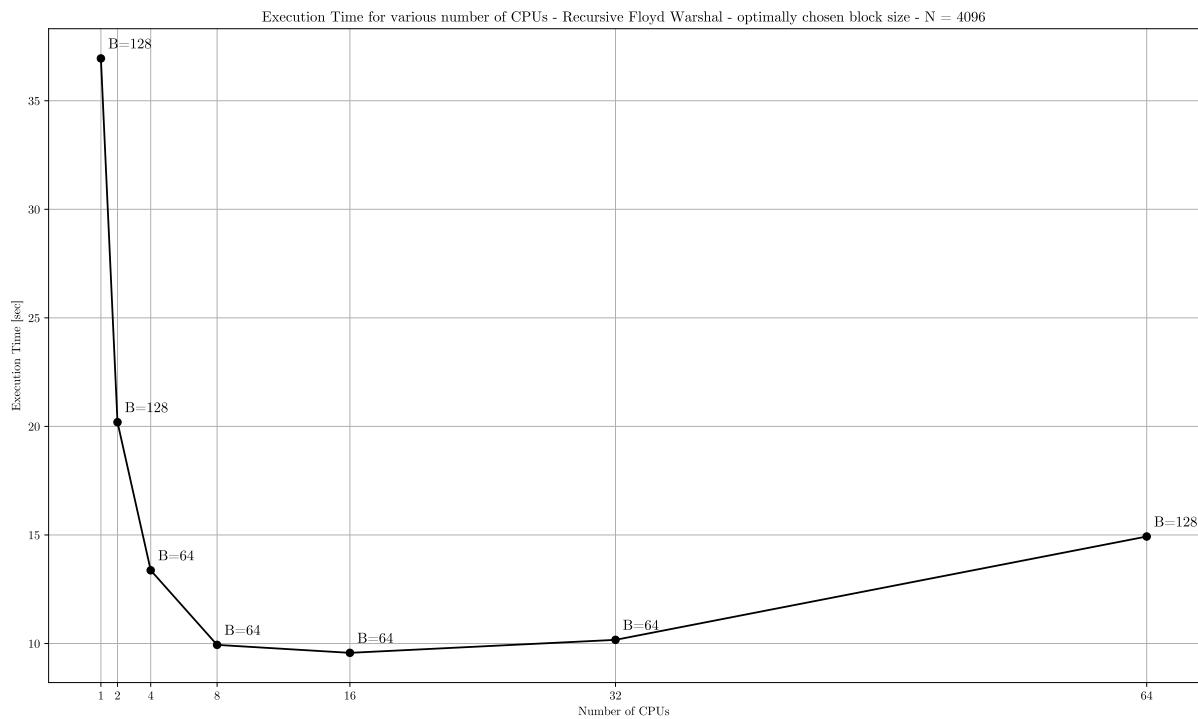
Σχήμα 27: Επιτάχυνση στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 1024$



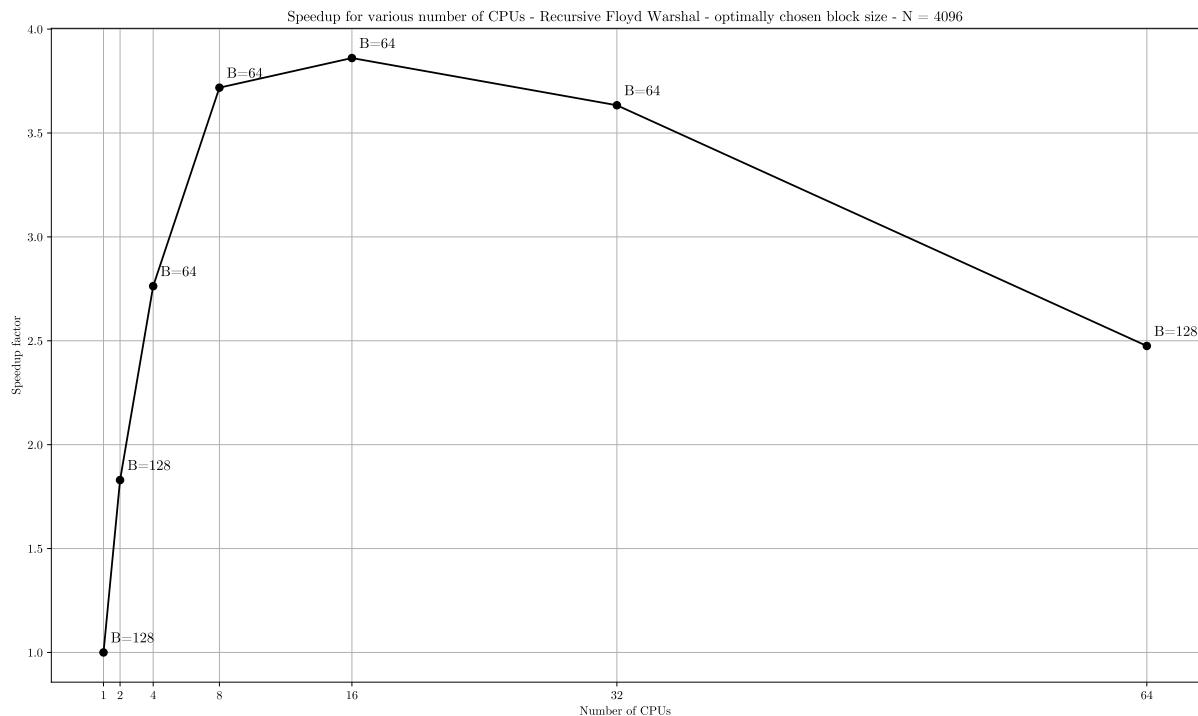
Σχήμα 28: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – N = 2048



Σχήμα 29: Επιτάχυνση στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – N = 2048



Σχήμα 30: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 4096$

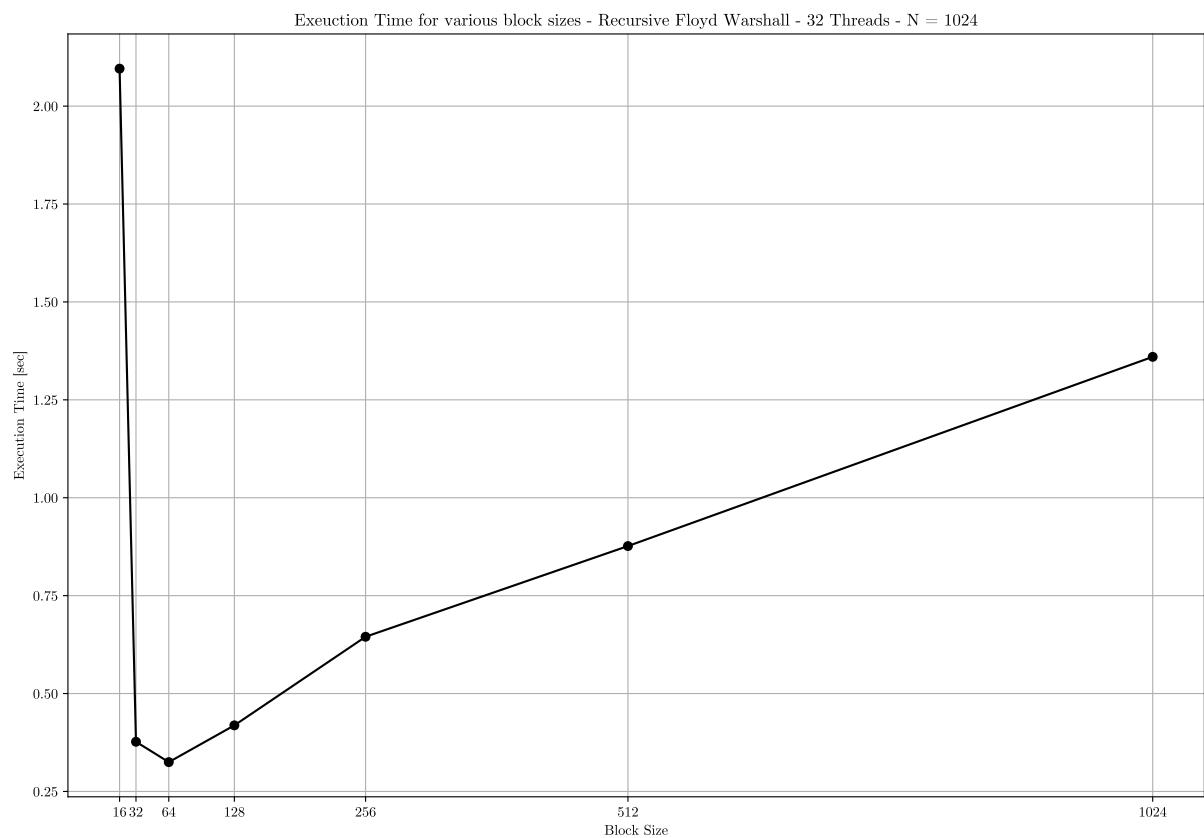


Σχήμα 31: Επιτάχυνση στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 4096$

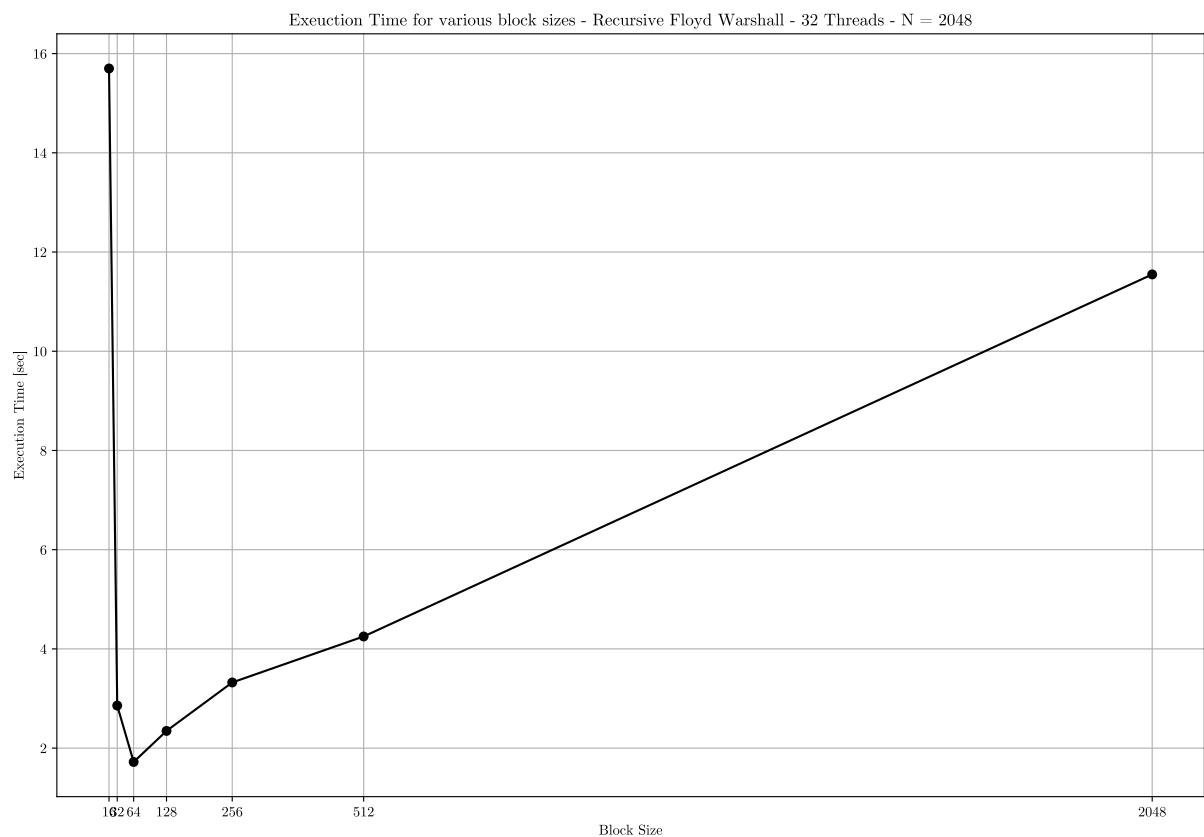
2.4.4 Μετρήσεις για διάφορα block sizes στην παραλληλη εκδοχή

Για λόγους πληρότητας, επαναλάβουμε την μελέτη για τον χρόνο εκτέλεσης ως προς το block size στην παραλληλοποιημένη εκδοχή, με 32 πυρήνες.

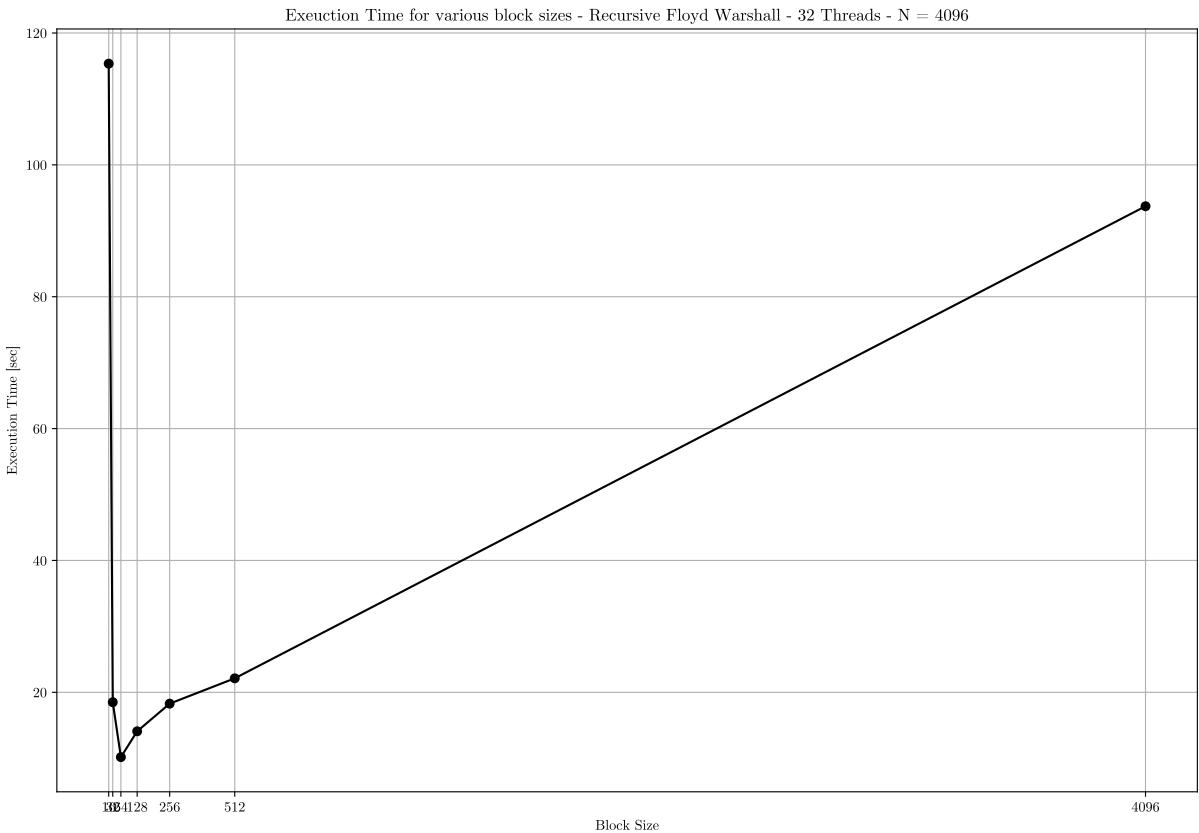
Τα γραφήματα για τους χρόνους εκτέλεσης φαίνονται στα σχήματα 32, 33, 34.



Σχήμα 32: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – 32 πυρήνες – N = 1024



Σχήμα 33: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – 32 πυρήνες – N = 2048



Σχήμα 34: Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – 32 πυρήνες – $N = 4096$

2.4.5 Παρατηρήσεις – συμπεράσματα

Ξεκινάμε με μερικές παρατηρήσεις για το μέγεθος block.

Αρχικά παρατηρούμε ότι ανεξάρτητα του μεγέθους του πινάκα, για σειριακή εκδοχή, ο βέλτιστος χρόνος επιτυγχάνεται για μέγεθος block $B = 128$.

Αντίθετα, στην παραλληλη εκδοχή το βέλτιστο μέγεθος του block γίνεται $B = 64$. Το B εκτός του ότι επηρεάζει την απόδοση στην cache, καθορίζει και το πόση παραλληλία εκτίθεται στην εκτέλεση οπότε είναι λογικό να επηράζεται το βέλτιστο B από την παραλληλοποίηση.

Σχετικά με την επιτάχυνση, παρατηρούμε ότι επιτυγχάνεται μέγιστη επιτάχυνση $\approx 4x$, για 16 πυρήνες. Για περισσότερους πυρήνες, η επιτάχυνση μάλιστα μειώνεται. Η επιτάχυνση και η κλιμάκωσή τους απέχουν ιδιαίτερα από τα ιδανικά, γεγονός που προβληματίζει. Ωστόσο, όπως θα δούμε στην επόμενο παράγραφο, σε μεγάλο βαθμό, ευθύνεται ο ίδιος ο αλγόριθμος που δεν εκθέτει αρκετή παραλληλία. Με άλλα λόγια, πρόκειται για μια συνέπεια του Νόμου Amdahl (μεγάλο σειριακό τμήμα).

2.4.6 Εξήγηση για την έλλειψη κλιμάκωσης

Η έλλειψη επιτάχυνσης και κλιμάκωσης αυτή, σε τέτοιο βαθμό, προβληματίζει αρκετά γιατί συνέβη.

Σκεφτόμαστε μήπως ο αλγόριθμος δεν εκθέτει αρκετή παραλληλία.

Προσπαθούμε να το ποσοτικοποιήσουμε αυτό. Για την ανάλυση αυτή αγνοούμε καθυστερήσεις από μνήμη, cache, κ.λπ. ώστε να βρούμε το μέγιστο ιδανικό speedup που θα μπορούσαμε να πάρουμε.

Εστω ότι το base case, για $n = B$ (που τρέχει πάντα σε έναν μόνο πυρήνα) απαιτεί για την εκτέλεσή του χρόνο w . Ιδανικά, όταν γίνει η εκτέλεση των 2 παραλληλων αναδρομικών κλήσεων, κάθε μία θα εκτελείται με τους μισούς διαθέσιμους πυρήνες. Με άλλα λόγια, θεωρούμε πως κάθε κλήση έχει ένα πλήθος διαθέσιμων πυρήνων, και όταν γίνονται σειριακές κλήσεις αυτές έχουν διαθέσιμες όλους τους πυρήνες, ενώ όταν γίνονται οι 2 παραλληλες αναδρομικές κλήσεις, κάθε μία έχει διαθέσιμους τους μισούς της γονικής κλήσης.

Αν μια κλήση έχει μόνο ένα διαθέσιμο πυρήνα, τα πάντα εκτελούνται σειριακά.

Έτσι, σε μια αναδρομική κλήση, που έχουμε ακόμα ≥ 2 διαθέσιμους πυρήνες, αφού οι 2 παράλληλες αναδρομικές κλήσεις θα γίνουν με μισούς πυρήνες κάθε μία, θα χρειαστούν τόσο χρόνο όσο 1 αναδρομική κλήση που εκτελείται σειριακά.

Όταν σε μια κλήση-γονέα, κάποιες κλήσεις-παιδιά γίνονται σειριακά, οι κλήσεις-παιδιά εξακολούθουν να μπορούν να γίνονται παράλληλα έχοντας διαθέσιμο όλους τους πυρήνες που έχει η κλήση-γονέας.

Έστω $t_{\text{parallel}}(n, \text{procs})$ και $t_{\text{serial}}(n)$ οι χρόνοι εκτέλεσης για μια παράλληλη κλήση με procs διαθέσιμους πυρήνες. Τότε:

$$\begin{aligned} t_{\text{parallel}}(B, \text{procs}) &= w \\ t_{\text{parallel}}(n, 1) &= t_{\text{serial}}(n) \\ t_{\text{parallel}}(n, \text{procs}) &= 4 \cdot t_{\text{parallel}}\left(\frac{n}{2}, \text{procs}\right) + 2 \cdot t_{\text{parallel}}\left(\frac{n}{2}, \frac{\text{procs}}{2}\right) \end{aligned} \quad (4)$$

και

$$\begin{aligned} t_{\text{serial}}(B) &= w \\ t_{\text{serial}}(n) &= 8 \cdot t_{\text{serial}}\left(\frac{N}{2}\right) \end{aligned} \quad (5)$$

Τότε η επιτάχυνση έχει τιμή:

$$\text{speedup}(n, \text{procs}) = \frac{t_{\text{serial}}(n)}{t_{\text{parallel}}(n, \text{procs})} \quad (6)$$

Με βάση τις παραπάνω αναδρομικές σχέσεις, με ένα σύντομο script Python, υπολογίζουμε τιμές της ιδανικά μέγιστης επιτάχυνσης λόγω παραλληλίας που μπορούμε να λάβουμε.

Ασχολούμαστε, ενδεικτικά για την περίπτωση $N = 4096$. Θέτουμε $B = 64$ που φάνηκε να είναι η βέλτιστη τιμή για $N = 4096$ στην παραλληλοποιημένη περίπτωση προηγουμένων.

Υπολογίζουμε την ιδανικά μέγιστη επιτάχυνση λόγω παραλληλίας για διάφορα πλήθη επεξεργαστών και τα απεικονίζουμε στο σχήμα 35.



Σχήμα 35: Ιδανικά μέγιστη επιτάχυνση – Αναδρομικός Floyd-Warshall – $N = 4096$ – $B = 512$

Παρατηρούμε, πως ακόμα για πολλούς πυρήνες δεν μπορούμε να πετύχουμε μεγάλη επιτάχυνση, επειδή **ο αλγόριθμος δεν εκθέτει τόση παραλληλία όσοι οι διαθέσιμοι πυρήνες**.

Ειδικότερα, βλέπουμε πως η μέγιστη επιτάχυνση που μπορούμε γενικά να πετύχουμε είναι 5.5x, καθώς μετά κυριαρχεί στον χρόνο εκτέλεσης του παραλληλοποιημένου προγράμματος το σειριακό τμήμα.

Με άλλα λόγια, αν θέλουμε καλύτερη επιτάχυνση ή/και καλύτερη επιτάχυνση, χρειαζόμαστε καλύτερο αλγόριθμο.

Η πραγματική επιτάχυνση που μετρήθηκε προηγούμενως είναι ακόμα μικρότερη από την ιδανική.

Με βάση τα προηγούμενα, σκεψτόμαστε μια ακόμα μικρή αλλαγή στην υλοποίηση, που ίσως βελτιώσει λίγο τις επιδόσεις. Όταν γίνουν οι δύο παραλληλες αναδρομικές κλήσεις, θέλουμε κάθε μία τους να έχει διαθέσιμους πυρήνες τους μισούς της γονικής κλήσης, ώστε ο χρόνος εκτέλεσης των δύο κλήσεων να είναι ίσος, οπότε να τελειώσουν ταυτόχρονα και να μην υπάρχει αδρανής χρόνος. Επίσης μια κλήση με μόνο 1 διαθέσιμο πυρήνα, ούτως ή αλλως θα πρέπει να εκτελεστεί σειριακά, οπότε δεν υπάρχει λόγος τότε να έχουμε το overhead των tasks, αλλά μπορούμε να την εκτελέσουμε απευθείας.

Για να το πετύχουμε αυτό, προσθέτουμε ένα επιπλέον όρισμα στην συνάρτηση FW_SR, τον ακέραιο procs. Στην αρχική κλήση δίνουμε τιμή το πλήθος των διαθέσιμων πυρήνων στο σύστημα (`omp_get_max_threads()`). Στις αναδρομικές σειριακές κλήσης δίνουμε τιμή την ίδια τιμή του γονέα και στις παραλληλες αναδρομικές κλήσης δίνουμε τιμή `procs/2`. Επίσης, μόνο αν `procs>1` μπαίνουμε στην παραλληλοποιημένη συνάρτηση με τα tasks, αλλιώς καλούμε τις αναδρομικές κλήσεις χωρίς tasks.

Παραθέτουμε τον κώδικα:

```

1 void FW_SR ( int **A, int arow, int acol,
2               int **B, int brow, int bcol,
3               int **C, int crow, int ccol,
4               int myN, int bsize, int procs)
5 {
6     if(myN<=bsize)
7         for(int k=0; k<myN; k++)
8             for(int i=0; i<myN; i++)
9                 for(int j=0; j<myN; j++)
10                    A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
11    else {
12        if (procs > 1) {
13            FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize, procs);
14
15            #pragma omp task
16            FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize, procs/2);
17            #pragma omp task if(0)
18            FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize, procs/2);
19
20            #pragma omp taskwait
21
22            FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize,
23                   procs);
24            FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize,
25                   procs);
26            #pragma omp task
27            FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize,
28                   procs/2);
29            #pragma omp task if(0)
30            FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize,
31                   procs/2);
32
33            #pragma omp taskwait
34
35            FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize, procs);
36        } else {
37            FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize, 0);
38            FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize, 0);
39            FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize, 0);
40            FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize, 0);
41        }
42    }
43 }
```

```

38     FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
39     ↵ myN/2, bsize, 0);
40     FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize, 0);
41     FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize, 0);
42     FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize, 0);
43 }
44 }

```

Οστόσο, οι χρόνοι εκτέλεσης, εν τέλει, προκύπτουν ίδιοι με όταν δεν γίνει αυτή η μεταβολή, γεγονός που σημαίνει ότι μάλλον ούτως ή άλλως από τον τρόπο που εκτελείται το πρόγραμμα από πριν, προκύπτει το ίδιο schedule που εδώ προσπαθούμε να επιβάλλουμε explicitly.

Έτσι, συμπερασματικά, η χαμηλότερη επίδοση από την θεωρητικά μέγιστη δυνατή πρέπει μάλλον να αποδοθεί σε θέματα που στην ιδανική ανάλυση θεωρήθηκαν αμελητέα, όπως η cache και η μνήμη. Η παραλληλία που υπάρχει στον αλγόριθμο είναι σχετικά ακανόνιστη και δεν υπάρχει στατικός προγραμματισμός του τι θα κάνει κάθε πυρήνας.

Αυτό, μάλλον, καθιστά πιθανό, ένας πυρήνας που καλείται να κάνει έναν υπολογισμό (στο base case) να είχε κάποια άσχετα δεδομένα προηγουμένως στην χρυφή του μνήμη, γεγονός που σημαίνει ότι με κάποια, μάλλον υψηλή πιθανότητα, σχεδόν κάθε φόρα που ένας πυρήνας κάνει υπολογισμός, πρέπει τα αντίστοιχα δεδομένα να μεταφερθούν από την κύρια μνήμη ή από την χρυφή μνήμη άλλου πυρήνα. Ισως, όταν πρόκειται για έναν πύρηνα μόνο του, να υπάρχει περισσότερη χωρική/χρονική τοπικότητα καθώς όλες οι κλήσεις γίνονται σειριακά. Με άλλα λόγια, ενδεχομένως, όταν γίνει η παραλληλοποίηση, το cache miss rate να αυξάνεται σε σχέση με την σειριακή υλοποίηση, οπότε η επίδοση να είναι χειρότερη από την αναμενόμενη.

2.4.7 Tiled Αλγόριθμος: Υλοποίηση, Μετρήσεις, Σχόλια

Παρατηρώντας πως ο αναδρομικός Floyd Warshall δεν πέτυχε καλές επιδόσεις στην παράλληλη εκδοχή του αναδρομικού αλγορίθμου.

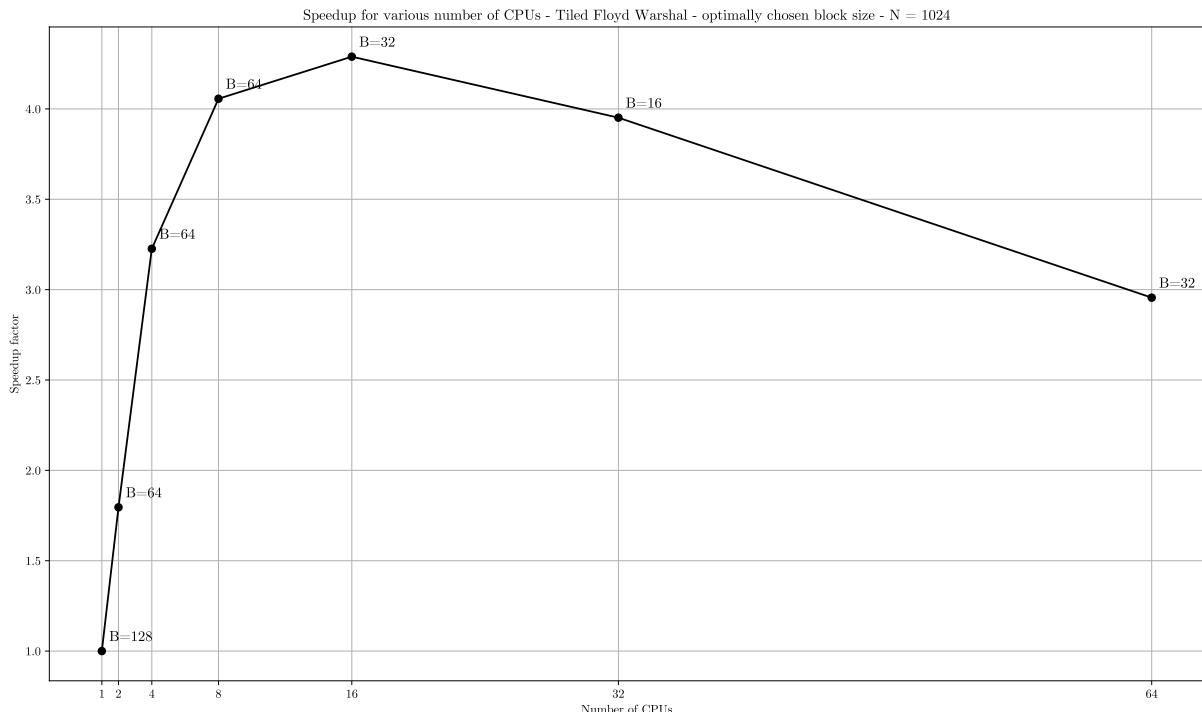
Για τον λόγο αυτό καταφύγαμε στην Tiled εκδοχή του Floyd Warshall.

Η παραλληλοποίηση εδώ γίνεται πάνω στους βρόχους, αντί με βάση tasks όπως έγινε προηγούμενως, καθώς, επίσης, στηρίζομαστε και σε static schedules. Λόγω του γεγονότος αυτό, οι επιδόσεις αναμένονται σημαντικά καλύτερες.

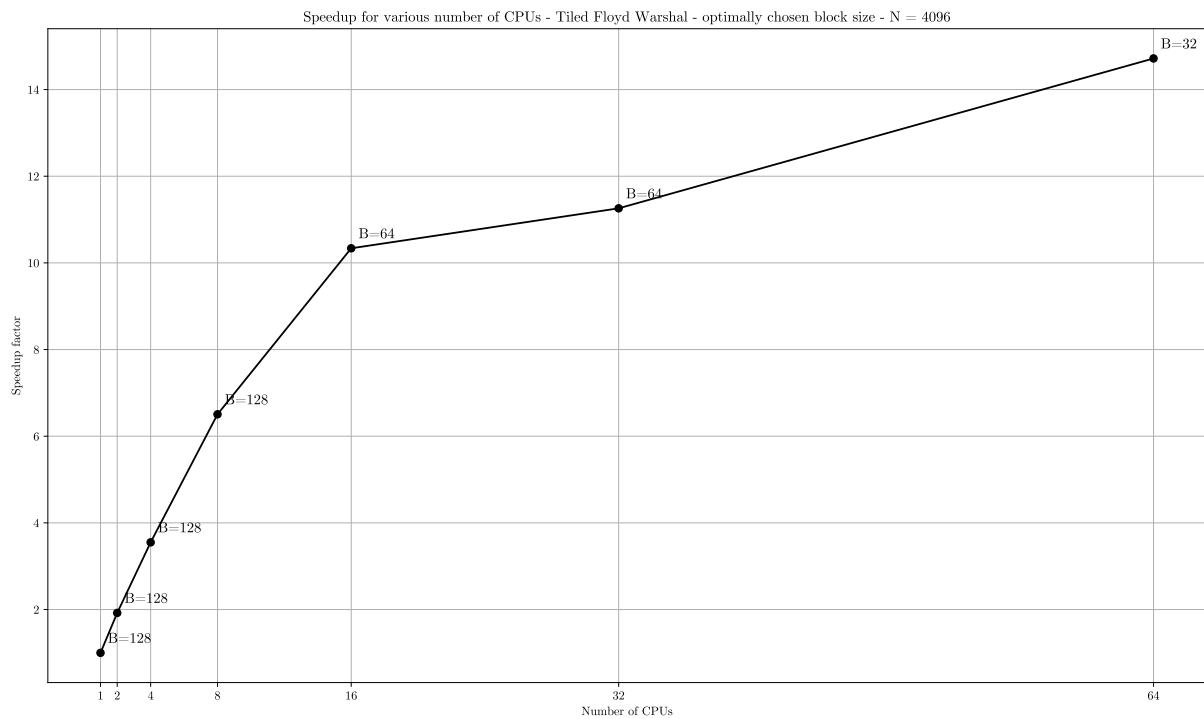
Δόθηκε έμφαση κατά την παραλληλοποίηση, να διατηρηθούν οι εξαρτήσεις δεδομένων, ώστε να διατηρηθεί η ορθότητα.

Για λόγους επιβεβαίωσης, συγχρίθηκαν για ίδια είσοδο τα αποτέλεσματα του παραλληλοποιημένου tiled με τον σειριακό αλγορίθμο και βρέθηκαν ίδια.

Εκτελέστηκε ο αλγόριθμος για διάφορα πλήθη πυρήνων. Για λόγους συντομίας, παραθέτουμε μόνο γραφήματα επιτάχυνσης. Για πίνακες μεγέθους 1024×1024 και 4096×4096 βλέπουμε το γράφημα επιτάχυνσης στα σχήματα 36 και 37 αντίστοιχα.



Σχήμα 36: Επιτάχυνση στον Tiled Floyd-Warshall για διάφορα πλήθη πυρήνων – $N = 1024$



Σχήμα 37: Επιτάχυνση στον Tiled Floyd-Warshall για διάφορα πλήθη πυρήνων – $N = 4096$

Παρατηρούμε ότι επιτυγχάνονται σημαντικά καλύτερες επιδόσεις σε σχέση με την αναδρομική υλοποίηση.

Στον μεγάλο πίνακα 4096×4096 επιτυγχάνεται μέγιστη επιτάχυνση 15x για 64 πυρήνες.

Ο ελάχιστος χρόνος που επιτυγχάνεται, ως απόλυτος αριθμός, είναι 2.3 δευτερόλεπτα.

Στον μικρό πίνακα, η κλιμάκωση είναι χειρότερη. Για 8 πυρήνες έχουμε 4x επιτάχυνση και για 16 πυρήνες έχουμε μέγιστη επιτάχυνση 4.3x. Για περισσότερους πυρήνες η επιδόση μειώνεται. Για την μείωση της επίδοσης αυτή ίσως να ευθύνεται το γεγονός μπαίνουν περισσότεροι κόμβοι NUMA, οπότε γίνονται προσβάσεις σε μνήμη απομακρυσμένου κόμβου, που είναι αρκετά αργές.

2.5 Ταυτόχρονες Δομές Δεδομένων

2.5.1 Περιγραφή

Θα μελετήσουμε την επίδραση στην χρονική επίδοση, δηλαδή στην ρυθμαπόδοση (throughput), διαφόρων τρόπων συγχρονισμού για την υλοποίηση μιας thread-safe ταξιονομημένης απλά συνδεδέμενης λίστας.

Οι τρόποι συγχρονισμού που μελετώνται είναι οι εξής:

1. Coarse-grain locking (cgl)
2. Fine-grain locking (fgl)
3. Optimistic synchronization (opt)
4. Lazy synchronization (lazy)
5. Non-blocking synchronization (nb)

2.5.2 Μετρήσεις – συμπεράσματα

Μετρήθηκαν οι χρόνοι εκτέλεσης για όλους του συνδυασμούς (καρτεσιανό γινόμενο) των παρακάτω μεταβλητών:

Πλήθος νήματων 1, 2, 4, 8, 16, 32, 64, 128 (ως 32 είναι φυσικοί πυρήνες, 33-64 είναι νήματα Simultaneous Multithreading (SMT ή Hyperthreading), 65-128 είναι δεύτερο νήμα εκτέλεσης για κάθε νήμα του υλικού)

Μέγεθος λίστας 1024, 8192

Φόρτος εργασίας 100-0-0, 80-10-10, 20-40-40, 0-50-50 (ποσοστό αναζητησεών-εισαγωγών-διαγραφών αντίστοιχα)

Τα αποτελέσματα φαίνονται στα επόμενα γραφήματα, όπου έχουν απεικονιστεί:

1. Η ρυθμαπόδοση (throughput) ως προς το πλήθος νήματων.
2. Η επιτάχυνση ($\frac{\text{throughput}}{\text{sequential}}$) ως προς το πλήθος νήματων.
3. Η αποδοτικότητα της παραλληλοποίησης ($\frac{\text{speedup}}{\text{threads}}$) ως προς το πλήθος νήματων. Εκφράζει το πόσο κοντά είναι το speedup στο ιδανικό speedup και έχει ιδανική μέγιστη τιμή 1.
4. Η ρυθμαπόδοση ως προς τον τύπο φόρτου εργασίας (σε bar plot) για διάφορα πλήθη νημάτων.
5. Η ρυθμαπόδοση ως προς το μέγεθος λίστας για διάφορα πλήθη νημάτων.
6. Heatmaps της ρυθμαπόδοσης ως προς το πλήθος νημάτων και τον τύπο φόρτου εργασίας για κάθε υλοποίηση συγχρονισμού και μέγεθος λίστας.

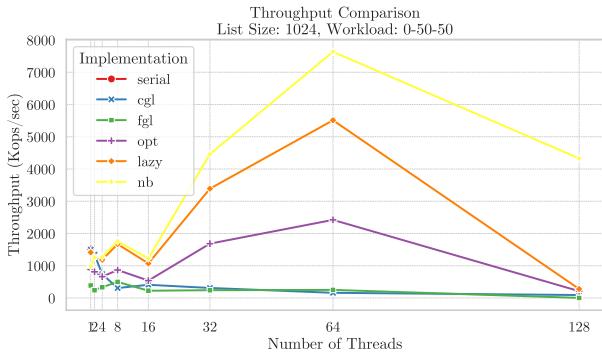
Από τα γραφήματα που ακολοθούν συμπεραίνουμε πως ο non-blocking συγχρονισμός είναι γενικά ο ταχύτερος, όμως με κοντική διαφορά από τον lazy συγχρονισμό, εκτός από την περίπτωση που γίνονται μόνο αναγνώσεις και καθόλου μεταβολές, οπότε ταχύτερος με αρκετή διαφορά είναι ο lazy συγχρονισμός. Οι υπόλοιποι συγχρονισμοί αποδίδουν συστηματικά χειρότερα από αυτούς τους δύο.

Επειδή στις περιπτώσεις που είναι καλύτερος ο non-blocking, η διαφορά είναι μικρή από το lazy, ενώ το αντίστροφο δεν ισχύει, η καλύτερη και πιο robust επιλογή φαίνεται να είναι ο non-blocking συγχρονισμός, καθώς σε κάθε φόρτιο θα αποδίδει σχεδόν βέλτιστα.

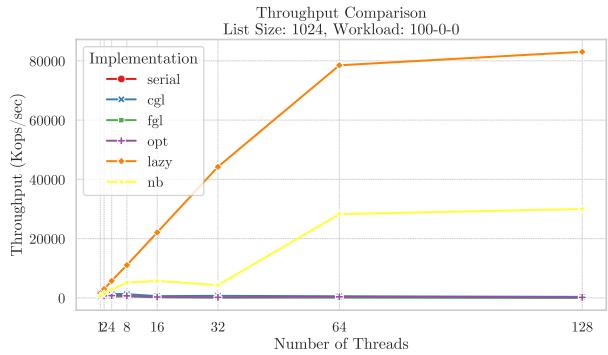
Η αποδοτικότητα της επιτάχυνσης (δηλαδή πόσο καλό αξιοποιούνται οι πρόσθετοι πυρήνες) φαίνεται να εξαρτάται από το μέγεθος της λίστας και να γίνεται καλύτερη για μεγαλύτερες λίστες. Η αποδοτικότητα είναι η καλύτερη στον φόρτο εργασίας χωρίς μεταβολές, γεγονός που είναι αναμενόμενο καθώς δεν δημιουργούνται εξαρτήσεις δεδομένων. Για τον φόρτο εργασίας με τις μεταβολές, στην μικρή λίστα, 1024 στοιχείων, η αποδοτικότητα δεν είναι ιδιαίτερα καλή, με επιτάχυνση $\approx 5x$ ακόμα για 64 νήματα, ενώ στην μεγάλη λίστα, 8192 στοιχείων, η αποδοτικότητα αυξάνεται σε πιο μέτρια τιμή, καθώς έχουμε επιτάχυνση $\approx 10x$ στα 64 νήματα.

Παρατηρούμε επίσης ότι στα 128 νήματα υπάρχει απότομη πτώση της επίδοσης στις περιπτώσεις που γίνονται μεταβολές στην λίστα, ειδικά όσο το ποσοστό των λειτουργιών με μεταβολές είναι μεγαλύτερο. Αυτό συμβαίνει, διότι με 128 νήματα εκτέλεσης αλλά 64 νήματα υλικού, το Λειτουργικό Σύστημα αναγκαστικά θα κάνει context switches. Αυτό θα έχει ως αποτέλεσμα, ένα νήμα ενώ διατηρεί ένα κλείδωμα, να το σταματά το Λειτουργικό Σύστημα. Τα κλειδώματα που χρησιμοποιούνται είναι ενεργού αναμονής. Έτσι, το ΛΣ θα βάλει προς εκτέλεση ένα άλλο νήμα εκτέλεσης, που απλά θα κάνει άσκοπα waiting, αφού το νήμα που μπορεί να ξεκλειδώσει δεν προοδεύει. Αξίζει να σημειωθεί ότι, ευτυχώς, είναι θέμα μόνο χρονικής επίδοσης και δεν προκύπτει deadlock που προηγούμενως δεν δημιουργόταν. Κάποια στιγμή το νήμα που έχει το κλείδωμα θα ξαναπεί προς εκτέλεση, θα ολοκληρώσει το

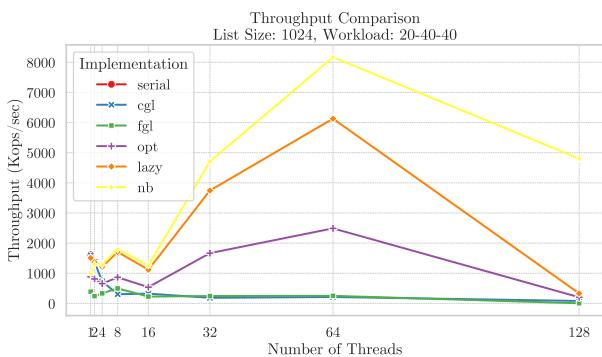
κρίσιμο τμήμα του (αυτό θα συμβεί αφού θεωρούμε ότι όταν δεν γίνονται context switches δεν υπάρχουν deadlocks) και θα απελευθερώσει το κλειδώμα, οπότε όταν εκ νέου επανέλθει το άλλο νήμα για εκτέλεση θα μπορέσει να λάβει το κλειδώμα και να μπει στο δικό του κρίσιμο τμήμα. Ο λόγος που το φαινόμενο αυτό συμβαίνει στους φόρτους εργασίας με μεταβολές, και μάλιστα με αυξανόμενη μείωση επίδοσης για αυξανόμενο ποσοστό μεταβολών, είναι πως στις “καλές” υλοποιήσεις συγχρονισμού, οι αναζητήσεις/αναγνώσεις δεν λαμβάνουν κλειδώματα, οπότε δεν προκύπτει το πρόβλημα που σχολιάστηκε.



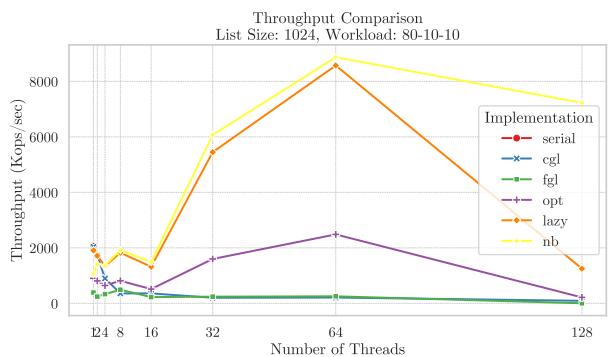
(α') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 0-50-50



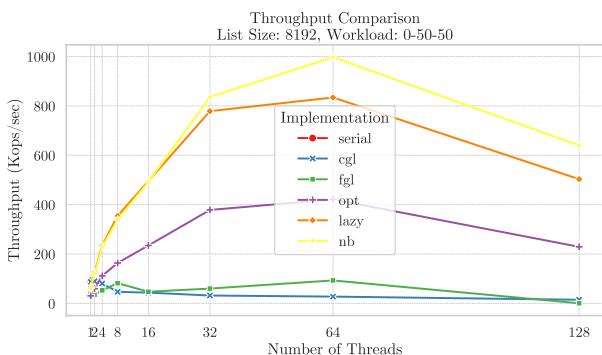
(β') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 100-0-0



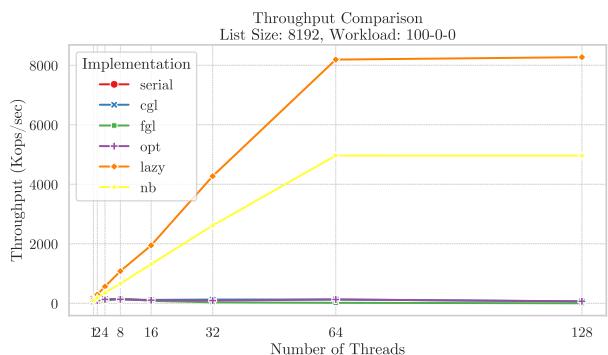
(γ') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 20-40-40



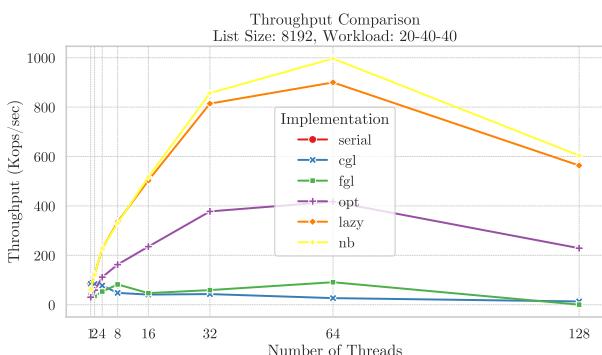
(δ') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 80-10-10



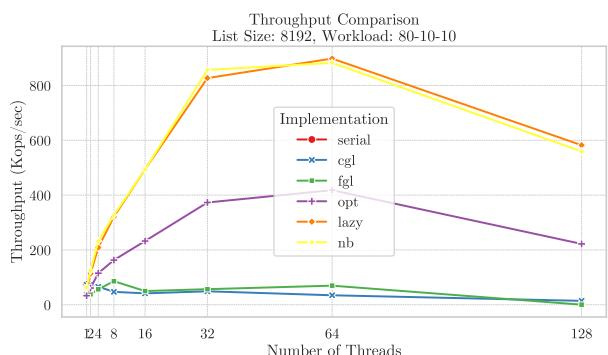
(α') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 0-50-50



(β') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 100-0-0



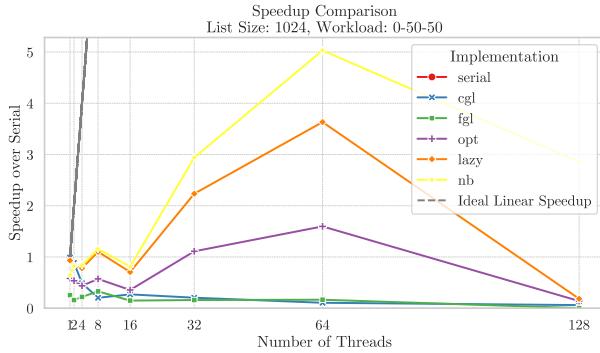
(γ') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 20-40-40



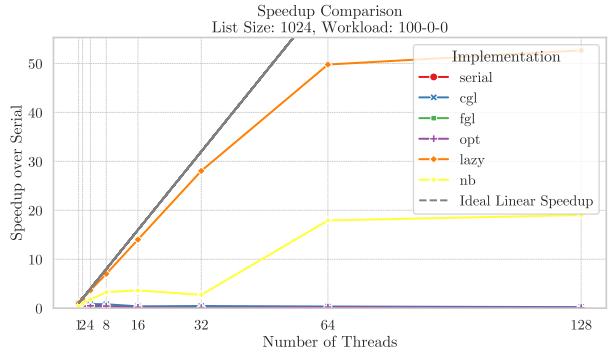
(δ') Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 80-10-10

Σχήμα 38: Ρυθμαπόδοση (Throughput) για διάφορους φόρτους εργασίας με μέγεθος λίστας 1024.

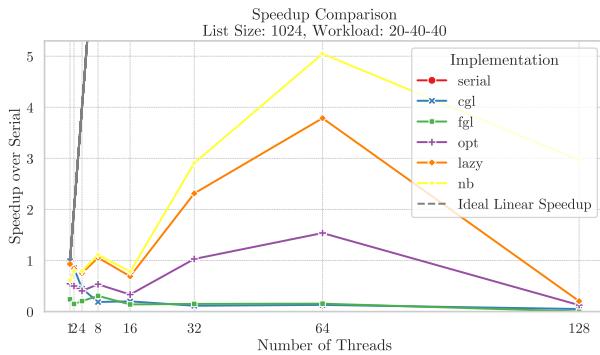
Σχήμα 39: Ρυθμαπόδοση (Throughput) για διάφορους φόρτους εργασίας με μέγεθος λίστας 8192.



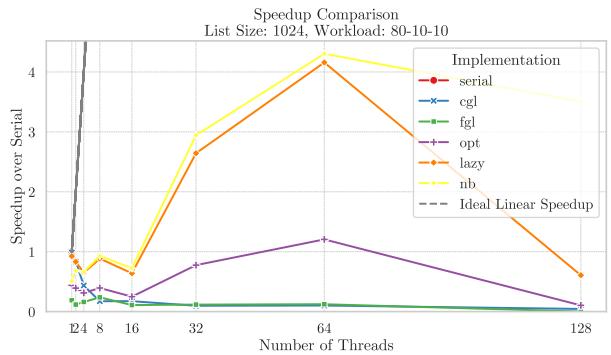
(α') Speedup για φόρτο εργασίας 0-50-50



(β') Speedup για φόρτο εργασίας 100-0-0

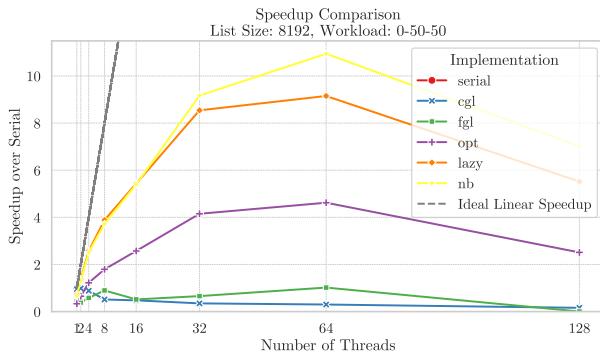


(γ') Speedup για φόρτο εργασίας 20-40-40

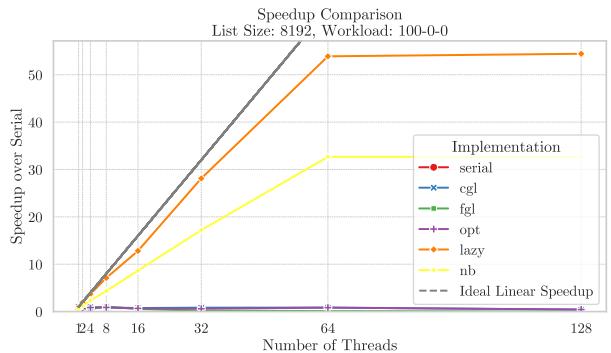


(δ') Speedup για φόρτο εργασίας 80-10-10

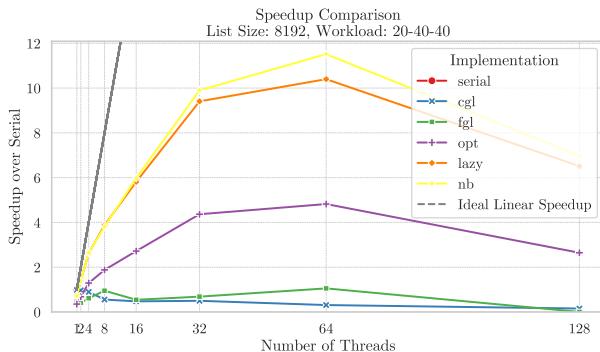
Σχήμα 40: Speedup για διαφορετικούς φόρτους εργασίας με μέγεθος λίστας 1024.



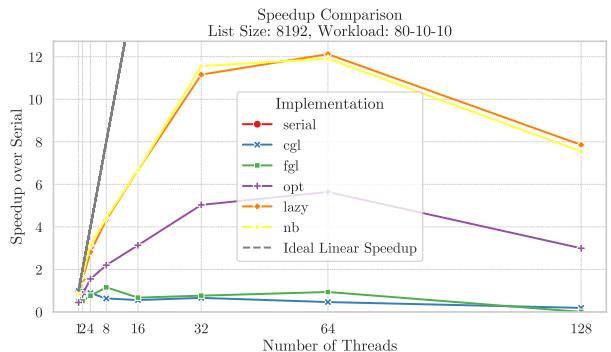
(α') Speedup για φόρτο εργασίας 0-50-50



(β') Speedup για φόρτο εργασίας 100-0-0

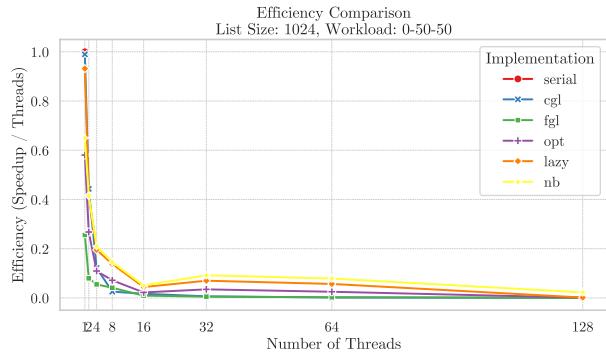


(γ') Speedup για φόρτο εργασίας 20-40-40

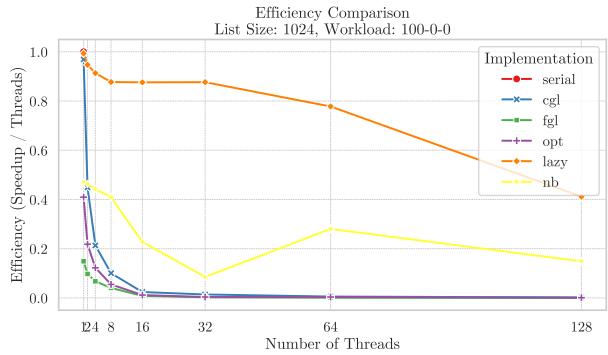


(δ') Speedup για φόρτο εργασίας 80-10-10

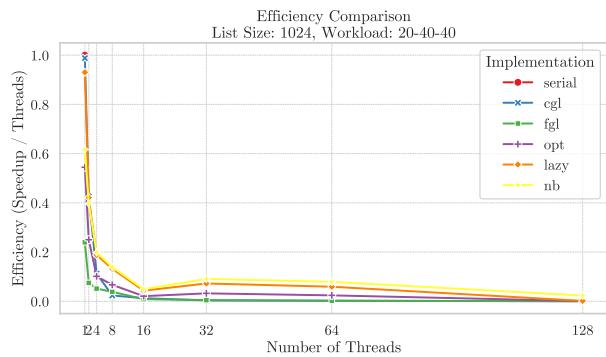
Σχήμα 41: Speedup για διαφορετικούς φόρτους εργασίας με μέγεθος λίστας 8192.



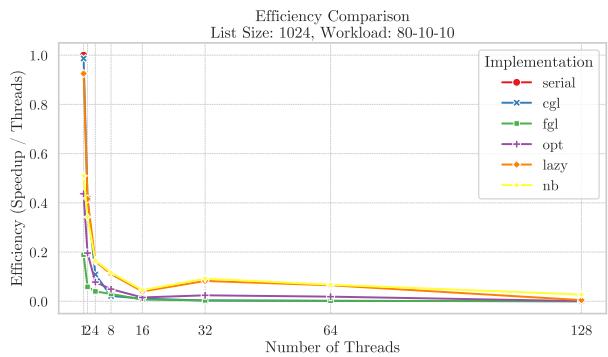
(α') Αποδοτικότητα με φόρτο εργασίας 0-50-50



(β') Αποδοτικότητα με φόρτο εργασίας 100-0-0

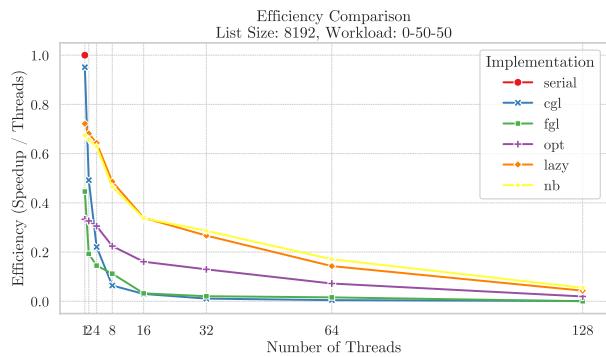


(γ') Αποδοτικότητα με φόρτο εργασίας 20-40-40

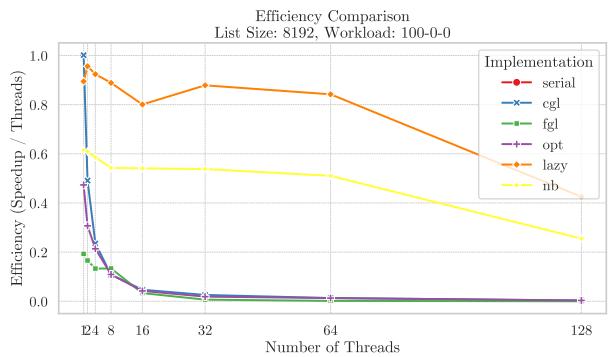


(δ') Αποδοτικότητα με φόρτο εργασίας 80-10-10

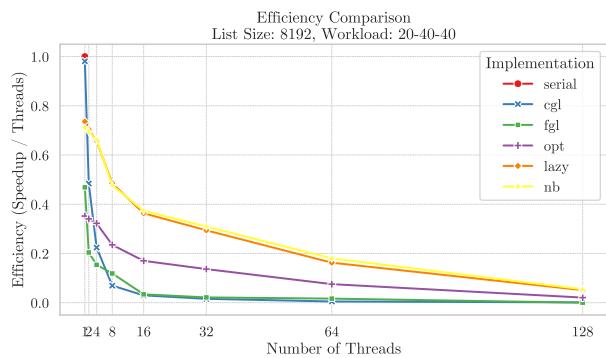
Σχήμα 42: Αποδοτικότητα για διάφορους φόρτους εργασίας με μέγεθος λίστας 1024.



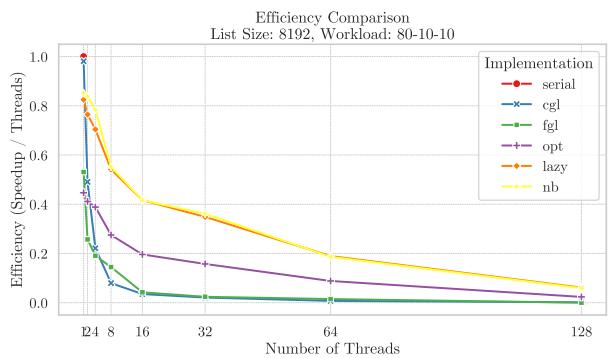
(α') Αποδοτικότητα με φόρτο εργασίας 0-50-50



(β') Αποδοτικότητα με φόρτο εργασίας 100-0-0

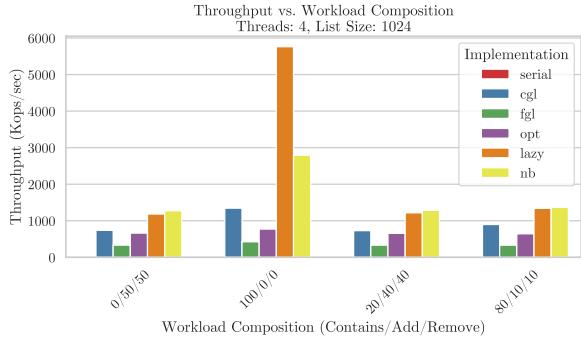


(γ') Αποδοτικότητα με φόρτο εργασίας 20-40-40

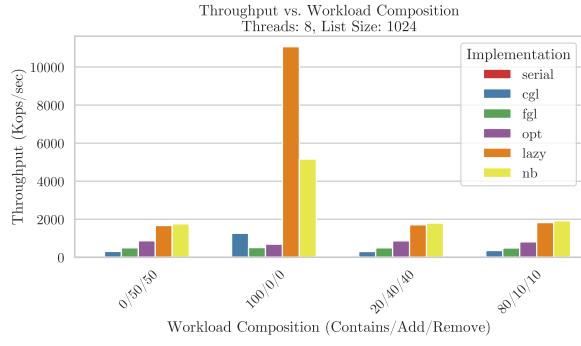


(δ') Αποδοτικότητα με φόρτο εργασίας 80-10-10

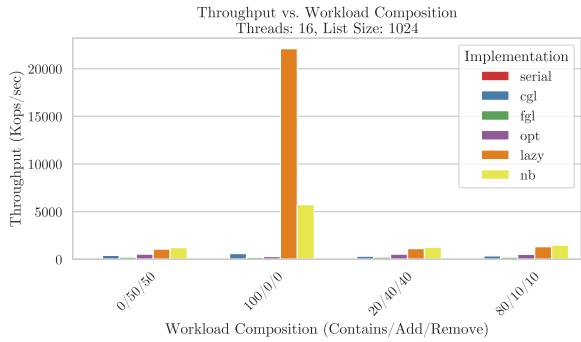
Σχήμα 43: Αποδοτικότητα για διάφορους φόρτους εργασίας με μέγεθος λίστας 8192.



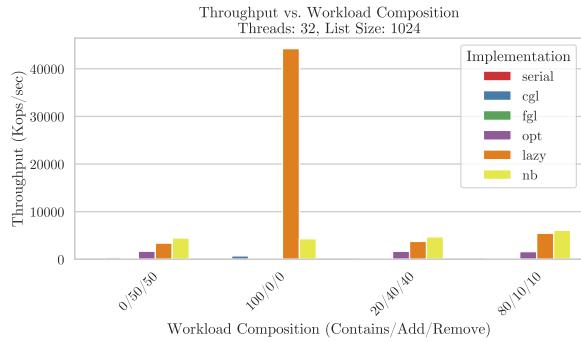
(α') Επίδραση Φορτίου για 4 Νήματα



(β') Επίδραση Φορτίου για 8 Νήματα

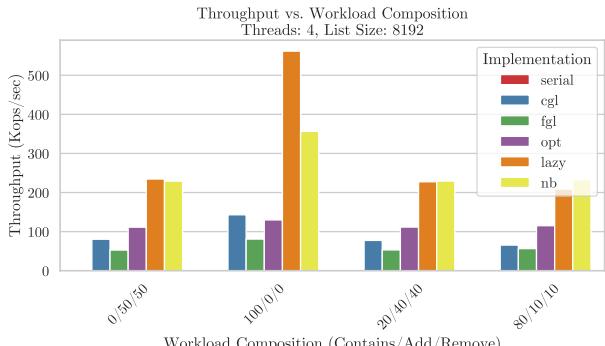


(γ') Επίδραση Φορτίου για 16 Νήματα

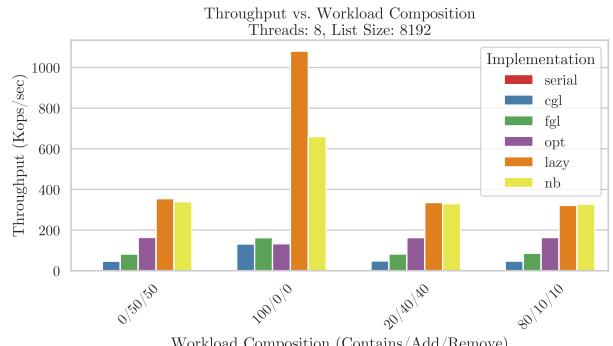


(δ') Επίδραση Φορτίου για 32 Νήματα

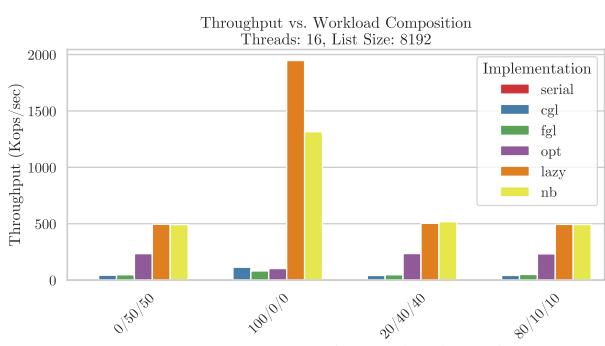
Σχήμα 44: Επίδραση Φορτίου για Διάφορους Αριθμούς Νημάτων με μέγεθος λίστας 1024.



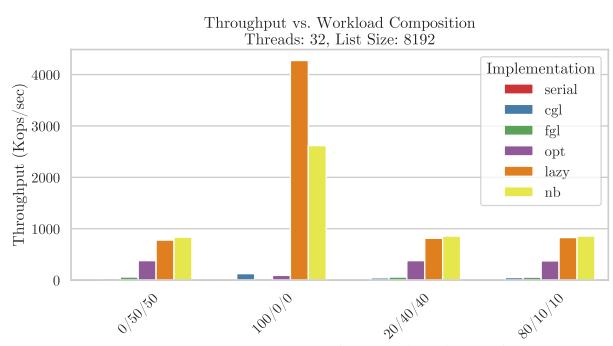
(α') Επίδραση Φορτίου για 4 Νήματα



(β') Επίδραση Φορτίου για 8 Νήματα

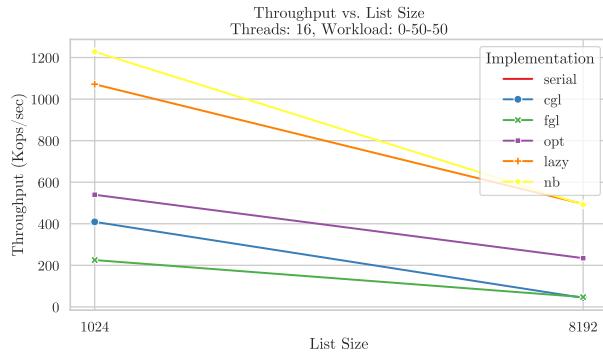


(γ') Επίδραση Φορτίου για 16 Νήματα

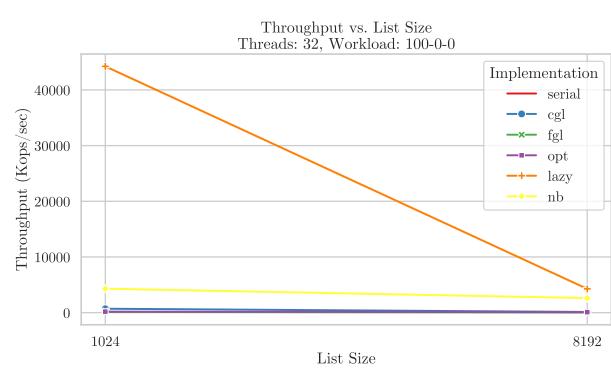
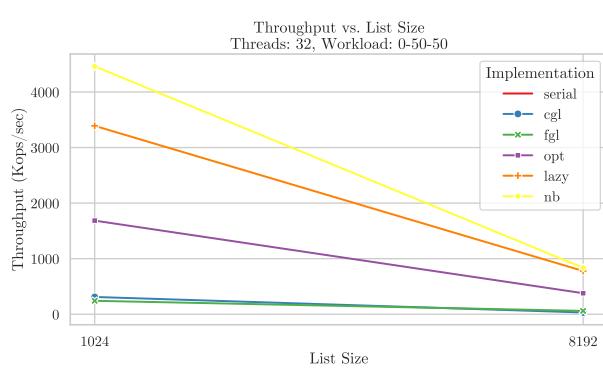
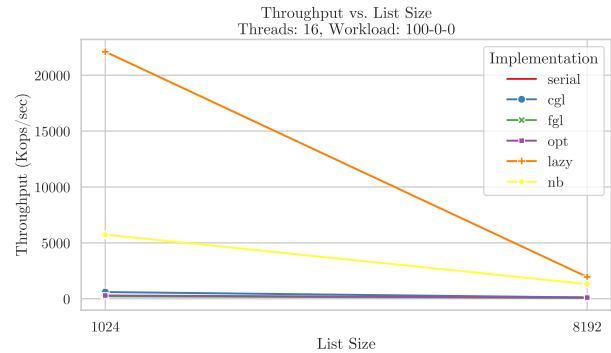


(δ') Επίδραση Φορτίου για 32 Νήματα

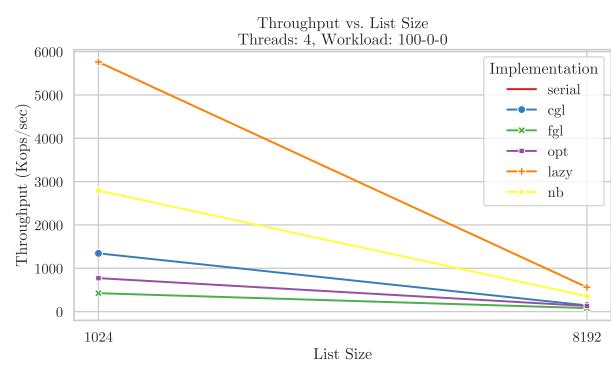
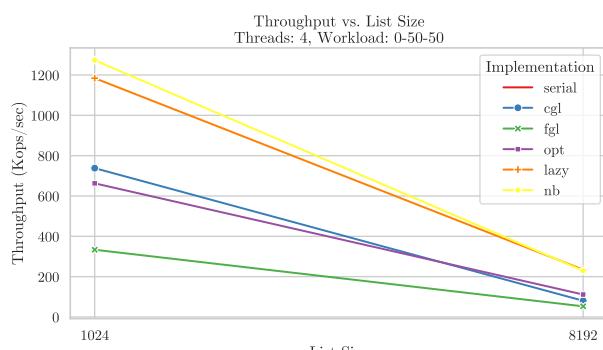
Σχήμα 45: Επίδραση Φορτίου για Διάφορους Αριθμούς Νημάτων με μέγεθος λίστας 8192.



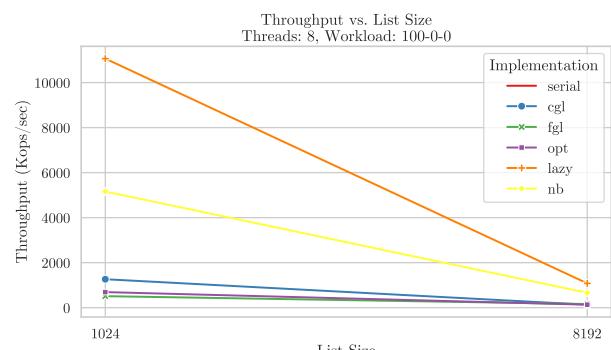
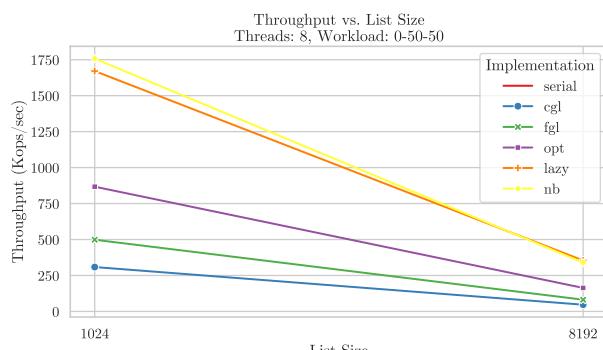
(α) Επίδραση Μεγέθους Λίστας για 16 Νήματα, Φορτίο 0-50-50 (β) Επίδραση Μεγέθους Λίστας για 16 Νήματα, Φορτίο 100-0-0



(γ') Επίδραση Μεγέθους Λίστας για 32 Νήματα, Φορτίο 0-50-50 (δ') Επίδραση Μεγέθους Λίστας για 32 Νήματα, Φορτίο 100-0-0

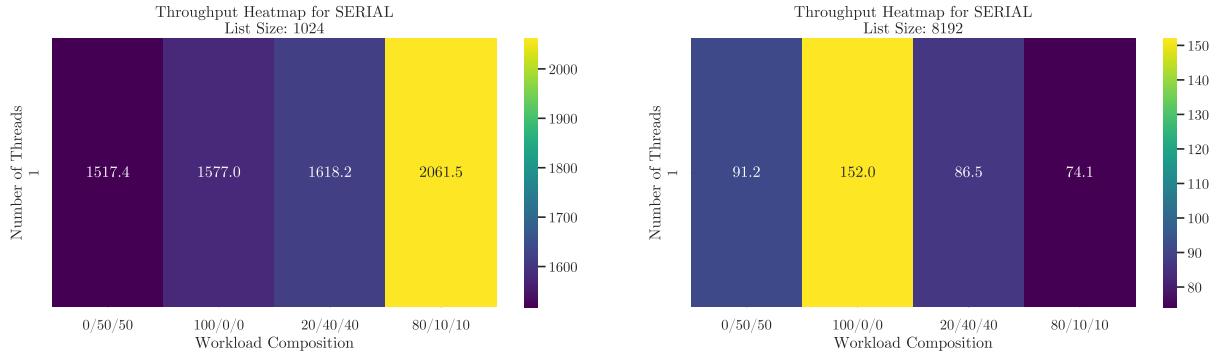


(ε') Επίδραση Μεγέθους Λίστας για 4 Νήματα, Φορτίο 0-50-50 0-0 (στ') Επίδραση Μεγέθους Λίστας για 4 Νήματα, Φορτίο 100-0-0



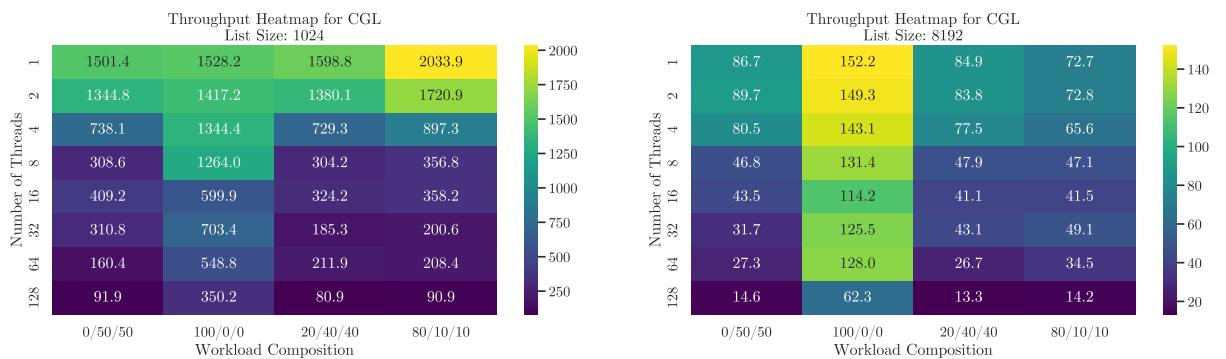
(ζ) Επίδραση Μεγέθους Λίστας για 8 Νήματα, Φορτίο 0-50-50 (η) Επίδραση Μεγέθους Λίστας για 8 Νήματα, Φορτίο 100-0-0

Σχήμα 46: Επίδραση Μεγέθους Λίστας για Διάφορους Αριθμούς Νημάτων και Φορτία

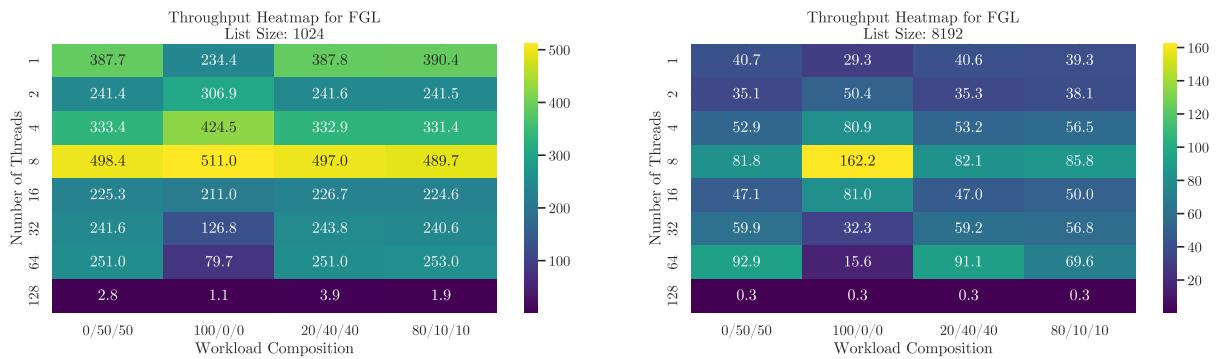


(α) Heatmap της σειριακής υλοποίησης για μέγεθος λίστας (β') Heatmap της σειριακής υλοποίησης για μέγεθος λίστας 1024
8192

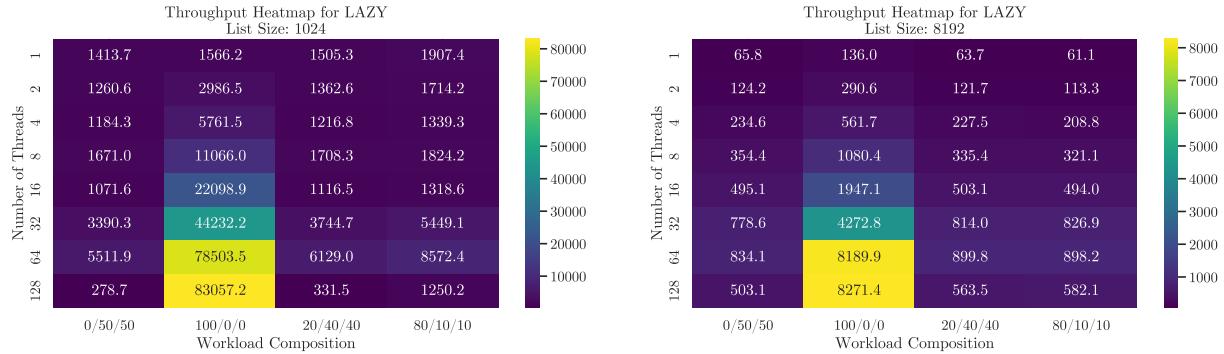
Σχήμα 47: Heatmaps για τη σειριακή υλοποίηση με διάφορα μεγέθη λίστας.



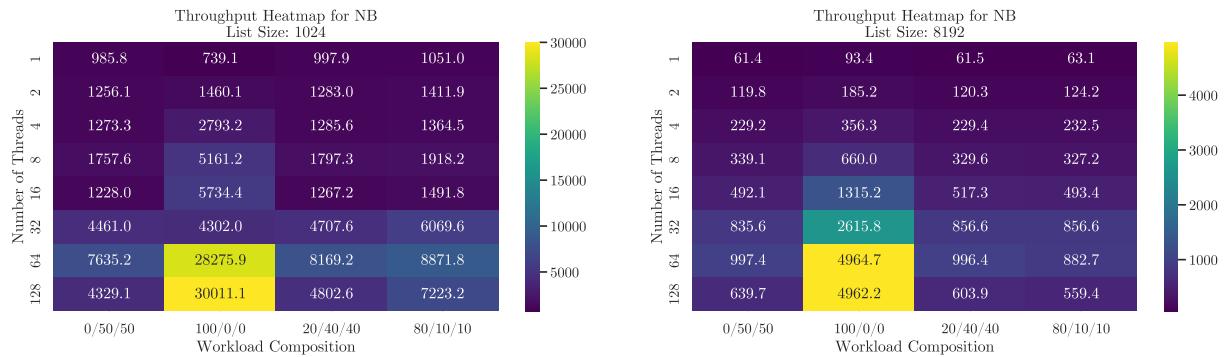
(α') Heatmap για την υλοποίηση CGL με μέγεθος λίστας 1024 (β') Heatmap για την υλοποίηση CGL με μέγεθος λίστας 8192



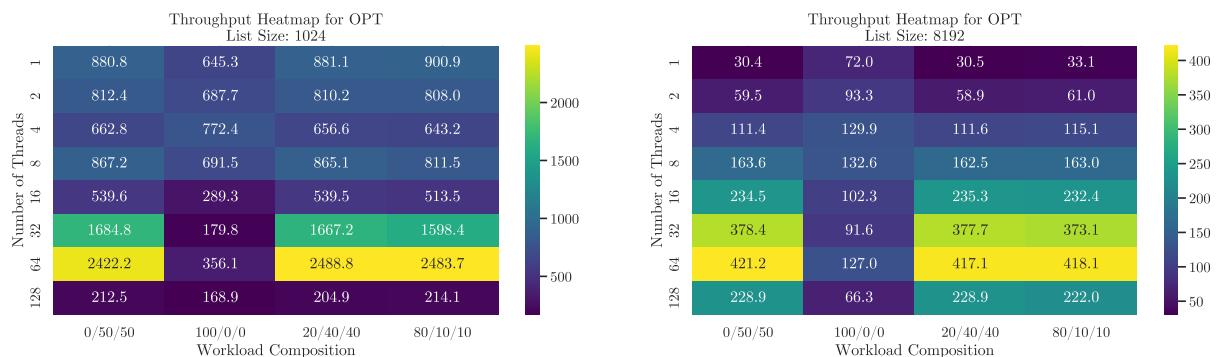
(γ') Heatmap για την υλοποίηση FGL με μέγεθος λίστας 1024 (δ') Heatmap για την υλοποίηση FGL με μέγεθος λίστας 8192



(ε') Heatmap για την υλοποίηση Lazy με μέγεθος λίστας 1024 (στ') Heatmap για την υλοποίηση Lazy με μέγεθος λίστας 8192



(ζ') Heatmap για την υλοποίηση Non-Blocking με μέγεθος λίστας 1024 (η') Heatmap για την υλοποίηση Non-Blocking με μέγεθος λίστας 8192



(θ') Heatmap για την υλοποίηση Optimistic με μέγεθος λίστας (ι') Heatmap για την υλοποίηση Optimistic με μέγεθος λίστας 1024
8192

Σχήμα 48: Heatmaps για διαφορετικές υλοποιήσεις με διάφορα μεγέθη λίστας. (συνέχεια)

3 Ζη Άσκηση – Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών – Αλγόριθμος KMeans

3.1 Naive υλοποίηση

3.1.1 Περιγραφή της υλοποίησης

Ο αλγόριθμος Kmeans αποτελείται από επανάληψη δύο διακριτών βήματων, την φάση όπου κάθε σημείο ανατίθεται στο κοντινότερο cluster, και την φάση που γίνεται ενημέρωση των clusters από τα σημεία που ανήκουν σε αυτό.

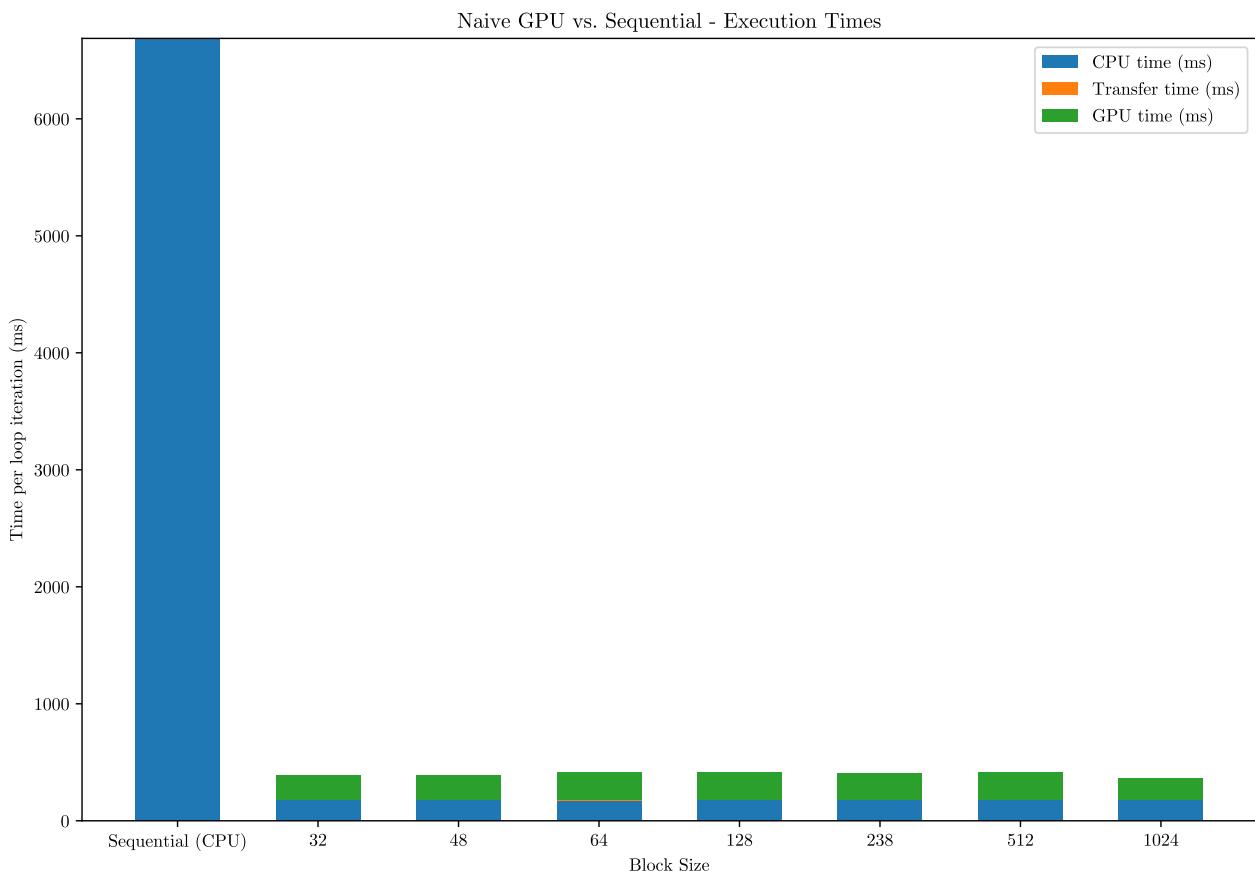
Σε αυτή την υλοποίηση ανατίθεται στην GPU η φάση ανάθεσης κάθε σημείου στο κοντινότερο cluster. Έτσι σε κάθε επανάληψη γίνεται αντίγραφη των clusters στην μνήμη της GPU, εκτελείται ο κώδικας ανάθεσης στην GPU, έπειτα αντιγράφεται ο πίνακας ανάθεσης κάθε σημείου σε cluster πίσω στην κύρια μνήμη, και τέλος η CPU υπολογίζει τα νέα κέντρα.

Στην GPU ανατίθεται σε κάθε νήμα ένα χωριστό CUDA thread. Ο χωρισμός σε thread blocks μπορεί, από απόψη ορθότητας, να είναι αυθαίρετος, καθώς δεν απαιτείται συγχρονισμός για κοινή δουλειά μεταξύ των νημάτων.

3.1.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1ο)

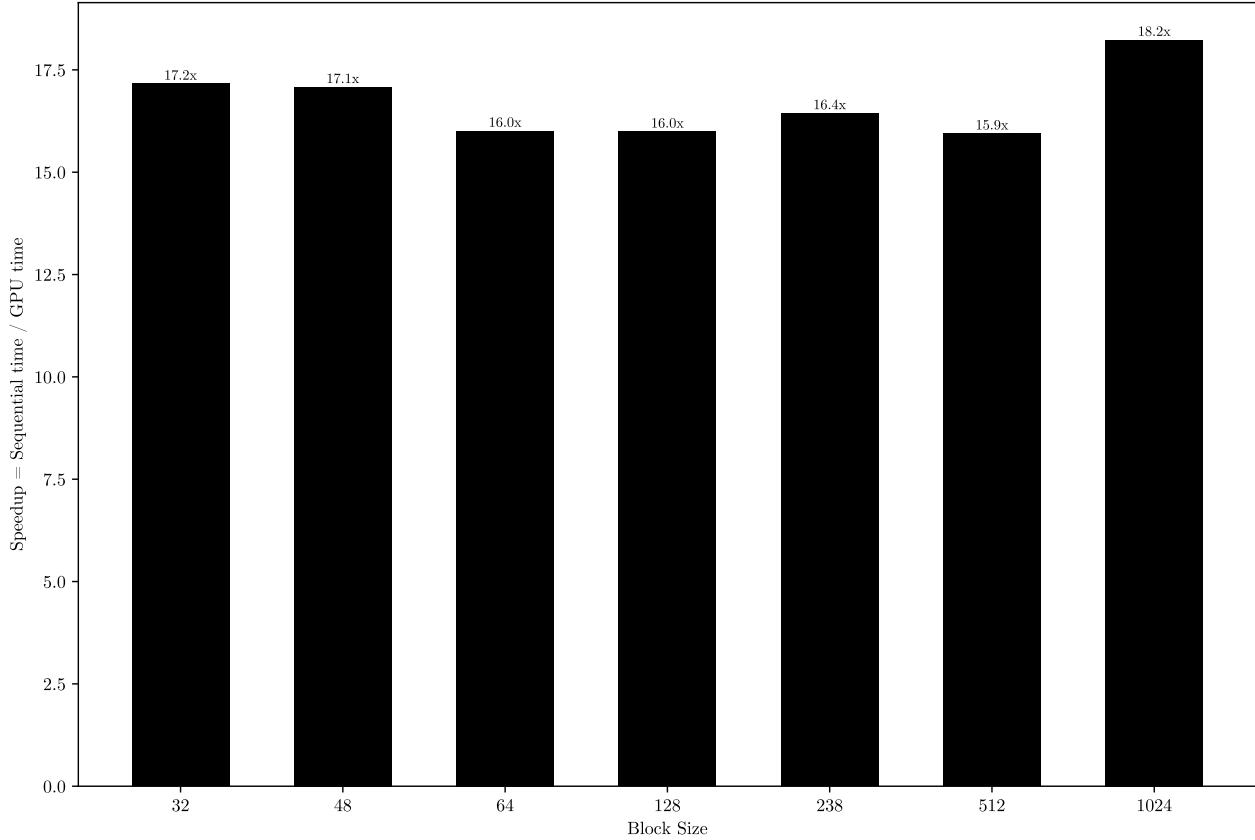
Πραγματοποιήθηκαν μετρήσεις για την naive αυτή υλοποίηση καθώς και για την σειριακή έκδοση σε CPU για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block size = {32, 48, 64, 128, 238, 512, 1024}.

Τα αποτελέσματα φαίνονται διαγραμματικά στο σχήμα 49 για τον χρόνο εκτέλεσης, και στο σχήμα 50 για την επιτάχυνση σε σχέση με την CPU.



Σχήμα 49: Χρόνος εκτέλεσης Naive GPU υλοποίησης του αλγόριθμου Kmeans

Speedup of Naive GPU vs. Sequential



Σχήμα 50: Επιτάχυνση (σε σχέση με CPU) Naive GPU υλοποίησης του αλγόριθμου Kmeans

3.1.3 Παρατηρήσεις – σχόλια (Ερώτημα 2o – α' μέρος)

Παρατηρούμε συνολική επιτάχυνση περίπου 17-18x. Ακριβέστερα, το τμήμα της ανάθεσης, που είναι και αυτό που επιταχύνθηκε στην GPU, έχει επιταχυνθεί περίπου 35x. Πρόκεται και για μια καλή επιτάχυνση, αν αναλογιστούμε ότι είναι μια αρκετά naive υλοποίηση, την στιγμή, μάλιστα, που η μέγιστη επιτάχυνση που λάβαμε με την βέλτιστη υλοποίηση στο NUMA μηχάνημα με τα 64 CPU νήματα ήταν περίπου 35-40x. (εκεί ήταν 4 Intel Xeon με 8 πυρήνες καθένας, και 2 SMT νήματα σε κάθε πυρήνα).

Παρόλαυτα, η επίδοση, ακόμα, δεν αξιολογείται ως ικανοποιητική με δεδομένο το πλήθος CUDA Cores της NVIDIA Tesla V100.

3.1.4 Παρατηρήσεις – σχόλια (Ερώτημα 2o – β' μέρος)

Η φάση του KMeans που αφορά στην εύρεση σε ποιο cluster ανήκει κάθε σημείο, είναι ένας αλγόριθμος που εκθέτει αρκετή παραλληλία, καθώς κάθε σημείο μπορεί να εξεταστεί από έναν διαφορετικό πυρήνα. Μάλιστα, όλοι οι πυρήνες προβλέπεται να τρέξουν τον ίδιο κώδικα, δεν υπάρχουν branches ανάλογα τις τιμές του κάθε σημείου, ούτε λογική που να οδηγεί σε άλλα μονοπάτια εκτέλεσης. Συνεπώς από αυτή την άποψη φαίνεται ένας καλός πυρήνας για GPU.

Επίσης, η αριθμητική ένταση του αλγόριθμου είναι αρκετά χαμηλή. Για κάθε 2 αριθμούς που φορτώνονται από την μνήμη (συντεταγμένη σημείου και συντεταγμένη cluster) γίνονται 1 διαφορά, 1 υπολογισμός και 1 άθροισμα, γεγονός που δείχνει μια αρκετά χαμηλή αριθμητική ένταση.

Φυσικά, η χαμηλή αριθμητική ένταση δεν είναι ιδανική έναντι του να είχαμε υψηλή αριθμητική ένταση, ωστόσο αποδίδει καλύτερα στην GPU έναντι της CPU, λόγω του υψηλού πολυνηματισμού της GPU που κρύβει τις καθυστερήσεις της μνήμης.

Συνολικά, το μόνο “άμεσο πρόβλημα” είναι πως οι προσβάσεις στους πίνακες *objects* και *clusters* γίνονται με ακανόνιστο τρόπο και όχι με γειτονικό τρόπο από γειτονικά threads του ίδιου warp, ωστόσο αυτό θα διορθωθεί στην επόμενη υλοποίηση.

Με άλλα λόγια, η φάση ανάθεσης του Kmeans χαρακτηρίζεται ως ένας σχετικά καλός πυρήνας για GPU.

Δεν τον σχολιάζουμε ως ιδανικό, διότι ο ιδανικός θα είχε και υψηλή αριθμητική ένταση.

3.1.5 Σχολιασμός του block size (Ερώτημα 3ο)

Παρατηρούμε ότι για τα διάφορα block size δεν παρατηρούνται σημαντικές διαφορές στην επιτάχυνση. Υπάρχει μια πτώση για 64 ως 512, ωστόσο στο 1024 η επιτάχυνση γίνεται μέγιστη. Με δεδομένο ότι δεν υπάρχει κοινή χρήση Shared Memory καθώς και κανένας συγχρονισμός μεταξύ των νημάτων του ίδιου Thread Block, οι μικρές αποκλίσεις στην επιτάχυνση αποδίδονται στον Thread Block Scheduler και στον Warp Scheduler.

Το ενδιαφέρον είναι πως για block size 48 και 238 δεν υπάρχει σημαντική αλλαγή στην επιτάχυνση. Αυτό φαίνεται εκ πρώτης όψης αρκετά περίεργο, διότι δεν είναι ακέραια πολλαπλάσια του warp size 32, οπότε προκύπτουν αντίστοιχα 25% και 7% σταθερά ανενεργά CUDA cores.

Όπως θα δούμε στην παράγραφο 3.2.3 υπάρχει εξήγηση στηριζόμενη, παραδόξως, στην L1 cache της GPU.

Σε κάθε περίπτωση, το γεγονός το ότι με inactive CUDA threads, η επίδοση δεν χειροτερεύει, είναι ήδη μια σημαντική ένδειξη πως το πρόγραμμα είναι memory bound, καθώς φαίνεται ότι αυξάνοντας τον ρυθμό που μπορούν να γίνουν υπολογισμοί (μεγαλύτερο SIMD utilization) η επίδοση δεν βελτιώνεται.

3.2 Transpose υλοποίηση

3.2.1 Περιγραφή της υλοποίησης

Παρατηρούμε ότι στην προηγούμενη υλοποίηση τα threads του ίδιου warp, όταν υπολογίζουν την Ευκλείδια Απόσταση, ζητάνε μακρινές διεύθυνσεις από την μνήμη, γεγονός που αξιολογείται ως αρκετά κακό για την επίδοση. Θα θέλαμε να είναι διπλανές, δηλαδή γετονικά νήματα του ίδιου warp να ζητάνε γειτονικές διεύθυνσεις.

Για να επιτευχθεί αυτό, χρησιμοποιούμε την αναπάρασταση του πίνακα objects και clusters στην ανάστροφη (transpose) μορφή τους σε σχέση με την naïve υλοποίηση, δηλαδή σε column-major αναπάρασταση. Με άλλα λόγια κάθε γραμμή του πίνακα δεν περιγράφει ένα σημείο, αλλά μια συντεταγμένη (αντίστοιχα μια στήλη περιγράφει ένα σημείο).

Ο υπολογισμός του transpose γίνεται στην αρχή του υπολογισμού για τον πίνακα objects οπότε και αντιγράφεται στην μνήμη της GPU, δηλαδή δεν απαιτείται να γίνεται σε κάθε επανάληψη.

Ο υπολογισμός του transpose γίνεται στην CPU (αν και θα μπορούσε να γίνεται και στην GPU – ωστόσο γίνεται στην CPU για απλότητα). Χρησιμοποιείται tiling ώστε τα δεδομένα να χωράνε συνεχώς στην L1 cache. Πιο συγκεκριμένα χρησιμοποιήθηκε block 32×32 , μεγεθούς $32 \cdot 32 \cdot 8 = 8192$ bytes που χωρά στην L1.

Σημείωση: Στα TODOs προτείνεται ο υπολογισμός του transpose για τον πίνακα clusters στο τέλος του υπολογισμού. Ωστόσο αυτό είναι μη αποδοτικό από την άποψη της CPU, καθώς στην φάση του update centroids χρησιμοποιείται η row-major μορφή για τα objects και η column-major για τα clusters. Πιο συγκεκριμένα η update centroids γίνεται από τον εξής κώδικα που έχει δοθεί ως σταθερός και δεν προβλέπεται η τροποποίηση του:

```
1 for (i = 0; i < numObjs; i++) {
2     /* find the array index of nestest cluster center */
3     index = membership[i];
4     /* update new cluster centers : sum of objects located within */
5     newClusterSize[index]++;
6     for (j = 0; j < numCoords; j++)
7         newClusters[j][index] += objects[i * numCoords + j];
8 }
9 /* average the sum and replace old cluster centers with newClusters */
10 for (i = 0; i < numClusters; i++) {
11     for (j = 0; j < numCoords; j++) {
12         if (newClusterSize[i] > 0)
13             dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
14         newClusters[j][i] = 0.0; /* set back to 0 */
15     }
16     newClusterSize[i] = 0; /* set back to 0 */
17 }
```

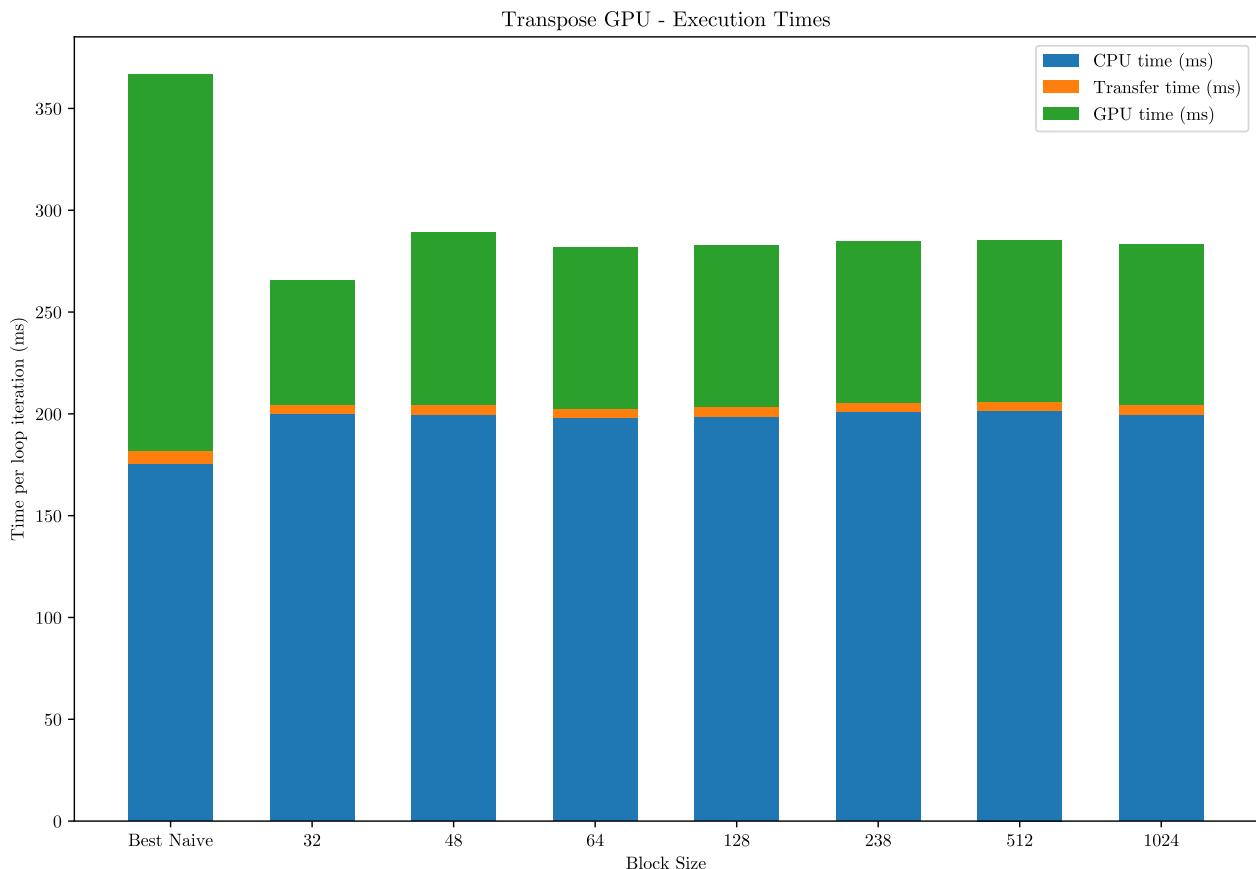
Έτσι οι προσβάσεις, στην CPU, στους πίνακες newClusters, dimClusters, γίνεται με μη συνεχή τρόπο, γεγονός που δεν συνέβαινε προηγούμενως, και που αυξάνει σημαντικά τα cache misses (στην CPU). Αν, ωστόσο, υπολογίζαμε σε κάθε επανάληψη τον transpose του πίνακα cluster και κανάμε το update centroids όπως πριν (ή

εναλλακτικά χρησιμοποιούσαμε τον transpose των objects για την update centroids) η επίδοση θα ήταν καλύτερη. Επειδή οι κώδικες αυτές δεν ήταν δηλωμένοι ως τμήματα που δίνονται για τροποποίηση, δεν τροποποιήθηκαν. Παρόλαυτα, όπως θα δούμε, οδηγούν σε αύξηση του χρόνου στην CPU.

3.2.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1o – α' μέρος)

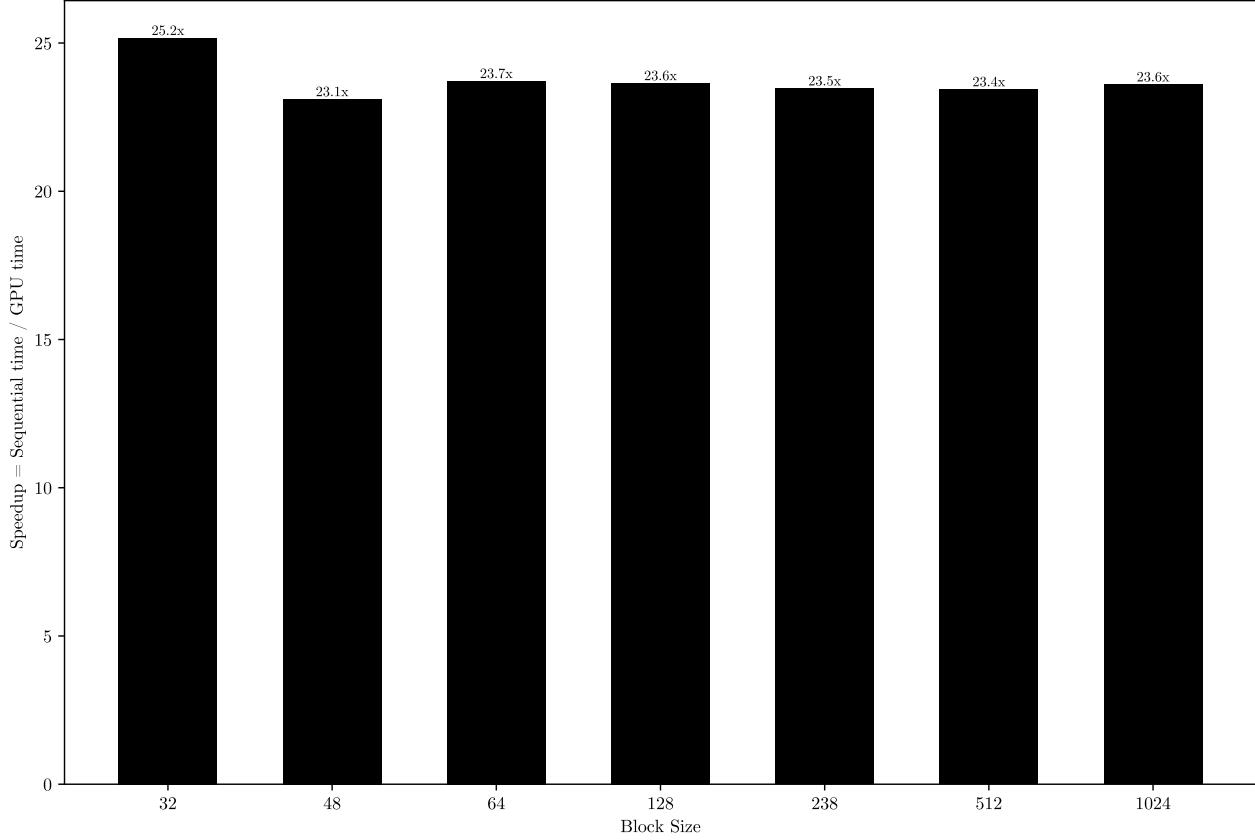
Πραγματοποιήθηκαν μετρήσεις για την transpose υλοποίηση καθώς και για την σειριακή έκδοση σε CPU για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block size = {32, 48, 64, 128, 238, 512, 1024}.

Τα αποτελέσματα φαίνονται διαγραμματικά στο σχήμα 51 για τον χρόνο εκτέλεσης, και στο σχήμα 52 για την επιτάχυνση σε σχέση με την CPU.



Σχήμα 51: Χρόνος εκτέλεσης Transpose GPU υλοποίησης του αλγόριθμου Kmeans

Speedup of Transpose GPU vs. Sequential



Σχήμα 52: Επιτάχυνση (σε σχέση με CPU) Transpose GPU υλοποίησης του αλγόριθμου Kmeans

3.2.3 Παρατηρήσεις – σχόλια (Ερώτημα 1o – β' μέρος)

Παρατηρούμε ότι η μέγιστη επιτάχυνση φτάνει το 25x, έναντι 18x στην naïve υλοποίηση, γεγονός που αντιστοιχεί σε 1.4x συνολική επιτάχυνση σε σχέση με την naïve υλοποίηση. Ακριβέστερα, το τιμήμα της GPU επιταχύνθηκε περίπου 3x.

Παρατηρούμε επίσης ότι ο χρόνος της CPU έχει αυξηθεί σε σχέση με την naïve υλοποίηση, γεγονός που συμβαίνει για τον λόγο που εξηγηθήκε στην Σημειώση της προηγούμενης παραγράφου 3.2.1.

Παρατηρούμε ότι η μέγιστη επιτάχυνση επιτυγχάνεται για block size 48. Αυτό αναφέρεται θεωρείται περίεργο, διότι $48 = 1 \cdot 32 + 16$ που σημαίνει ότι στα μισά warps χρησιμοποιείται το μισό warp, δηλαδή κατά μεση περίπτωση, το 25% των CUDA Cores είναι ανενεργό.

Η απάντηση (αναβλήθηκε στην naïve υλοποίηση για εδώ, διότι εκεί μπορούσε ίσως φαινομενικά να αποδοθεί στις προσβάσεις των clusters που γίνονται ακανόνιστα) εντοπίζεται στην L1 cache.

Στις Nvidia GPU, στην αρχιτεκτονική Volta, (σε αυτήν ανήκει η Nvidia Tesla V100 που χρησιμοποιούμε) υπάρχει η Unified Data Cache, που σε κάθε Streaming Multiprocessor, είναι 128KB.

Η shared memory είναι ένα τιμήμα αυτής της μνήμης, που τίθεται κατά την εκκίνηση (launch) του kernel σε κάποια από τις τιμές 0, 8, 16, 32, 64 ή 96 KB. Η ποσότητα που απομένει χρησιμοποιείται ως hardware-managed L1 cache.

Οι ιδιότητες της L1 cache αυτής, καθώς και γενικά οι λεπτομέρειες της αρχιτεκτονικής, δεν ανακοινώνονται από την Nvidia, και συνεπώς δεν μπορούμε να είμαστε σίγουροι για την λειτουργία της. Ωστόσο χρησιμοποιώντας ειδικά benchmarks, διάφοροι έχουν προσπαθήσει να κάνουν reverse engineer την αρχιτεκτονική. Πιο συγκεκριμένα στο report <https://arxiv.org/pdf/1804.06826.pdf> έχει γίνει μια μελέτη για την αρχιτεκτονική Volta.

Το ενδιαιρέρον είναι, που φαίνεται να επιβεβαιώνεται και από άλλες μελέτες σε παλαιότερες αρχιτεκτονικές, πως η πολιτική αντικάταστασης φαίνεται πως δεν είναι η LRU. Κατά τα λοιπά, το cache line φαίνεται να είναι στα 32 bytes, οπότε αν οι προσβάσεις γίνονται ακολουθιακά από τα νήματα ενός warp και ευθυγραμμισμένα, η επίδοση δεν ωφελείται από χωρική τοπικότητα. Ωφελείται ωστόσο από χρονική τοπικότητα.

Το πρόγραμμά μας, εμφανίζει χαρακτηριστικά χρονικής τοπικότητας, καθώς σε κάθε επαναλάβη το ίδιο σημείο συγκρίνεται πολλές φορές με όλα τα clusters. Μάλιστα, το πρόγραμμα μας, αφελείται και από κοινή χρήση του

ίδιου cluster αν αυτό βρίσκεται στην L1. Συνεπώς, προβλέπεται διαφορά στην επίδοση, ανάλογα με το αν το σημείο μπορέσει να παραμείνει στην L1 cache από την σύγκριση με ένα cluster μέχρι την σύγκριση με επόμενο cluster, καθώς επίσης και αν το cluster μπορέσει να μείνει στην L1 μέχρι όλα τα νήματα να το έχουν χρησιμοποιήσει.

Δεν υπάρχει ανταγωνισμός σε ίδια δεδομένα – εκτός του delta – ώστε να υπάρχουν invalidations λόγω πρωτοκόλλου συνάρτεις μνήμης (cache coherence protocol). Επίσης θα αγνοήσουμε τα Conflict miss, που προκύπτουν λόγω της διεύθυνσιοδότησης, θεωρώντας ότι το Set Associativity τα κρύβει ωκανοποιητικά αυτά. Συνεπώς τα cache invalidations προκύπτουν αποκλειστικά λόγω capacity, επειδή διαφορετικά threads κάνουν προσβάσεις στην μνήμη.

Κάθε σημείο έχει μέγεθος $32 \cdot 8 = 256$ bytes. Συνεπώς στην L1 χωράνε 512 σημεία. Τα σημεία αυτά είναι τα clusters, συνολικά πλήθους 64, και τα objects. Αν και η L1 φαίνεται ότι δεν είναι LRU από την μελέτη που δόθηκε παραπάνω, θα κάνουμε την προσέγγιση να θεωρήσουμε ότι τα παλιά clusters διαγράφονται από την κρυφή μνήμη. Επίσης θεωρούμε αμελητέο το ένα cache block για το delta. Με όλα λόγια, θεωρούμε ότι τα $512 - 1 \approx 512$ σημεία που χωράνε στην L1 διατίθεται για τα σημεία (objects) που ανατίθεται σε κάθε thread. Συνεπώς για να χωράνε στην L1, πρέπει να έχουμε το πολύ 512 νήματα σε κάθε Streaming Multiprocessor. Με δεδομένο ότι στην αρχιτεκτονική Volta ανατίθεται 32 thread blocks σε κάθε SM, θέλουμε πλήθος νημάτων ανά thread block, δηλαδή block size, περίπου 16, που παραδόξως είναι κάτω από ένα warp.

Πράγματι, εκτελώντας με block size 16, παρόλο που είναι το μισό warp size, οπότε η μισή GPU είναι inactive, ο χρόνος εκτέλεσης μειώθηκε κατά 30%.

Συμπερασματικά, το block size είναι ανάλογο του πλήθους CUDA threads που εκτελούνται σε έναν Streaming Multiprocessor (ανεξαρτήτως του πλήθους warps), που είναι ανάλογο των απαιτήσεων σε cache. Συνεπώς η αύξηση του block size σημαίνει αύξηση απαιτήσεων σε cache, και αν ξεπεραστεί το μέγεθος της cache, σημαίνει πως οι προσβάσεις θα καταληγούν στην μνήμη αντί στην cache. Έτσι, για μεγάλα block size, παρόλο που γίνονται ταχύτερα υπολογισμοί, οι μεταφορές από την μνήμη επιβραδύνονται, και έτσι συνολικά η επίδοση χειροτερεύει.

3.2.4 Ερμηνεία βελτίωσης επίδοσης έναντι της naive υλοποίησης (Ερώτημα 2o)

Όπως έχει εξηγηθεί και προηγουμένως, στην naive υλοποίηση τα 32 νήματα κάθε warp ζητάνε από την global memory διεύθυνσεις οι οποίες απέχουν μεταξύ τους ένα ολόκληρο σημείο (δηλαδή $32 \cdot 8 = 256$ bytes), οπότε πρέπει να γίνουν 32 διαφορετικές memory transactions.

Αντίθετα, στην transpose υλοποίηση και τα 32 νήματα του κάθε warp ζητάνε το καθένα μια διεύθυνση που ξεκινά στο τέλος των δεδομένων που ζητά το προηγούμενο του νήμα. Έτσι, όλο το warp ζητά ένα ενιαίο συνεχόμενο block από την μνήμη, που μπορεί να γίνει με μια ενιαία memory transaction, γεγονός που είναι σημαντικά ταχύτερο.

3.3 Shared Memory υλοποίηση

3.3.1 Περιγραφή της υλοποίησης

Στην φάση ανάθεσης κάθε σημείου στο κοντινότερο cluster, όλο το thread block χρησιμοποιεί τα ίδια clusters. Συνεπώς μπορούμε να τα προφορτώσουμε στην shared memory και να τα χρησιμοποιήσουμε από εκεί όλα τα threads του thread block. Αν και θα μπορούσαμε σε κάθε γύρο σύγκρισης με ένα cluster, να διαγράψουμε το παλιό cluster, να φορτώνουμε ένα νέο και να κάνουμε ένα `__syncthreads()`, επειδή το μέγεθος των clusters είναι σχετικά μικρό, προκειμένου να αποφύγουμε το κόστος συγχρονισμού, τα φορτώνουμε εξαρχής όλα.

Η αντιγραφή στην shared memory από την global memory γίνεται με κατανεμημένο τρόπο από όλα τα threads του thread blocks.

Γίνεται αναθέτοντας σε κάθε CUDA thread να μεταφέρει το στοιχείο που αντιστοιχεί στο id του, ώστε threads του ίδιου warp να μεταφέρουν γειτονικά στοιχεία. Παρατίθεται το σχετικό απόσπασμα κάτω:

```
1 for (int i = threadIdx.x; i < numClusters*numCoords; i += blockDim.x) {
2     shmemClusters[i] = deviceClusters[i];
3 }
```

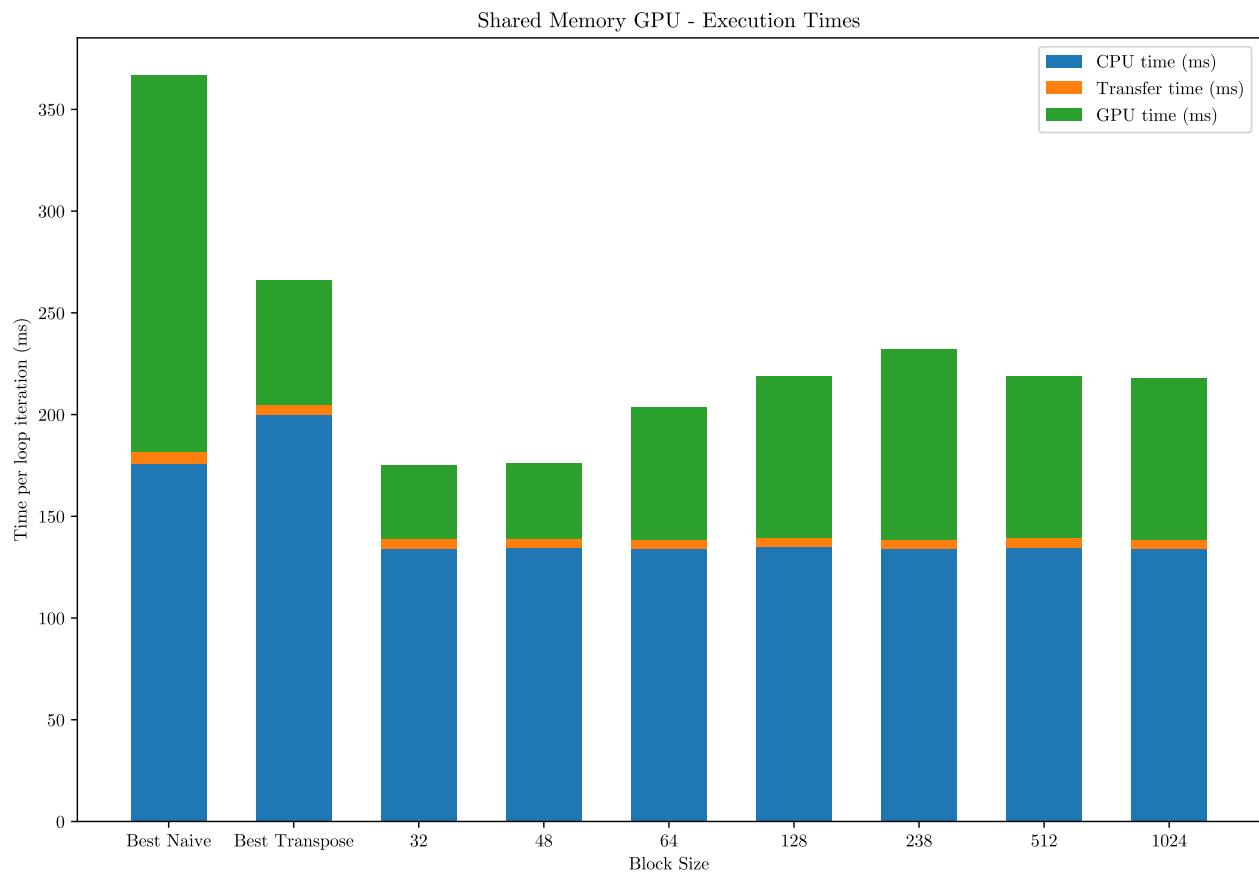
Ο απαιτούμενος χώρος shared memory που πρέπει να εκχωρηθεί σε κάθε thread block είναι:

$$\text{numClusters} \cdot \text{numCoords} \cdot \text{sizeof(double)} = 64 \cdot 32 \cdot 8 = 16KiB$$

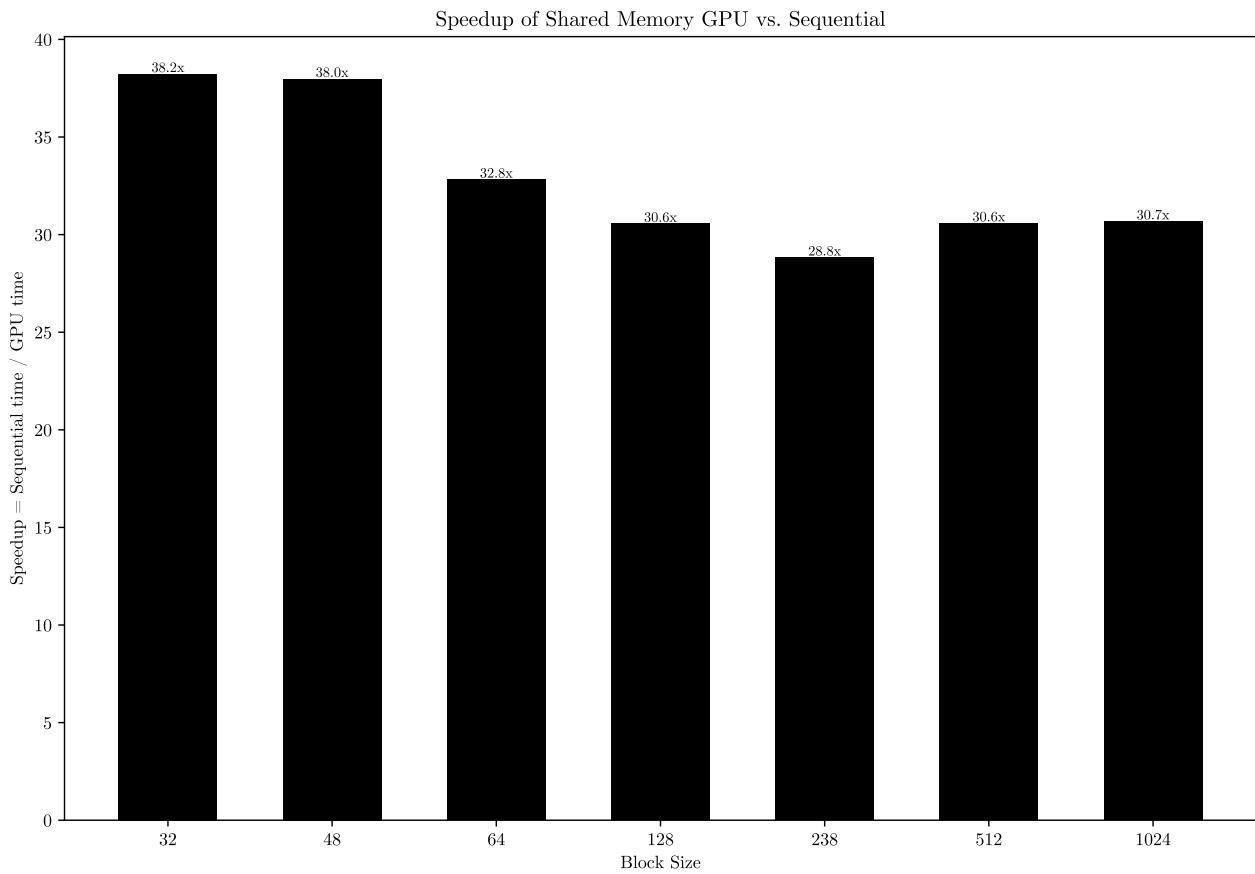
3.3.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση

Πραγματοποιήθηκαν μετρήσεις για την Shared Memory υλοποίηση καθώς και για την σειριακή έκδοση σε CPU για το configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10} και για block size = {32, 48, 64, 128, 238, 512, 1024}.

Τα αποτελέσματα φαίνονται διαγραμματικά στο σχήμα 53 για τον χρόνο εκτέλεσης, και στο σχήμα 54 για την επιτάχυνση σε σχέση με την CPU.



Σχήμα 53: Χρόνος εκτέλεσης Shared GPU υλοποίησης του αλγόριθμου Kmeans



Σχήμα 54: Επιτάχυνση (σε σχέση με CPU) Shared GPU υλοποίησης του αλγόριθμου Kmeans

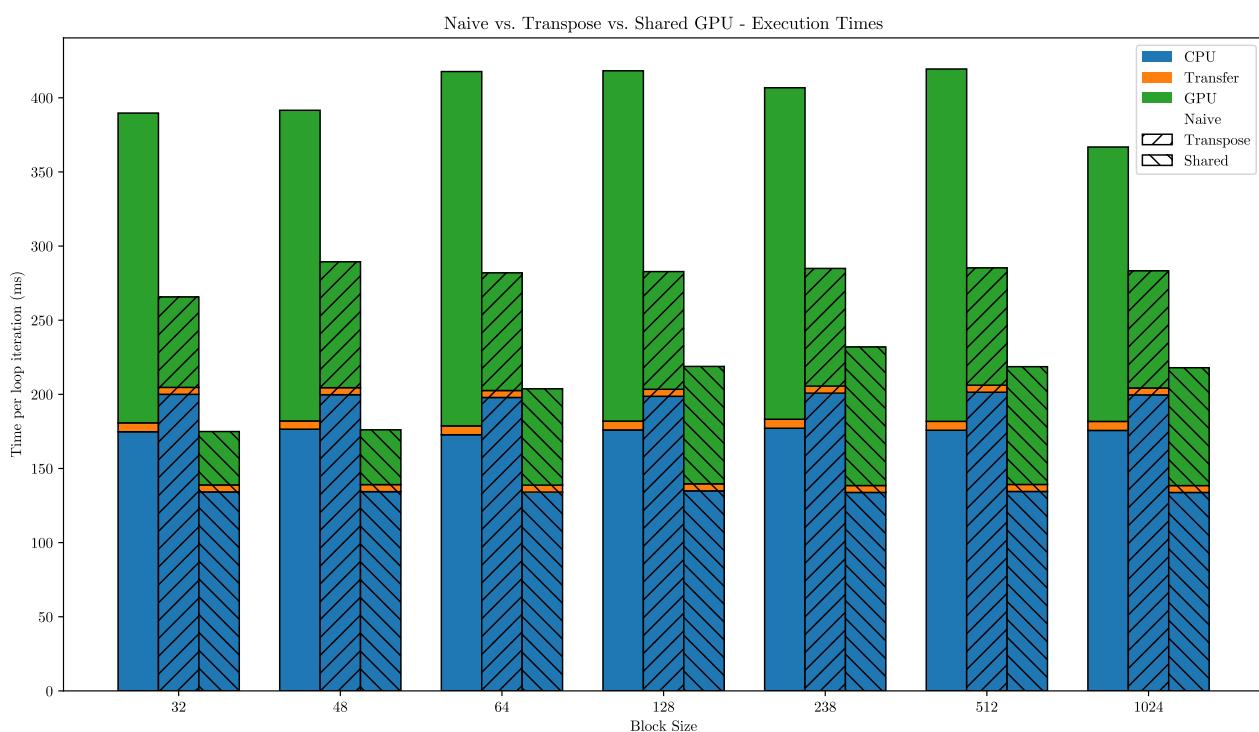
3.3.3 Παρατηρήσεις – σχόλια

Παρατηρούμε ότι η μέγιστη επιτάχυνση φτάνει το 38x, έναντι 25.2x στην transpose υλοποίηση, γεγονός που αντιστοιχεί σε 1.5x συνολική επιτάχυνση σε σχέση με την transpose υλοποίηση.

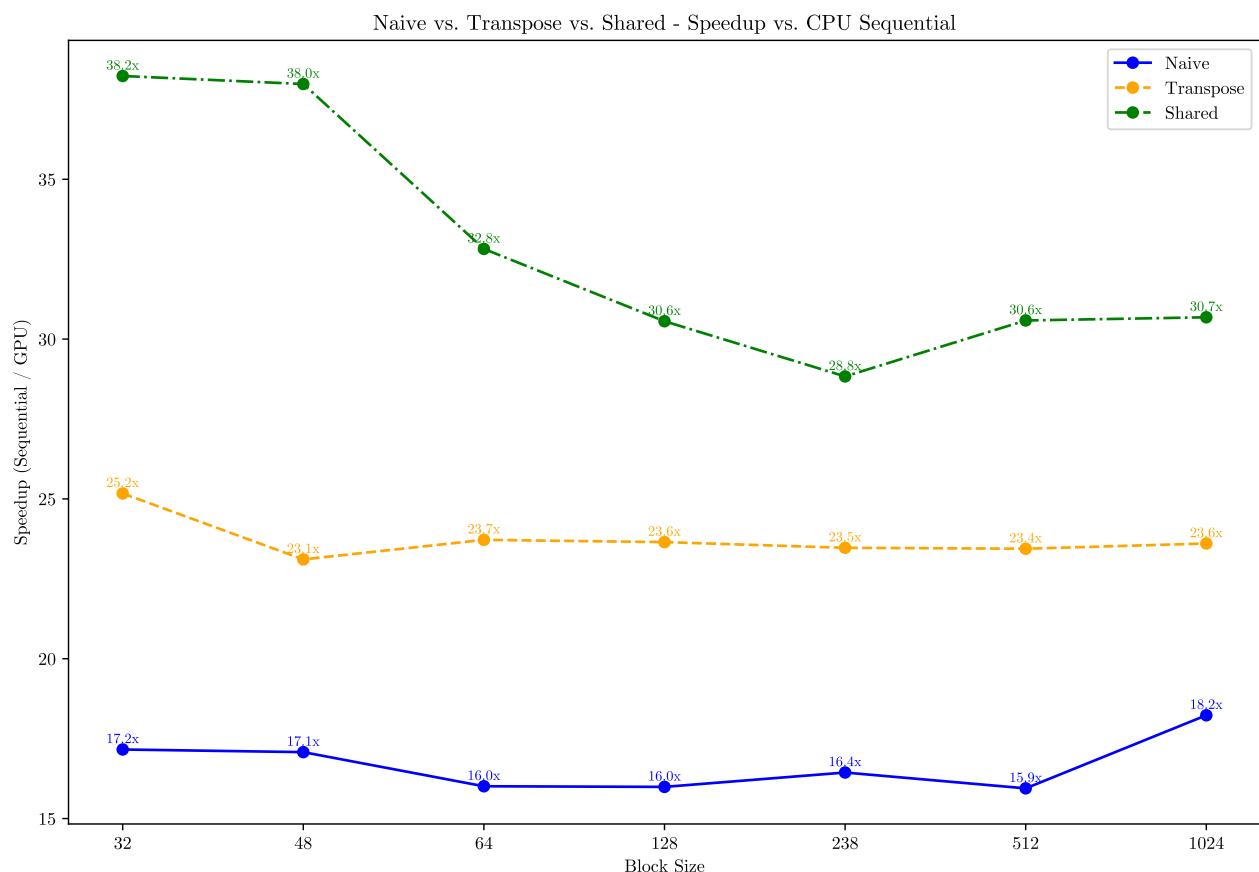
Η αύξηση επίδοσης είναι αρκετά καλή. Αν και φαινομενικά μπορεί να αναμέναμε μεγαλύτερη επιτάχυνση, στην πραγματικότητα προηγούμενως ήδη φορτώνονταν τα clusters στην L1 cache, απλά αυτό γινόταν με hardware-managed τρόπο που ήταν διαφανής προς τον προγραμματιστή.

3.3.4 Απεικόνιση των επιδόσεων όλων των υλοποιήσεων (Ερώτημα 1o – α' μέρος)

Απεικονίζονται διαγραμματικά οι χρονικές επιδόσεις και τριών υλοποιήσεων (naive, transpose, shared) ως προς το block size, στο σχήμα 55 για τον χρόνο εκτέλεσης, και στο σχήμα 56 για την επιτάχυνση σε σχέση με την CPU.



Σχήμα 55: Χρόνος εκτέλεσης των υλοποίησεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans



Σχήμα 56: Επιτάχυνση (σε σχέση με CPU) των υλοποιήσεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans

3.3.5 Επίδραση του block size (Ερώτημα 1o – β' μέρος)

Παρατηρούμε ότι ο χρόνος εκτέλεσης χειροτερεύει για μεγαλύτερα block size, γεγονός που δεν παρατηρούνται στην transpose και στην naive υλοποίηση.

Το γεγονός αυτό αποδίδεται σε bank conflicts στην shared memory. Καθώς αυξάνεται το block size, ο ίδιος πίνακας clusters της shared memory, διαβάζεται από περισσότερα νήματα, και μάλιστα όλα τα νήματα ζητάνε κάθε φορά το ίδιο cluster. Επειδή, κάθε bank της shared memory δίνει ένα μέγιστο εύρος ζώνης, καθώς αυξάνεται το block size, δηλαδή το πλήθος νημάτων που ζητάνε ταυτόχρονα δεδομένα από ίδιο bank, ο χρόνος εξυπηρέτησης των αιτημάτων από την shared memory αυξάνει. Επειδή το υπολογιστικό τμήμα είναι σχετικά σύντομο, η GPU αδυνατεί να κρύψει το latency αυτό και έτσι ο συνολικός χρόνος εκτέλεσης αυξάνει. Όταν η GPU κάνει schedule out ένα νήμα που περιμένει δεδομένα από την shared memory, κάνει schedule in ένα άλλο νήμα που και αυτό ζητάει δεδομένα από το ίδιο bank της shared memory και έτσι τα latencies αυξάνουν, οπότε γίνεται ακόμα πιο δύσκολο να τα κρύψουν οι υπολογισμοί.

Συνεχίζοντας όσα σχολιάστηκαν στην παράγραφο 3.2.3, μελέταμε μήπως προέκυψε και μεγαλύτερη έλλειψη της L1 cache. Κάθε Streaming Multiprocessor διαθέτει 128KB Unified Data Cache, εκ των οποίων η shared memory είναι ένα τμήμα αυτής της μνήμης που τίθεται κατά την εκκίνηση (launch) του kernel σε κάποια από τις τιμές 0, 8, 16, 32, 64 ή 96 KB. Η ποσότητα που απομένει χρησιμοποείται ως hardware-managed L1 cache.

Ο πίνακας των clusters καταλαμβάνει χώρο $64 \cdot 32 \cdot 8 = 16384$ bytes = 16 KiB στην shared memory. Συνεπώς, σε αντίθεση με προηγούμενως, αυτό δημιουργεί περιορισμό στο πόσα thread blocks μπορούν να ανατεθούν σε κάθε Streaming Multiprocessor έναντι του μεγίστου 32 που ισχύει για την Nvidia Tesla V100. Πιο συγκεκριμένα επιλέγεται το μέγιστο shared memory από την Unified Data Cache του Streaming Multiprocessor και έτσι ανατίθενται $\lfloor \frac{96}{16} \rfloor = 6$ thread blocks σε κάθε Streaming Multiprocessor. Έτσι απομένουν, $128 - 96 = 32$ KiB για L1 cache σε κάθε Streaming Multiprocessor. Ακολουθώντας την ίδια ανάλυση με της παραγράφου 3.2.3, αφού κάθε σημείο των objects καταλαμβάνει $32 \cdot 8 = 256$ bytes, στην L1 cache χωράνε $\lfloor \frac{32768}{256} \rfloor = 128$ σημεία. Συνεπώς προκειμένου να χωράνε τα σημεία στην L1 σε κάθε Streaming Multiprocessor, πρέπει να ανατίθεται το πολύ 128 νήματα, δηλαδή σε κάθε thread block να ανατίθεται το πολύ $\lfloor \frac{128}{6} \rfloor = 21$ σημεία που είναι συνεπώς το μέγιστο block size, ώστε τα σημεία των objects να χωράνε στην L1 cache και να μην συμβαίνουν capacity misses. Συνεπώς η προσφορά για L1 cache είναι μεγαλύτερη από ότι στην transpose υλοποίηση, όποτε δεν οφείλεται άμεσα η έλλειψη της L1 cache για την χειροτέρευση έναντι της transpose.

Με απλούστερα λόγια, στην παρούσα υλοποίηση αντιστοιχούν $\frac{32}{6} = 5.33$ KiB L1 cache ανά thread block, σε αντίθεση με $\frac{128}{32} = 4$ KiB στις naive και transpose (ανεξάρτητα του block size). Συνεπώς ως προς την L1, είναι πιο ευνοϊκή η κατάσταση.

3.4 Σύγκριση υλοποίησεων – Bottleneck Analysis

3.4.1 Bottleneck στο iterative μέρος του αλγορίθμου (Ερώτημα 1o)

Ήδη στην naive υλοποίηση, αλλά κυριότερα στις transpose και shared memory, ο χρόνος υπολογισμού της CPU είναι που κυριαρχεί στον χρόνο έκτελεσης.

Πιο συγκεκριμένα, στον καλύτερο χρόνο που έχει επιτευχθεί, που είναι στην shared memory υλοποίηση, για block size 32, η CPU καταλαμβάνει το 77% του χρόνου, η GPU το 21% του χρόνου και οι μεταφορές μεταξύ κύριας μνήμης και μνήμης GPU το 2% του χρόνου.

Με βάση τον Νόμο του Amdahl, η μέγιστη επιτάχυνση ως προς αυτήν την υλοποίηση που μπορούμε να επιτύχουμε βελτιστοποιώντας τον πυρήνα της GPU είναι μόλις:

$$\frac{1}{1 - 0.21} = 1.27x$$

δηλαδή από το 38x σε σχέση με την αρχική υλοποίηση σε CPU να πάμε σε περίπου 48x.

Στην πράξη βέβαια ακόμα και αυτή η αύξηση δεν είναι εφικτή διότι υποθέτει ότι ο χρόνος της GPU θα μηδενιστεί.

Αν αντίθετα μειώσουμε κατά μόλις 25% τον χρόνο της CPU θα πάρουμε περίπου ίδια επιτάχυνση, στόχος που φαίνεται σημαντικά πιο ρεαλιστικός. Μάλιστα, φαίνεται ότι ο χρόνος της CPU θα μπορούσε να επιταχυνθεί ίσως σημαντικά παραπάνω από αυτό, με πολύ μεγαλύτερα κέρδη στην τελική επίδοση.

Με άλλα λόγια, το bottleneck στην τρέχουσα υλοποίηση είναι οι υπολογισμοί που γίνονται στην CPU.

3.4.2 Μετρήσεις Επίδοσης και Διαγραμματική Απεικόνιση για 2o Configuration για όλες τις υλοποιήσεις (Ερώτημα 2o – α' μέρος)

Πραγματοποιήθηκαν μετρήσεις για τις naive, transpose και shared memory GPU υλοποιήσεις καθώς και για την σειριακή έκδοση σε CPU για το 2o configuration {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} και για block size = {32, 48, 64, 128, 238, 512, 1024}.

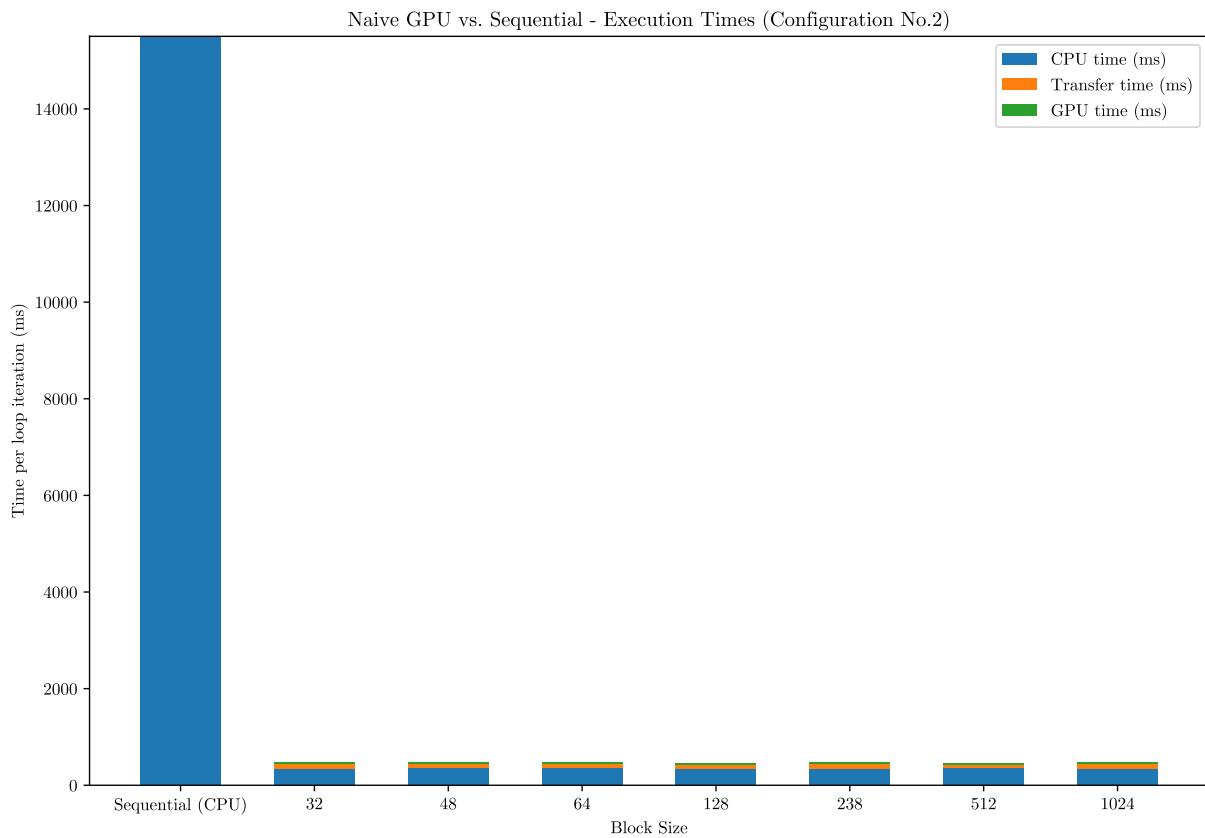
Τα αποτελέσματα φαίνονται διαγραμματικά:

Για την naive υλοποίηση Στο σχήμα 57 για τον χρόνο εκτέλεσης, και στο σχήμα 58 για την επιτάχυνση σε σχέση με την CPU.

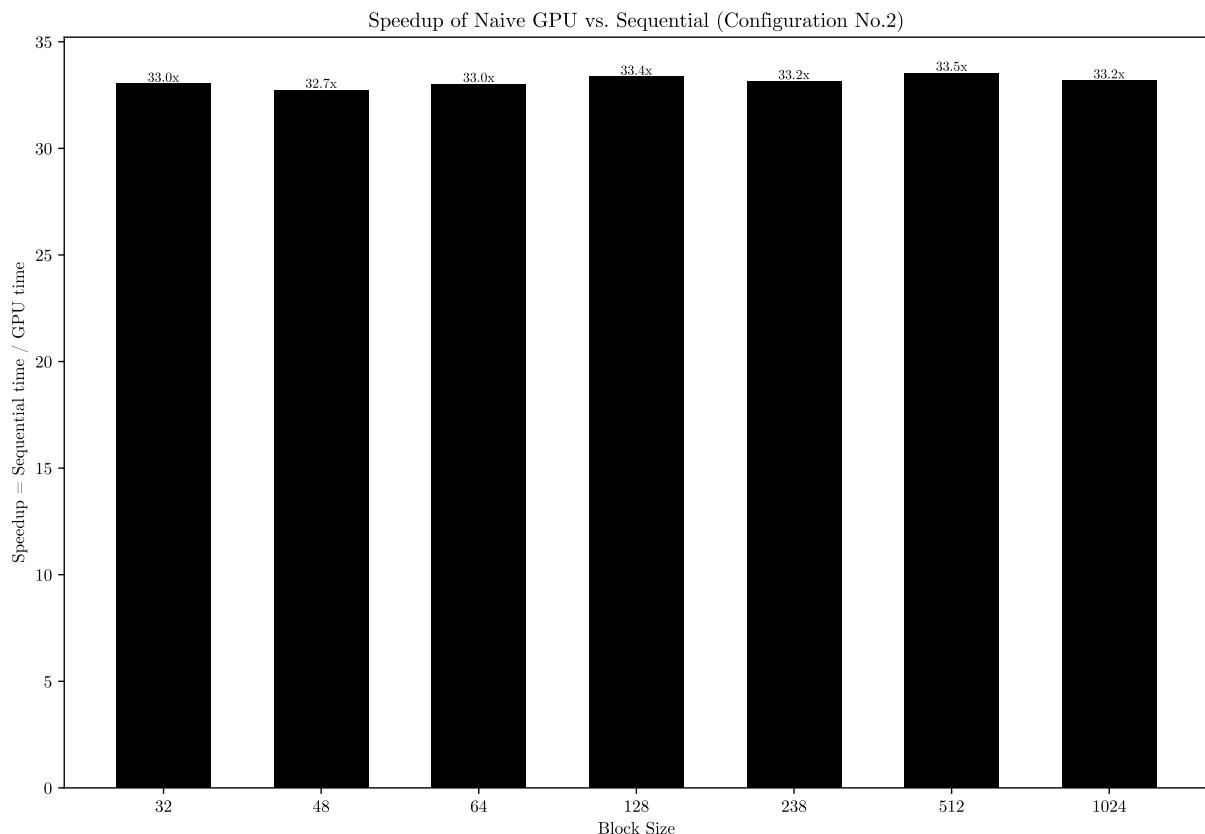
Για την transpose υλοποίηση Στο σχήμα 59 για τον χρόνο εκτέλεσης, και στο σχήμα 60 για την επιτάχυνση σε σχέση με την CPU.

Για την shared υλοποίηση Στο σχήμα 61 για τον χρόνο εκτέλεσης, και στο σχήμα 62 για την επιτάχυνση σε σχέση με την CPU.

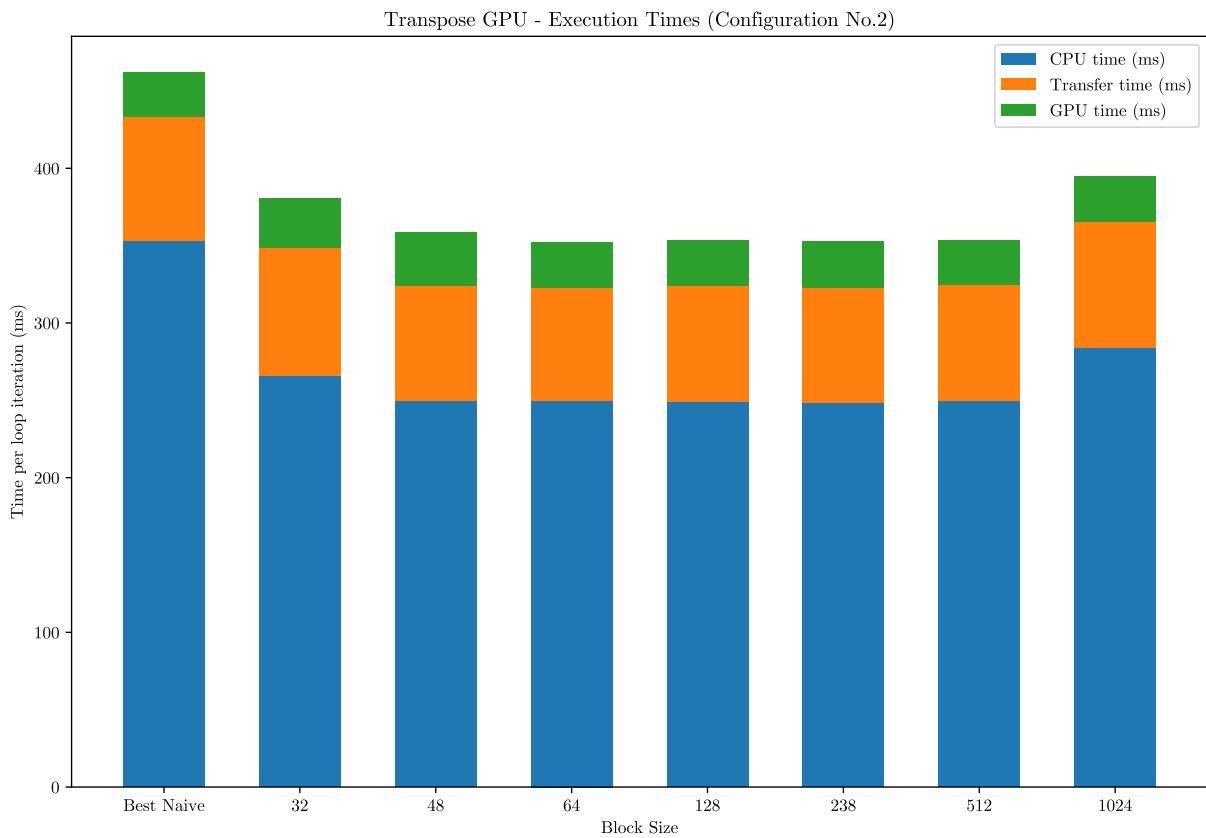
Συνολικά και για τις τρεις υλοποιήσεις Στο σχήμα 63 για τον χρόνο εκτέλεσης, και στο σχήμα 64 για την επιτάχυνση σε σχέση με την CPU.



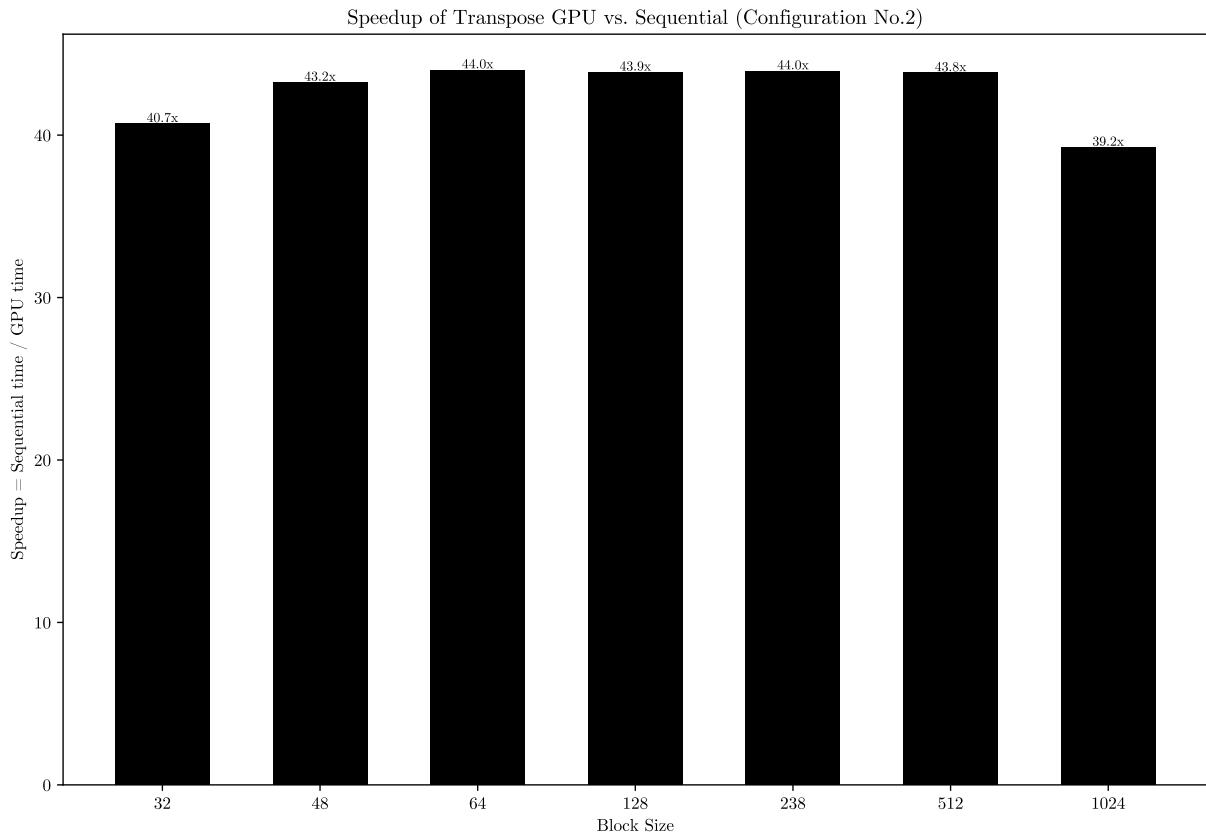
Σχήμα 57: Χρόνος εκτέλεσης Naive GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)



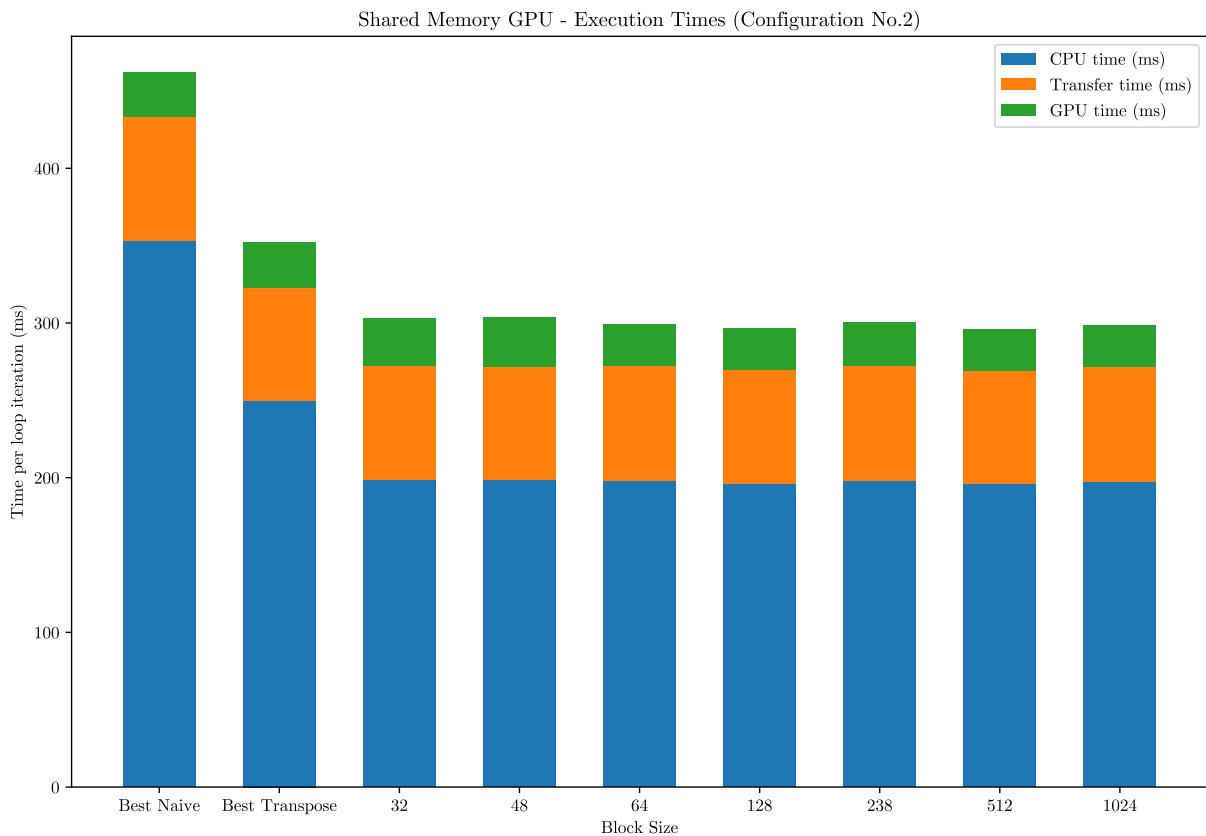
Σχήμα 58: Επιτάχυνση (σε σχέση με CPU) Naive GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)



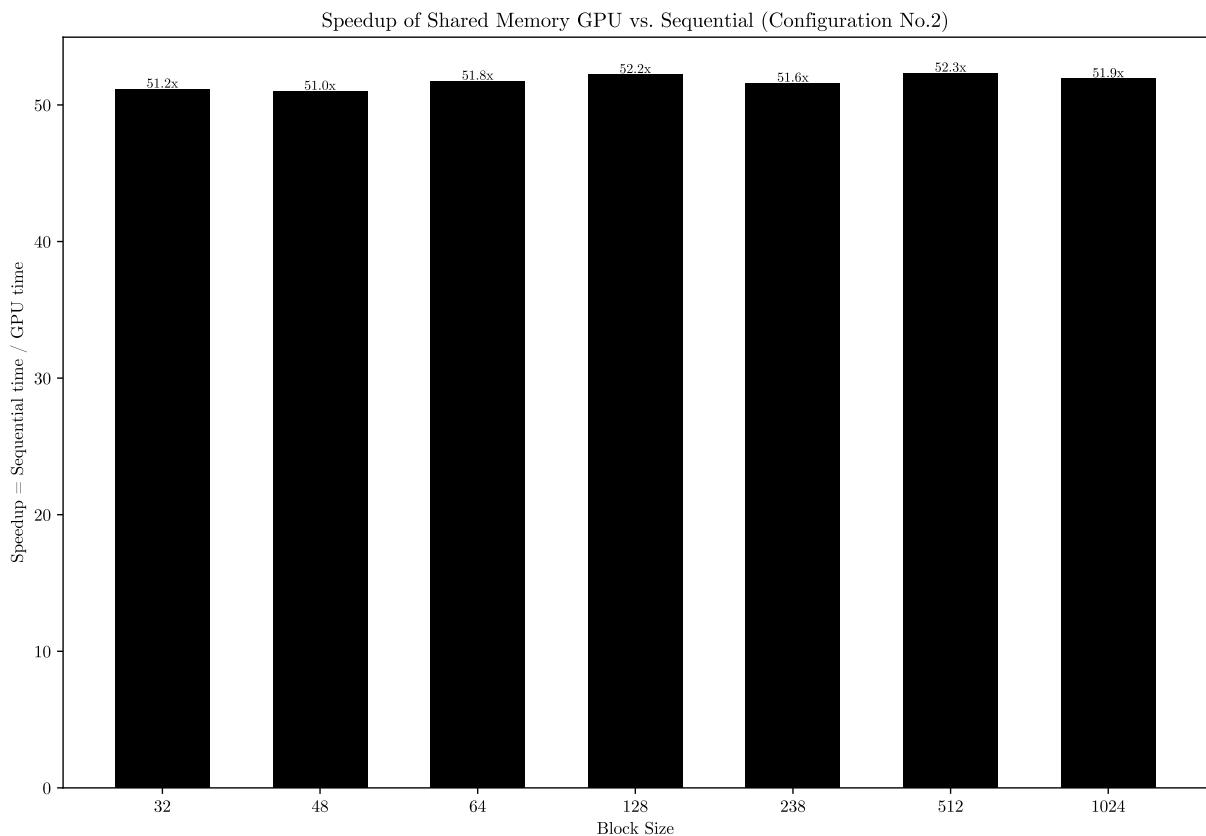
Σχήμα 59: Χρόνος εκτέλεσης Transpose GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)



Σχήμα 60: Επιτάχυνση (σε σχέση με CPU) Transpose GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)

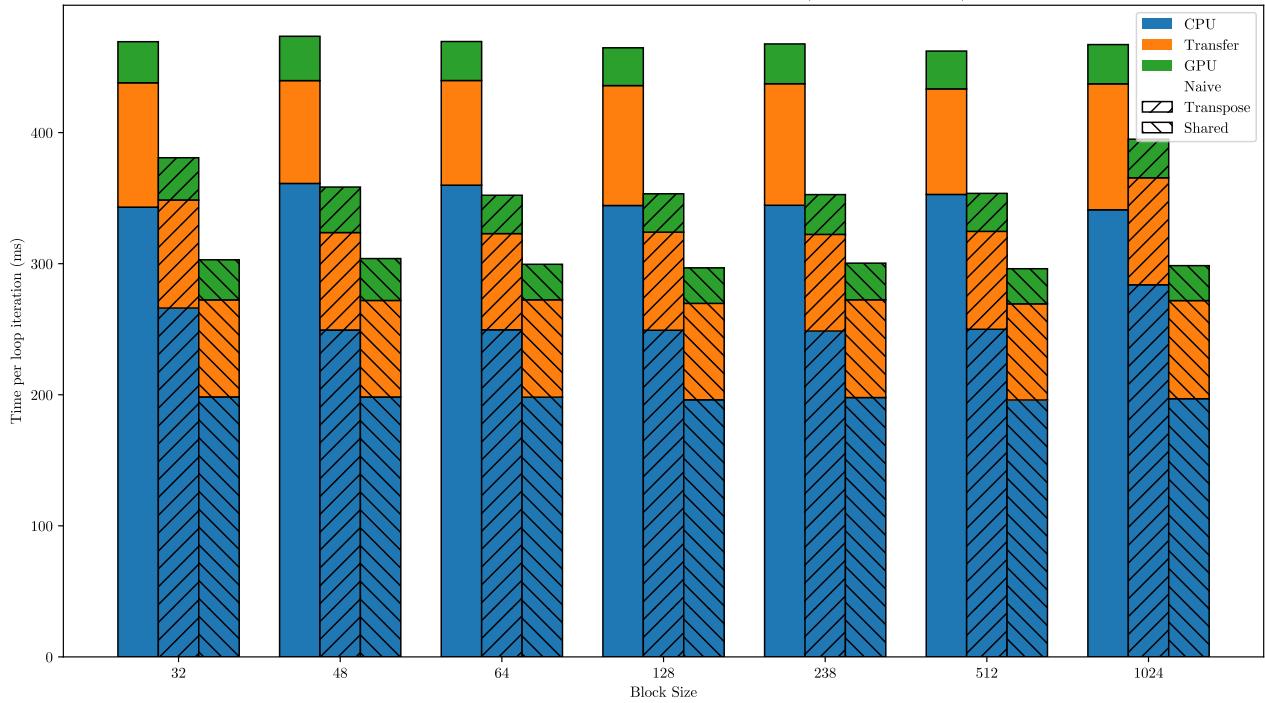


Σχήμα 61: Χρόνος εκτέλεσης Shared Memory GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)



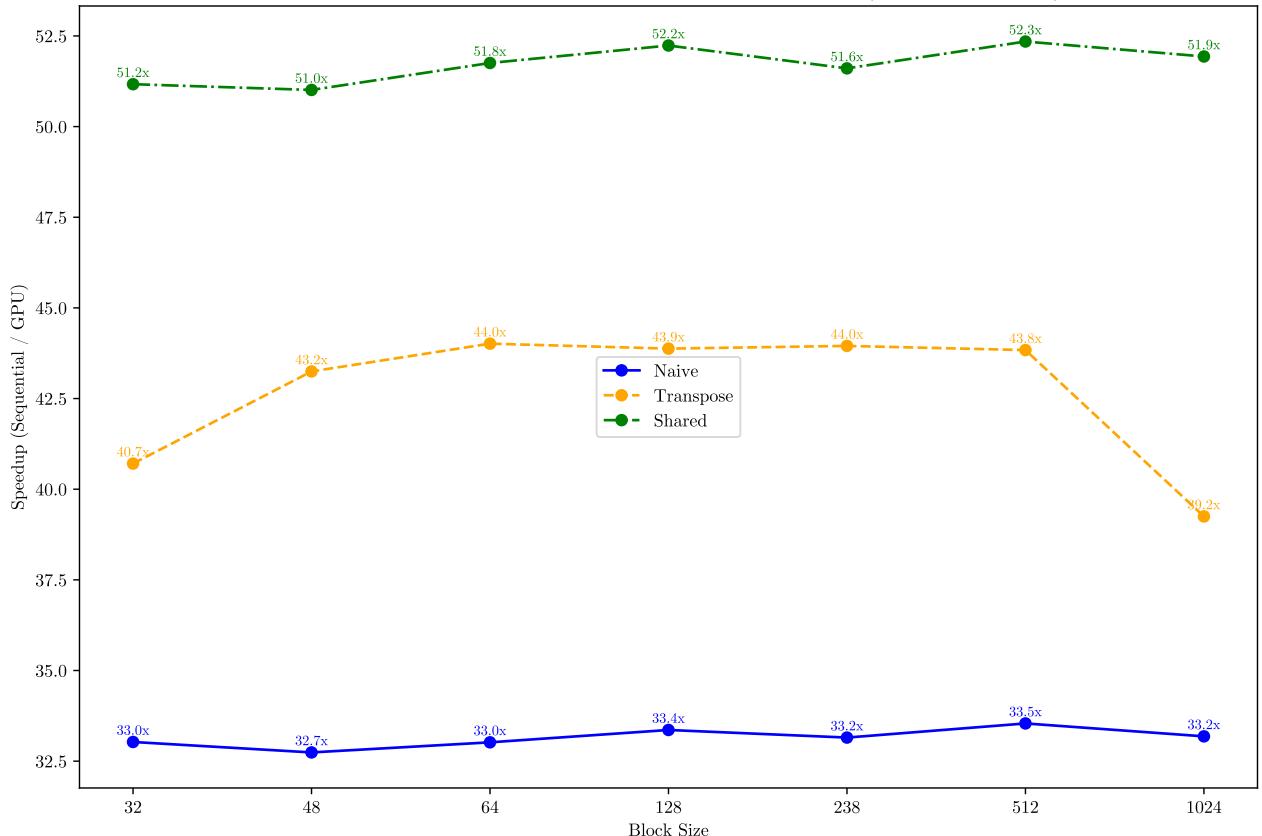
Σχήμα 62: Επιτάχυνση (σε σχέση με CPU) Shared Memory GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)

Naive vs. Transpose vs. Shared GPU - Execution Times (Configuration No.2)



Σχήμα 63: Χρόνος εκτέλεσης των υλοποίησεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans (2o Configuration)

Naive vs. Transpose vs. Shared - Speedup vs. CPU Sequential (Configuration No.2)



Σχήμα 64: Επιτάχυνση (σε σχέση με CPU) των υλοποιήσεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans (2o Configuration)

3.4.3 Παρατηρήσεις – σχόλια μεταξύ των δύο Configurations (Ερώτημα 2o – β' μέρος)

Σε πρώτο επίεδο, παρατηρούμε ότι οι συνολικοί χρόνοι εκτέλεσης σχεδόν έγιναν διπλάσιοι. Επίσης οι χρόνοι μεταφορών μεταξύ κύριας μνήμης και μνήμης GPU επίσης αυξήθηκαν σημαντικά. Παρόλαυτα, η επιτάχυνση των υλοποιήσεων GPU έναντι CPU αυξήθηκε.

Μειώνοντας από 32 σε 2 (16 φορές) το πλήθος συντεταγμένων αλλα κρατώντας σταθερό το μέγεθος σε bytes του dataset (σε 1024MiB), αυξήθηκε 16 φορές το πλήθος σημειών objects.

Αυτό σημαίνει ότι αυξήθηκε 16 φορές το μέγεθος του πίνακα membership που αντιγράφεται σε κάθε επανάληψη από την μνήμη GPU στην κύρια μνήμη, και αυτός είναι ο λόγος που παρατηρούμε τόσο σημαντική αύξηση στους χρόνους μεταφορών.

Λόγω της αύξησης σημειών αυξάνεται και ο χρόνος της φάσης update centroids στην CPU. Αυτό μπορεί να αποδοθεί στο μέγεθος της L3 cache. Ο Intel Xeon 4114 έχει 13.75MB L3 cache. Στο 1o configuration ο πίνακας membership έχει μέγεθος $4 \cdot \frac{1024 \text{ MiB}}{32.8 \text{ bytes}} = 16 \text{ MiB}$ ενώ αντίστοιχα στο 2o configuration έχει μέγεθος 256 MiB. Όταν εκτελείται το CudaMemcpy, η GPU αντιγράφει τα δεδομένα σε έναν DMA buffer στην κύρια μνήμη, και έπειτα ο driver αντιγράφει αυτά τα δεδομένα στους buffers του προγράμματος μας, όπου του ζητήσαμε. Έτσι, όταν τρέξει ο βρόχος update centroids, ο πίνακας membership έχει μόλις αντιγραφεί από τον driver, συνεπώς βρίσκεται στην L3 cache. Κατ' επέκταση στο 1o configuration, εν μέρει χωρά στην L3 cache, ενώ στο 2o configuration σχεδόν καθόλου, οπότε οι επιδόσεις στο 1o configuration είναι σημαντικά καλύτερες.

Παρατηρούμε, επίσης, και μια σημαντική διαφοροποίηση στην συμπεριφορά ως προς το block size στην Shared Memory υλοποίηση, που θα αναδειχθεί ακόμα παραπάνω στην All GPU υλοποίηση. Στην Shared Memory υλοποίηση, στο 1o Configuration, τα μικρά block sizes, δίνουν σημαντικά μικρότερο χρόνο εκτέλεσης, γεγονός που όπως εξηγήθηκε παραπάνω, αποδίδεται στην περιορισμένη χωρητικότητα της L1 cache. Αντίθετα, όμως, στο 2o Configuration, η χρονική επίδοση στην Shared Memory υλοποίηση φαίνεται να είναι σχεδόν ανεξάρτητη του block size. Το γεγονός αυτό προκύπτει επειδή οι συντεταγμένες ανά σημείο μειώθηκαν σημαντικά, οπότε ο χώρος που καταλάμβανε κάθε σημείο μειώθηκε και αυτός σημαντικά, οπότε πλέον στην L1 cache χωράνε σημαντικά περισσότερα σημεία και έτσι ακόμα και στην Shared Memory υλοποίηση (όπου αποδίδεται μικρό μέρος της Unified Data Cache στην L1 cache) υπάρχει μικρός ανταγωνισμός για χώρο στην L1 cache, ακόμα και στα μεγάλα block sizes που αναθέτουν πολλά σημεία ανά Streaming Multiprocessor (και άρα άνα “φυσική” L1 cache).

3.4.4 Καταλληλότητα shared memory GPU υλοποίησης για arbitrary Configurations (Ερώτημα 2o – γ' μέρος)

Η τρέχουσα shared memory GPU υλοποίηση δεν είναι κατάλληλη για arbitrary configurations, διότι στην υλοποίηση αυτή πρέπει όλος ο πίνακας των clusters να χωρά στην Shared Memory. Στην καλύτερη περίπτωση, ένα 1 thread block θα ανατεθεί σε κάθε Streaming Multiprocessor, και θα του αφιερωθεί το μέγιστο της Unified Data Cache που μπορεί να αφιερωθεί σε Shared Memory, δηλαδή 96KB. Αυτά επαρκούν για 12288 αριθμούς κινητής υποδιαστολής διπλής ακριβείας (double), που είναι το μέγιστο γινόμενο numClusters · numCoords που θα μπορέσει να τρέξει στην GPU. Για configurations που υπερβαίνουν αυτό το όριο, ο πυρήνας δεν θα μπορέσει καν να εκκινήσει διότι θα ζητάμε περισσότερη μνήμη από την φυσικά διαθέσιμη, οπότε θα λαμβάνουμε σφάλμα χρόνου εκτέλεσης.

Στην πραγματικότητα, όπως είχαμε σχολιάσει παραπάνω, δεν χρειάζονται όλα τα clusters σε όλο τον χρόνο υπολογισμού. Όλα τα νήματα διαβάζουν διαδοχικά από το πρώτο προς το τελευταίο cluster και από την πρώτη προς την τελευταία συντεταγμένη, χωρίς να ξαναδιαβάζουν δεδομένα από πριν. Έτσι, θα μπορούσαμε να φορτώνουμε τημήματα του πίνακα clusters στην shared memory, να συγχρονίζουμε το thread block με __syncthreads(), να προοδεύει ο υπολογισμός, να συγχρονίζουμε εκ νεύου το thread block και έπειτα να φορτώνουμε ένα ακόμα τημήμα στην shared memory, κ.ο.κ. Ο λόγος που δεν το κάναμε είναι διότι στα παρόντα μεγέθη του configuration θέλαμε να αποφύγουμε το τόσο συνεχόμενο συγχρονισμό που θα επέφερε χρονικό κόστος. Ωστόσο, καθώς μεγάλωνουν τα μεγέθη, ούτως ή αλλως θα χωρούσαν αρκετά στοιχεία στην shared memory, οπότε θα μεσολαβούσε σχετικά αρκετός χρόνος μεταξύ των συγχρονισμών, οπότε θα ήταν πιο ανεκτό.

3.4.5 Επιλογή block size με την συνάρτηση cudaMemcpyMaxPotentialBlockSize (Ερώτημα 3o – BONUS 1)

Δοκιμάστηκε η χρήση της συνάρτησης cudaMemcpyMaxPotentialBlockSize για την επιλογή του block size. Σκοπός της συνάρτησης αυτής είναι ο υπολογισμός του block size ώστε να μεγιστοποιείται το occupancy, δηλαδή ο λόγος CUDA threads ανά Streaming Multiprocessor προς το μέγιστο πλήθος CUDA threads ανά Streaming Multiprocessor, που στην Nvidia Tesla V100 είναι 2048 νήματα. Ωστόσο, λόγω του περιορισμένου πλήθους καταχωρητών και ποσότητας shared memory ανά Streaming Multiprocessor, η μέγιστη εφικτή λύση ενδεχομένως να μην είναι 2048, αλλά κάποιος μικρότερος αριθμός.

Η συνάρτηση επέστρεψε για την naïve υλοποίηση block size 768 και για όλες τις άλλες υλοποιήσεις (transpose, shared, All-GPU, All-GPU Delta Redction) 1024. Το μειωμένο block size για την naïve υλοποίηση οφείλεται στο γεγονός ότι ο πνcc χρησιμοποιήσε περισσότερους καταχωρητές σε αυτή την υλοποίηση.

Για block size 1024 έχουν γίνει ήδη όλες οι εκτελέσεις και τα σχετικά αποτελέσματα υπάρχουν σε όλα τα γραφήματα, και έχουν ήδη σχολιαστεί παραπάνω οι λόγοι που έχει μειωμένη επίδοση. Ανίστοιχα, για το block size 768 που προέκυψε για την naïve υλοποίηση, προκύπτουν προσεγγιστικά παρόμοιοι χρόνοι με το block size 512.

Αν και τα αποτελέσματα δεν είναι βέλτιστα, είναι χειρότερα (αναφερόμενοι αποκλειστικά στον χρόνο GPU) κατά περίπου 30% στις naïve και transpose υλοποιήσεις, οπότε η `cudaOccupancyMaxPotentialBlockSize` χαρακτηρίζεται ως μια καλή ευριστική. Αντίθετα, στις υλοποιήσεις με shared memory (Shared, All-GPU, All-GPU Delta) συμβαίνει το ανάποδο, και το προτεινόμενο block size δίνει 2x φορές επιβράδυνση έναντι του βέλτιστου. Αυτό προκύπτει, διότι όπως εξηγείται και παραπάνω αλλά και παρακάτω, σε αυτές τις υλοποιήσεις δεν στοχεύουμε στο μέγιστο occupancy, καθώς αυτό οδηγεί σε αυξημένο ανταγωνισμό για την χωρητικότητα της L1 cache σε κάθε Streaming Multiprocessor.

3.5 All-GPU Υλοποίηση

3.5.1 Περιγραφή της υλοποίησης

Παρατηρώντας προηγουμένως πως bottleneck είναι οι υπολογισμοί στην CPU, στην υλοποίηση αυτή επιθυμούμε να μεταφέρουμε τους υπολογισμούς αυτούς στην GPU με σκοπό την επιτάχυνσή τους.

Αυτό θα γίνει με δύο kernels που θα επηρεάζουν ίδια δεδομένα. Ο 1ος υπολογίζει το άθροισμα σημειών για τα νέα clusters και το πλήθος σημειών ανά cluster, και ο δεύτερος αναλαμβάνει να διαιρέσει τα άθροισμα με το πλήθος σημείων.

Για τον σκοπό αυτό, ο 1ος πυρήνας αντί να υπολογίζει τον πίνακα `membership`, υπολογίζει απευθείας το άθροισμα σημείων ανά νέο cluster. Για να αποφευχθούν τα atomics, διατηρούμε έναν πίνακα στην shared memory για κάθε thread block, όπου υπολογίζουμε το μερικό άθροισμα σημείων για κάθε cluster σε κάθε thread block. Η άθροιση ανά thread block γίνεται με atomics. Ωστόσο, αυτό δεν αποτελεί πρόβλημα διότι σε μια δεδομένη χρονική στιγμή, ο Streaming Multiprocessor εκτελεί ένα warp, οπότε αν τα 32 nήματα του warp αντιστοιχούν σε σημεία σε διαφορετικό cluster (λόγω τυχαιότητας της κατανομής σημειών σε clusters αυτό θα ισχύσει) δεν προκύπτει ανταγωνισμός κατά τις ατομικές ενημερώσεις ή τουλάχιστον προκύπτει ελάχιστος. Στο τέλος, τα μερικά άθροισμα πλέον όλου του thread block αθροίζονται με `atomicAdd` σε ένα ενιαίο άθροισμα στην global memory (η άθροιση γίνεται κατανεμημένα χωριστά για κάθε στοιχείο των μερικών άθροισμάτων – πλήθους `numClusters` · `numCoords` – από όλο το thread block).

Με άλλα λόγια, υλοποιούμε ένα reduction 2 επιπέδων, πρώτα μεταξύ των nημάτων ενός thread block, στην shared memory, και έπειτα μεταξύ των thread blocks, στην global memory. Αφού υλοποιούμε αυτού του είδους το reduction, το υλοποιούμε και για το `delta` προκειμένου να κερδίσουμε και σε αυτό επίδοση.

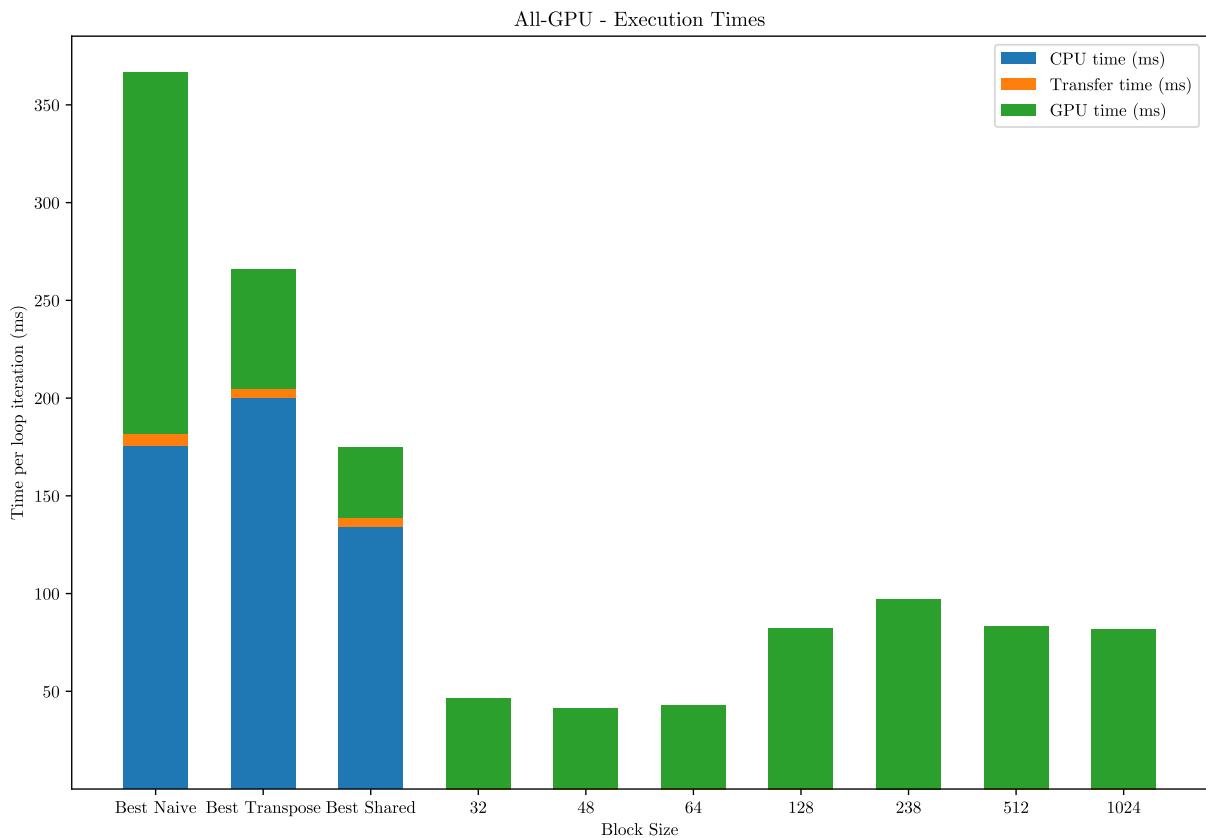
Ο 2ος πυρήνας εκκινεί αφού ο 1ος ολοκληρωθεί (η CPU αναλαμβάνει τον συγχρονισμό μέσω `cudaDeviceSynchronize`) και απλά διαιρεί τα άθροισμα σημειών ανά cluster με το πλήθος σημειών ανά cluster. (Ο μόνος λόγος που χρειάζεται ο 2ος πυρήνας να είναι διαχριτός από τον 1ο, είναι για να εξασφαλίσουμε ότι ο 1ος πυρήνας θα έχει ολοκληρωθεί, που δεν μπορεί να γίνει διαφορετικά, καθώς η CUDA δεν υποστηρίζει barrier μεταξύ των thread blocks.) Στον 2ο πυρήνα προφορτώνουμε τα cluster sizes στην shared memory ώστε να τα χρησιμοποιήσουμε για όλο το cluster χωρίς να χρειάζονται επαναφορτώσεις από την global memory. Για τον 2ο πύρηνα, θέσαμε το block size που μεγιστοποιεί το occupancy (δεν υπάρχει χρονική τοπικότητα εδώ, εκτός των cluster sizes που τοποθετήθηκαν όμως στην shared memory, ώστε να είχε σημασία η L1 εδώ). Έτσι το block size επιλέχθηκε με χρήση της συνάρτησης `cudaOccupancyMaxPotentialBlockSize`.

3.5.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1o – α' μέρος)

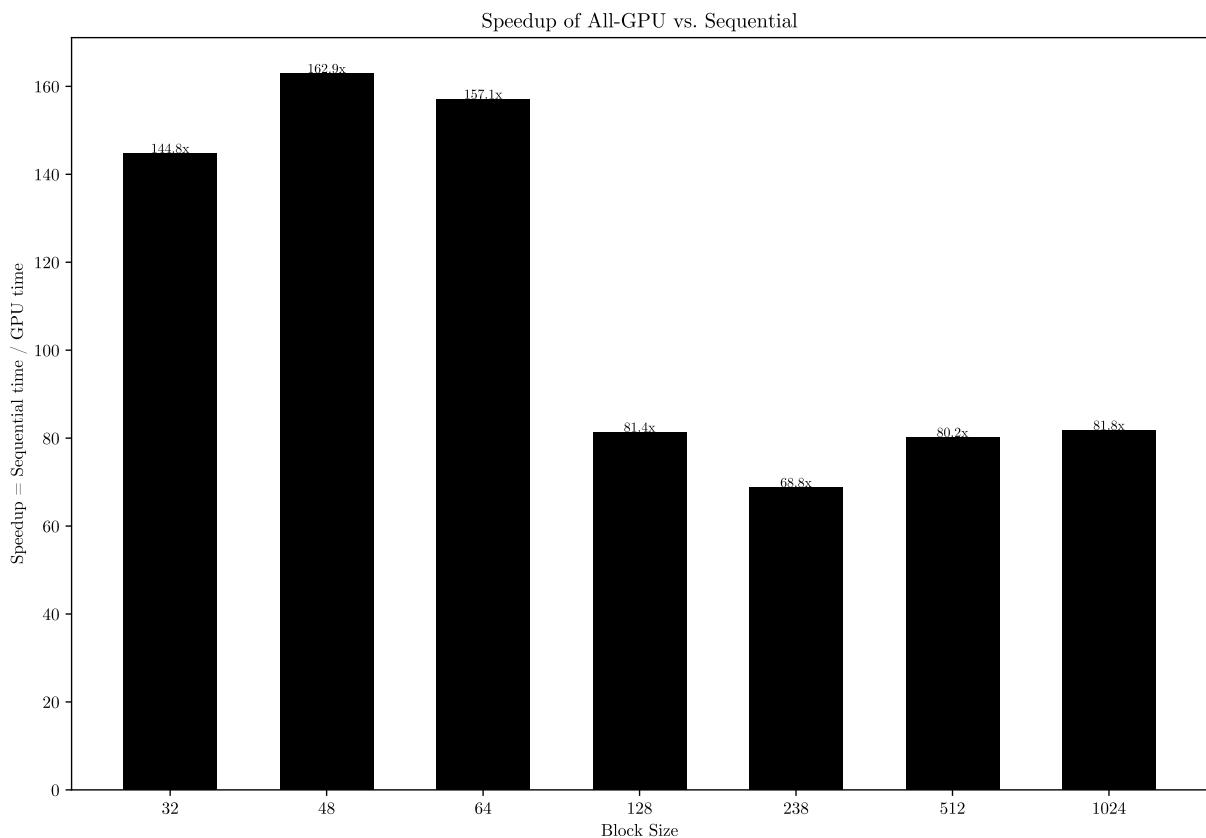
Πραγματοποιήθηκαν μετρήσεις για την All-GPU υλοποίηση για το 1o configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}, για το 2o configuration {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} καθώς και για `block size` = {32, 48, 64, 128, 238, 512, 1024}.

Τα αποτελέσματα φαίνονται διαγραμματικά:

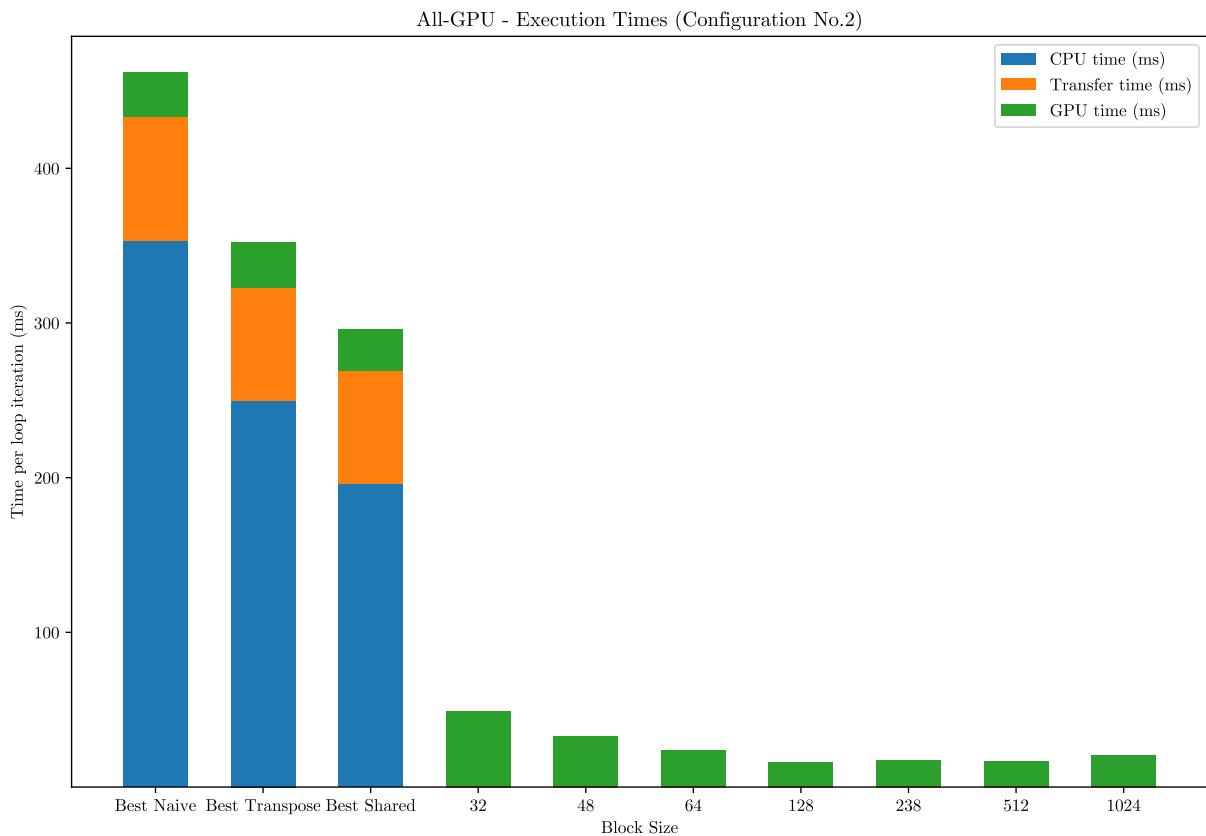
- Για το 1o Configuration στο σχήμα 65 για τον χρόνο εκτέλεσης, και στο σχήμα 66 για την επιτάχυνση σε σχέση με την CPU.
- Για το 2o Configuration στο σχήμα 67 για τον χρόνο εκτέλεσης, και στο σχήμα 68 για την επιτάχυνση σε σχέση με την CPU.



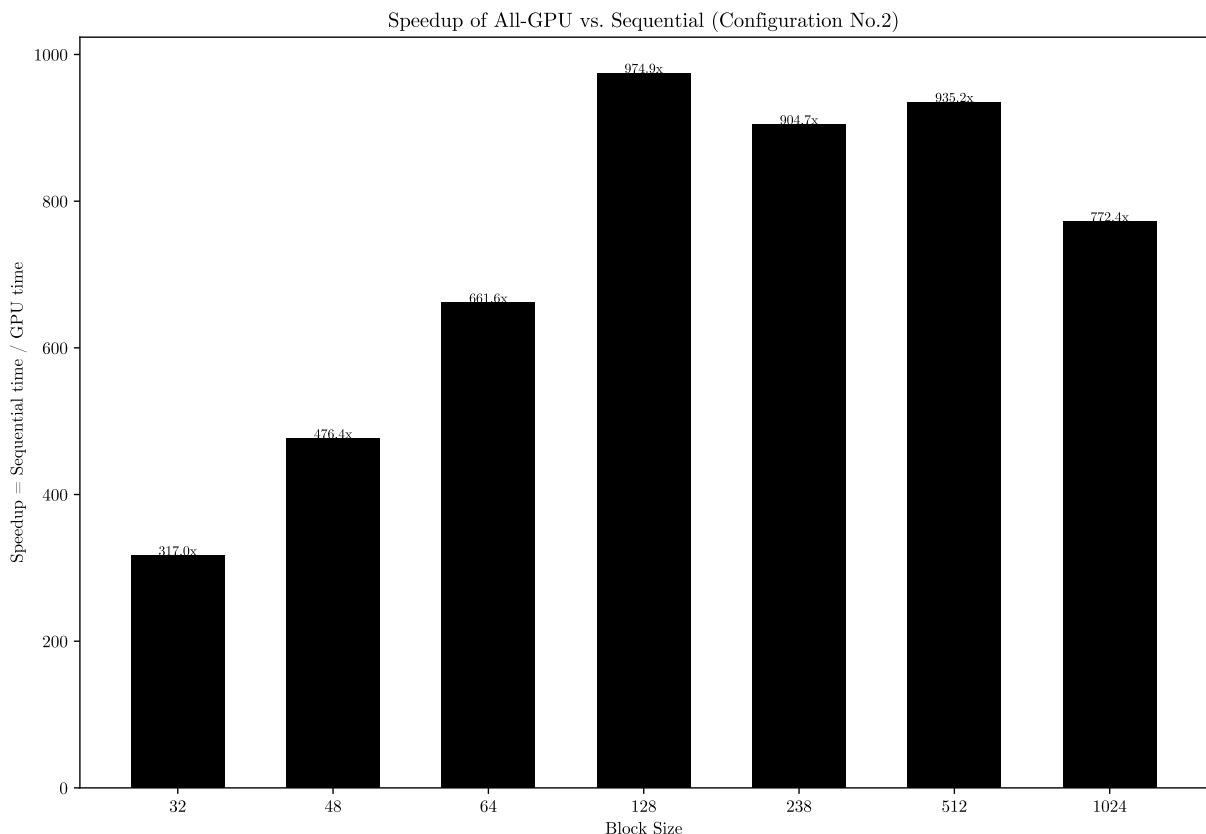
Σχήμα 65: Χρόνος εκτέλεσης All-GPU υλοποίησης του αλγόριθμου Kmeans



Σχήμα 66: Επιτάχυνση (σε σχέση με CPU) All-GPU υλοποίησης του αλγόριθμου Kmeans



Σχήμα 67: Χρόνος εκτέλεσης All-GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)



Σχήμα 68: Επιτάχυνση (σε σχέση με CPU) All-GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)

3.5.3 Παρατηρήσεις – σχόλια (Ερώτημα 1o – β' μέρος)

Η επίδοση είναι σημαντικά καλύτερη, αγγίζοντας σχεδόν 160x στο 1o Configuration και σχεδόν 1000x στο 2o Configuration, σε σχέση με την CPU. Οι επιταχύνσεις αυτές αντιστοιχούν σε 4x και 20x αντίστοιχα στα δύο Configurations σε σχέση με το βέλτιστο block size στην Shared Memory υλοποίηση.

Ο χρόνος GPU έχει ελάχιστα αυξηθεί σε σχέση με την Shared Memory υλοποίηση, όμως έχουν εξαφανιστεί οι χρόνοι της CPU και των μεταφορών, και λόγω αυτού έχει προκύψει η δραματική αύξηση της επιτάχυνσης.

Αν και δεν διαθέτουμε profiling στοιχεία που να το αποδεικνύουν, πιστεύουμε πως τα atomic operations στην global memory, δηλαδή το reduction 2ου επιπέδου μεταξύ των thread blocks, που είναι το στοιχείο που πιστεύουμε πειρισσότερο πως θα οδηγήσει σε καθυστερήσεις, δεν επηρεάζει τελικά τόσο σημαντικά, καθώς βλέπουμε τον χρόνο GPU να μην έχει αυξηθεί σημαντικά σε σχέση με την Shared Memory υλοποίηση. Αυτό μπορεί να δικαιολογηθεί καθώς ο χρόνος εκτέλεσης ενός thread block μέχρι να φτάσει στο τέλος του που γίνονται τα atomics είναι αρκετά μεγάλος, οπότε αφενός μεν τυχαίες καθυστερήσεις μειώνουν τον συναγωνισμό στα atomics, αφετέρου δε η GPU μπορεί στην αναμονή για τα atomics να ξεκινήσει άλλο thread block.

Το σχεδόν 1000x που δίνει το 2o Configuration, θεωρείται πλέον αρκετά καλό, καθώς η NVIDIA Tesla V100 έχει 2560 CUDA Cores, που είναι πλέον παραπλήσιο της επιτάχυνσης.

Το 160x του 1o Configuration, σε αντιπαράθεση με το 1000x, πάντως φαίνεται λίγο προβληματικό. Θα αναλύσουμε την διαφορά περισσότερο στο Ερώτημα 4o, όπου θα το απόδοσουμε στα atomics στην global memory.

Με άλλα λόγια, το επόμενο βήμα, που θα επιχειρούσαμε για βελτίωση της επίδοσης, είναι να γίνει reduction μεταξύ των thread blocks χωρίς χρήση atomics στην global memory (ενδεικτικά μπορεί να γίνει με έναν τρίτο kernel που θα αναλάβει το reduction).

3.5.4 Επίδραση του block size (Ερώτημα 2o)

Στην Shared Memory υλοποίηση, είχαμε καλύτερη επίδοση για μικρά block sizes στο 1o Configuration (λόγω της επίδρασης της L1), ενώ στο 2o Configuration είχαμε περίπου παρόμοια επίδοση για όλα τα block sizes.

Στην παρούσα All-GPU υλοποίηση, όμως, το block size φαίνεται να αποκτά πολύ πιο σημαντικό ρόλο. Και πάλι στο 1o Configuration έχουν καλύτερες επιδόσεις τα μικρά block sizes, όμως στο 2o Configuration έχουν καλύτερες επιδόσεις, και σχεδόν με 2-3x, τα μεγάλα block sizes.

Όπως έχει σχολιαστεί παραπάνω, το 1o Configuration, απαιτεί από μόνο του μικρό block size ώστε να καταφέρουν τα σημεία να χωράνε στην L1 cache, γεγονός που αποκτά ακόμα μεγαλύτερη σημασία εδώ, καθώς εκτός των συγκρίσεων γίνεται και μια ακόμα άθροιση, στο μερικό άθροισμα των clusters, στο τέλος.

Αντίθετα, στο 2o Configuration, η συμπεριφορά ως προς το block size άλλαξε, από το να είναι σχεδόν ανεξάρτητη, στο να εξαρτάται ισχυρά από αυτό. Αυτό συμβαίνει, επειδή μεγαλύτερο block size, σημαίνει πως στο 1o επίπεδο reduction, δηλαδή εντός του thread block, αθροίζονται περισσότερα σημεία, οπότε απομένει λίγοτερο reduction να γίνει στο 2o επίπεδο, δηλαδή μεταξύ των thread blocks, που είναι και το πλεόν χρονοβόρο καθώς απαιτεί atomic ενημερώσεις στην global memory.

Με άλλα λόγια, υψηλότερο block size, οδηγεί σε απαίτηση λίγοτερων ατομικών ενημερώσεων, που έχουν υψηλό χρονικό κόστος, οπότε η χρονική επίδοση βελτιώνεται σημαντικά.

Στο 2o Configuration, επειδή το πλήθος συντεταγμένων μειώνεται, χωράνε περισσότερα σημεία στην L1 cache, οπότε η L1 δεν πέζει τόσο την επίδοση όσο οι ατομικές ενημέρωσεις.

Παρατηρούμε πως αν και εκτελούμε τον ίδιο κώδικα, τα δύο Configurations ευνοούνται για διαφορετικούς λόγους και για διαφορετικά φαινόμενα από διαφορετικές τιμές block size, γεγονός που καταδεικνύει την σημασία, για ένα πρόγραμμα που τρέχει σε GPU, του κατάλληλου profiling και της μελέτης των παραμέτρων με εξειδίκευση στις διάφορες εισόδους που αναμένονται να δοθούν.

3.5.5 Καταλληλότητα του τμήματος update_centroids για GPU (Ερώτημα 3o)

Το τμήμα update_centroids δεν είναι εξαιρετικό για GPU, διότι ξεκινώντας από τα membership κάθε σημείου, πρέπει ουσιαστικά να κάνει ένα “μη κανονικό” reduction, δηλαδή πρωτικά μια λειτουργία Shuffling (σαν μια λειτουργία Gather στο πλαίσιο των Scatter/Gather αναφορών). Με άλλα λόγια, συνεχόμενα σημεία, πρέπει να αντιστοιχηθούν σε διαφορετικά clusters. Ακόμα και αν δεν υπήρχε η καθυστέρηση που δίνουν τα atomics, οι προσβάσεις που γίνονται από τα 32 νήματα ενός warp δεν είναι διπλάνες, οπότε από μόνο του αυτό θα δώσει καθυστέρησεις. Η χρήση ατομικών ενημερώσεων χειροτερεύει ακόμα παραπάνω την επίδοση.

Επιπρόσθετα, προκειμένου να μπορέσει να γίνει η διαιρέση με το cluster size χρειάζεται όλες οι αθροίσεις από όλα τα thread blocks να έχουν ολοκληρωθεί, οπότε χρειάζεται ένα barrier, που επειδή δεν υποστηρίζεται, η CPU αναλαμβάνει τον συγχρονισμό αυτό, εκκινώντας διαφορετικό πυρήνα.

Πάντως, το τμήμα `update_centroids` που μπαίνει στον 2o πυρήνα, δηλαδή η διαιρέση με το cluster size, που ονομάζεται βέβαια `update_centroids` στον κώδικα όμως περιλαμβάνει μόνο αυτό το κόμματι του `update_centroids` είναι ένας σχετικά καλός πυρήνας για GPU. Τα στοιχεία επεξεργάζονται όλα ανεξάρτητα, δεν υπάρχει ανάγκη για συγχρονισμό (από τον ίδιο τον αλγόριθμο), και όλα τα στοιχεία χρειάζεται να εκτελέσουν ακριβώς την ίδια λειτουργία. Το μόνο ίσως αρνητικό χαρακτηριστικό του πυρήνα αυτού είναι πως ενώ πολλά στοιχεία (συντεταγμένες ενός cluster) διαιρούνται με το ίδιο cluster size, στην μνήμη γειτονικές είναι συντεταγμένες διαφορετικών clusters, οπότε προκειμένου να έχουμε γειτονικές φορτώσεις από τις global memory των συντεταγμένων των clusters, καταλήγουμε σε 32 διαφορετικές προσβάσεις μνήμης από τα 32 ήματα του warp. Αυτό, ώστοσο, δεν προκύπτει από τον αλγόριθμο, αλλά από τον τρόπο που οργανώσαμε τις δομές δεδομένων στην μνήμη, φυσικά όμως απολαμβάνοντας άλλα πλεονεκτήματα στον 1o πυρήνα. Επίσης ένα δεύτερο αρνητικό του πυρήνα είναι πως έχει διαιρέση, που είναι γενικά μια χρονοβόρα πράξη, αν και αυτό μάλλον δύσκολα μπορεί να αποφευχθεί ή να θεωρηθεί ως χαρακτηριστικό που ευνοεί κάποια άλλη αρχιτεκτονική.

3.5.6 Διαφορά επίδοσης μεταξύ των δύο Configurations (Ερώτημα 4o)

Όπως αναφέρθηκε παραπάνω, στην “Επίδραση του block size (Ερώτημα 2o)” αλλά και στην πρώτη σύγκριση που έγινε στην παράγραφο 3.4.3, το 1o Configuration ευνοείται σημαντικά περισσότερο από λιγότερο ανταγωνισμό για την χωρητικότητα της L1 cache σε έναν Streaming Multiprocessor, ενώ στο 2o Configuration αυτός ο παράγοντας γίνεται πολύ λιγότερο σημαντικός (λόγω της μείωσης του αριθμού συντεταγμένων, χωράνε, όπως σχολιάστηκε, περισσότερα σημεία στην L1 cache, οπότε μειώνεται ο ανταγωνισμός για την χωρητικότητά της) και αναδεικνύεται ως πιο σημαντικός παράγοντας οι μείωση των χρονοβόρων ατομικών ενημερώσεων, που γίνεται με την αύξηση του block size (ώστε να γίνει περισσότερο reduction στο 1o επίπεδο reduction, δηλαδή εντός του thread block, αντί του 2ou επιπέδου reduction, δηλαδή μεταξύ των thread blocks).

Ο συνδυασμός των δύο παραγόντων εν τέλει ευνοεί σημαντικά την επίδοση του 2ou Configuration. Πιο συγκεκριμένα, το 2o Configuration έναντι του 1ou, καταφέρνει να δουλέψει σε μεγάλα block sizes χωρίς να χάσει σε επίδοση, ενώ το 1o Configuration και αυτό πράγματι κερδίζει από τον ίδιο παράγοντα στα μεγάλα block sizes, όμως σε αυτά η επίδοση του γίνεται σημαντικά χειρότερη λόγω της L1. Για να γίνω σαφέστερος, το 2o Configuration κερδίζει διπλά, καθώς δεν είναι ότι αντισταθμίζει την απώλεια της L1 με τις μειωμένες ενημερώσεις (αυτό προκύπτει διότι στην Shared Memory υλοποίηση η επίδοση ήταν περίπου ανεξάρτητη του block size), αλλά αντίθετα, κερδίζει και την καλή επίδοση λόγω L1 και την καλή επίδοση λόγω μειωμένων ατομικών ενημερώσεων.

Το συμπέρασμα αυτό μπορεί να γίνει καλύτερα αντιληπτό συγκρίνοντας επιτάχυνση σε σχέση με την Shared Memory υλοποιησή. Το 2o Configuration, δίνει περίπου 20x φορές μεγαλύτερη επιτάχυνση, σε σχέση με την Shared Memory υλοποίηση, ενώ το 1o Configuration μόλις 4x. Η διαφορά αυτή προκύπτει, επειδή το 1o Configuration αναγκάζεται να υποστεί το μικρό block size και τις αυξημενές ατομικές ενημερώσεις.

3.6 All-GPU Delta Reduction Υλοποίηση (BONUS 2)

3.6.1 Περιγραφή της υλοποίησης

Στην υλοποίηση αυτό σκοπός είναι να υλοποιήσουμε πιο αποδοτικά, σε μορφή δέντρου, το άθροισμα των delta. Αναφερόμαστε στο 1o επίπεδο reduction, δηλαδή εντός του thread block. Θα υλοποιήσουμε αποδοτικά την άθροιση αυτή με χρήση των **Warp-Level Primitives**.

Το μέγιστο block size είναι $1024 = 32 \cdot 32$, δηλαδή το μέγιστο thread block αποτελείται από 32 warps, που για καλή μας τυχή είναι ίσο με το μέγεθος του ενός warp. Έτσι, σπάω το reduction σε δύο φάσεις. Στην πρώτη φάση, αθροίζω τις τιμές από τα νήματα ενός warp και ένα νήμα από κάθε warp έχει το άθροισμα του warp. Στην δεύτερη φάση, μαζεύω τις 32 τιμές από κάθε warp, στα νήματα ενός μόνο warp και αθροίζω εκ νέου τις τιμές. Έτσι τελικά προκύπτει το συνολικό άθροισμα όλου του thread block.

Στην γενική περίπτωση, αρκεί να μπορούμε να υπολογίσουμε το άθροισμα τιμών ενός warp. Θα μπορούσαμε να το κάνουμε με tree reduction με τις ενδιάμεσες τιμές στην shared memory. Ωστόσο, η NVIDIA GPUs, υποστηρίζουν warp-level primitives, όπου μπορούμε να μεταφέρουμε τιμές αυστηρά εντός του ίδιου warp από ένα νήμα σε ένα άλλο (αυτός είναι και ο λόγος που κάνουμε σε δύο φάσεις το reduction εντός του thread block, για να εκμεταλλευτούμε αυτές τις instructions).

Για την 1η φάση, τα πράγματα είναι ακόμα πιο ευνοϊκά από αυτό, καθώς κάθε νήμα δίνει 1 bit (0 ή 1 ανάλογα αν ισχύει ή όχι μια συνθήκη – βάζουμε επιπλέον την συνθήκη να είναι `&& tid<numObjjs` ώστε να αθροίζουμε από όλο το thread block). Έτσι, αρκεί να μεταφέρουμε αυτά τα bits σε ένα νήμα από κάθε warp και έπειτα αυτό να μετρήσει πλήθος 1 στις τιμές που θα λάβει. Αυτό γίνεται αποδοτικά με την χρήση του warp-level primitive `_ballot_sync`. Σε αυτό όλα τα νήματα του warp “ψηφίζουν 0 ή 1” και όλα τα νήματα του warp πάρουν μια 32-bit τιμή με τις “ψήφους”. Αρκεί συνεπώς, να μετρήσουμε, με την `_popc` το πλήθος των 1 στην τιμή αυτή.

Στην 2η φάση, αρχικά μεταφέρουμε τις τιμές από ένα νήμα κάθε warp στα 32 νήματα ενός μόνο warp που θα αναλάβει την άθροιση. Η μεταφορά γίνεται μέσω της shared memory.

Έπειτα, αυτή την φορά, τα νήματα του warp έχουν μεγαλύτερες τιμές από 1 bit, οπότε θα κάνουμε ένα κανονικό tree reduction εντός όμως του warp. Αντί να επικοινωνούμε τις ενδιάμεσες τιμές μέσω της shared memory, χρησιμοποιούμε το warp-level primitive `_shfl_down_sync` προκειμένου να μεταφέρουμε σε κάθε επίπεδο του δέντρου τις τιμές στα σωστά νήματα. Ακριβέστερα, το reduction ώστε από τα 32 νήματα ενός warp να αποκτήσει το 0-οστό του το άθροισμα των 32 τιμών γίνεται ως εξής:

```
1 res = <VALUE PER THREAD>
2 for (int i = 16; i > 0; i /= 2)
3     res += __shfl_down_sync(0xffffffff, res, i); // "sending" to the "lower half" each time.
4 // the 0th thread of the warp now has the sum of the 32 threads of the warp.
```

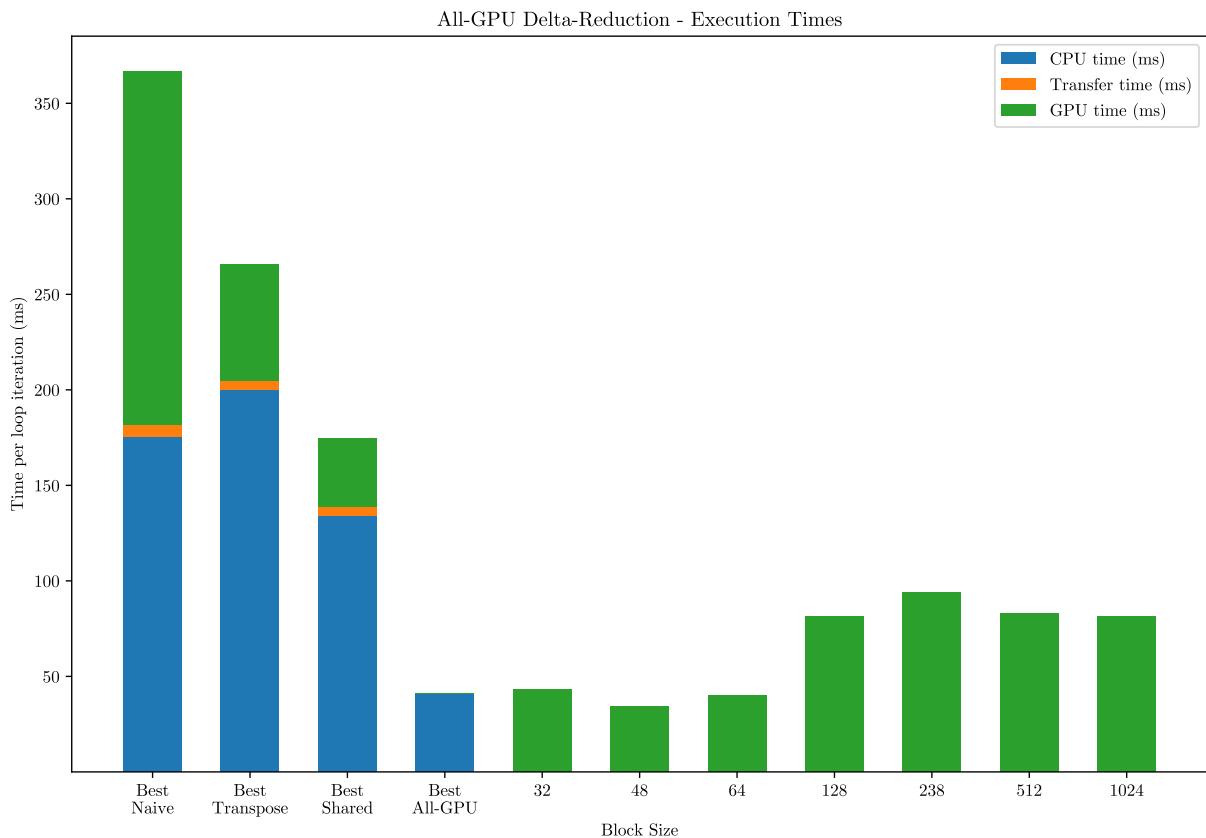
Ο τρόπος αυτός για reduction, σε 2 φάσεις, πρώτα εντός κάθε warp, και μεταξύ ξανά εντός ενός warp, είναι πιο αποδοτικός σε σχέση με ένα απλό tree reduction, διότι χρειάζεται μια μόνο επικοινωνία μέσω shared memory από την 1η φάση στην 2η φάση, η υπόλοιπη επικοινωνία γίνεται κατά την εκτέλεση με warp-level primitives (και μεταφέρονται κατά την ίδια την εκτέλεση, καθώς κάθε εντολή είναι ουσιαστικά μια διανυσματική εντολή για όλο το warp.)

3.6.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (Ερώτημα 1o – α' μέρος)

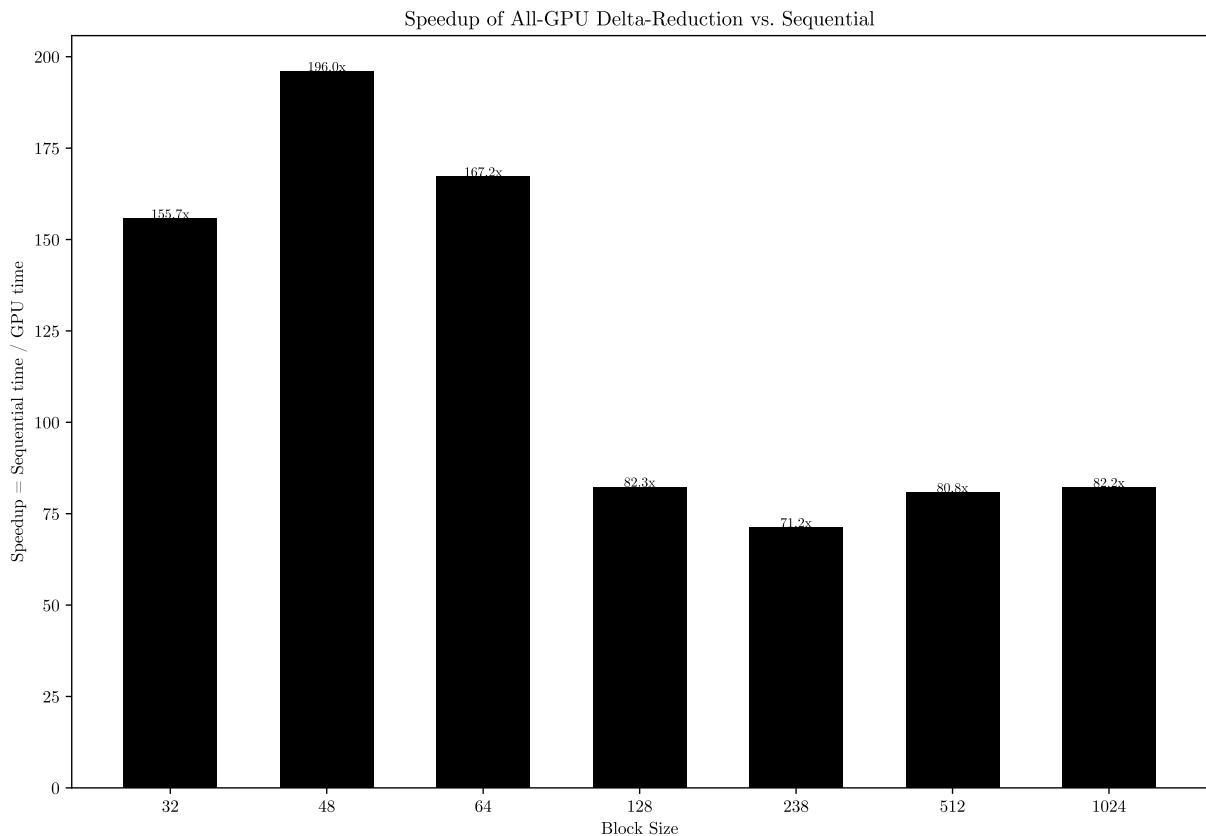
Πραγματοποιήθηκαν μετρήσεις για την All-GPU Delta Reduction υλοποίηση για το 1o configuration {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}, για το 2o configuration {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10} καθώς και για `block size` = {32, 48, 64, 128, 238, 512, 1024}.

Τα αποτελέσματα φαίνονται διαγραμματικά:

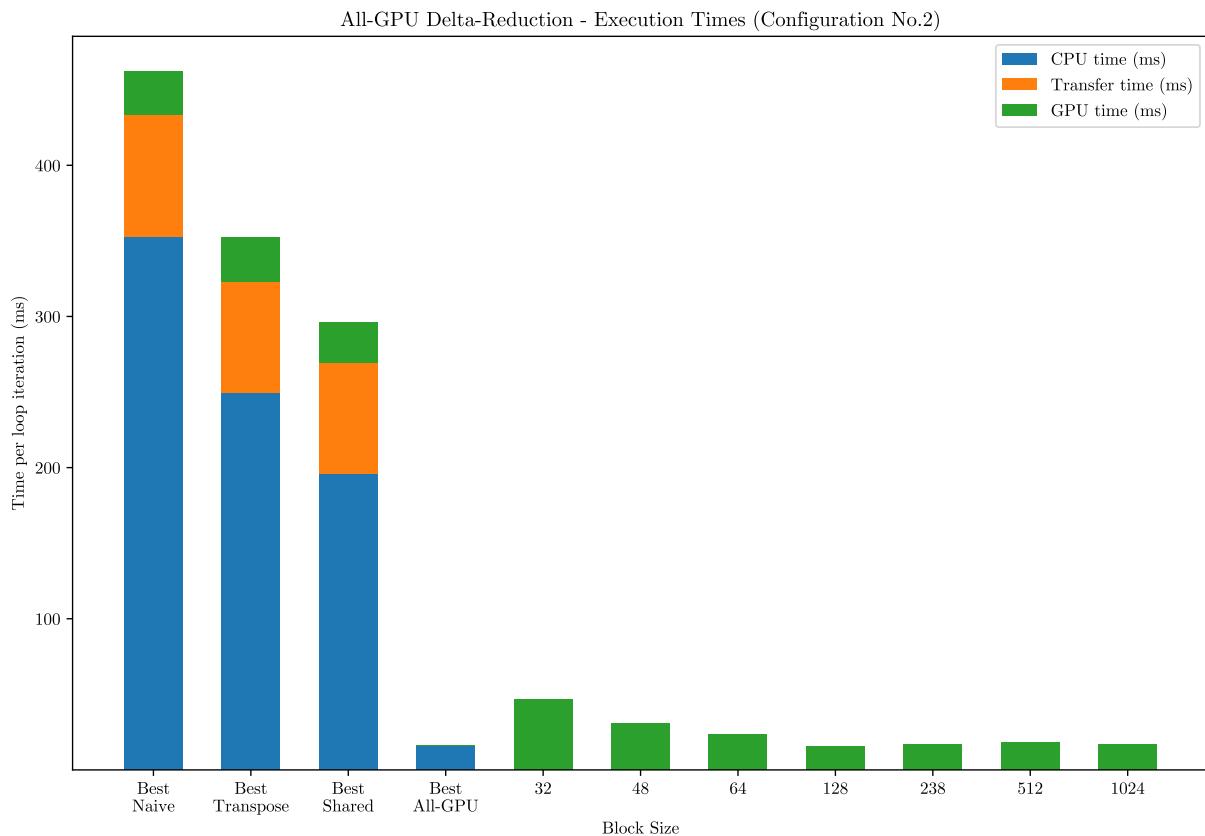
- Για το 1o Configuration στο σχήμα 69 για τον χρόνο εκτέλεσης, και στο σχήμα 70 για την επιτάχυνση σε σχέση με την CPU.
- Για το 2o Configuration στο σχήμα 71 για τον χρόνο εκτέλεσης, και στο σχήμα 72 για την επιτάχυνση σε σχέση με την CPU.



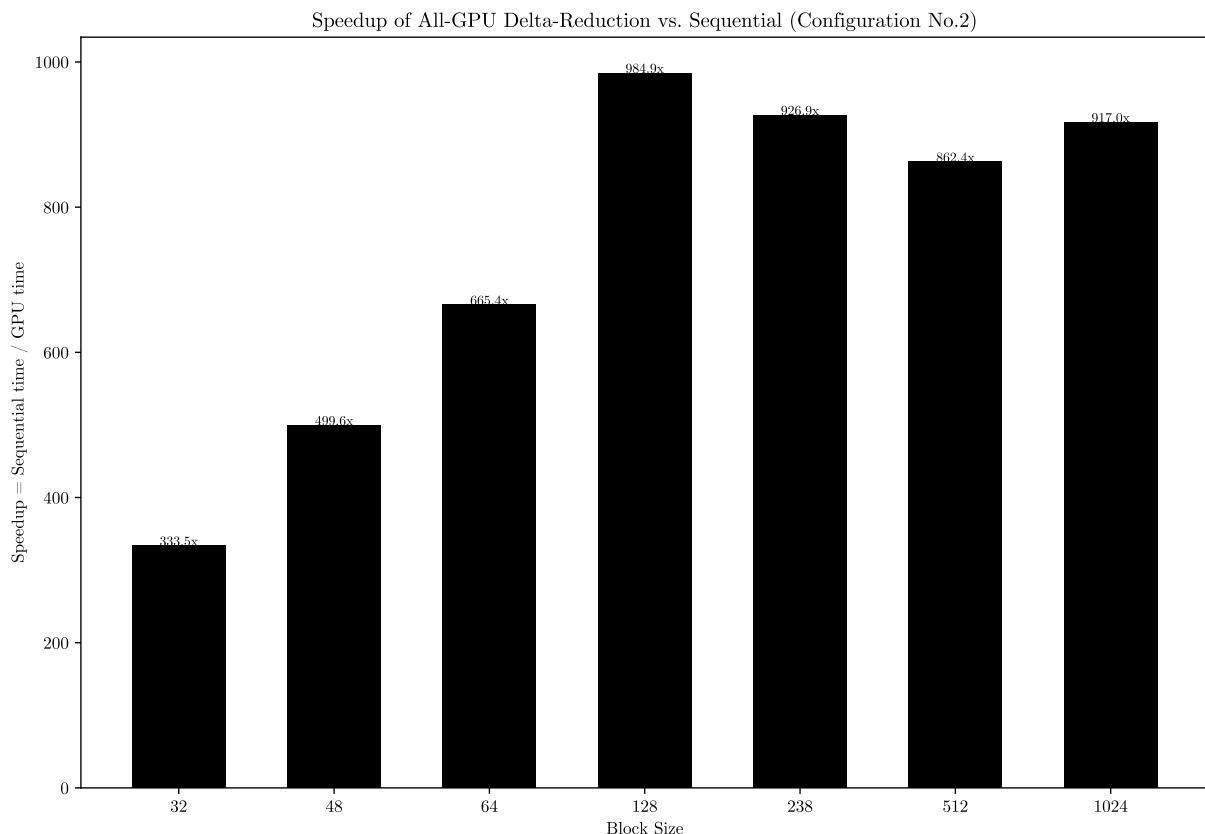
Σχήμα 69: Χρόνος εκτέλεσης All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans



Σχήμα 70: Επιτάχυνση (σε σχέση με CPU) All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans



Σχήμα 71: Χρόνος εκτέλεσης All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans (2o Configuration)



Σχήμα 72: Επιτάχυνση (σε σχέση με CPU) All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans (2o Configuration)

3.6.3 Παρατηρήσεις – Σχόλια (Ερώτημα 1ο – β' μέρος)

Παρατηρείται μια μικρή αύξηση της επιτάχυνση, σχεδόν 20%, στο 1o Configuration.

Η βελτίωση της επίδοσης συμβαίνει λόγω της εξάλειψης των ατομικών ενημερώσεων που γίνονται στην μεταβλητή delta εντός του thread block, οι οποίες είναι ισάριθμες με το block size. Αντί να γίνονται ατομικές ενημέρωσεις στην shared memory, έχουμε εντολές που εκτελούνται χωρίς καθόλου χρήση καμίας μνήμης (warp-level primitives) και επιπλέον έχουμε μια μόνο επικοινωνία μέσω της shared memory που γίνεται όμως χωρίς atomics, αλλά ο συγχρονισμός διασφαλίζεται μέσω ενός μόνο barrier (`_syncthread()`). Έτσι, αφενός μεν, γλιτώνουμε το κόστος της σειροποιήσης των ατομικών μεταβολών, που είναι και το σημαντικότερο, γλιτώνουμε όμως επίσης και το κόστος ανάγνωσης-εγγραφής στην shared memory.

Στο 2o Configuration η επίδοση δεν μεταβάλλεται σημαντικά. Αν και δεν διαθέτουμε profiling στοιχεία, φαίνεται πως αυτό θα μπορούσε να εξηγηθεί επειδή το 2o Configuration επιτυγχάνει καλές επιδόσεις σε μεγάλα block size σε αντίθεση με το 1o Configuration που επιτυγχάνει καλές επιδόσεις σε μικρά block size (τα αίτια αυτού του φαινόμενο έχει εξηγηθεί και αναλυθεί σε προηγούμενες παραγράφους). Το μεγάλο block size, δηλαδή με πολλά warps, ενδεχομένως εκθέτει περισσότερη παραλληλία, επιτρέποντας στον thread block scheduler να θέσει προς εκτέλεση άλλα warps ενόσω κάποιο warp περιμένει να του δοθεί πρόσβαση για κάποια ατομική μεταβολή, γεγονός που μάλλον δεν μπορεί να γίνει αν το thread block αποτελείται από λίγα warps. Από την άλλη πλευρά, μπορεί η επιτάχυνση που λαμβάνουμε στο 2o Configuration να μην εξαρτάται τόσο σημαντικά από τις ατομικές ενημερώσεις στην shared memory, αλλά από κάποιον άλλο παράγοντα πλέον, όπως ενδεικτικά από τις ατομικές ενημερώσεις στην global memory.

3.6.4 Επίδραση Block Size (Ερώτημα 2o)

Παρατηρούμε πως στο 1o Configuration η διαφορά ανάμεσα στο καλύτερο block size και στα υπόλοιπα γίνεται πιο έντονη. Αυτό φαίνεται να συμβαίνει, λόγω της μεγαλύτερης χρήσης shared memory (για τα ενδιάμεσα αποτελέσματα του delta), που αυξάνει ακόμα περισσότερο την πίεση για χωρητικότητα στην L1 cache.

Στο 2o Configuration η καμπύλη block size-χρόνου είναι περίπου ίδια, όμως έχει μετατοπιστεί “δεξιότερα”, προς μεγαλύτερα block size, δηλαδή επιτυγχάνουμε παρόμοιες επιδόσεις σε μεγαλύτερα block sizes. Αυτό είναι λογικό να συμβεί, διότι το δεντρικό reduction δεν έχει ανταγωνισμό για τον κοινό πόρο μιας shared μεταβλητής, που κάθε νήμα να χρειάζεται να λάβει ατομική πρόσβαση σε αυτό, οπότε κλιμακώνει πολύ καλύτερα για μεγαλύτερα block sizes. Πιο συγκεκριμένα, μάλιστα, το δεντρικό reduction, όπως το έχουμε υλοποιήσει με 2 φάσεις με warp-level primitives, έχει μια πρώτη φάση που γίνεται τελείως ανεξάρτητα ανά warp, και μια δεύτερη φάση που θέλει σταθερό χρόνο όσο και αν είναι το block size (αν το block size είναι μικρότερο από 32 warps, έχουμε θέσει τις αντίστοιχες τιμές των μη-υπαρκτών warps σε 0 στην 2η φάση, ώστε να μην χρειάζονται branches που θα καθυστερούσαν την εκτέλεση).

4 4η Άσκηση – Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης – MPI

4.1 Αλγόριθμος Kmeans

4.1.1 Περιγραφή της υλοποίησης

Τλοποιήσαμε τον αλγόριθμο Kmeans στο MPI, δηλαδή σε μοντέλο περάσματος μηνυμάτων.

Το σύνολο σημείων χωρίστηκε και ανατέθηκε ισομερώς στις επιμέρους διεργασίες.

Θέλουμε τα μέγιστο μέρος που θα πάρει κάποια εργασία να είναι το ελάχιστο δυνατό, διότι αυτό είναι που θα καθορίσει τον χρόνο εκτέλεσης. Συνεπώς το υπόλοιπο (πλήθος σημείων mod πλήθος διεργασιών) το κατανέμουμε με 1 σημείο/διεργασία.

Η διεργασία με $rank == 0$ (έστω *master* διεργασία) αναλαμβάνει την δημιουργία του dataset και την κατανομή τους στις επιμέρους διεργασίες.

Σε κάθε γύρω του αλγόριθμου όλες οι διεργασίες διαθέτουν τον τρέχοντα πίνακα των clusters. Στο τέλος κάθε γύρου με μια κλήση MPI_Allreduce υπολογίζονται σε όλες τις διεργασίες τα αθροίσματα σημείων ανά νέο cluster καθώς και το μέγεθος κάθε cluster. Στο τέλος κάθε διεργασία υπολογίζει τα νέα clusters διαιρώντας τους αριθμούς αυτούς.

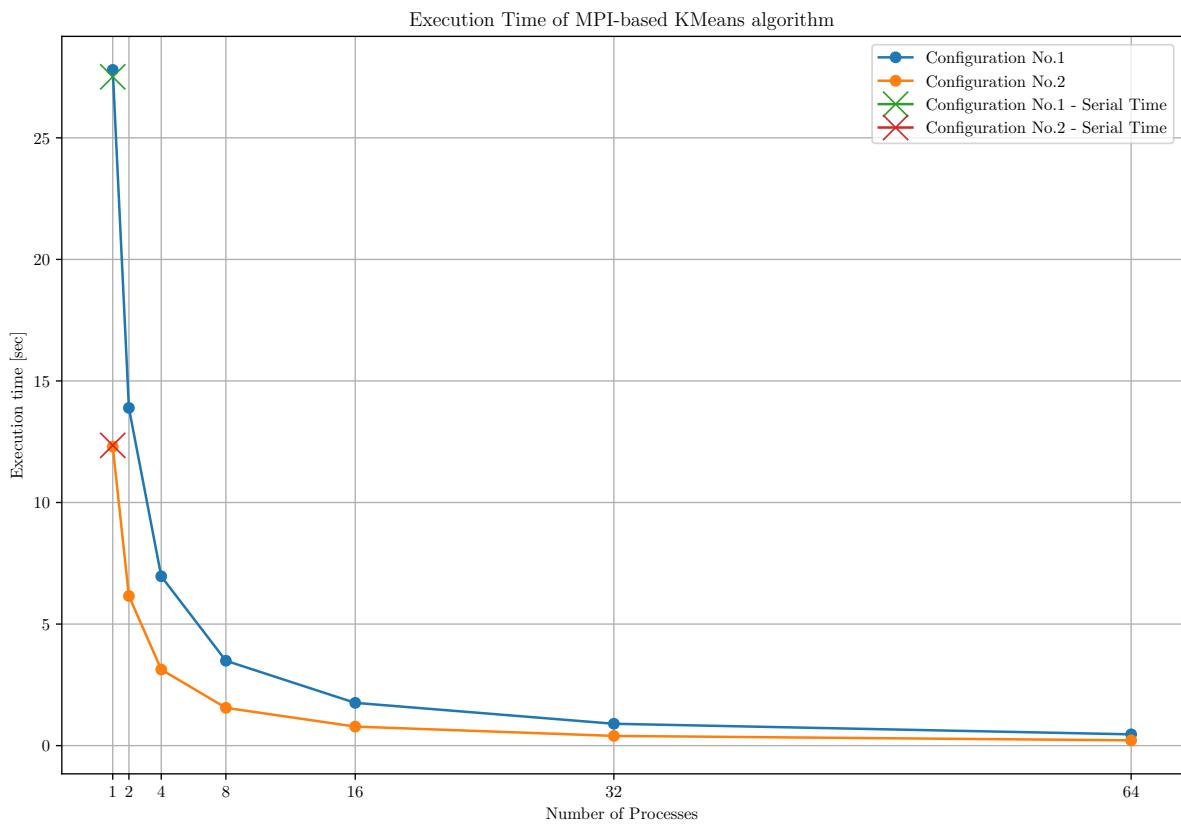
4.1.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση

Μετρήσαμε τον χρόνο εκτέλεσης για 1 ως 64 διεργασίες και ειδικότερα για ως 8 φυσικούς κόμβους με ως 8 επεξεργαστές (πυρήνες)/κόμβο και 1 διεργασία/πυρήνα.

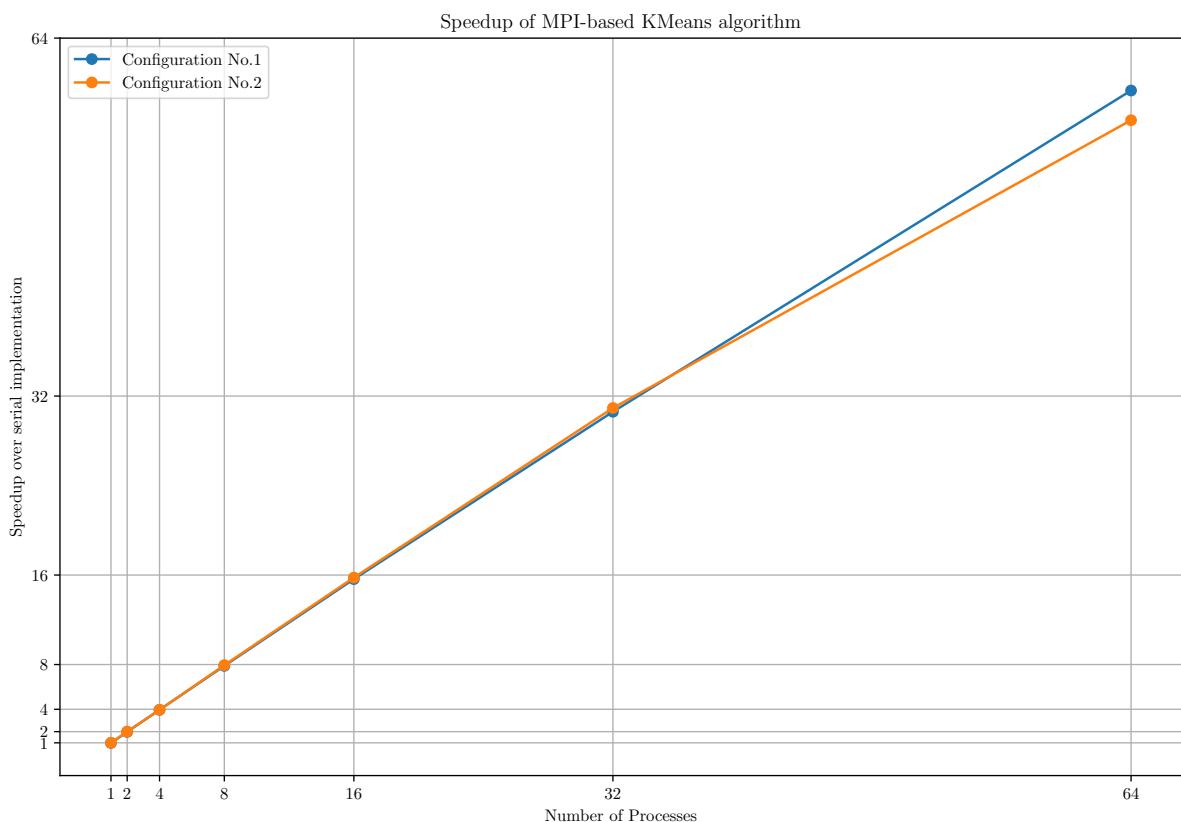
Δοκιμάσαμε δύο Configurations, τα ίδια με την 2η Άσκηση. Το 1o, που είναι και εδώ ζητούμενο, είναι το {Size, Coords, Clusters, Loops} = 256, 16, 32, 10. Το 2o, που στην 2η Άσκηση ήταν και το memory-bound που είχε τις χειρότερες επιδόσεις, είναι το {Size, Coords, Clusters, Loops} = 256, 1, 4, 10.

Τα αποτελέσματα φαίνονται για τον χρόνο εκτέλεσης στο σχήμα 73 και για την επιτάχυνση ως προς την σειριακή έκδοση στο σχήμα 74.

Σημειώνεται, πως σειριακή έκδοση νοείται η αρχική υλοποίηση χωρίς MPI και όχι εκτέλεση της MPI υλοποίησης με 1 διεργασία.



Σχήμα 73: Χρόνος εκτέλεσης MPI-based υλοποίησης αλγορίθμου KMeans ως συνάρτηση του πλήθους διεργασιών για 2 Configurations



Σχήμα 74: Επιτάχυνση ως προς την σειριακή υλοποίησης MPI-based υλοποίησης αλγορίθμου KMeans ως συνάρτηση του πλήθους διεργασιών για 2 Configurations

4.1.3 Παρατηρήσεις – Σχόλια

Παρατηρούμε πως λαμβάνουμε περίπου την μέγιστη ιδανική επιτάχυνση, δηλαδή όσο και το πλήθος των διεργασίων. Μέχρι και τις 32 διεργασίες είναι σχεδόν πλήρως ιδανική, ενώ στις 64 παρατηρείται μια μικρή μείωση της κλιμακωσμότητας.

Τα 2 Configurations φαίνεται να επιτυγχάνουν και τα δύο παρόμοια επιτάχυνση, και μάλιστα και τα δύο σχεδόν ιδανική, συμπεριφορά που είναι αρκετά διαφορετική από αυτή που είδαμε στο NUMA μηχάνημα στην 2η Άσκηση.

4.1.4 Σύγκριση με την 2η Άσκηση (υλοποίηση σε αρχιτεκτονική κοινής μνήμης) – BONUS

Παρατηρούμε πως οι χρονικές επιδόσεις, και η κλιμακωσμότητα είναι σημαντικά καλύτερες σε σχέση με την 2η Άσκηση, γεγονός που σε πρώτη φάση φαίνεται περίεργο, καθώς στην 2η Άσκηση χρησιμοποιούσαμε ένα cache-coherent μηχάνημα που επιτρέπει πολύ ταχύτερη, τόσο ως προς το latency όσο και ως προς το εύρος ζώνης, επικοινωνία μεταξύ των πυρήνων, σε σχέση με το δίκτυο μεταξύ κόμβων που χρησιμοποιείται εδώ. Παραξενεύει, ιδιαίτερα, η επίδοση του 2ου Configuration, που εδώ προκύπτει σχεδόν ίδια με του 1ου, ενώ στο cache-coherent μηχάνημα προέκυπτε σημαντικά διαφορετική.

Η διαφορά αποδίδεται στο εύρος ζώνης της μνήμης. Στο cache-coherent μηχάνημα, είχαμε, βέβαια, γρήγορη επικοινωνία, όμως όλες οι διεργασίες χρησιμοποιούσαν την ίδια φυσική μνήμη, που μπορούσε να δώσει αθροιστικά σε όλα τα νήματα σταθερό εύρος ζώνης. Έτσι, το εύρος ζώνης μνήμης/νήμα ήταν αντιστρόφως ανάλογο του πλήθους νημάτων και έτσι πολύ σύντομα το πρόγραμμα φρασσόταν από το εύρος ζώνης μνήμης, γινόταν δηλαδή memory-bound.

Αντίθετα, εδώ κάθε (φυσικός) κόμβος έχει χωριστή φυσική μνήμη, και έτσι καθώς αυξάνεται το πλήθος κόμβων το εύρος ζώνης/διεργασία παραμένει σταθερό. Έτσι, το υπολογιστικό τμήμα του αλγόριθμου παραμένει compute-bound. Μάλιστα, στην παρούσα υλοποίηση, όλα τα νήματα έχουν χωριστό αντίγραφο του πίνακα clusters, ακόμα και όταν βρίσκονται στον ίδιο φυσικό κόμβο. Αν και δεν επαληθεύτηκε ότι συμβαίνει, πιθανώς, στον ίδιο φυσικό κόμβο, οι πίνακες clusters χωριστών διεργασιών να αντιστοιχήσουν σε χωριστά memory banks, και έτσι τελικά, αν το σύστημα μνήμης το υποστηρίζει, να μπορέσαν οι διεργασίες ακόμα και εντός του ίδιου κόμβου να απολέψουν μεγαλύτερο συνολικό ευρός ζώνης από ότι στην υλοποίηση με τον κοινό πίνακα για όλες τις διεργασίες του κόμβου. (ακόμα και στο NUMA μηχάνημα είχαμε 1 αντίτυπο του πίνακα clusters/NUMA κόμβο.)

Με άλλα λόγια, επαληθεύουμε το συμπέρασμα πως ένα κατανεμημένο σύστημα, παρόλο που υποφέρει από αυξημένα κόστη επικοινωνίας, είναι πολύ πιο κλιμακώσιμο από ένα κεντρικοποιημένο σύστημα, καθώς όλοι οι πόροι κατανείμονται χωριστά, και δεν προκύπτει ανταγωνισμός για κάποιον κοινό πόρο, όπως συμβαίνει στις αρχιτεκτονικές κοινής μνήμης.

Αξίζει, πάντως, να σημειωθεί, πως ακόμα και στην MPI υλοποίηση, στις 64 διεργασίες, όπου έχουμε το μέγιστο πλήθος διεργασιών/φυσικό κόμβο, παρατηρείται στο 2o Configuration μια μικρή επιβράδυνση έναντι του 1ou Configuration, καθώς στο σημείο αυτό ο ανταγωνισμός για εύρος ζώνης μνήμης αρχίζει να αυξάνεται και το πρόγραμμα βαίνει προς την περιοχή του memory-bound.

4.2 Αριθμητική Επίλυση της Εξίσωσης Θερμότητας σε 2Δ ορθογώνιο χωρίο (με 3 μεθόδους)

4.2.1 Περιγραφή της υλοποίησης των 3 αριθμητικών μεθόδων

Χρησιμοποιήθηκαν 3 διαφορετικές μέθοδοι αριθμητικής επίλυσης, η μέθοδος Jacobi, η μέθοδος Gauss-Seidel Successive Over-Relaxation (GaussSeidelSOR) και η μέθοδος Red-Black Gauss-Seidel Successive Over-Relaxation (RedBlackSOR).

Για την υλοποίηση χωρίζουμε το $x \times y$ grid σε ορθογώνια μεγέθους $Px \times Py$ που εκχωρούνται σε κάθε διεργασία. Σε κάθε επανάληψη, οι διεργασίες εκτελούν υπολογισμούς εντός του ορθογωνίου τους και στο τέλος της επανάληψης επικοινωνούν με τους γείτονες που έχουν τα 4 γειτονικά ορθογώνια με αυτές, τα σύνορα των ορθογωνίων τους (τις τιμές στις 4 πλευρές), με τρόπο ανάλογο της κατά περίπτωση μεθόδου όπως θα δούμε.

Στην μέθοδο Jacobi, στο τέλος κάθε επανάληψης η κάθε διεργασία απλά ανταλλάσσει με τους 4 γείτονές της τις τιμές στις 4 πλευρές του ορθογωνίου της.

Στην μέθοδο GaussSeidelSOR, η διεργασία στην αρχή της i -οστής επανάληψης περιμένει να της σταλούν τιμές για την τρέχουσα επανάληψη από την βόρεια και δυτική διεργασία και στο τέλος της επανάληψης στέλνει τις τιμές που υπολογίσει και στους 4 γείτονες ενώ επίσης παραλαμβάνει από τον νότιο και ανατολικό γείτονα της τιμές της επανάληψης. Αυτό έχει ως αποτέλεσμα, η διεργασία να έχει στην αρχή της επανάληψης, τις τιμές του βόρειου και δυτικού γείτονα για την ίδια επανάληψη και του νότιου και ανατολικού γείτονα για την προηγούμενη επανάληψη, όπως ακριβώς απαιτείται. Σημειώνεται, πως με αυτή την υλοποίηση, πρωτικά έχουμε υλοποίησει κάτι σαν pipeline, καθώς στην αρχή μπορεί να τρέξει μόνο η τέρμα πάνω-αριστερά διεργασία, αμέσως μετά, αυτή και οι διεργασίες της

“2ης αντιδιαγωνιού”, κ.ο.κ. Αυτό σημαίνει, πως υπάρχει ένα χρονικό διάστημα κάποιων επαναλήψεων (πλήθους όσο το άθροισμα μεγάλης και μικρής πλευράς του grid διεργασιών - 2) στην αρχή, και αντίστοιχα στο τέλος, που δεν έχουμε πλήρη παραλληλοποίηση, και άρα που δεν έχουμε πλήρη επιτάχυνση. Ωστόσο, αν το πλήθος εκτελούμενων επαναλήψεων είναι πολύ μεγαλύτερο από αυτό το πλήθος επαναλήψεων η καθυστέρηση αυτή θα είναι αμελητέα. Εδώ για το μέγιστο πλήθος διεργασιών, 64, έχουμε μέγιστη πλευρά 8, και τρέχουμε 256 επαναλήψεις, οπότε δεν έχουμε πλήρη παραλληλοποίηση σε κλάσμα $\frac{28}{256} = 10.9\%$ των επαναλήψεων. Με σύντομους υπολογισμούς, η μέγιστη επιτάχυνση που αναμένουμε, αν σε κάθε επανάληψη έχουμε την μέγιστη δυνατή επιτάχυνση (θεωρώντας ότι το σύστημα δουλεύει πρακτικά σε $256 + 14 + 14$ γύρους σταθερού χρόνου, όπου υπολογίζουν οι διεργασίες που μπορούν), είναι:

$$\frac{1}{256 + 14 + 14} (2 \cdot (1 + (1 + 2) + (1 + 2 + 3) + \dots + (1 + \dots + 8) + (1 + \dots + 8 + 7) + (1 + \dots + 8 + \dots + 1)) + 64 \cdot 254) = \\ \frac{1}{272} (2 \cdot 512 + 16256) = 60.84 \times \quad (7)$$

Η απώλεια χρόνου συνεπώς λόγω του αρχικού και τελικού διαστήματος με μειωμένη παραλληλία δεν αξιολογείται ως σημαντική για την τελική επιτάχυνση.

Στην **μέθοδο RedBlackSOR**, η κάθε επανάληψη χωρίζεται σε 2 φάσεις, την φάση Red και την φάση Black όπου γίνονται οι ενημερώσεις των Red και Black στοιχείων αντίστοιχα. Η επικοινωνία είναι παρόμοια με την μέθοδο Jacobi (δεν υπάρχει το πρόβλημα με το pipeline δηλαδή). Μετά την φάση Red επικοινωνούνται με τους γείτονες μόνο τα Red στοιχεία (στοιχεία των 4 συνόρων με stride 2) και αντίστοιχα μετά την φάση Black μόνο τα Black στοιχεία (στοιχεία των 4 συνόρων με stride 2 αλλά διαφορετικό ± 1 offset σε σχέση με τα στοιχεία Red). Συνολικά, συνεπώς, σε κάθε επανάληψη επικοινωνούνται μόνο μια φορά τα στοιχεία των 4 συνόρων, όπως και προηγουμένως. Ωστόσο, το γεγονός ότι σε μια χρονική στιγμή χρειάζεται να σταλούν μόνο τα μισά στοιχεία δημιουργεί το μισό φορτίο στο δίκτυο διασύνδεσης, γεγονός που μπορεί με την σειρά του να βελτιώσει τις επιδόσεις. Επιπλέον, σε αντίθεση με την μέθοδο GaussSeidelSOR, από την εκκίνηση του αλγορίθμου μέχρι και το τέλος του, όλες οι διεργασίες μπορούν συνεχώς να εκτελούν χρήσιμους υπολογισμούς.

4.2.2 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση (για σταθερό πλήθος επαναλήψεων – χωρίς ελέγχο σύγκλισης)

Μετρήσαμε τον χρόνο εκτέλεσης για 1 ως 64 διεργασίες και ειδικότερα για ως 8 φυσικούς κόμβους με ως 8 επεξεργαστές (πυρήνες)/κόμβο και 1 διεργασία/πυρήνα. Δοκιμάστηκαν 3 μεγέθη grids, τα 2048×2048 , 4096×4096 και 6144×6144 για κάθε μία από τις μεθόδους.

Στο σενάριο αυτό εκτελούμε χωρίς έλεγχο σύγκλισης, δηλαδή για σταθερό πλήθος $T = 256$ επαναλήψεις.

Τα αποτελέσματα επιτάχυνσης (τόσο αναφορικά με τον συνολικό χρόνο εκτέλεσης όσο και αναφορικά αποκλειστικά με τον χρόνο υπολογισμών – δηλαδή χωρίς τον χρόνο επικοινωνίας) ως προς την αντίστοιχη σειριακή υλοποίηση (της κατά περίπτωση μεθόδου) ως συνάρτηση του πλήθους διεργασιών φαίνονται:

1. Για την μέθοδο **Jacobi**:

- (α') Για το grid 2048×2048 στο σχήμα 75.
- (β') Για το grid 4096×4096 στο σχήμα 76.
- (γ') Για το grid 6144×6144 στο σχήμα 77.

2. Για την μέθοδο **GaussSeidelSOR**:

- (α') Για το grid 2048×2048 στο σχήμα 78.
- (β') Για το grid 4096×4096 στο σχήμα 79.
- (γ') Για το grid 6144×6144 στο σχήμα 80.

3. Για την μέθοδο **RedBlackSOR**:

- (α') Για το grid 2048×2048 στο σχήμα 81.
- (β') Για το grid 4096×4096 στο σχήμα 82.
- (γ') Για το grid 6144×6144 στο σχήμα 83.

Επιπλέον συγχριτικά μεταξύ τους οι χρόνοι και οι επιταχύνσεις των 3 μεθόδων φαίνονται:

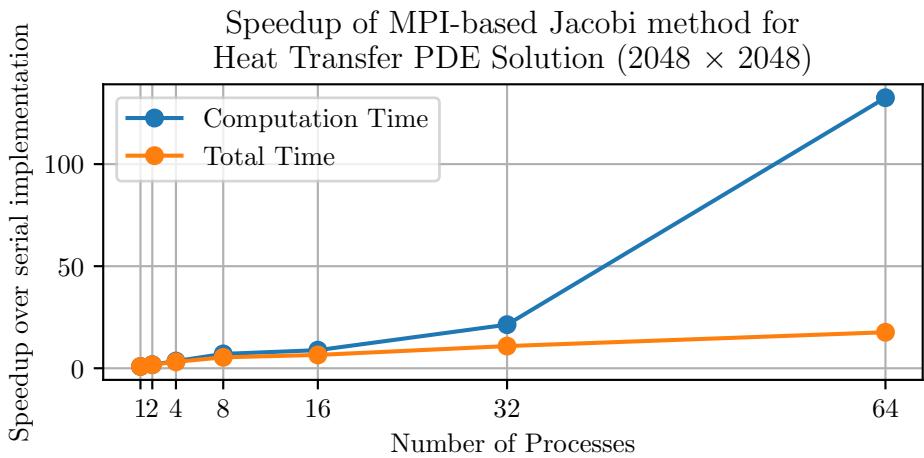
1. Για το grid 2048×2048 στο σχήμα 85 για την επιτάχυνση και στο σχήμα 84 για τον ολικό χρόνο εκτέλεσης.

2. Για το grid 4096×4096 στο σχήμα [87](#) για την επιτάχυνση και στο σχήμα [86](#) για τον ολικό χρόνο εκτέλεσης.

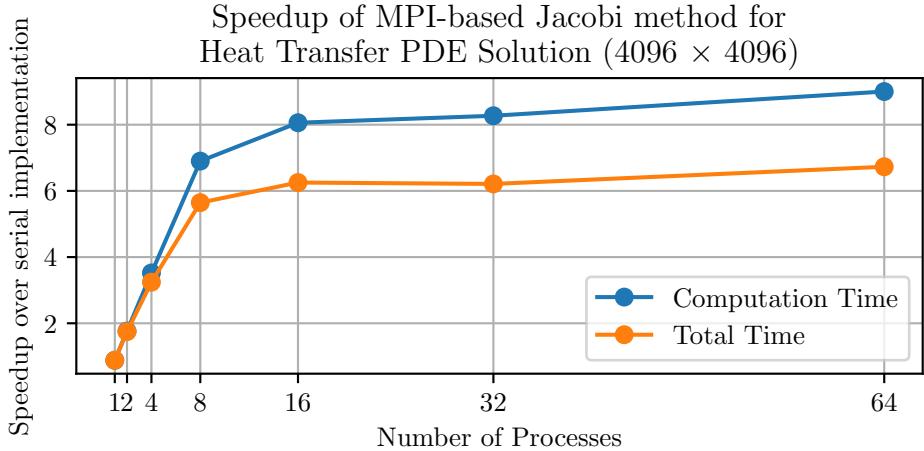
3. Για το grid 6144×6144 στο σχήμα [89](#) για την επιτάχυνση και στο σχήμα [88](#) για τον ολικό χρόνο εκτέλεσης.

Τέλος τα ζητούμενα της εκφώνησης διαγράμματα με μπάρες για τους χρόνους εκτέλεσης φαίνονται:

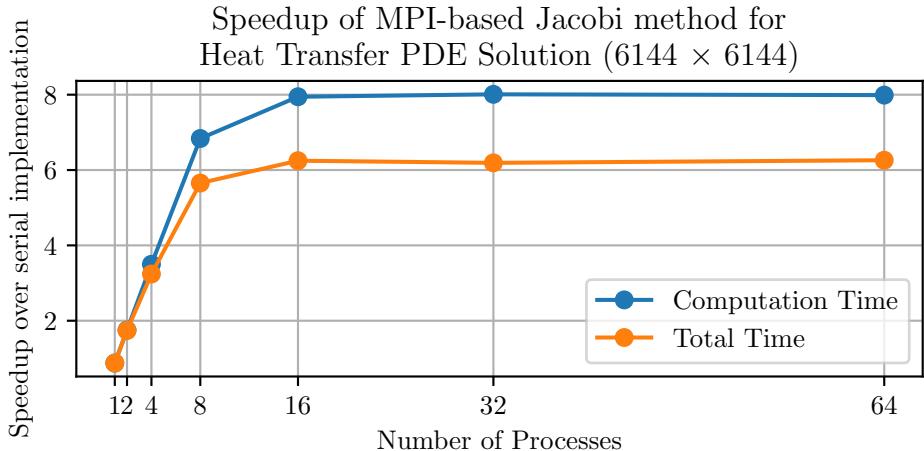
1. Για το grid 2048×2048 στο σχήμα [90](#).
2. Για το grid 4096×4096 στο σχήμα [91](#).
3. Για το grid 6144×6144 στο σχήμα [92](#).



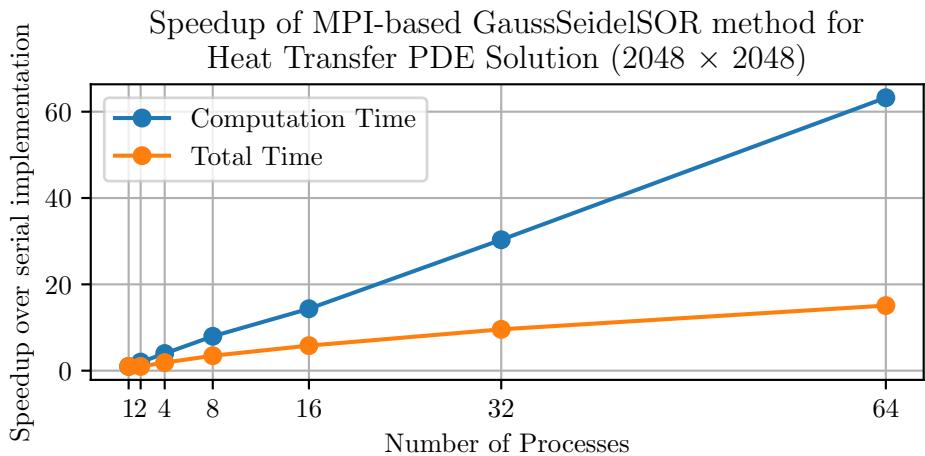
Σχήμα 75: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου Jacobi ως συνάρτηση του πλήθους διεργασιών σε grid 2048×2048



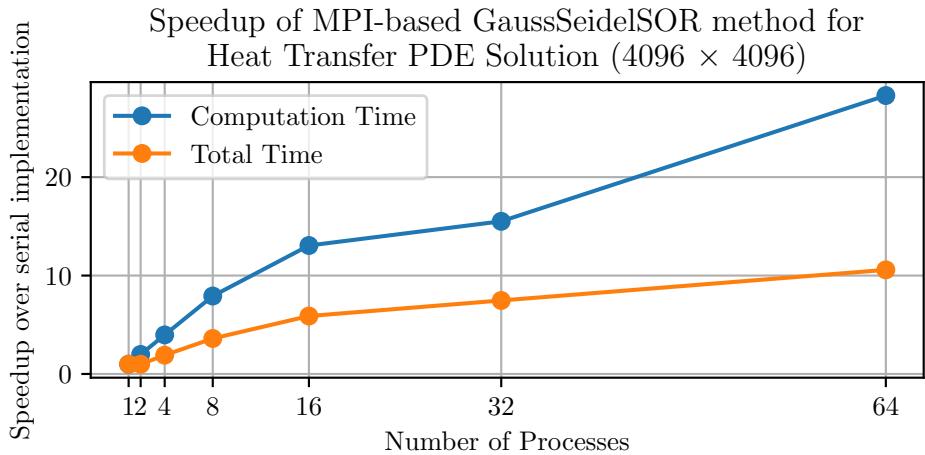
Σχήμα 76: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου Jacobi ως συνάρτηση του πλήθους διεργασιών σε grid 4096×4096



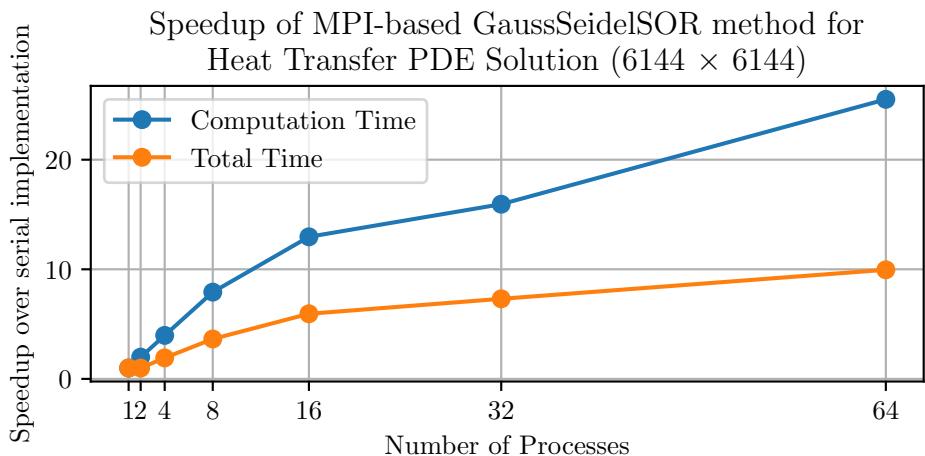
Σχήμα 77: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου Jacobi ως συνάρτηση του πλήθους διεργασιών σε grid 6144×6144



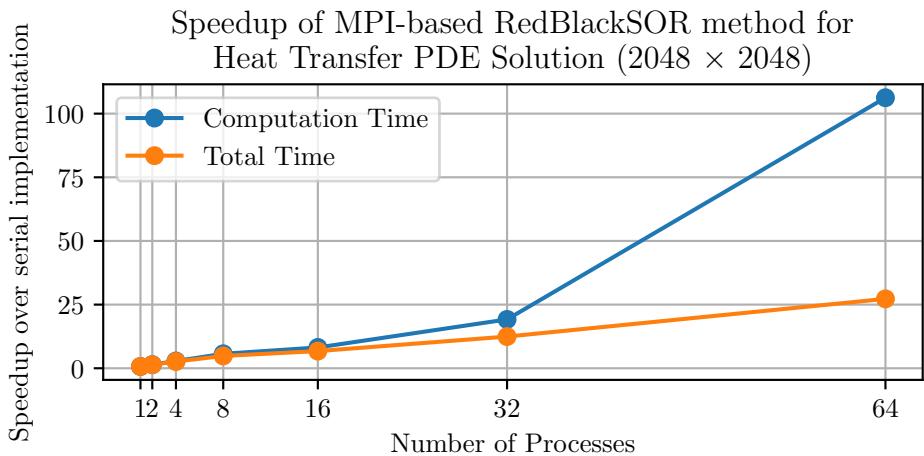
Σχήμα 78: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 2048×2048



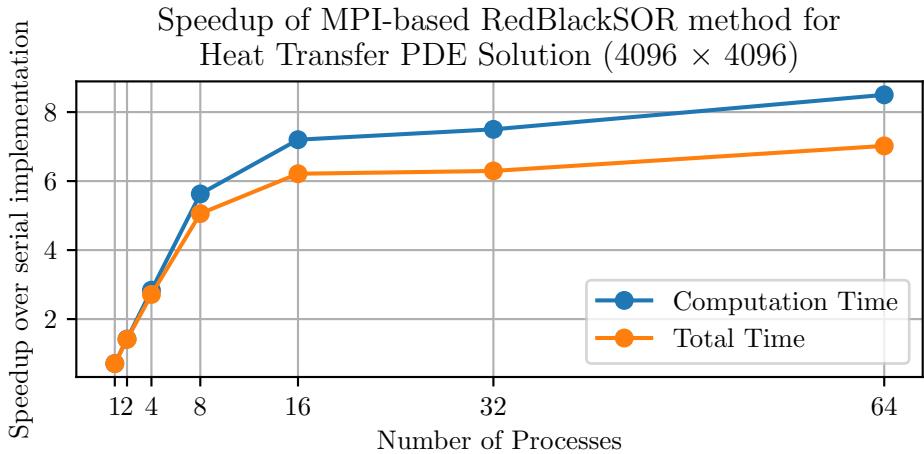
Σχήμα 79: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 4096×4096



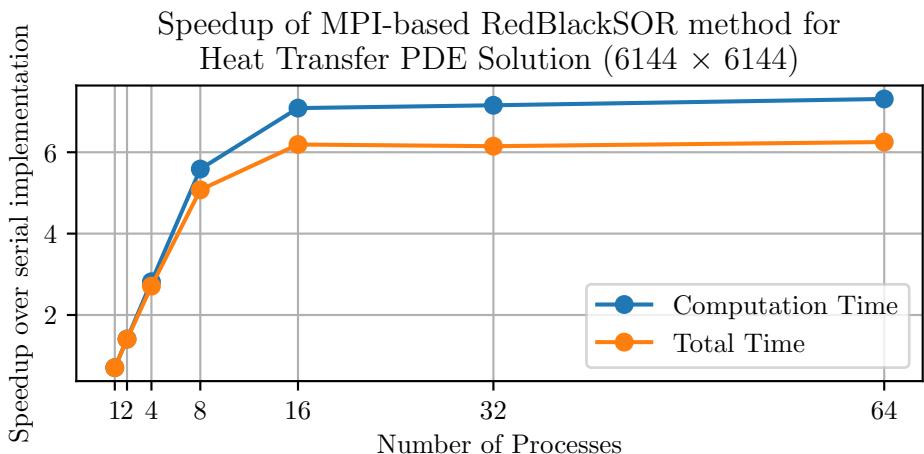
Σχήμα 80: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 6144×6144



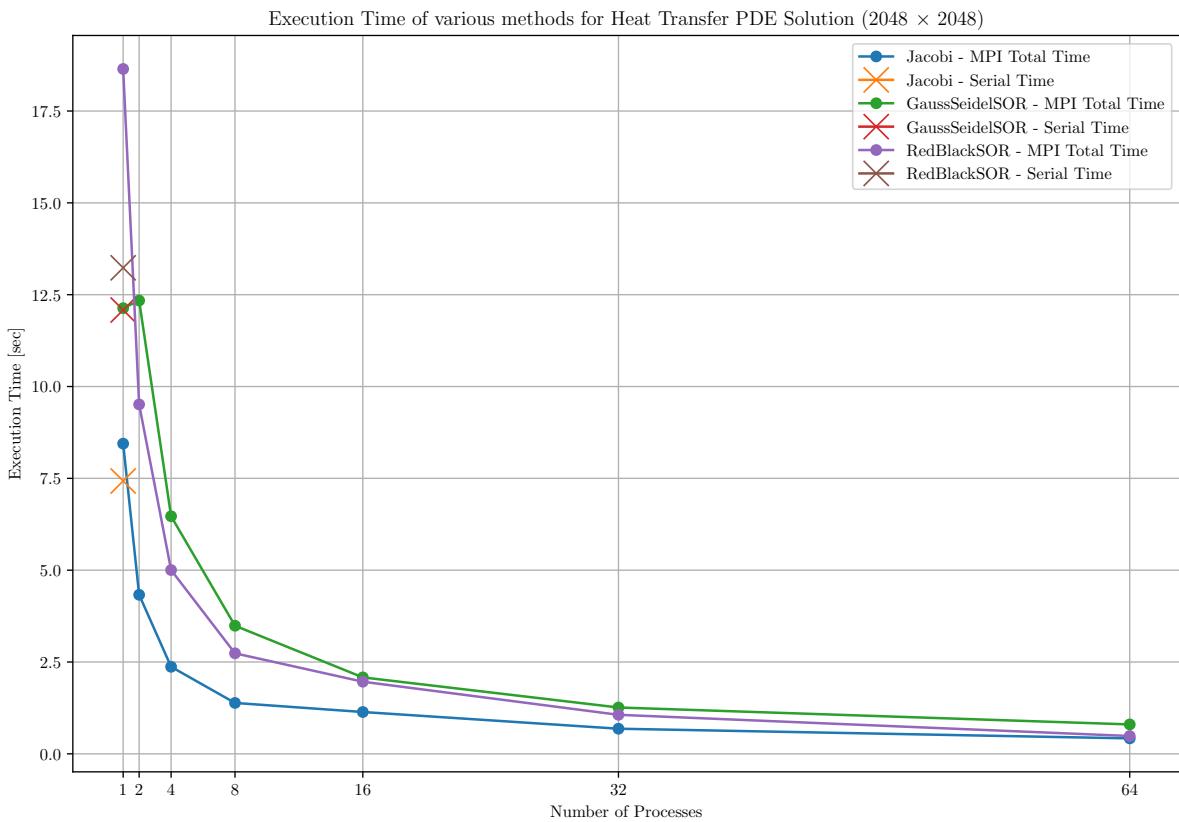
Σχήμα 81: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 2048×2048



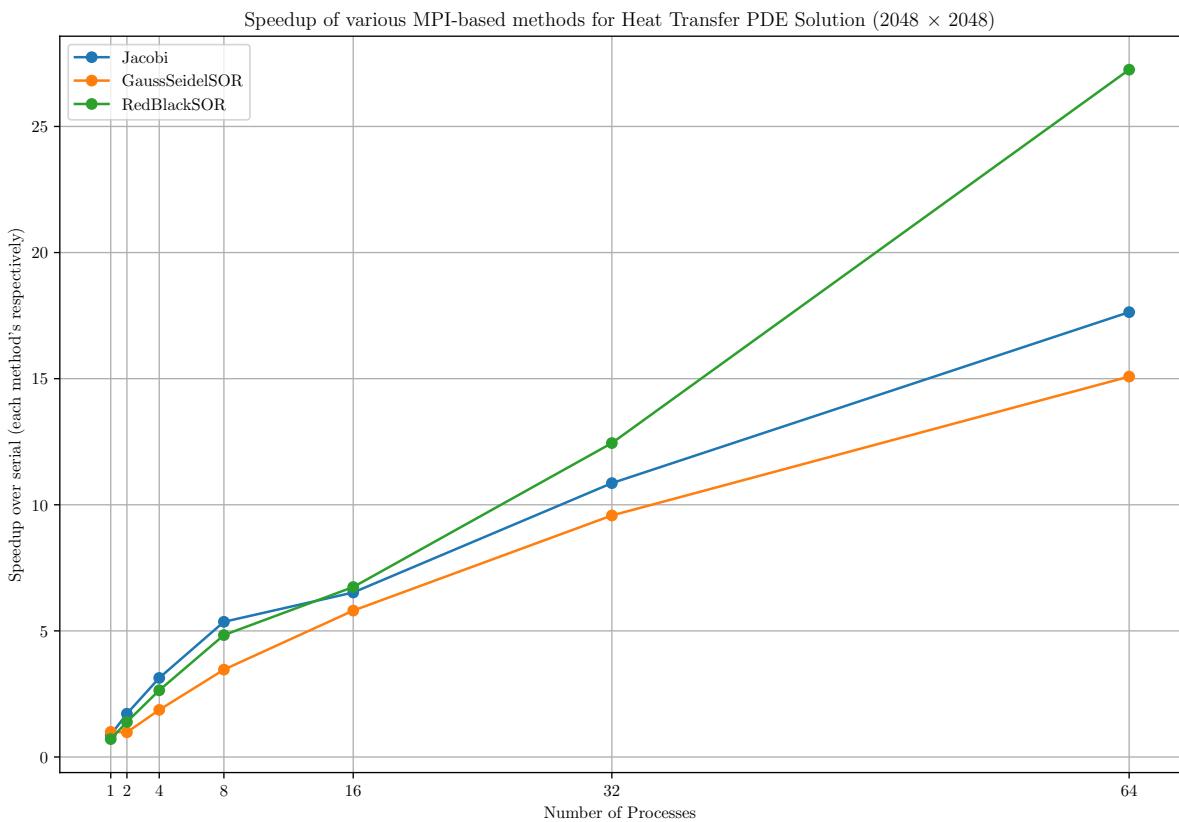
Σχήμα 82: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 4096×4096



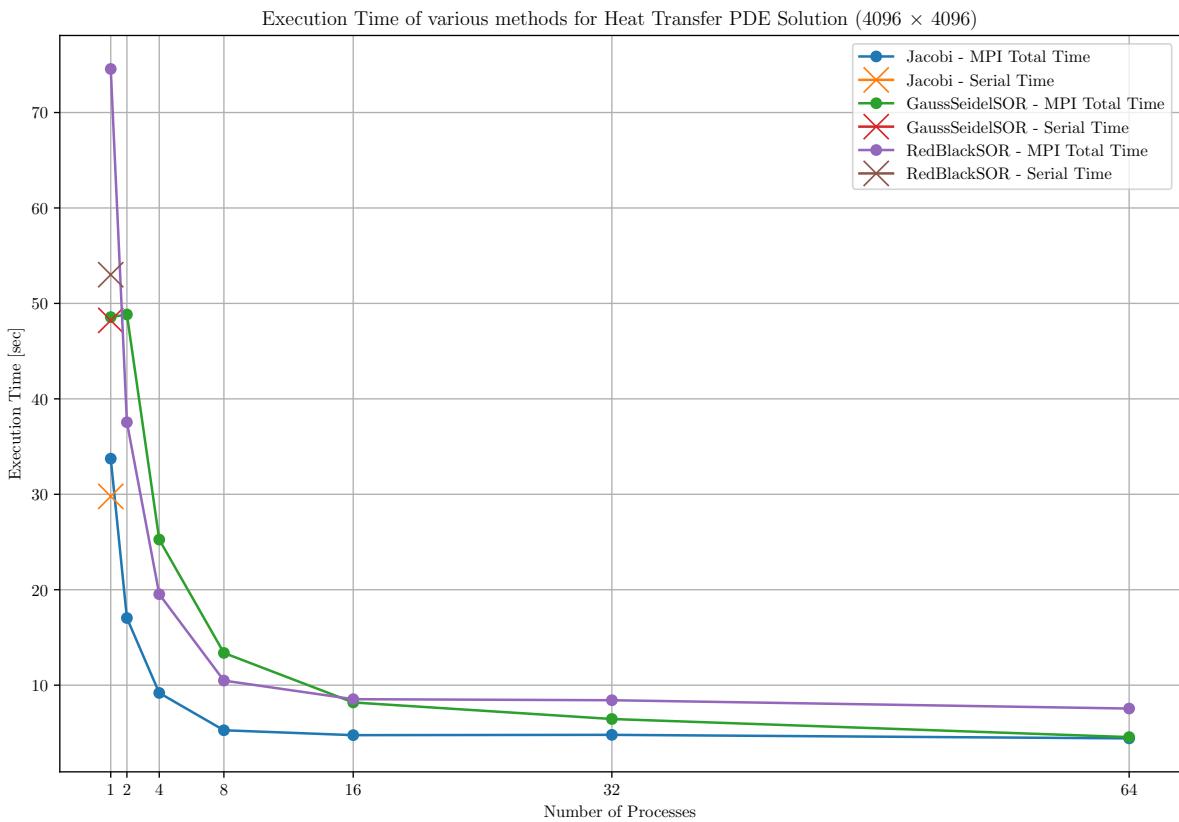
Σχήμα 83: Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 6144×6144



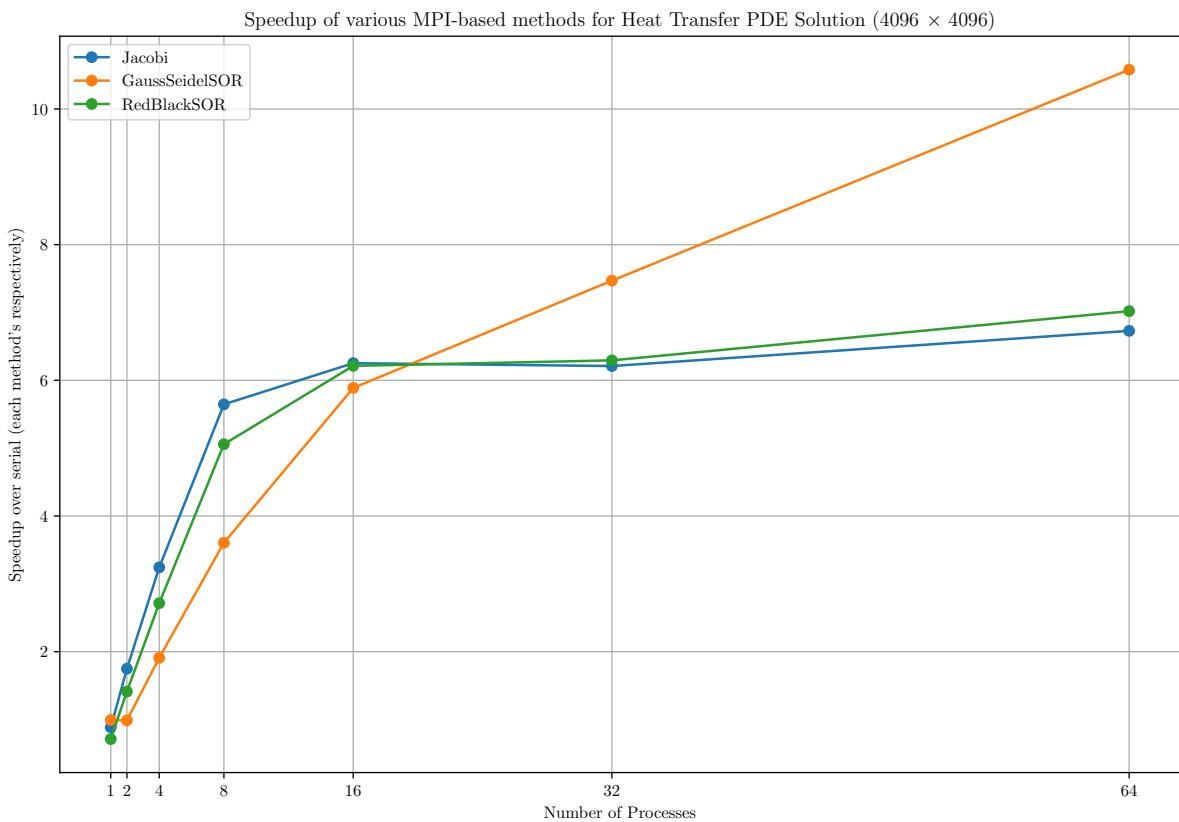
Σχήμα 84: Συγκριτική απεικόνιση χρόνων εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 2048×2048 ως συνάρτηση του πλήθους διεργασιών



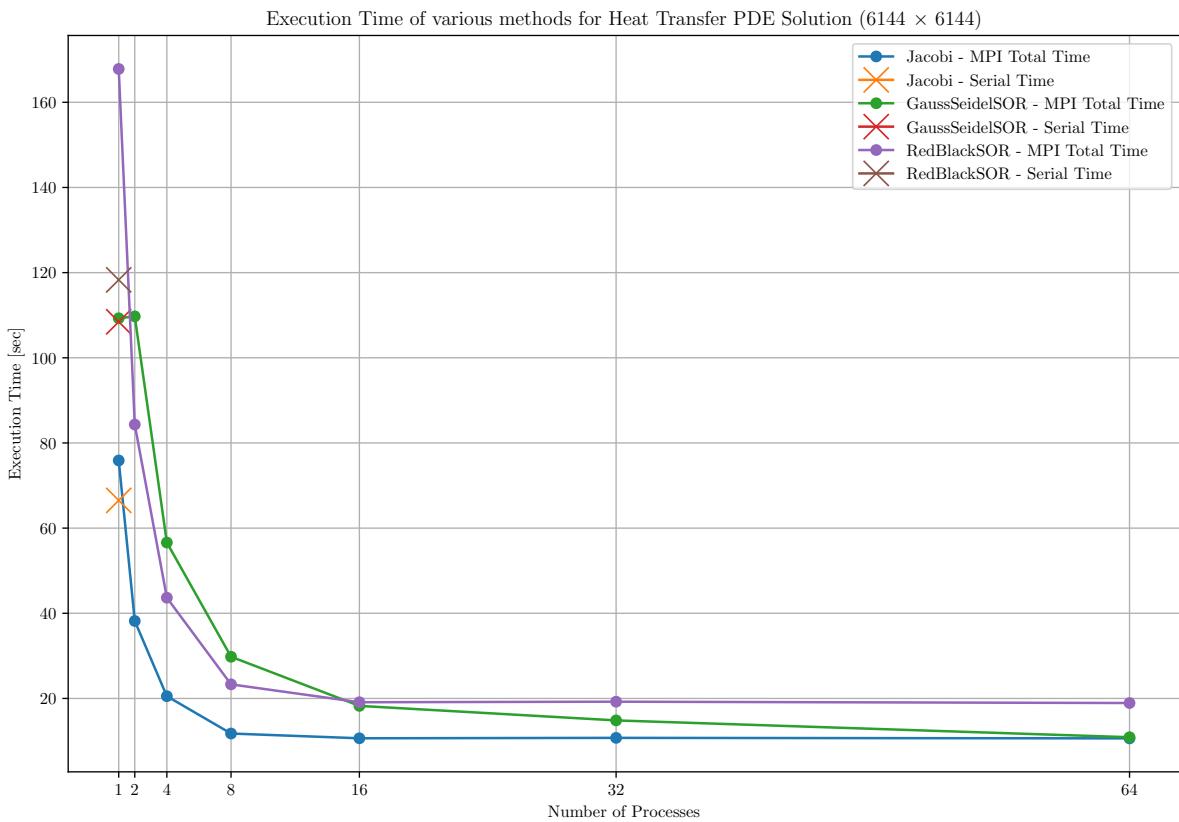
Σχήμα 85: Συγκριτική απεικόνιση επιταχύνσεων (ολικού χρόνου εκτέλεσης) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 2048×2048 ως συνάρτηση του πλήθους διεργασιών



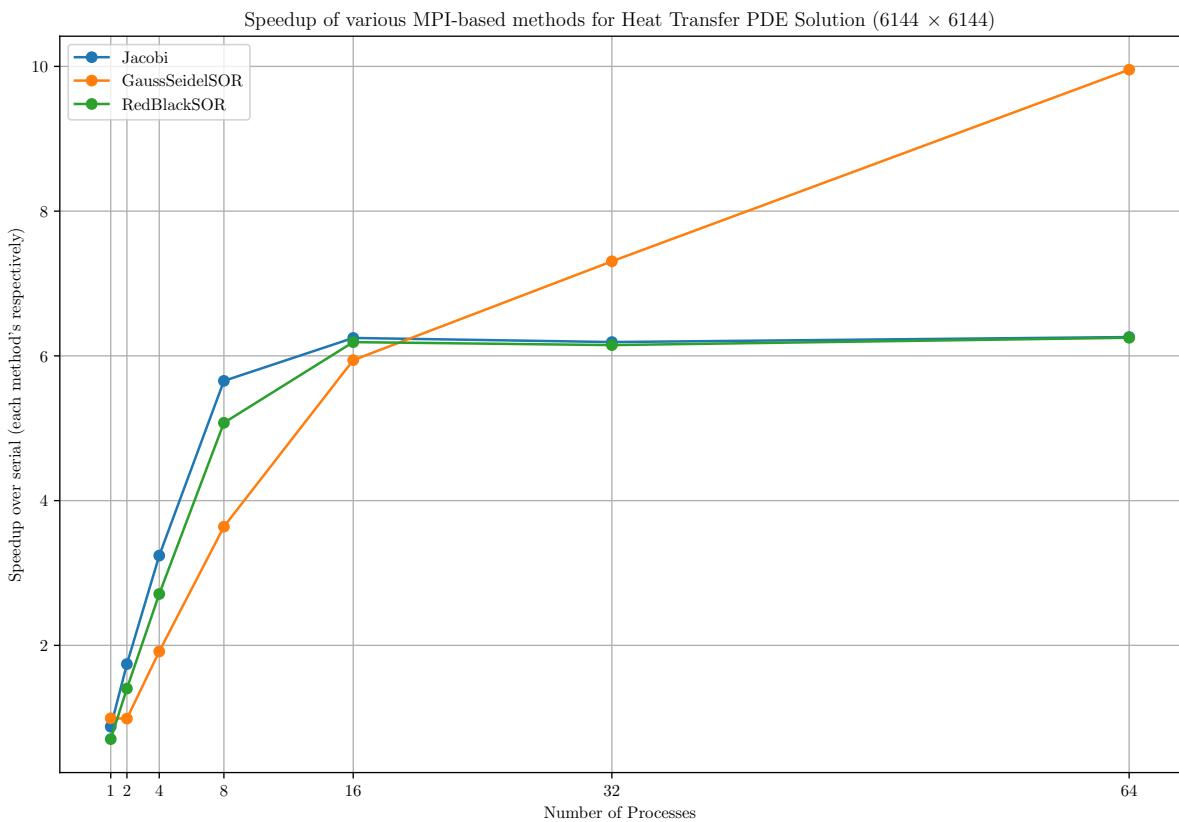
Σχήμα 86: Συγκριτική απεικόνιση χρόνων εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 4096×4096 ως συνάρτηση του πλήθους διεργασιών



Σχήμα 87: Συγκριτική απεικόνιση επιταχύνσεων (ολικού χρόνου εκτέλεσης) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 4096×4096 ως συνάρτηση του πλήθους διεργασιών

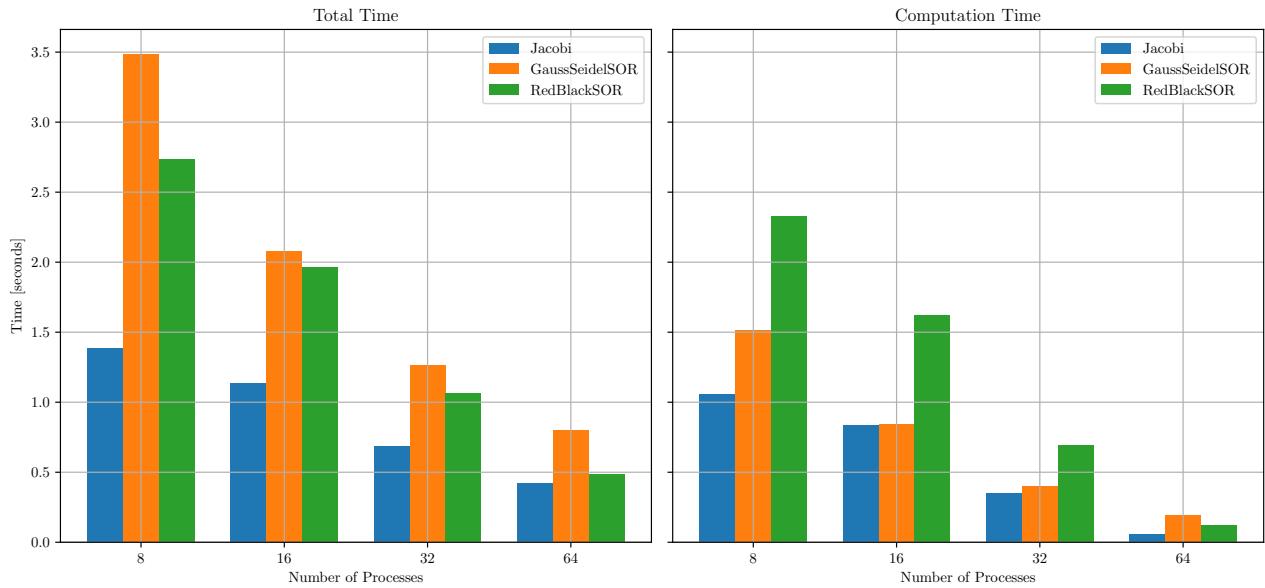


Σχήμα 88: Συγκριτική απεικόνιση χρόνων εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 6144×6144 ως συνάρτηση του πλήθους διεργασιών



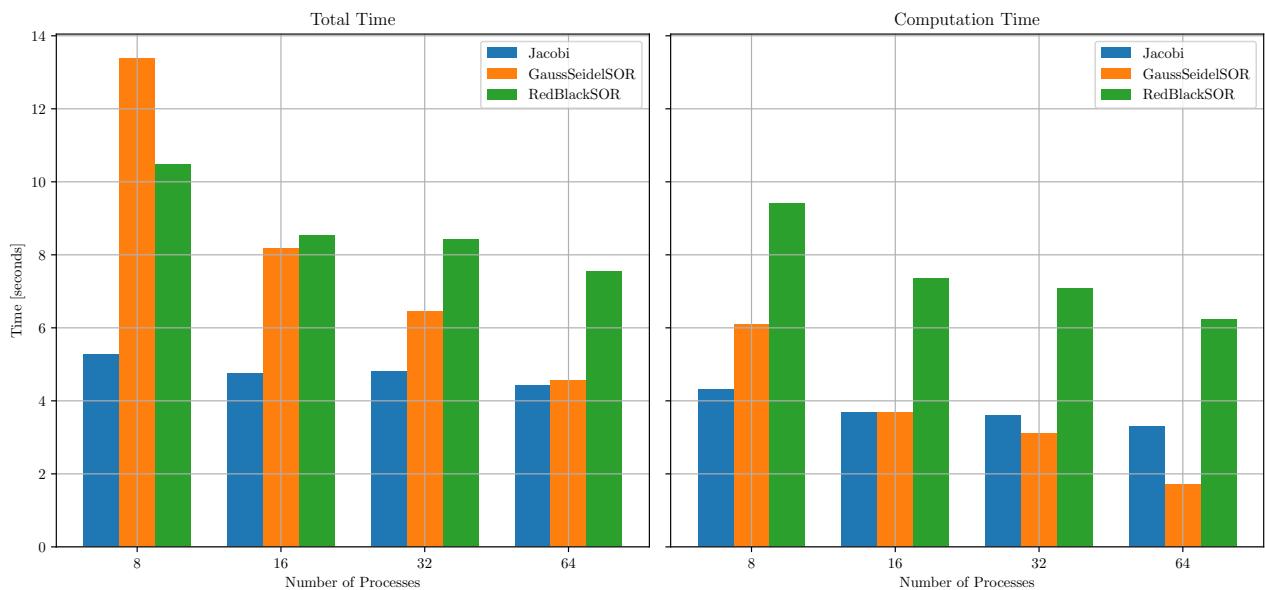
Σχήμα 89: Συγκριτική απεικόνιση επιταχύνσεων (ολικού χρόνου εκτέλεσης) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 6144×6144 ως συνάρτηση του πλήθους διεργασιών

Execution Time of various methods for Heat Transfer PDE Solution (2048×2048)

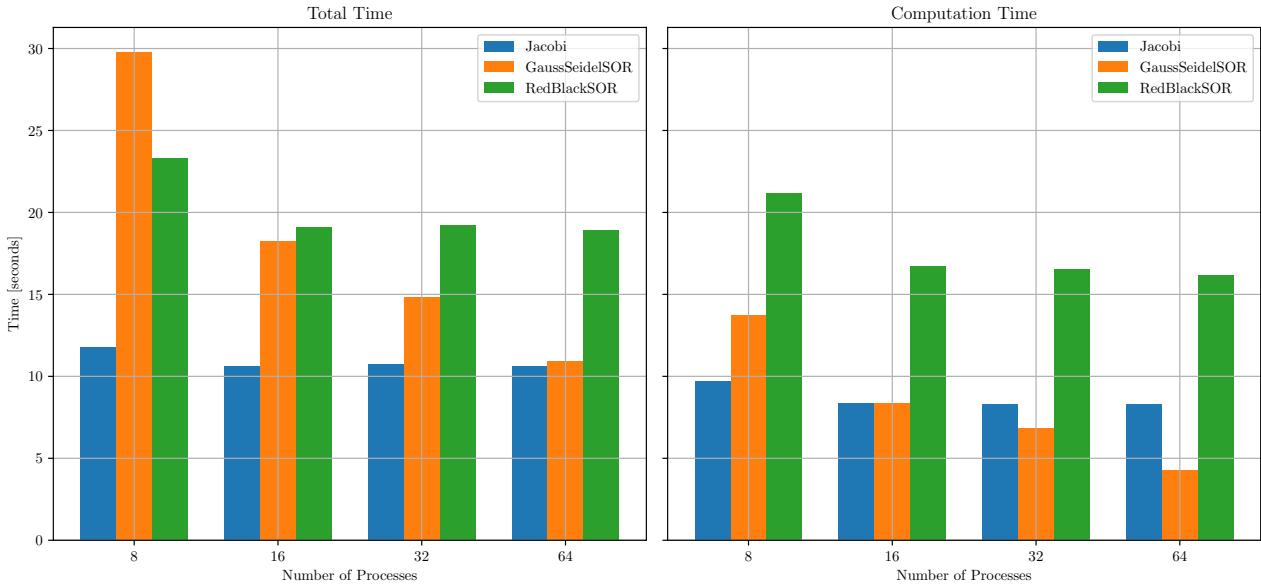


Σχήμα 90: Συγκριτική απεικόνιση χρόνων εκτέλεσης σε διάγραμμα με μπάρες (ζητούμενο από την εκφώνηση διάγραμμα) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 2048×2048

Execution Time of various methods for Heat Transfer PDE Solution (4096×4096)



Σχήμα 91: Συγκριτική απεικόνιση χρόνων εκτέλεσης σε διάγραμμα με μπάρες (ζητούμενο από την εκφώνηση διάγραμμα) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 4096×4096

Execution Time of various methods for Heat Transfer PDE Solution (6144×6144)

Σχήμα 92: Συγκριτική απεικόνιση χρόνων εκτέλεσης σε διάγραμμα με μπάρες (ζητούμενο από την εκφώνηση διάγραμμα) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 6144×6144

4.2.3 Παρατηρήσεις – σχόλια

Σε πρώτο επίπεδο, σχολιάζουμε την **κλιμακωσιμότητα** του ολικού χρόνου εκτέλεσης, αρχικά συνολικά για όλες τις μεθόδους. Παρατηρούμε πως λαμβάνουμε επιτάχυνση κοντά στην ιδανική, ως και τις 8 διεργασίες, η κλιμακωσιμότητα μειώνεται λίγο στις 16 διεργασίες, και για περισσότερες από 16 εργασίες γίνεται σταθερή και σχεδόν ίση με των 16 διεργασιών, δηλαδή δεν υπάρχει κλιμάκωση πέραν των 16 διεργασιών. Το φαινόμενο αυτό, προκύπτει λόγω του εύρους ζώνης διαύλου μνήμης. Μέχρι τις 8 διεργασίες, για κάθε διεργασία που εισάγουμε, εισάγουμε και έναν νέο κόμβο, με 1 διεργασία/κόμβο, που φέρει και ανεξάρτητα την δική του ανεξάρτητη φυσική μνήμη. Από τις 8 διεργασίες και πάνω, διατηρούμε σταθερό πλήθος κόμβο, και αυξάνουμε τις διεργασίες/κόμβο, χρησιμοποιώντας περισσότερους πυρήνες/κόμβο. Αυξάνοντας όμως τους πυρήνες/κόμβο, διαμοιράζουμε και το εύρος ζώνης μνήμης του κόμβου σε αυτούς. Τα προγράμματα εδώ, για αρχή εποπτικά, φαίνονται να είναι memory-bound, καθώς σχεδόν για κάθε έναν αριθμό που φορτώνεται από την μνήμη γίνεται κατά μέσο περίπου μόλις μια πράξη. Προκειμένου να βεβαιωθούμε πως αυτό συμβαίνει και όχι κάποιο άλλο φαίνομενο π.χ. λόγω της επικοινωνίας, **υλοποιήσαμε σε OpenMP** την μέθοδο Jacobi και επαληθεύτηκε πως σε έναν κόμβο λαμβάνουμε για 2 φυσικούς πυρήνες, μια μικρή επιτάχυνση έναντι της σειριακή υλοποίησης, που έπειτα μένει σταθερή όσους πυρήνες και αν βάλουμε. Μάλιστα, το ίδιο ακριβώς παρατηρείται και στην MPI υλοποίηση, όπου στις 16 διεργασίες (με 2 διεργασίες/κόμβο) βλέπουμε μια μικρή – μικρότερη της ιδανικής – επιτάχυνση έναντι των 8 διεργασιών (με 1 διεργασία/κόμβο), και έπειτα για >16 διεργασίες (με >2 διεργασίες/κόμβο) βλέπουμε σχεδόν σταθερή επιτάχυνση.

TL;DR Με άλλα λόγια, η κλιμακωσιμότητα βλάπτεται λόγω της (σχεδόν πλήρους) έλλειψης κλιμακωσιμότητας εντός ενός κόμβου και όχι λόγω της επικοινωνίας μεταξύ διαφορετικών κόμβων. Αν χρησιμοποιούσαμε 64 κόμβους ανεξάρτητους, πιστεύουμε πως θα βλέπαμε σημαντικά καλύτερη κλιμακωσιμότητα στην επιτάχυνση.

Ενδιάφερον έχει η τεράστια, super-linear, επιτάχυνση ίση με περίπου 120x (σχεδόν διπλάσια της ιδανικής 64x!) για τον χρόνο υπολογισμών στην μέθοδο Jacobi στον πίνακα 2048×2048 . Το φαινόμενο αυτό εξηγείται λόγω της χρυφής μνήμης (cache) που έχει κάθε πυρήνας. Καθώς αυξάνεται το πλήθος διεργασιών, σε κάθε διεργασία (και συνεπώς σε κάθε πυρήνα) αναλογεί (αντιστρόφως ανάλογα) μικρότερου μέγεθους πίνακας για επεξεργασία. Συνεπώς αν το αρχικό grid έχει κατάλληλο μέγεθος, από ένα πλήθος διεργασιών και πάνω, ο πίνακας κάθε διεργασίας θα μπορέσει να χωρέσει στην cache, παραμένοντας έτσι σε εντός της cache από επανάληψη σε επανάληψη, με τεράστια βελτίωση στην χρονική επίδοση, όπως και παρατηρείται εδώ. Φυσικά, ο χρόνος επικοινωνίας, δεν μπορεί να κλιμακωθεί με τέτοιο ρυθμό, για αυτό και ο ολικός χρόνος τελικά δεν κλιμακώνει τόσο και έτσι προκύπτει το χάσμα, τόσο μεγάλο μόνο σε αυτή την περίπτωση, μεταξύ ολικού χρόνο και χρόνου υπολογισμών (τόσο σε απόλυτα νούμερα όσο και κλιμακωσιμότητα).

Συγκριτικά μεταξύ των μεθόδων, παρατηρείται πως, για ίδιο πλήθος επαναλήψεων, η μέθοδος Jacobi απαιτεί,

σε απόλυτους αριθμούς, λιγότερο χρόνο από τις δύο άλλες μεθόδους, γεγονός που θεωρείται λογικό καθώς έχει την απλότερη επικοινωνία. Για ως 16 διεργασίες, η RedBlackSOR είναι ταχύτερη της GaussSeidelSOR και για πάνω από 16 διεργασίες ισχύει το αντίστροφο, χωρίς ωστόσο σημαντικές διαφορές. Αυτή η διαφορά, μπορεί να αποδοθεί στη καλύτερη χωρική τοπικότητα της RedBlackSOR έναντι της GaussSeidelSOR (που θέλει συνεχώς πρόσβαση σε 2 πίνακες), καθώς σε υψηλότερο αριθμό διεργασιών/κόμβο, όπως σχολιάστηκε πριν, το πρόγραμμα γίνεται πολύ πιο έντονα memory-bound και έτσι η χωρική τοπικότητα διαδραματίζει τότε πολύ πιο σημαντικό ρόλο.

Από την άποψη της επιτάχυνσης και της κλιμακωσιμότητας, σχολιάζουμε τα μεγέθη 4096, 6144 καθώς στο 2048 παρατηρείται η παραπάνω ιδιοτροπία με την cache που ήδη εξηγήθηκε (με τις διαφορές μεταξύ των δύο μεθόδων να προκύπτουν τότε λόγω της διαφοράς στην χωρική τοπικότητα των προσβάσεων). Παρατηρείται πως η μέθοδος Jacobi εμφανίζει ελάχιστα μεγαλύτερη επιτάχυνση από την RedBlackSOR, ωστόσο οι δύο επιταχύνσεις γίνονται σχεδόν ίσες για ≥ 16 διεργασίες.

Η μέθοδος GaussSeidelSOR, αν και έχει την μικρότερη επιτάχυνση ως τις 16 διεργασίες, κλιμακώνει πολύ καλύτερα, και έτσι για > 16 διεργασίες ξεπερνά σημαντικά τις επιδόσεις των άλλων μεθόδων, φτάνοντας στο μέγιστο πλήθος διεργασιών, σχεδόν σε διπλάσια επιτάχυνση. Αυτό προκύπτει διότι χρησιμοποιεί τα μισά στοιχεία από την τρέχουσα εκτέλεση, που θα βρίσκονται στην cache, και έτσι η απαίτηση για εύρος ζώνης μνήμης υποδιαπλασιάζεται, οπότε το πρόγραμμα απομακρύνεται από την memory-bound περιοχή.

4.2.4 Μετρήσεις επίδοσης και διαγραμματική απεικόνιση για πλήθος επαναλήψεων μέχρι την σύγκλιση

Στην πράξη, δεν ενδιαφέρει η εκτέλεση κάποιου σταθερού πλήθους επαναλήψεων ενός αλγορίθμου, αλλά η εκτέλεση επαναλήψεων ως την σύγκλιση. Έτσι, εκτός της επιτάχυνσης, της κλιμακωσιμότητας, κ.λπ., που ήδη μελετήθηκαν, κρίνεται σκόπιμο, και μάλιστα με υψηλή σημασία, να μελετηθεί ο τελικός χρόνος εκτέλεσης για την επίλυση ενός προβλήματος.

Αν η ταχύτητα σύγκλισης διαφέρει σημαντικά, μπορεί να συμφέρει ένας αλγόριθμος, που φαίνομενικά φάνηκε να δίνει χειρότερες επιδόσεις στην εκτέλεση κάθε βήματος.

Στο σενάριο αυτό εκτελούμε επαναλήψεις μέχρι να φτάσουμε στην σύγκλιση που καθορίζεται από το δοθέν χριτήριο πως η καινουρία τιμή κάθε στοιχείου πρέπει να διάφερει από την παλαιά, το πολύ μια σταθερά $\epsilon > 0$.

Σημειώνεται πως το πλήθος επαναλήψεων αυτό, ως την σύγκλιση, δεν είναι ίδιο για τις 3 μεθόδους.

Εκτελέσαμε τις 3 μεθόδους και στην αμιγώς σειριακή υλοποίησή τους καθώς και στην MPI-based με 64 διεργασίες.

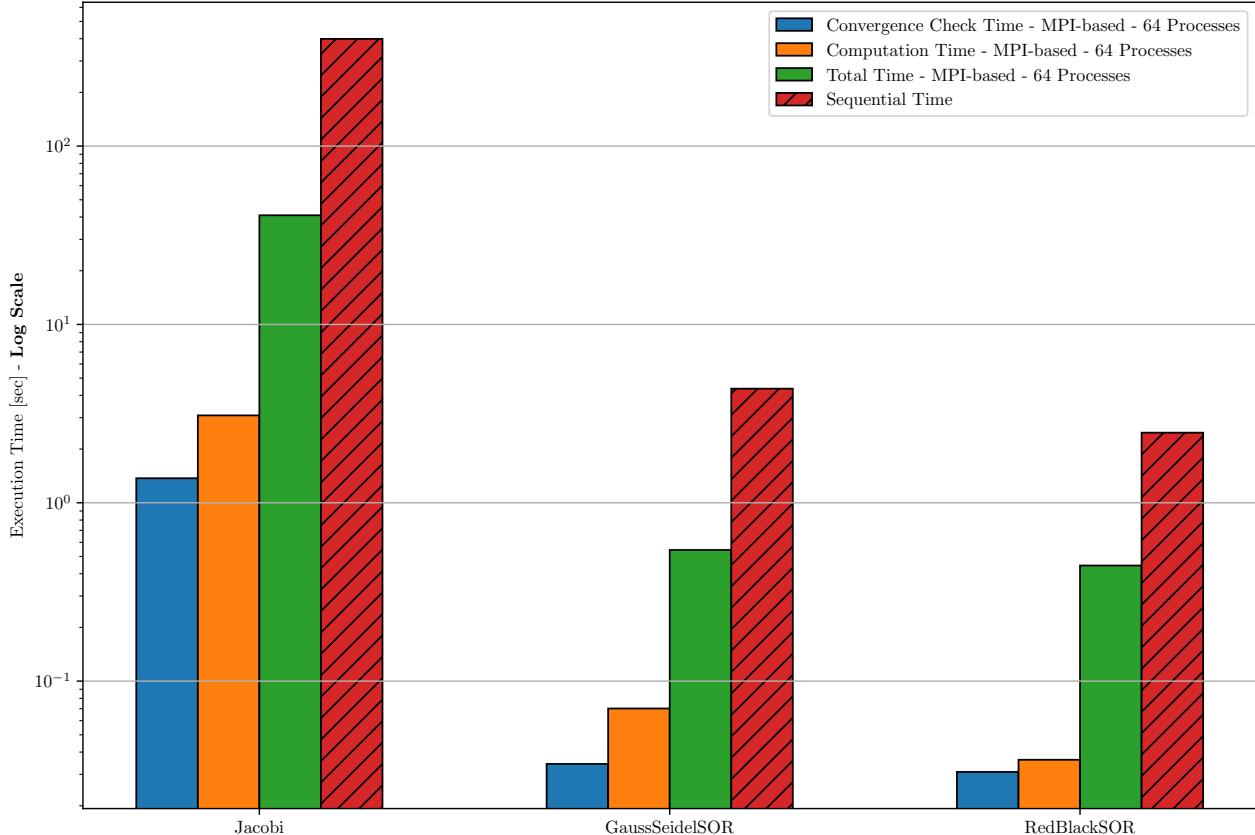
Χρησιμοποιήθηκε grid διαστάσεων 512×512 .

Τα αποτελέσματα για τους χρόνους εκτέλεσης της σειριακής υλοποίησης καθώς και της MPI-based (δίνεται και ολικό και αναλύεται σε χρόνο υπολογισμού και χρόνο ελέγχου σύγκλισης) φαίνονται στον πίνακα 2 και απεικονίζονται στο σχήμα 93. Το γράφημα έχει τον κατακόρυφο άξονα (αξόνας χρόνου) σε λογαριθμική κλίμακα ώστε να μπορούν να απεικονιστούν όλοι χρόνοι σε ένα διάγραμμα, καθώς οι διαφορές μεταξύ των χρόνων, δύπως φαίνονται, είναι τεράστιες.

Μέθοδος	Σειριακός Χρόνος	Ολικός Χρόνος MPI	Χρόνος Υπολογισμών MPI	Χρόνος Ελέγχου Σύγκλισης MPI
Jacobi	399.133240	40.929662	3.090690	1.372567
GaussSeidelSOR	4.363700	0.543638	0.070230	0.034320
RedBlackSOR	2.472030	0.444859	0.036220	0.030966

Πίνακας 2: Χρόνοι εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR, για εκτέλεση ως την σύγκλιση, σε grid 512×512 , για σειριακή υλοποίηση καθώς και MPI-based υλοποίηση με 64 διεργασίες

Execution Time for Heat Equation PDE Solution (until convergence) - 512×512



Σχήμα 93: Χρόνοι εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR, για εκτέλεση ως την σύγκλιση, σε grid 512×512 , για σειριακή υλοποίηση καθώς και MPI-based υλοποίηση με 64 διεργασίες

4.2.5 Παρατηρήσεις – σχόλια – δεδομένης και της ταχύτητας σύγκλισης

Παρατηρούμε πως υπάρχουν τεράστιες διαφορές στους χρόνους εκτέλεσης μεταξύ των αλγορίθμων, που προκύπτουν επειδή απαιτούν σημαντικά διαφορετικά πλήθη επαναλήψεων για να φτάσουν στην σύγκλιση.

Μάλιστα, προκύπτει το ενδιαφέρον συμπέρασμα πως είναι ταχύτερο να τρέξουμε σειριακά κάποια από τις μεθόδους GaussSeidelSOR ή RedBlackSOR, παρά να βάλουμε 64 διεργασίες στην MPI-based υλοποίηση της μεθόδου Jacobi!

Οι διαφορές λόγω διαφορετικού πλήθους επαναλήψεων, φαίνεται σε τελική ανάλυση, να υπερβαίνουν όλες τα συμπεράσματα στα οποία οδηγήθηκαμε προηγουμένως για την επίδοση σε σταθερό πλήθος επαναλήψεων, και τελικά οι διαφορές στην επίδοση μεταξύ των μεθόδων να καθορίζονται αποκλειστικά από το πλήθος επαναλήψεων.

Στην πράξη, θα επιλέγαμε την μέθοδο RedBlackSOR, καθώς επιτυγχάνει το ελάχιστο πλήθος επαναλήψεων και εν τέλει το ελάχιστο χρόνο εκτέλεσης με σημαντική διαφορά. Θα επιλέγαμε 8 ή 16 διεργασίες καθώς επιτυγχάνουν πρακτικά τον ίδιο χρόνο εκτέλεσης, όπως είδαμε προηγουμένως, οπότε το να χρησιμοποιήσουμε παραπάνω θα ήταν σπατάλη πόρων, που θα μπορούσαν ίσως να χρησιμοποιηθούν (ταυτόχρονα) για άλλο σκόπο. Γενικά, αν είχαμε περισσότερους (φυσικούς) κόμβους θα μπορούσαμε να τους χρησιμοποιήσουμε, πάντως με ως 1-2 πυρήνες/φυσικό κόμβο, διότι με παραπάνω δεν επιτυγχάνεται πρόσθετη επιτάχυνση.

Κατάλογος σχημάτων

1	Μετρήσεις ολικού χρόνου εκτέλεσης για το παραλληλοποιημένο Game of Life	4
2	Μετρήσεις επιτάχυνσης για το Game of Life για το παραλληλοποιημένο Game of Life	4
3	Μετρήσεις επίδοσης στο παραλληλοποιημένο Game of Life για $N = 4096$ και για διάφορους τύπους δεδομένων πίνακα	7
α'	Επιτάχυνση	7
β'	Ολικός χρόνος εκτέλεσης	7
4	Χρόνος εκτέλεσης για naïve παραλληλοποιημένο KMeans	10
5	Επιτάχυνση για naïve παραλληλοποιημένο KMeans	10
6	Χρόνος εκτέλεσης για naïve παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού	11
7	Επιτάχυνση για naïve παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού	12
8	Επιτάχυνση για naïve παραλληλοποιημένο KMeans – σύγχριση εκτελέσεων	13
9	Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού	15
10	Επιτάχυνση για reduction παραλληλοποιημένο KMeans – νήματα εκτέλεσης σε σταθερά νήματα υλικού	16
11	Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – 2o configuration	17
12	Επιτάχυνση για reduction παραλληλοποιημένο KMeans – 2o configuration	18
13	Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – διορθωμένη υλοποίηση – 2o configuration	20
14	Επιτάχυνση για reduction παραλληλοποιημένο KMeans – διορθωμένη υλοποίηση – 2o configuration	21
15	Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation	25
16	Επιτάχυνση για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation	26
17	Χρόνος εκτέλεσης για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation - 1o configuration	27
18	Επιτάχυνση για reduction παραλληλοποιημένο KMeans – NUMA-aware allocation – 1o configuration	27
19	Χρόνοι εκτέλεσης KMeans για διάφορες υλοποιήσεις κλειδωμάτων	33
20	Επιτάχυνση KMeans για διάφορες υλοποιήσεις κλειδωμάτων	33
21	Χρόνοι εκτέλεσης KMeans για διάφορες υλοποιήσεις κλειδωμάτων – εστίαση ως 8 πυρήνες	34
22	Επιτάχυνση KMeans για διάφορες υλοποιήσεις κλειδωμάτων – εστίαση ως 8 πυρήνες	34
23	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – σειριακή εκδοχή – $N = 1024$	36
24	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – σειριακή εκδοχή – $N = 2048$	37
25	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – σειριακή εκδοχή – $N = 4096$	37
26	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 1024$	39
27	Επιτάχυνση στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 1024$	39
28	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 2048$	40
29	Επιτάχυνση στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 2048$	40
30	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 4096$	41
31	Επιτάχυνση στον Αναδρομικό Floyd-Warshall για διάφορα πλήθη πυρήνων – παραλληλοποιημένη εκδοχή – $N = 4096$	41
32	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – 32 πυρήνες – $N = 1024$	42
33	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – 32 πυρήνες – $N = 2048$	42
34	Χρόνος εκτέλεσης στον Αναδρομικό Floyd-Warshall για διάφορα block sizes – 32 πυρήνες – $N = 4096$	43
35	Ιδανικά μέγιστη επιτάχυνση – Αναδρομικός Floyd-Warshall – $N = 4096$ – $B = 512$	44
36	Επιτάχυνση στον Tiled Floyd-Warshall για διάφορα πλήθη πυρήνων – $N = 1024$	47
37	Επιτάχυνση στον Tiled Floyd-Warshall για διάφορα πλήθη πυρήνων – $N = 4096$	48
38	Ρυθμαπόδοση (Throughput) για διάφορους φόρτους εργασίας με μέγεθος λίστας 1024.	51
α'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 0-50-50	51
β'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 100-0-0	51

γ'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 20-40-40	51
δ'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 80-10-10	51
39	Ρυθμαπόδοση (Throughput) για διάφορους φόρτους εργασίας με μέγεθος λίστας 8192.	51
α'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 0-50-50	51
β'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 100-0-0	51
γ'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 20-40-40	51
δ'	Ρυθμαπόδοση (Throughput) με φόρτο εργασίας 80-10-10	51
40	Speedup για διαφορετικούς φόρτους εργασίας με μέγεθος λίστας 1024.	52
α'	Speedup για φόρτο εργασίας 0-50-50	52
β'	Speedup για φόρτο εργασίας 100-0-0	52
γ'	Speedup για φόρτο εργασίας 20-40-40	52
δ'	Speedup για φόρτο εργασίας 80-10-10	52
41	Speedup για διαφορετικούς φόρτους εργασίας με μέγεθος λίστας 8192.	52
α'	Speedup για φόρτο εργασίας 0-50-50	52
β'	Speedup για φόρτο εργασίας 100-0-0	52
γ'	Speedup για φόρτο εργασίας 20-40-40	52
δ'	Speedup για φόρτο εργασίας 80-10-10	52
42	Αποδοτικότητα για διάφορους φόρτους εργασίας με μέγεθος λίστας 1024.	53
α'	Αποδοτικότητα με φόρτο εργασίας 0-50-50	53
β'	Αποδοτικότητα με φόρτο εργασίας 100-0-0	53
γ'	Αποδοτικότητα με φόρτο εργασίας 20-40-40	53
δ'	Αποδοτικότητα με φόρτο εργασίας 80-10-10	53
43	Αποδοτικότητα για διάφορους φόρτους εργασίας με μέγεθος λίστας 8192.	53
α'	Αποδοτικότητα με φόρτο εργασίας 0-50-50	53
β'	Αποδοτικότητα με φόρτο εργασίας 100-0-0	53
γ'	Αποδοτικότητα με φόρτο εργασίας 20-40-40	53
δ'	Αποδοτικότητα με φόρτο εργασίας 80-10-10	53
44	Επίδραση Φορτίου για Διάφορους Αριθμούς Νημάτων με μέγεθος λίστας 1024.	54
α'	Επίδραση Φορτίου για 4 Νήματα	54
β'	Επίδραση Φορτίου για 8 Νήματα	54
γ'	Επίδραση Φορτίου για 16 Νήματα	54
δ'	Επίδραση Φορτίου για 32 Νήματα	54
45	Επίδραση Φορτίου για Διάφορους Αριθμούς Νημάτων με μέγεθος λίστας 8192.	54
α'	Επίδραση Φορτίου για 4 Νήματα	54
β'	Επίδραση Φορτίου για 8 Νήματα	54
γ'	Επίδραση Φορτίου για 16 Νήματα	54
δ'	Επίδραση Φορτίου για 32 Νήματα	54
46	Επίδραση Μεγέθους Λίστας για Διάφορους Αριθμούς Νημάτων και Φορτία	55
α'	Επίδραση Μεγέθους Λίστας για 16 Νήματα, Φορτίο 0-50-50	55
β'	Επίδραση Μεγέθους Λίστας για 16 Νήματα, Φορτίο 100-0-0	55
γ'	Επίδραση Μεγέθους Λίστας για 32 Νήματα, Φορτίο 0-50-50	55
δ'	Επίδραση Μεγέθους Λίστας για 32 Νήματα, Φορτίο 100-0-0	55
ε'	Επίδραση Μεγέθους Λίστας για 4 Νήματα, Φορτίο 0-50-50	55
στ'	Επίδραση Μεγέθους Λίστας για 4 Νήματα, Φορτίο 100-0-0	55
ζ'	Επίδραση Μεγέθους Λίστας για 8 Νήματα, Φορτίο 0-50-50	55
η'	Επίδραση Μεγέθους Λίστας για 8 Νήματα, Φορτίο 100-0-0	55
47	Heatmaps για τη σειριακή υλοποίηση με διάφορα μεγέθη λίστας.	56
α'	Heatmap της σειριακής υλοποίησης για μέγεθος λίστας 1024	56
β'	Heatmap της σειριακής υλοποίησης για μέγεθος λίστας 8192	56
48	Heatmaps για διαφορετικές υλοποιήσεις με διάφορα μεγέθη λίστας. (συνέχεια)	57
α'	Heatmap για την υλοποίηση CGL με μέγεθος λίστας 1024	57
β'	Heatmap για την υλοποίηση CGL με μέγεθος λίστας 8192	57
γ'	Heatmap για την υλοποίηση FGL με μέγεθος λίστας 1024	57
δ'	Heatmap για την υλοποίηση FGL με μέγεθος λίστας 8192	57
ε'	Heatmap για την υλοποίηση Lazy με μέγεθος λίστας 1024	57
στ'	Heatmap για την υλοποίηση Lazy με μέγεθος λίστας 8192	57
ζ'	Heatmap για την υλοποίηση Non-Blocking με μέγεθος λίστας 1024	57
η'	Heatmap για την υλοποίηση Non-Blocking με μέγεθος λίστας 8192	57
θ'	Heatmap για την υλοποίηση Optimistic με μέγεθος λίστας 1024	57
ι'	Heatmap για την υλοποίηση Optimistic με μέγεθος λίστας 8192	57
49	Χρόνος εκτέλεσης Naive GPU υλοποίησης του αλγόριθμου Kmeans	58

50	Επιτάχυνση (σε σχέση με CPU) Naive GPU υλοποίησης του αλγόριθμου Kmeans	59
51	Χρόνος εκτέλεσης Transpose GPU υλοποίησης του αλγόριθμου Kmeans	61
52	Επιτάχυνση (σε σχέση με CPU) Transpose GPU υλοποίησης του αλγόριθμου Kmeans	62
53	Χρόνος εκτέλεσης Shared GPU υλοποίησης του αλγόριθμου Kmeans	64
54	Επιτάχυνση (σε σχέση με CPU) Shared GPU υλοποίησης του αλγόριθμου Kmeans	65
55	Χρόνος εκτέλεσης των υλοποίησεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans	66
56	Επιτάχυνση (σε σχέση με CPU) των υλοποιήσεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans	66
57	Χρόνος εκτέλεσης Naive GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	69
58	Επιτάχυνση (σε σχέση με CPU) Naive GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	69
59	Χρόνος εκτέλεσης Transpose GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	70
60	Επιτάχυνση (σε σχέση με CPU) Transpose GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	70
61	Χρόνος εκτέλεσης Shared Memory GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	71
62	Επιτάχυνση (σε σχέση με CPU) Shared Memory GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	71
63	Χρόνος εκτέλεσης των υλοποίησεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans (2o Configuration)	72
64	Επιτάχυνση (σε σχέση με CPU) των υλοποιήσεων Naive, Transpose, Shared Memory GPU υλοποιήσεων του αλγόριθμου Kmeans (2o Configuration)	72
65	Χρόνος εκτέλεσης All-GPU υλοποίησης του αλγόριθμου Kmeans	75
66	Επιτάχυνση (σε σχέση με CPU) All-GPU υλοποίησης του αλγόριθμου Kmeans	75
67	Χρόνος εκτέλεσης All-GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	76
68	Επιτάχυνση (σε σχέση με CPU) All-GPU υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	76
69	Χρόνος εκτέλεσης All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans	80
70	Επιτάχυνση (σε σχέση με CPU) All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans	80
71	Χρόνος εκτέλεσης All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	81
72	Επιτάχυνση (σε σχέση με CPU) All-GPU Delta Reduction υλοποίησης του αλγόριθμου Kmeans (2o Configuration)	81
73	Χρόνος εκτέλεσης MPI-based υλοποίησης αλγορίθμου KMeans ως συνάρτηση του πλήθους διεργασιών για 2 Configurations	84
74	Επιτάχυνση ως προς την σειριακή υλοποίησης MPI-based υλοποίησης αλγορίθμου KMeans ως συνάρτηση του πλήθους διεργασιών για 2 Configurations	84
75	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου Jacobi ως συνάρτηση του πλήθους διεργασιών σε grid 2048×2048	88
76	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου Jacobi ως συνάρτηση του πλήθους διεργασιών σε grid 4096×4096	88
77	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου Jacobi ως συνάρτηση του πλήθους διεργασιών σε grid 6144×6144	88
78	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 2048×2048	89
79	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 4096×4096	89
80	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 6144×6144	89
81	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 2048×2048	90
82	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 4096×4096	90
83	Επιτάχυνση της MPI-based υλοποιήσης της μεθόδου GaussSeidelSOR ως συνάρτηση του πλήθους διεργασιών σε grid 6144×6144	90
84	Συγκριτική απεικόνιση χρόνων εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 2048×2048 ως συνάρτηση του πλήθους διεργασιών	91
85	Συγκριτική απεικόνιση επιταχύνσεων (ολικού χρόνου εκτέλεσης) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 2048×2048 ως συνάρτηση του πλήθους διεργασιών	91
86	Συγκριτική απεικόνιση χρόνων εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 4096×4096 ως συνάρτηση του πλήθους διεργασιών	92
87	Συγκριτική απεικόνιση επιταχύνσεων (ολικού χρόνου εκτέλεσης) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 4096×4096 ως συνάρτηση του πλήθους διεργασιών	92

88	Συγκριτική απεικόνιση χρόνων εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 6144×6144 ως συνάρτηση του πλήθους διεργασιών	93
89	Συγκριτική απεικόνιση επιταχύνσεων (ολικού χρόνου εκτέλεσης) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 6144×6144 ως συνάρτηση του πλήθους διεργασιών	93
90	Συγκριτική απεικόνιση χρόνων εκτέλεσης σε διάγραμμα με μπάρες (Ζητούμενο από την εκφώνηση διάγραμμα) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 2048×2048	94
91	Συγκριτική απεικόνιση χρόνων εκτέλεσης σε διάγραμμα με μπάρες (Ζητούμενο από την εκφώνηση διάγραμμα) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 4096×4096	94
92	Συγκριτική απεικόνιση χρόνων εκτέλεσης σε διάγραμμα με μπάρες (Ζητούμενο από την εκφώνηση διάγραμμα) για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR σε grid 6144×6144	95
93	Χρόνοι εκτέλεσης για τις μεθόδους Jacobi, GaussSeidelSOR, RedBlackSOR, για εκτέλεση ως την σύγκλιση, σε grid 512×512 , για σειριακή υλοποίηση καθώς και MPI-based υλοποίηση με 64 διεργασίες	98