Συστήματα Παράλληλης Επεξεργασίας

Εξαμηνιαία Άσκηση

Ομάδα parlab17
Νικόλαος Παγώνας, el18175
Αγγελική Εμμανουέλα Συρρή, el18811

1 Εξοικείωση με το περιβάλλον προγραμματισμού

1.1 Εισαγωγή

Σκοπός της παρούσας άσκησης είναι η εξοικείωση με τις υποδομές του εργαστηρίου μέσω της παραλληλοποίησης του *Conway's Game of Life* σε αρχιτεκτονικές κοινής μνήμης, με χρήση του OpenMP.

1.2 Conway's Game of Life

Το *Conway's Game of Life* εξελίσσεται σε ένα δισδιάστατο ταμπλό, κάθε κελί του οποίου μπορεί να είναι ζωντανό (1) ή νεκρό (0). Κάθε χρονική στιγμή, κάθε κελί αλλάζει κατάσταση με βάση την κατάσταση των 8 γειτόνων του, ακολουθώντας τους εξής κανόνες:

- Αν ένα κελί είναι ζωντανό και έχει λιγότερους από 2 γείτονες πεθαίνει *(μοναξιά)*
- Αν ένα κελί είναι ζωντανό και έχει περισσότερους από 3 γείτονες πεθαίνει *(υπερπληθυσμός)*
- Αν ένα κελί είναι ζωντανό και έχει 2 ή 3 γείτονες παραμένει ζωντανό *(επιβίωση)*
- Αν ένα κελί είναι νεκρό και έχει ακριβώς 3 γείτονες γίνεται ζωντανό *(αναπαραγωγή)*

Η σειριακή υλοποίηση του *Game of Life* δίνεται έτοιμη στο αρχείο `/home/parallel/pps/2022-2023/a1/Game_Of_Life.c`.

1.3 Μεταγλώττιση και υποβολή εργασιών στις συστοιχίες

Για τη μεταγλώττιση χρησιμοποιούμε το αρχείο `make_on_queue.sh`, ενώ για το τρέξιμο του προγράμματος χρησιμοποιούμε το `run_on_queue.sh`. Τα αρχεία αυτά τα υποβάλλουμε σαν εργασίες στην ουρά `parlab` του Torque, με τις εντολές:

```
qsub -q parlab {make,run}_on_queue.sh
```

Στο αρχείο `run_on_queue.sh` ορίζουμε και τη μεταβλητή περιβάλλοντος `OMP_NUM_THREADS`, η οποία καθορίζει τον αριθμό νημάτων που θα χρησιμοποιηθούν για την παραλληλοποίηση.

1.4 Ανάπτυξη παράλληλου προγράμματος στο μοντέλο κοινού χώρου διευθύνσεων με χρήση του OpenMP

Για να παραλληλοποιήσουμε το `Game_Of_Life.c` στο μοντέλο κοινού χώρου διευθύνσεων θα χρησιμοποιήσουμε το OpenMP. Παρατηρούμε ότι η καρδιά της υλοποίησης είναι ο τριπλός βρόχος:

```
for (t = 0; t < T ;t++) {
  for (i = 1; i < N-1; i++) {
    for (j = 1; j < N-1; j++) {
      nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
             + previous[i][j-1] + previous[i][j+1] \
             + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
      if (nbrs == 3 || (previous[i][j]+nbrs == 3))
        current[i][j] = 1;
      else
        current[i][j] = 0;
    }
  }
    /* ... */
}
```

Είναι εύκολο να δούμε ότι δεν μπορούμε να κάνουμε παραλληλοποίηση ως προς τη μεταβλητή χρόνου, αφού για να υπολογίσουμε την επόμενη κατάσταση του ταμπλό (πίνακας `current`), χρειάζεται να έχουν υπολογιστεί πρώτα όλα τα κελιά του πίνακα `previous`. Ωστόσο, οι δύο εσωτερικοί βρόχοι μπορούν να τρέξουν παράλληλα, αφού κάθε κελί του πίνακα `current` χρειάζεται μόνο τις γειτονικές τιμές στον πίνακα `previous` και δεν υπάρχει εξάρτηση μεταξύ των κελιών του `current`.

Επομένως, εισάγουμε στον κώδικα το #pragma directive που ακολουθεί:

```
#pragma omp parallel for shared(N, previous, current) private(i, j, nbrs)
```

Για τον λόγο που εξηγήσαμε προηγουμένως, το εισάγουμε πριν τους δύο εσωτερικούς βρόχους:

```
for (t = 0; t < T; t++) {
  #pragma omp parallel for shared(N, previous, current) private(i, j, nbrs)
  for (i = 1; i < N-1; i++) {
    for (j = 1; j < N-1; j++) {
      nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
             + previous[i][j-1] + previous[i][j+1] \
             + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
      if (nbrs == 3 || (previous[i][j] + nbrs == 3))
        current[i][j] = 1;
      else
        current[i][j] = 0;
    }
  }
  /* ... */
}
```

Οι μεταβλητές `N`, `previous` και `current` επιλέχθηκαν να είναι *shared*, διότι όλα τα νήματα πρέπει να έχουν κοινή εικόνα για το περιεχόμενο των δύο πινάκων και τις διαστάσεις τους. Αντίθετα, οι μεταβλητές `i`, `j` και `nbrs` επιλέχθηκαν να είναι *private*, διότι κάθε νήμα λειτουργεί σε διαφορετική θέση του ταμπλό, και υπολογίζει διαφορετικό αριθμό γειτόνων από τα υπόλοιπα νήματα.

1.5 Μετρήσεις επίδοσης

Η μέτρηση του χρόνου εκτέλεσης είναι ενσωματωμένη στο `Game_Of_Life.c`, όπου χρησιμοποιείται η συνάρτηση:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

στην αρχή και στο τέλος του τριπλού βρόχου. Με τη γραμμή:

```
printf("GameOfLife: Size %d Steps %d Time %lf\n", N, T, time);
```

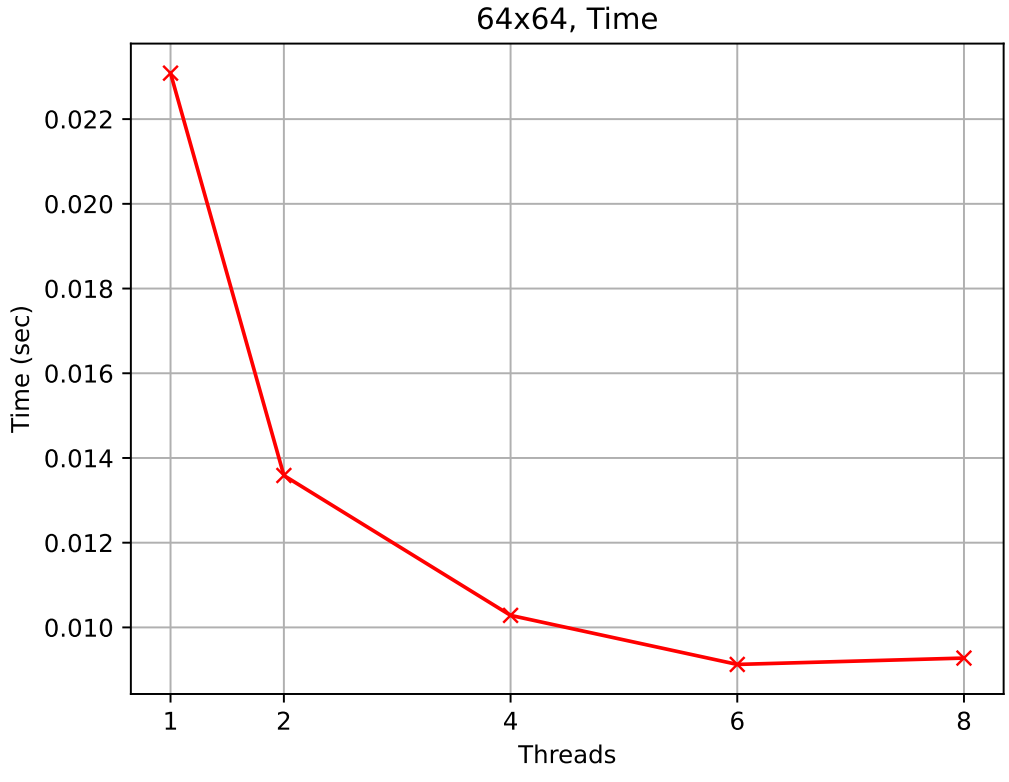
Μπορούμε να εξάγουμε την επίδοση της κάθε εκτέλεσης στο αρχείο εξόδου.

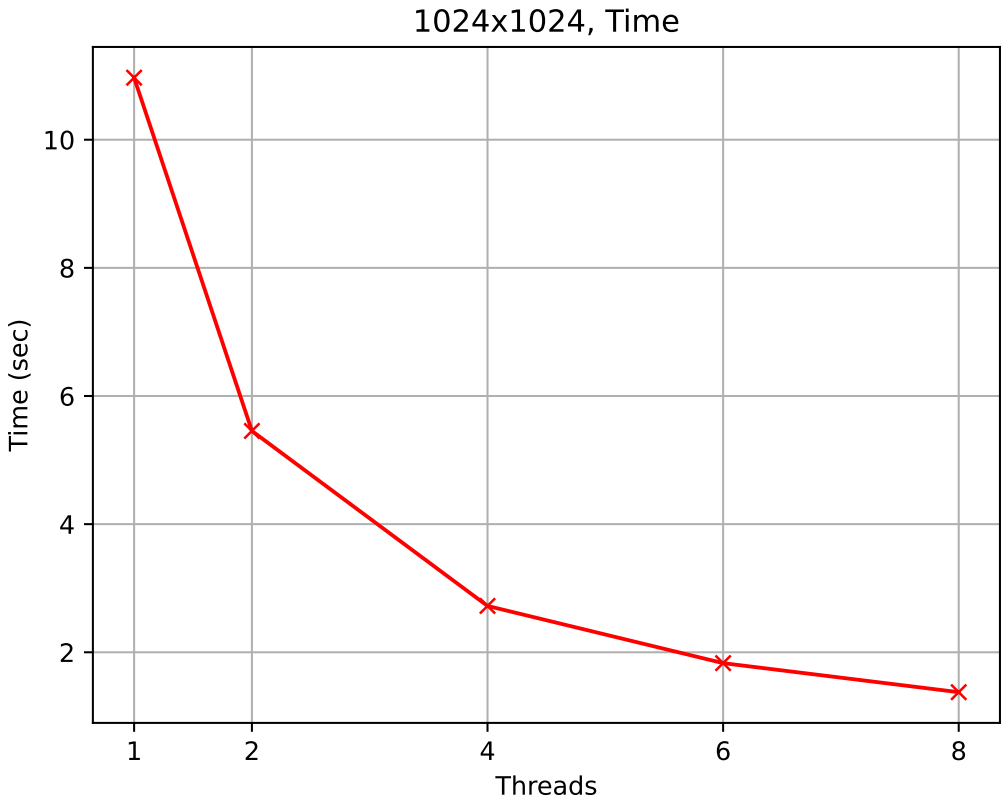
1.6 Αποτελέσματα

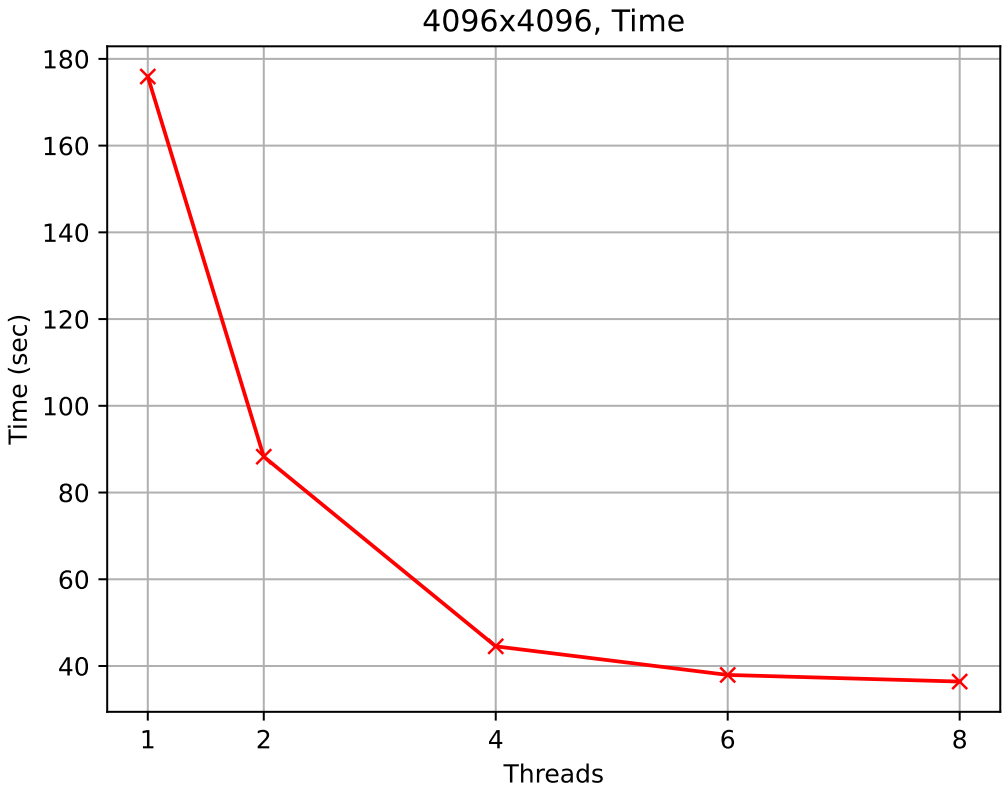
Στο αρχείο `run_Game_Of_Life.out` βρίσκονται τα αποτελέσματα των εκτελέσεων για $1, 2, 4, 6, 8$ πυρήνες και μεγέθη ταμπλό 64×64, 1024×1024 και 4096×4096. Σε όλες τις περιπτώσεις το παιχνίδι τρέχει για 1000 γενιές. Έχουμε για τον χρόνο εκτέλεσης σε δευτερόλεπτα:

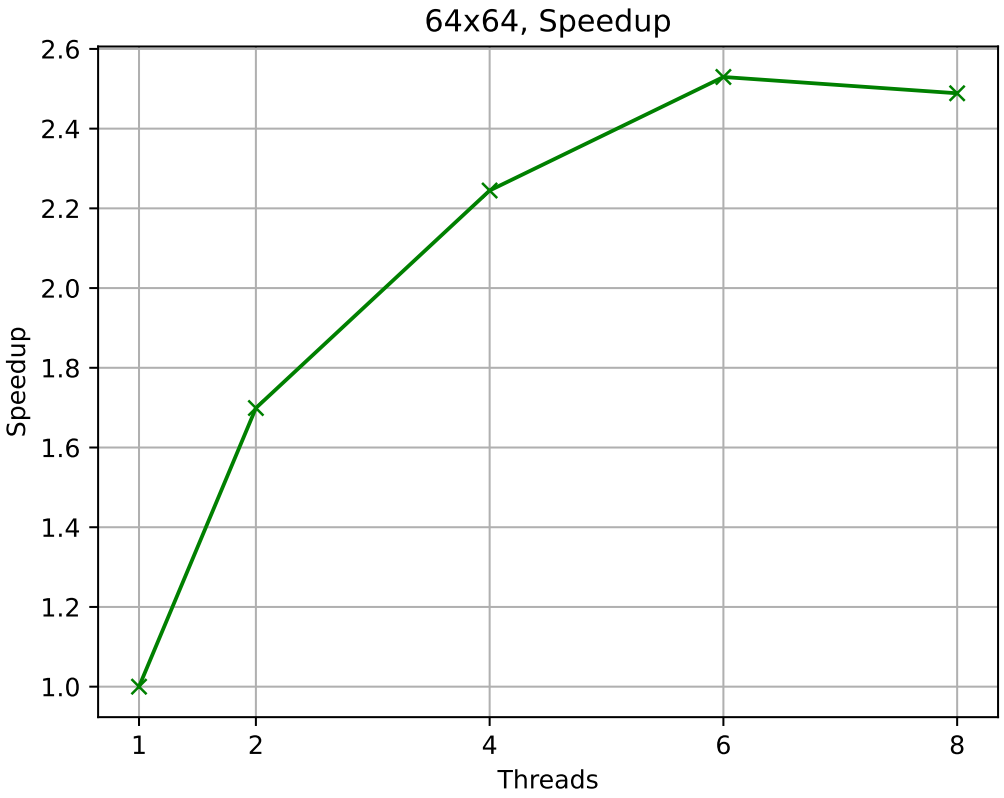
Μέγεθος ταμπλό / Αριθμός νημάτων	64×64	1024×1024	4096×4096
1	0.023085	10.968392	175.924056
2	0.013587	5.454778	88.268872
4	0.010284	2.723389	44.534155
6	0.009126	1.830635	37.940541
8	0.009276	1.377333	36.407225

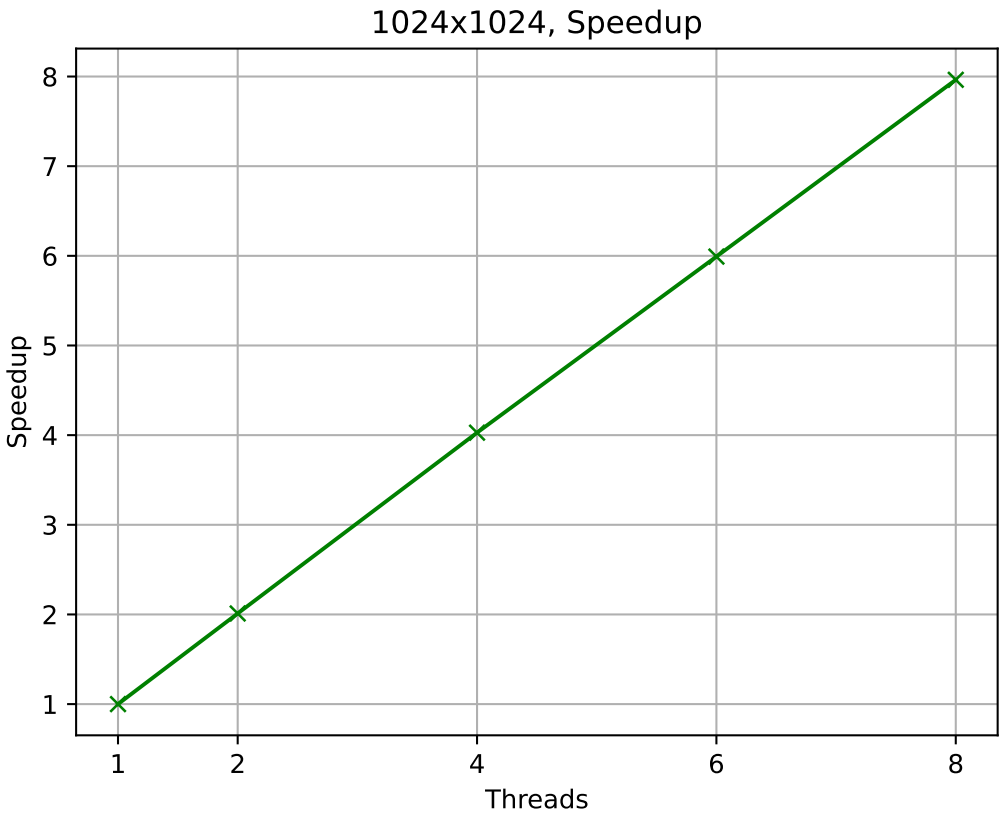
Χρησιμοποιώντας τους παραπάνω χρόνους και το αρχείο `plot.py`, δημιουργούμε τα παρακάτω διαγράμματα χρόνου και επιτάχυνσης. Η επιτάχυνση ορίζεται ως $S = T_S/T_P$, όπου T_S ο χρόνος εκτέλεσης του σειριακού προγράμματος και T_P ο χρόνος εκτέλεσης του παράλληλου προγράμματος με P νήματα:

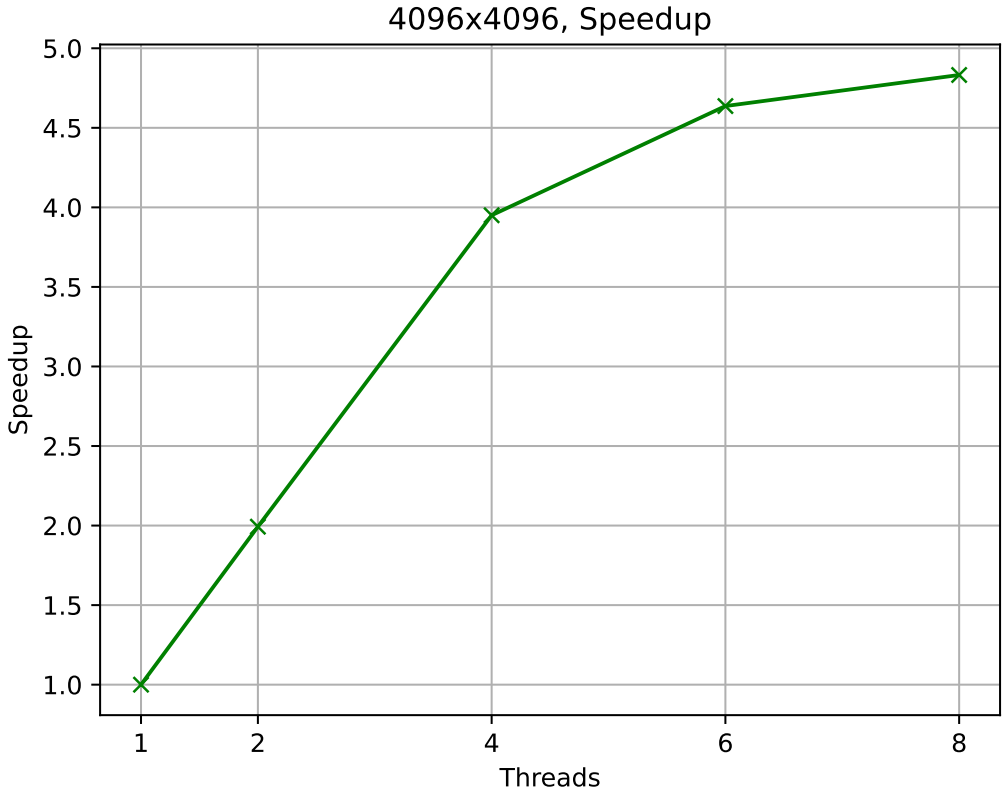



1024x1024, Time
Time (sec)
10
8
6
4
2
1
2
4
6
8
Threads


4096x4096, Time
Time (sec)
180
160
140
120
100
80
60
40
1
2
4
6
8
Threads


64x64, Speedup
Speedup
Threads
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
1
2
4
6
8


1024x1024, Speedup
Speedup
Threads
8
7
6
5
4
3
2
1
1
2
4
6
8



1.7 Σχολιασμός

Στόχος της παραλληλοποίησης είναι η μείωση του χρόνου εκτέλεσης μέσω της αξιοποίησης των πόρων του υπολογιστή. Στη βέλτιστη περίπτωση, η μείωση του χρόνου εκτέλεσης είναι ανάλογη του αριθμού των νημάτων. Για παράδειγμα, ο διπλασιασμός των νημάτων οδηγεί σε υποδιπλασιασμό του χρόνου εκτέλεσης.

Παρατηρούμε τα παρακάτω:

- Για την περίπτωση 64×64, η επιτάχυνση είναι χειρότερη της γραμμικής. Όσο αυξάνεται ο αριθμός των νημάτων, η διαφορά στην επιτάχυνση είναι όλο και μικρότερη. Μάλιστα, όταν μεταβαίνουμε από 6 σε 8 νήματα ο χρόνος εκτέλεσης αυξάνεται αντί να μειώνεται.
- Για την περίπτωση 1024×1024, η επιτάχυνση είναι γραμμική. Κάθε φορά που διπλασιάζονται τα νήματα, ο χρόνος εκτέλεσης υποδιπλασιάζεται, με άλλα λόγια ο λόγος T_S/T_P είναι κάθε φορά ίσος με P.
- Για την περίπτωση 4096×4096, η επιτάχυνση είναι γραμμική μέχρι και τα 4 νήματα, ενώ για 6 και 8 νήματα είναι αρκετά χειρότερη της γραμμικής.

Απ' ό,τι φαίνεται, η ιδανική περίπτωση δεν ανταπεξέρχεται πάντα στην πραγματικότητα. Για να ερμηνεύσουμε τα παραπάνω αποτελέσματα, δεν πρέπει να ξεχνάμε ότι, σε σχέση με την σειριακή εκτέλεση, η παραλληλοποίηση εισάγει επιπλέον κόστος για τη *διαχείριση των νημάτων* (δημιουργία/συγχρονισμός των νημάτων, αναμονή να τελειώσουν όλα τα νήματα, συλλογή του τελικού αποτελέσματος). Επιπλέον, η *συμφόρηση στο διάδρομο μνήμης* αποτελεί εμπόδιο όσο οι προσβάσεις στη μνήμη αυξάνονται. Έτσι, ανάλογα με το μέγεθος του ταμπλό, το επιπλέον κόστος δρα με διαφορετικό τρόπο στην εκάστοτε περίπτωση.

Όταν το ταμπλό είναι μικρό, όπως στην περίπτωση 64×64, το κόστος διαχείρισης των νημάτων είναι συγκρίσιμο με το κόστος εκτέλεσης του αλγορίθμου, και άρα έχει σημαντική επίδραση στο τελικό αποτέλεσμα. Γι' αυτό και για 8 νήματα η επίδοση φτάνει να είναι μέχρι και χειρότερη από ό,τι με 6 νήματα.

Όσο το μέγεθος του ταμπλό αυξάνεται όμως, φαίνεται η επίδραση αυτού του κόστους να είναι όλο και λιγότερο ισχυρή, όπως διακρίνεται στην περίπτωση 1024×1024, όπου η επιτάχυνση είναι γραμμική.

Βέβαια, η αύξηση του μεγέθους φέρνει αργά ή γρήγορα την υπερβολική συμφόρηση στο διάδρομο μνήμης, κάτι που φαίνεται στην περίπτωση 4096×4096. Ένα ακόμα πιθανό πρόβλημα που εμφανίζεται όσο αυξάνεται το μέγεθος του ταμπλό είναι ότι πλέον δεν μπορούν όλα τα δεδομένα να βρίσκονται στην cache του κάθε πυρήνα. Αυτό έχει ως αποτέλεσμα να απαιτείται επιπλέον χρόνος για την μεταφορά των απαραίτητων δεδομένων στην μνήμη.

1.8 Ειδικές Αρχικοποιήσεις

Στα δύο αρχεία `.gif` συμπεριλαμβάνονται δύο ενδιαφέρουσες αρχικοποιήσεις:

- *Gosper Glider Gun:* Αυτή η αρχικοποίηση δημιουργεί συνεχώς δομές που ονομάζονται "gliders", λόγω του τρόπου που κινούνται στο ταμπλό. Ήταν η πρώτη συνεχώς αυξανόμενη σε μέγεθος αρχικοποίηση (infinitely-growing pattern) που ανακαλύφθηκε, το 1970.
- *Snark Loop*: Μία άλλη ενδιαφέρουσα αρχικοποίηση, που ανήκει στην κατηγορία των ταλαντωτών (oscillators) επειδή επαναλαμβάνεται περιοδικά. Η συγκεκριμένη έχει περίοδο 43.

2 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

2.1.1 Εισαγωγή/Σκοπός της άσκησης

Σκοπός της άσκησης είναι η παραλληλοποίηση του αλγορίθμου K-means με δύο τρόπους, χρησιμοποιώντας το OpenMP (προγραμματιστικό μοντέλο κοινού χώρου διευθύνσεων).

2.1.2 Ο αλγόριθμος K-means

Ο αλγόριθμος K-means χρησιμοποιείται για το διαχωρισμό N αντικειμένων σε k μη επικαλυπτόμενες ομάδες-clusters. Σε ψευδοκώδικα:

```
until convergence (or fixed loops)
  for each object
    find nearest cluster
  for each cluster
    calculate new cluster center coordinates.
```

Οι δύο τρόποι παραλληλοποίησης που θα υλοποιήσουμε είναι οι εξής:

- Προσθήκη κατάλληλων εντολών συγχρονισμού ώστε να γίνει σωστά η παράλληλη ενημέρωση του πίνακα `newClusters`.
- Χρήση τοπικών (copied) πινάκων για κάθε νήμα, ώστε να μην υπάρχει ανάγκη συγχρονισμού, και συγκέντρωση (reduction) στον τελικό πίνακα newClusters.

2.1.3 Δεδομένα

Για την ανάπτυξη των παραπάνω υλοποιήσεων χρησιμοποιούμε τους σκελετούς που δίνονται στο directory `/home/parallel/pps/2022-2023/a2/kmeans`. Το αρχείο `seq_kmeans.c` αντιστοιχεί στην σειριακή υλοποίηση, το `omp_naive_kmeans.c` αντιστοιχεί στην υλοποίηση με κατάλληλες εντολές συγχρονισμού, ενώ το `omp_reduction_kmeans.c` αντιστοιχεί στην υλοποίηση με την χρήση copied πινάκων. Με το `file_io.c` γίνεται η δημιουργία του dataset, ενώ το `main.c` καλεί την κατάλληλη υλοποίηση του K-means κάθε φορά.

2.1.4 shared clusters

1. Εδώ παραλληλοποιούμε την naive έκδοση του αλγορίθμου. Το κομμάτι που άλλαξε σε σχέση με τον προϋπάρχοντα σκελετό είναι το εξής:

```
#pragma omp parallel for private(i, j, index)
for (i=0; i<numObjs; i++) {
  // find the array index of nearest cluster center
  index = find_nearest_cluster(
      numClusters,
      numCoords,
      &objects[i*numCoords],
      clusters
  );

  /* ... */

  // update new cluster centers : sum of objects located within
  // enforce atomic access to shared "newClusterSize" array
  #pragma omp atomic
  newClusterSize[index]++;

  for (j=0; j<numCoords; j++) {
    // enforce atomic access to shared "newClusters" array
    #pragma omp atomic
    newClusters[index*numCoords + j] += objects[i*numCoords + j];
  }
}
```

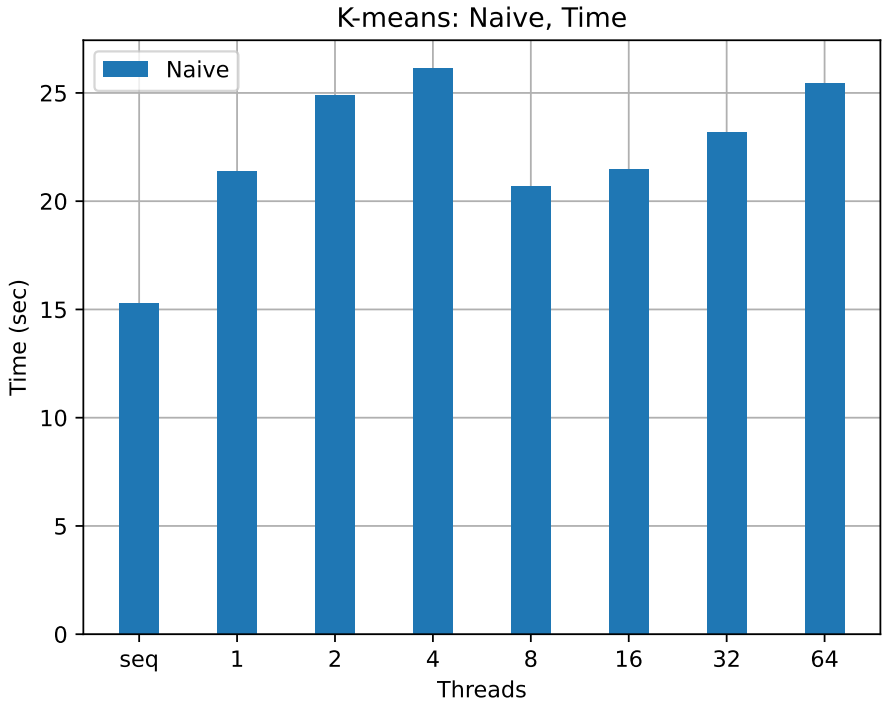
Η παραλληλοποίηση γίνεται με την χρήση του parallel for, στο οποίο οι μεταβλητές `i, j, index` είναι *private* (αντιγραμμένες), αφού:

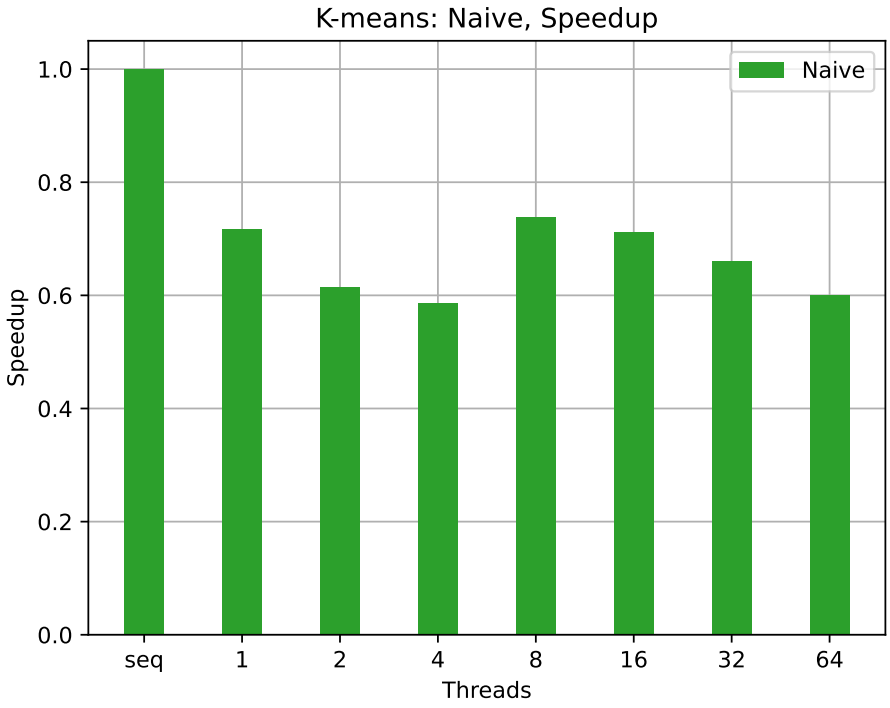
- Τα `i, j` με τα οποία αποκτούμε πρόσβαση στα στοιχεία των πινάκων, πρέπει να είναι διαφορετικά για κάθε thread.
- Το `index` που προκύπτει από την εύρεση του κοντινότερου cluster είναι διαφορετικό ανά `i`, και άρα πρέπει να είναι επίσης διαφορετικό για κάθε thread.

Το βασικό σημείο προσοχής σε αυτή την υλοποίηση είναι ότι έχουμε διαμοιραζόμενα δεδομένα, οπότε πρέπει να φροντίσουμε για τον σωστό συγχρονισμό πρόσβασης στα δεδομένα αυτά. Συγκεκριμένα, επιβάλλουμε την ατομική πρόσβαση στην μνήμη με το `#pragma omp atomic`, μία φορά κατά το γράψιμο του `newClusterSize`, και μία φορά κατά το γράψιμο του `newClusters`.

Το `#pragma omp atomic` είναι προτιμότερο από το `#pragma omp critical` για απλά memory writes, όπως στην περίπτωση μας, αφού εκμεταλλευόμαστε τα atomic operations που προσφέρει το υλικό, και άρα έχουμε αυξημένη επίδοση. Πραγματοποιούμε μετρήσεις στο μηχάνημα *sandman*, για το configuration $\{\text{Size, Coords, Clusters, Loops}\} = \{256, 16, 16, 10\}$, για threads $= \{1, 2, 4, 8, 16, 32, 64\}$.

Ακολουθούν τα barplot διαγράμματα χρόνου εκτέλεσης και speedup:





Παρατηρούμε ότι η naive υλοποίηση με συγχρονισμό όχι απλά δεν βελτιώνει την επίδοση, αλλά είναι χειρότερη από την σειριακή υλοποίηση, για όλους τους αριθμούς νημάτων. Φαίνεται ότι το κόστος για συγχρονισμό ξεπερνάει το όποιο όφελος μπορεί να είχε η παραλληλία της συγκεκριμένης υλοποίησης.

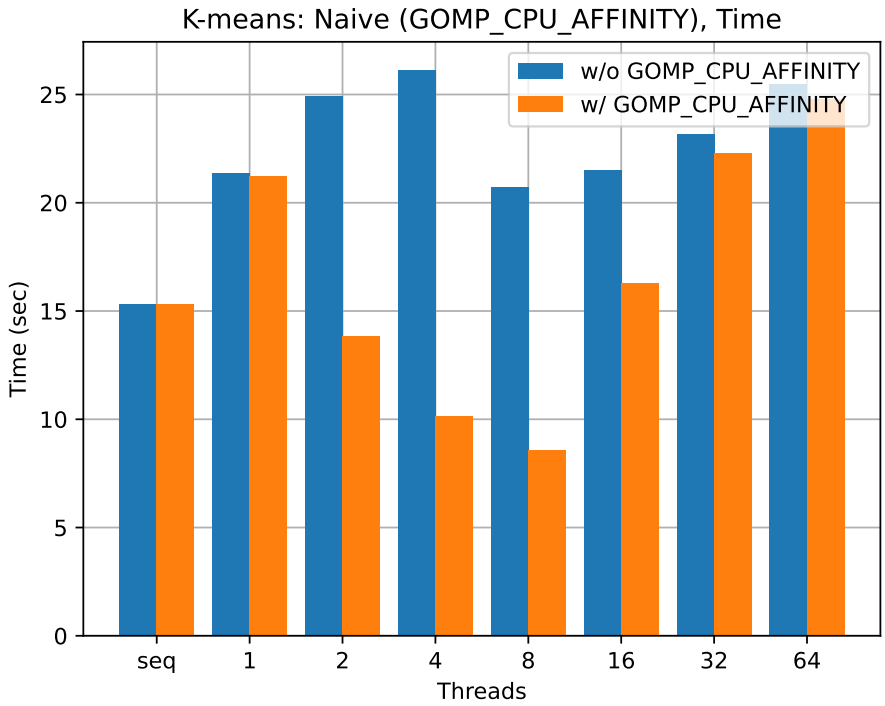
2. Σε αυτό το ερώτημα χρησιμοποιούμε τη μεταβλητή περιβάλλοντος `GOMP_CPU_AFFINITY` ως εξής:

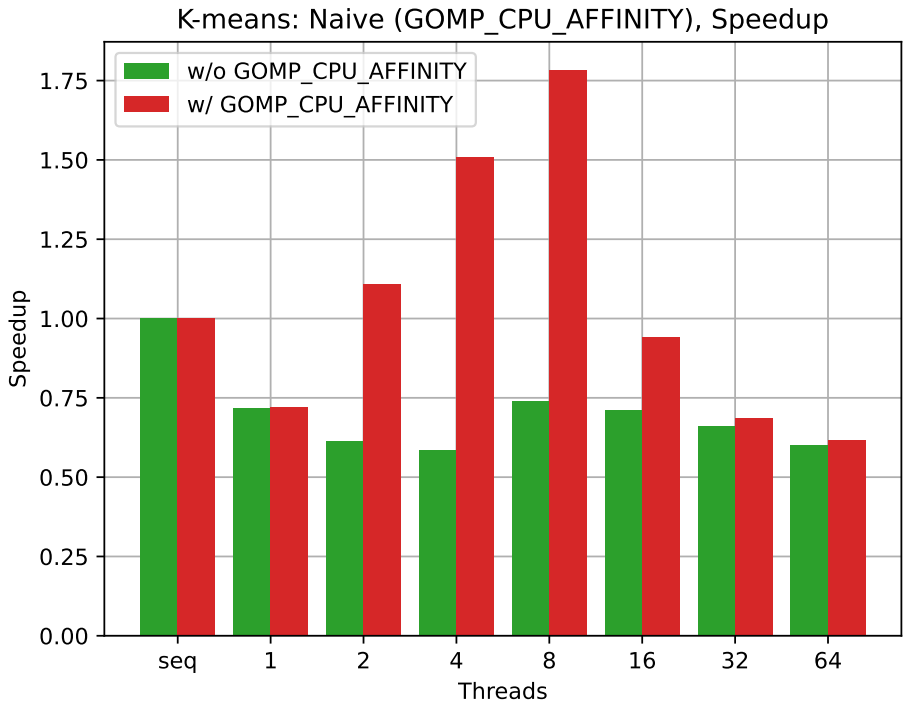
```
GOMP_CPU_AFFINITY="0-$(($threads-1))"
```

δηλαδή η μεταβλητή παίρνει τιμή ανάλογη του αριθμού των νημάτων:

Νήματα	GOMP_CPU_AFFINITY
1	0
2	0-1
4	0-3
8	0-7
16	0-15
32	0-31
64	0-63

Αυτό σημαίνει ότι το OpenMP thread 0 θα πάει στο hardware thread 0, το OpenMP thread 1 θα πάει στο hardware thread 1 κοκ. Το σημαντικό είναι ότι τα νήματα του OpenMP προσδένονται σε πυρήνες για όλη την εκτέλεση, κάτι που ευνοεί την τοπικότητα των δεδομένων, αφού πλέον κάθε νήμα επενεργεί στα δεδομένα του πυρήνα του, και γίνεται καλύτερη χρήση των caches. Πραγματοποιούμε τις προηγούμενες μετρήσεις, αυτή τη φορά χρησιμοποιώντας το `GOMP_CPU_AFFINITY`, και συγκρίνουμε τα αποτελέσματα:





Βλέπουμε ότι η τοπικότητα έχει σηματική επίδραση στην επίδοση, με χαρακτηριστικό παράδειγμα την περίπτωση των 8 threads, όπου πετυχαίνουμε το μέγιστο speedup, της τάξης του 1.75.

Σημειώνουμε ότι από εδώ και στο εξής, για τα ερωτήματα που αφορούν τον αλγόριθμο K-means, η μεταβλητή `GOMP_CPU_AFFINITY` *θα είναι πάντα ορισμένη με τον τρόπο που αναφέραμε προηγουμένως.*

2.1.5 copied clusters and reduce

1. Παραλληλοποιούμε αυτή την έκδοση του αλγορίθμου, αρχικά αγνοώντας τα hints που αφορούν το false-sharing. Για την παραλληλοποίηση χρησιμοποιούμε πάλι parallel for:

```
#pragma omp parallel for private(index, i, j, k)
```

όπου οι μεταβλητές `index, i, j` και `k` που χρησιμοποιούμε για το indexing των πινάκων είναι private, για τους ίδιους λόγους που εξηγήσαμε παραπάνω.
Όπως αναφέραμε παραπάνω, στη δεύτερη υλοποίηση δεν χρησιμοποιούμε εντολές συγχρονισμού, αλλά κάθε νήμα επενεργεί σε μία δικιά του (αντιγραμμένη) έκδοση των global πινάκων `newClusterSize` και `newClusters`. Για τον λόγο αυτό εισάγουμε τους πίνακες:

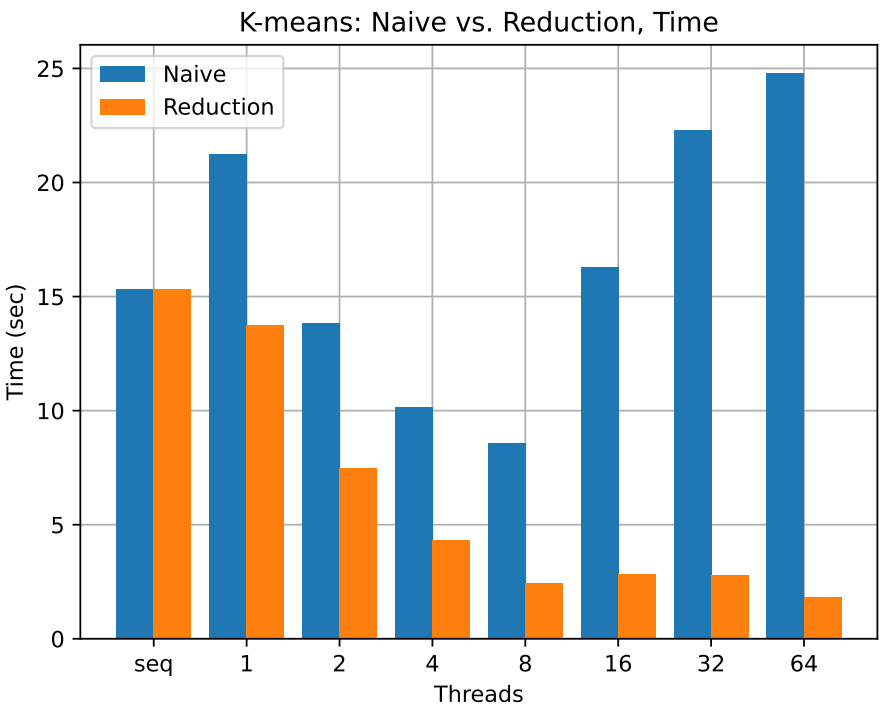
- `local_newClusterSize[nthreads][numClusters]`
- `local_newClusters[nthreads][numClusters][numCoords]`

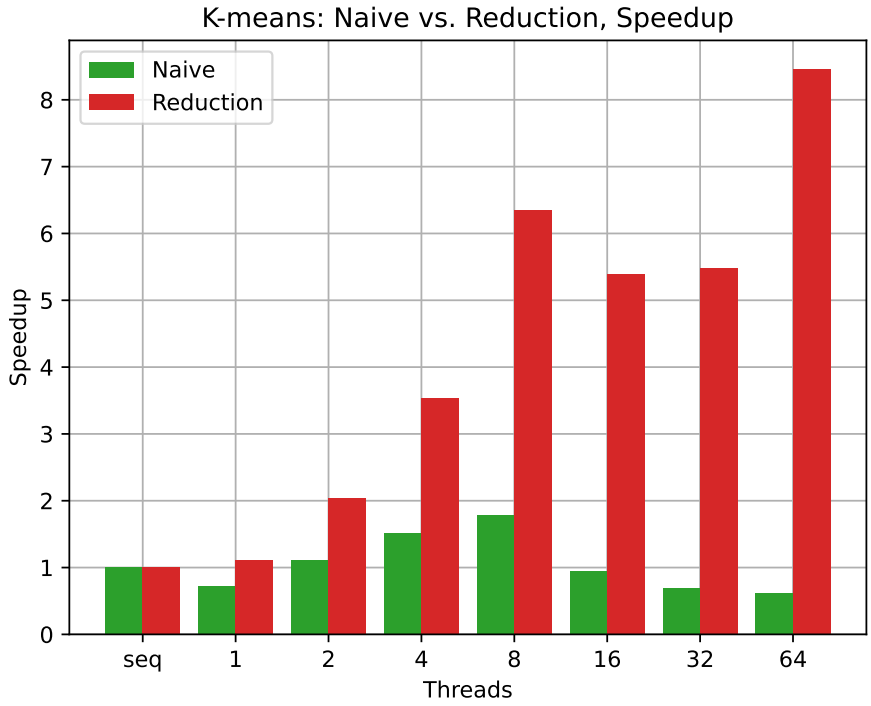
Οι πίνακες αυτοί πρέπει κάθε φορά να μηδενίζονται, όπως γινόταν για τους πίνακες `newClusterSize` και `newClusters` στην naive υλοποίηση. Για το μηδενισμό αυτό χρησιμοποιούμε το `#pragma omp parallel`. Στο τέλος της κάθε επανάληψης του αλγορίθμου, ένα νήμα αναλαμβάνει να συνδυάσει τα επιμέρους αποτελέσματα (reduction) και να τα ενσωματώσει στους global πίνακες.

Τα βασικά σημεία της υλοποίησης αποτυπώνονται ακολούθως:

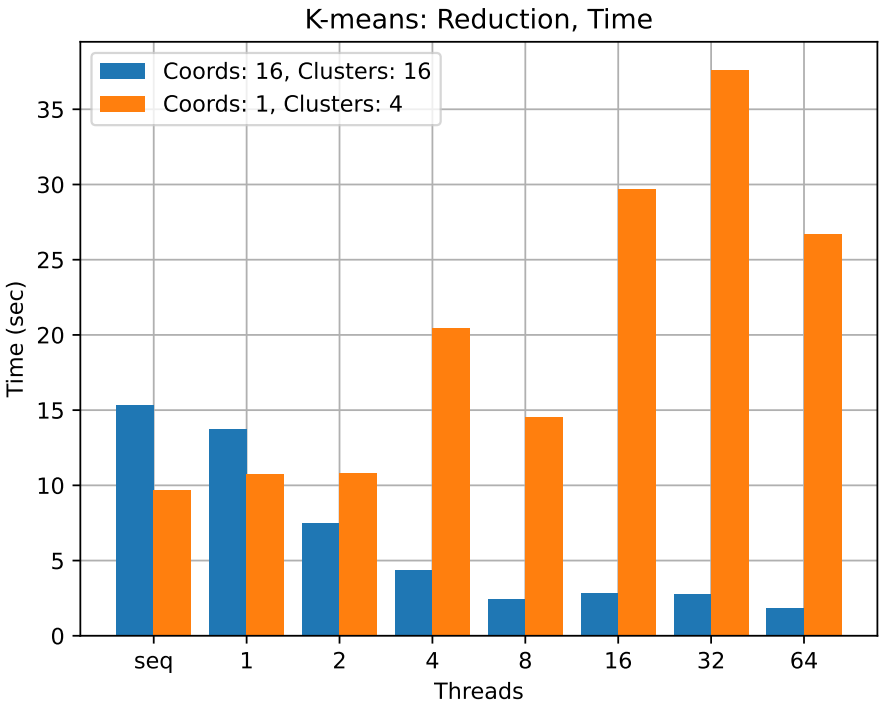
```
// Initiliaze local cluster data to zero (separate for each thread)
#pragma omp parallel private(i, j, k)
{
  for (i=0; i<numClusters; i++) {
    k = omp_get_thread_num();

    for (j=0; j<numCoords; j++)
      local_newClusters[k][i * numCoords + j] = 0.0;

    local_newClusterSize[k][i] = 0;
  }
}

#pragma omp parallel for private(index, i, j, k)
for (i=0; i<numObjs; i++) {
  index = find_nearest_cluster(
    numClusters,
    numCoords,
    &objects[i*numCoords],
    clusters
  );

  /* ... */

  // Collect cluster data in local arrays (local to each thread)
  k = omp_get_thread_num();
  local_newClusterSize[k][index]++;

  for (j=0; j<numCoords; j++)
    local_newClusters[k][index*numCoords + j] += objects[i*numCoords + j];
}

// Reduction of cluster data from local arrays to shared.
// This operation will be performed by one thread
for (k=0; k<nthreads; k++) {
  for (i=0; i<numClusters; i++) {
    newClusterSize[i] += local_newClusterSize[k][i];

    for (j=0; j<numCoords; j++)
      newClusters[i*numCoords + j] += local_newClusters[k][i*numCoords + j];
  }
}
```

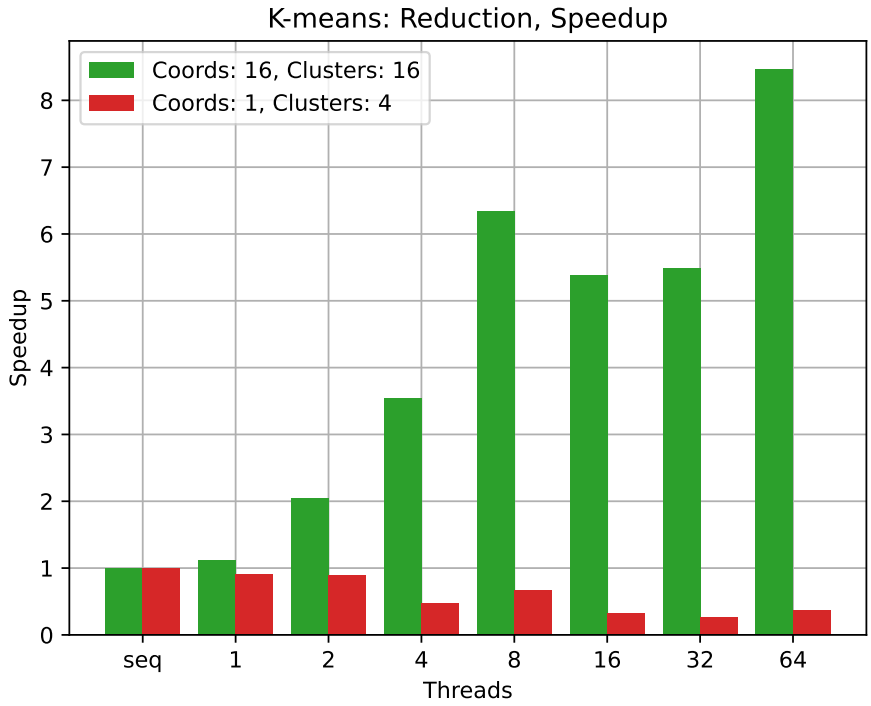
Είναι σαφές ότι με αυτόν τον τρόπο οι απαιτήσεις σε μνήμη πολλαπλασιάζονται κατά έναν παράγοντα nthreads, όμως αναμένουμε να κερδίσουμε σε ταχύτητα, αφού δεν απαιτείται πλέον συγχρονισμός μεταξύ των νημάτων. Πράγματι, όπως φαίνεται και από τα παρακάτω διαγράμματα, η δεύτερη υλοποίηση είναι σημαντικά καλύτερη από την naive, για όλους τους αριθμούς νημάτων.





2. Αυτή τη φορά δοκιμάζουμε το configuration $\{\text{Size, Coords, Clusters, Loops}\} = \{256, 1, 4, 10\}$ και συγκρίνουμε με τις προηγούμενες μετρήσεις:





Παρατηρούμε ότι για το configuration $\{256, 1, 4, 10\}$ η επίδοση είναι σημαντικά χειρότερη, και μάλιστα χειρότερη από την σειριακή. Αυτό συμβαίνει διότι η τοποθέτηση των δεδομένων στις μνήμες των νημάτων γίνεται με μη-βέλτιστο τρόπο. Συγκεκριμένα:

- Μόνο ένα νήμα αναλαμβάνει να μηδενίσει την μνήμη αρχικά, και άρα όλη η μνήμη που αντιστοιχεί στον πίνακα `local_newClusters` πηγαίνει στο node όπου βρίσκεται το συγκεκριμένο νήμα λόγω της πολιτικής first-touch του Linux. Αυτό καθυστερεί τα υπόλοιπα νήματα που δεν βρίσκονται στο ίδιο node, καθώς χρειάζονται περισσότερο χρόνο για να αποκτήσουν πρόσβαση στην μνήμη που τους "αναλογεί" (και κανονικά θα έπρεπε να βρίσκεται πιο κοντά σε αυτά)

- Το μέγεθος των blocks του πίνακα `local_numClusters` που χρησιμοποιούνται από διαφορετικά threads (`numCoords`×`numClusters`×`sizeof(float)`) δεν είναι πλέον 1024 bytes, αλλά 16, και έτσι χωράνε >1 blocks σε μία cache line. Μπορούμε να βρούμε το μέγεθος της cache line τρέχοντας στον *sandman* την παρακάτω εντολή, η οποία μας επιστρέφει την τιμή 64:

  ```
  cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
  ```

  Αυτό σημαίνει ότι εμφανίζονται φαινόμενα false sharing (δηλαδή συνεχές invalidation για δεδομένα που δεν είναι πραγματικά μοιραζόμενα, απλώς τυχαίνει να είναι στην ίδια cache line).

Για να αντιμετωπίσουμε τα παραπάνω προβλήματα, τροποποιούμε την υλοποίησή μας και αλλάζουμε το παρακάτω for loop, από:

```
for (k=0; k<nthreads; k++) {
  local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(
    numClusters,
    sizeof(**local_newClusterSize)
  );

  local_newClusters[k] = (typeof(*local_newClusters)) calloc(
    numClusters * numCoords,
    sizeof(**local_newClusters)
  );
}
```

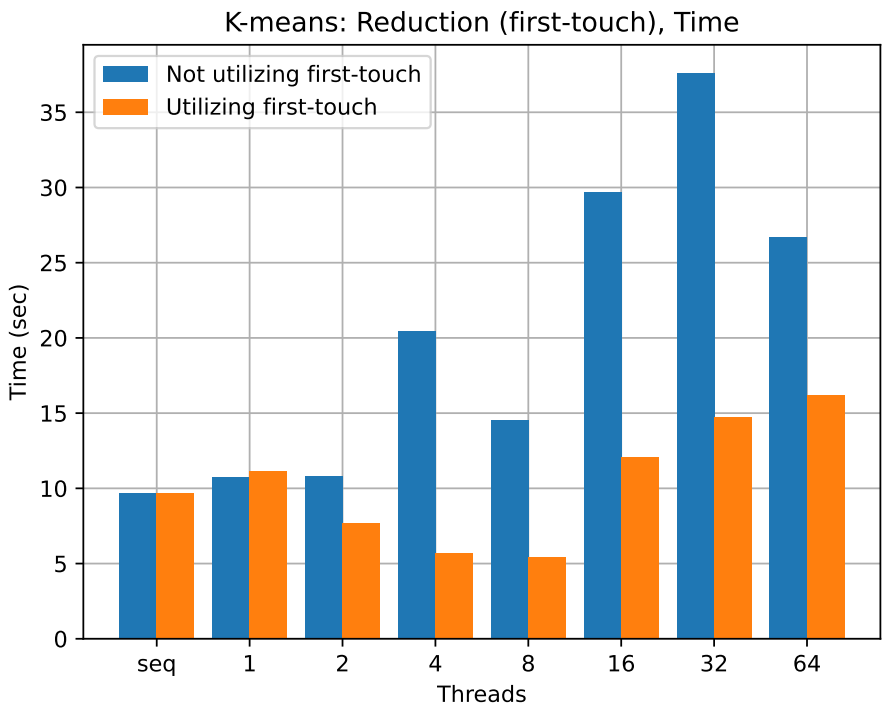
σε:

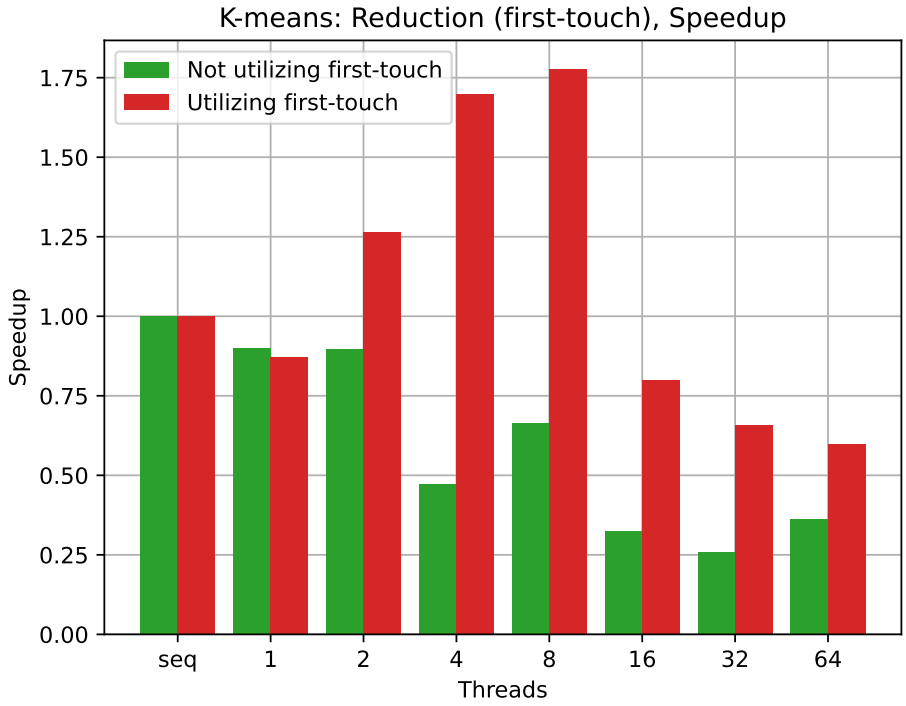
```
#pragma omp parallel private(k)
{
  k = omp_get_thread_num();
  local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(
    numClusters,
    sizeof(**local_newClusterSize)
  );

  local_newClusters[k] = (typeof(*local_newClusters)) calloc(
    numClusters * numCoords,
    sizeof(**local_newClusters)
  );
}
```

Με αυτόν τον τρόπο, κάθε thread μηδενίζει τη μνήμη που του αντιστοιχεί, και διασφαλίζει ότι η μνήμη αυτή (σε granularity σελίδας) θα έρθει στο NUMA node του κάθε thread. Επιπλέον, κατά την αρχικοποίηση, τα δεδομένα έρχονται στην cache του πυρήνα στον οποίο τρέχει το κάθε νήμα, και μειώνεται το false sharing.

Η τροποποιημένη υλοποίηση βρίσκεται στο αρχείο `omp_reduction_kmeans_fs.c`.

Επαναλαμβάνουμε τις μετρήσεις και συγκρίνουμε με την αρχική υλοποίηση:





Είναι εμφανές ότι η επίδοση της "First-touch-aware" υλοποίησης είναι σημαντικά καλύτερη. Ο καλύτερος χρόνος που πετυχαίνουμε είναι $\sim$5 sec (για 8 threads), ενώ με την απλή reduction υλοποίηση πετυχαίνουμε $\sim$10 sec (για 1-2 threads).

3. Εδώ προσπαθούμε να εκμεταλλευτούμε τα NUMA χαρακτηριστικά του μηχανήματος, όσον αφορά τον διαμοιρασμό του πίνακα `objects` στα memory nodes. Συγκεκριμένα, ο σκοπός μας είναι ο πίνακας `objects` να μοιραστεί όσο το δυνατόν γίνεται κοντά στα threads που θα εκτελέσουν memory reads πάνω του. Έχοντας προσδέσει τα threads σε πυρήνες με τη μεταβλητή `GOMP_CPU_AFFINITY`, και

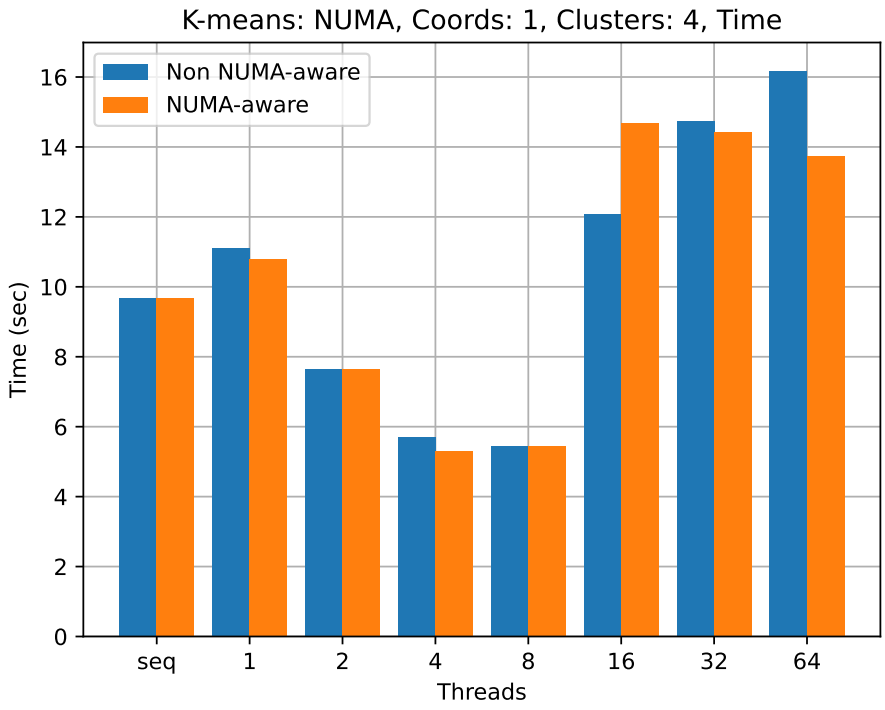
λόγω του static scheduling του parallel for που έχουμε ορίσει, μπορούμε να πούμε ότι ο διαμοιρασμός των δεδομένων θα γίνει κατά προσέγγιση ως εξής:

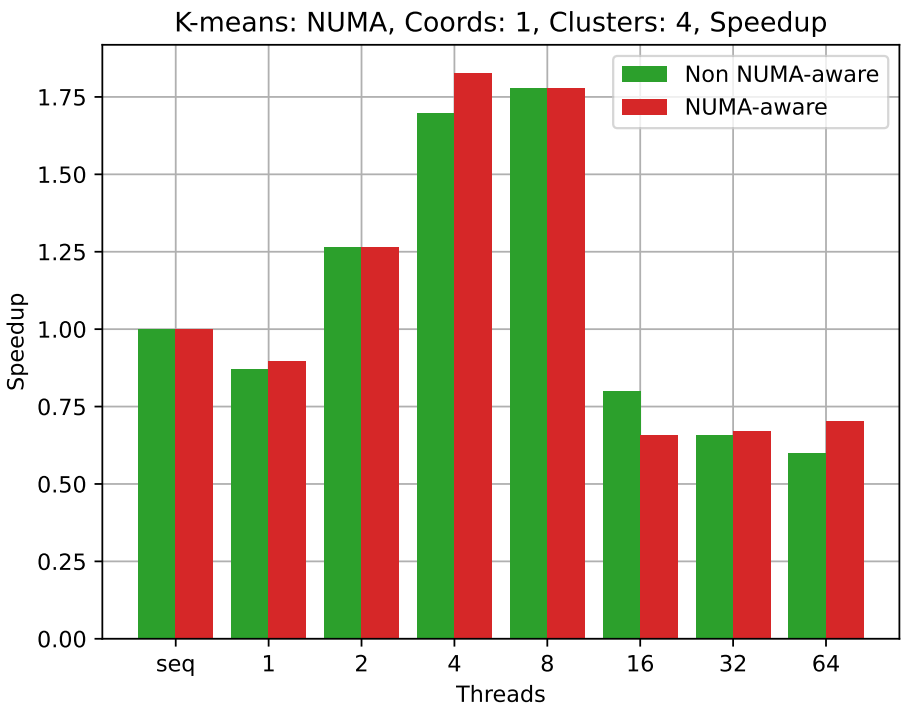
- Θέσεις 0 μέχρι `numObjs/nthreads` → 1ο thread
- Θέσεις `numObjs/nthreads` μέχρι 2×`numObjs/nthreads` → 2ο thread
- Θέσεις 2×`numObjs/nthreads` μέχρι 3×`numObjs/nthreads` → 3ο thread
- ...κοκ

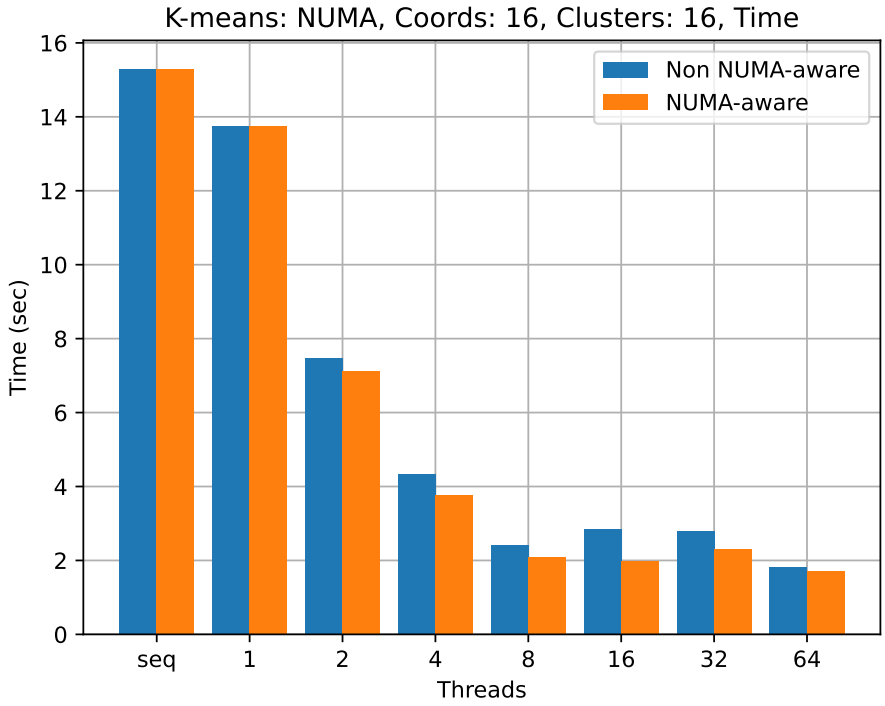
Έτσι, αν εκμεταλλευτούμε την πολιτική first-touch, μπορούμε να τροποποιήσουμε το `file_io.c` ως εξής:

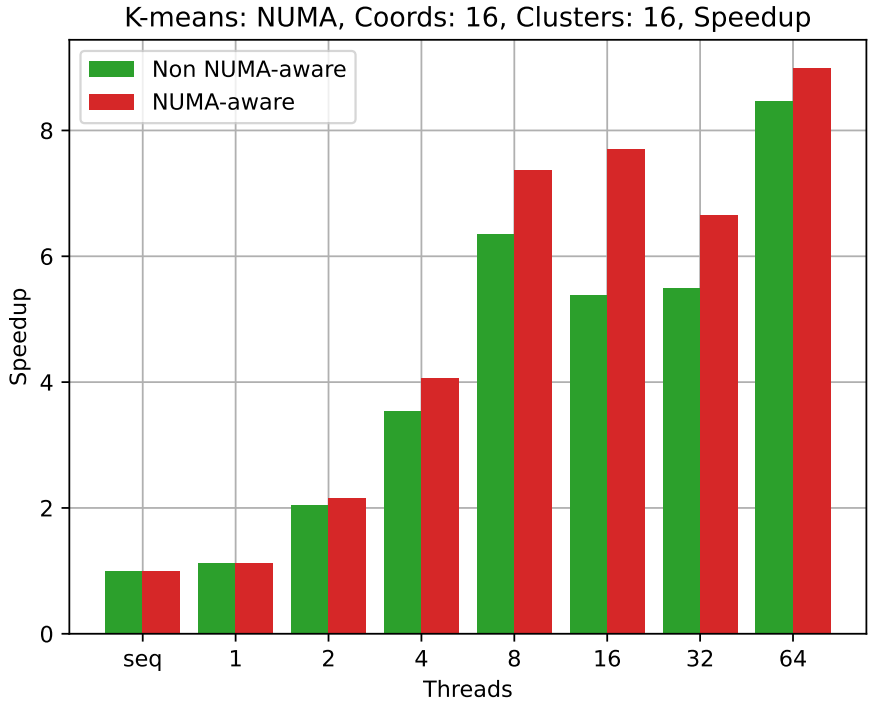
```
#pragma omp parallel for private(i, j)
for (i=0; i<numObjs; i++) {
  unsigned int seed = i;
  for (j=0; j<numCoords; j++) {
    objects[i*numCoords + j] = (rand_r(&seed) / ((float) RAND_MAX)) * val_range;
    /* ... */
  }
}
```

Με αυτόν τον τρόπο, κάθε thread κάνει touch τα δεδομένα που το αφορούν, και τα οποία θα ανατεθούν στο NUMA node του πυρήνα του. Εκτελούμε μετρήσεις για τα configuration {Size, Coords, Clusters, Loops} = $\{256, 1, 4, 10\}$ και $\{256, 16, 16, 10\}$:




K-means: NUMA, Coords: 1, Clusters: 4, Speedup
Non NUMA-aware
NUMA-aware
Speedup
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
seq
1
2
4
8
16
32
64
Threads


K-means: NUMA, Coords: 16, Clusters: 16, Time
Non NUMA-aware
NUMA-aware
Time (sec)
0
2
4
6
8
10
12
14
16
seq
1
2
4
8
16
32
64
Threads



Παρατηρούμε ότι για το configuration $\{256, 1, 4, 10\}$ υπάρχει πολύ μικρή βελτίωση σε σχέση με πριν, εκτός από την περίπτωση των 16 threads, όπου η NUMA-aware υλοποίηση έχει σημαντική καθυστέρηση της τάξης των ~ 3 δευτερολέπτων. Για το configuration $\{256, 16, 16, 10\}$ υπάρχει παντού μια μικρή βελτίωση, της τάξης των 0.5-1 δευτερολέπτων. Κάτι τέτοιο μας δείχνει ότι το κυρίαρχο bottleneck δεν είναι η ανομοιόμορφη πρόσβαση στην μνήμη, αλλά το κόστος των υπολογισμών (compute bound).

2.1.6 Αμοιβαίος Αποκλεισμός - Κλειδώματα

Σε αυτό το σημείο θα αξιολογήσουμε διαφορετικές υλοποιήσεις κλειδωμάτων για αμοιβαίο αποκλεισμό. Γι' αυτόν το σκοπό θα επιστρέψουμε στην *shared clusters* υλοποίηση του K-means. Αυτή τη φορά αντί για `#pragma omp atomic` θα χρησιμοποιηθούν τα εξής είδη κλειδωμάτων:

- **nosync_lock**: Εδώ δεν παρέχεται αμοιβαίος αποκλεισμός, οπότε και τα αποτελέσματα δεν θα είναι τα σωστά. Χρησιμοποιείται μόνο ως μέτρο σύγκρισης για την αξιολόγηση των άλλων κλειδωμάτων, αφού δεν έχει καθόλου overhead, και άρα αναμένουμε ο χρόνος εκτέλεσης να είναι ο ελάχιστος.
- **pthread_mutex_lock**: Το `pthread_mutex_t` κλείδωμα που παρέχει η βιβλιοθήκη `pthreads`.
- **pthread_spin_lock**: Το `pthread_spinlock_t` κλείδωμα που παρέχει η βιβλιοθήκη `pthreads`.
- **tas_lock**: Το *test-and-set* κλείδωμα, όπως έχει περιγραφεί στο μάθημα.
- **ttas_lock**: Το *test-and-test-and-set* κλείδωμα, όπως έχει περιγραφεί στο μάθημα.
- **array_lock**: Το array-based κλείδωμα, όπως έχει περιγραφεί στο μάθημα.
- **clh_lock**: Ένα είδος κλειδώματος που βασίζεται στην χρήση μιας συνδεδεμένης λίστας. Λεπτομέρειες στο βιβλίο The Art of Multiprocessor Programming, Κεφάλαιο 7.

Επιπλέον, έχουμε και μια υλοποίηση που χρησιμοποιεί και το `#pragma omp critical` για την προστασία του κρίσιμου τμήματος.

Συνολικά, θα χρησιμοποιηθούν τα εξής αρχεία:

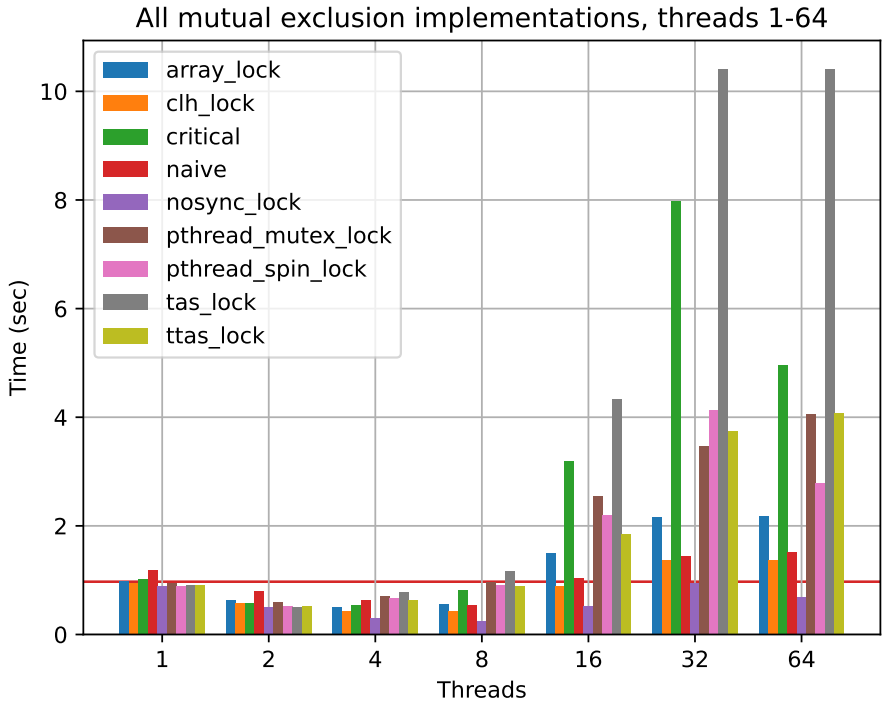
- `omp_naive_kmeans.c`: Χρησιμοποιείται το `#pragma omp atomic` για την προστασία των διαμοιραζόμενων δεδομένων.
- `omp_lock_kmeans.c`: Χρησιμοποιούνται τα διάφορα κλειδώματα για την προστασία του κρίσιμου τμήματος.
- `omp_critical_kmeans.c`: Χρησιμοποιείται το `#pragma omp critical` για την προστασία του κρίσιμου τμήματος.

Το `Makefile` ενημερώνεται ώστε συνολικά να παράγει τα εκτελέσιμα:

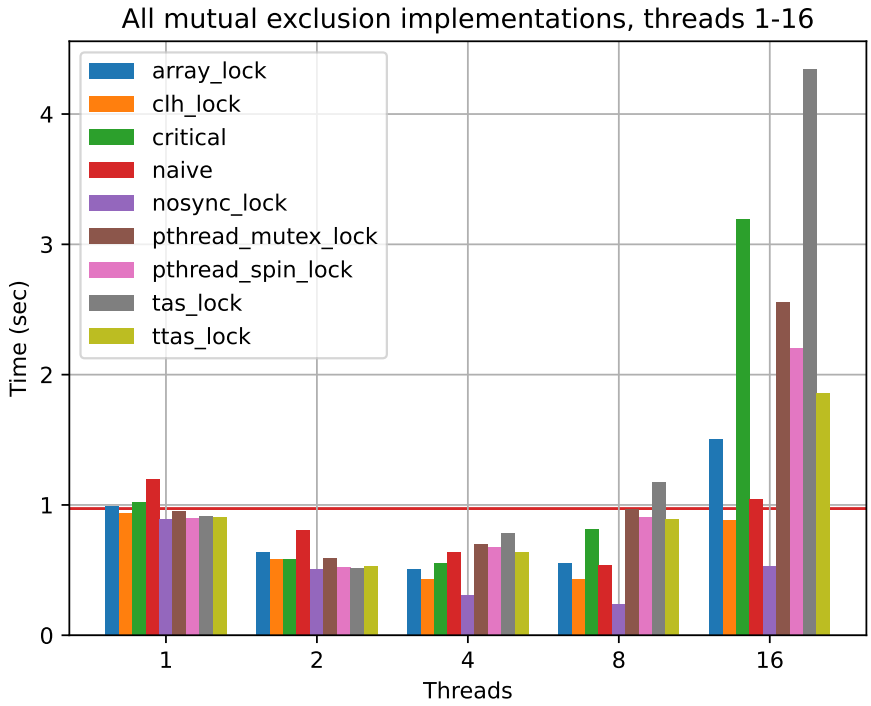
- `kmeans_seq`: Η σειριακή υλοποίηση.
- `kmeans_omp_naive`: Η υλοποίηση με `#pragma omp atomic`.
- `kmeans_omp_critical`: Η υλοποίηση με `#pragma omp critical`.
- `kmeans_omp_<lock-type>_lock`: Ένα εκτελέσιμο για κάθε τύπο κλειδώματος.

Εκτελούμε μετρήσεις για κάθε εκτελέσιμο, για το configuration {Size, Coords, Clusters, Loops} = $\{16, 16, 16, 10\}$ για threads = $\{1, 2, 4, 8, 16, 32, 64\}$ στο μηχάνημα *sandman*. Υπενθυμίζουμε ότι, όπως και παραπάνω, εφαρμόζουμε thread-binding μέσω της μεταβλητής `GOMP_CPU_AFFINITY`.

Σημείωση: Στις μετρήσεις που ακολουθούν, η κόκκινη οριζόντια γραμμή υποδηλώνει κάθε φορά τον σειριακό χρόνο εκτέλεσης, δηλαδή τον χρόνο που χρειάζεται το εκτελέσιμο `kmeans_seq` για το ίδιο configuration.



Βλέπουμε ότι για 32 και 64 νήματα καμία υλοποίηση δεν είναι καλύτερη από την σειριακή, εκτός από την `nosync_lock`, επομένως για λόγους ευκρίνειας δείχνουμε τις ίδιες μετρήσεις για 1-16 νήματα:



Σχολιασμός

Ταξινομούμε τις υλοποιήσεις ως προς τον χρόνο εκτέλεσης για 64 threads, από την χειρότερη στην καλύτερη:

1. **tas_lock**: Στο *test-and-set* κλείδωμα, το νήμα λειτουργεί ατομικά πάνω σε 2 μεταβλητές (state και test). Θέτει το state στην τιμή 1 και μεταφέρει την προηγούμενη τιμή του state στο test. Αν η τιμή του test είναι 0, τότε το νήμα μπορεί να μπει στο κρίσιμο τμήμα, αλλιώς πρέπει να επαναλάβει την διαδικασία. Είναι εμφανές ότι αυτή η διαδικασία εισάγει πολλή περιττή κίνηση στον διάδρομο, αφού το *set* κομμάτι του *test-and-set* γίνεται ακόμα κι αν το νήμα δεν καταφέρει να πάρει το κλείδωμα. Αν λάβουμε υπόψιν και το πρωτόκολλο συνάφειας κοινής μνήμης, έχουμε μεγάλο αριθμό από invalidations και η επίδοση χειροτερεύει επικίνδυνα. Αυτό είναι εμφανές από τις μετρήσεις, όπου το *test-and-set* κλείδωμα έχει σταθερά την χειρότερη επίδοση, για 4-64 νήματα. Το μόνο θετικό είναι ότι υλοποιείται εύκολα.

2. **critical**: Στην περίπτωση που χρησιμοποιούμε το `#pragma omp critical`, απλώς περικλείουμε το κρίσιμο τμήμα σε ένα block κώδικα και αφήνουμε το OpenMP να διαχειριστεί τις λεπτομέρειες της υλοποίησης. Αυτό έχει το εμφανές πλεονέκτημα της προγραμματιστικής ευκολίας, αφού δεν χρειάζεται καν να επιλέξουμε ποιο είδος κλειδώματος χρειαζόμαστε, πόσο μάλλον να υλοποιήσουμε κάποιο δικό μας κλείδωμα. Μειονέκτημα της συγκεκριμένης επιλογής είναι η έλλειψη ελέγχου ως προς το κλείδωμα που θα χρησιμοποιήσουμε. Όπως δείχνουν οι μετρήσεις μας, η κλιμακωσιμότητα αυτής της υλοποίησης δεν είναι καθόλου καλή, αφού από τα 16 νήματα και πάνω οι χρόνοι εκτέλεσης είναι απαγορευτικά μεγάλοι.

3. **ttas_lock**: Το *test-and-test-and-set* κλείδωμα αποτελεί μία βελτιστοποίηση του *test-and-set*, αφού αρχικά γίνεται μόνο ανάγνωση της κατάστασης του κλειδώματος (στην local cache), και μόνο όταν το κλείδωμα είναι ελεύθερο ξεκινά η διεκδίκηση (και άρα γράψιμο) της

μεταβλητής state. Αποφεύγονται έτσι τα προβλήματα της υπερβολικής χρήσης του διαδρόμου και των επάλληλων invalidations. Η βελτιστοποίηση είναι εμφανής σε σχέση με το *test-and-set* κλείδωμα, ειδικά όσο ο αριθμός των νημάτων αυξάνεται. Βλέπουμε ότι για 16 νήματα ο χρόνος εκτέλεσης είναι ο μισός σε σχέση με πριν, με τη διαφορά να αυξάνεται ακόμα περισσότερο στα 32 και 64 νήματα. Να επισημάνουμε ότι τόσο το *ttas_lock* όσο και το *tas_lock* δεν είναι starvation free. Τέλος, με προσθήκη exponential backoff μπορούμε να μειώσουμε περισσότερο τον συνωστισμό στον διάδρομο κατά την διάρκεια της διεκδίκησης, ώστε να βελτιώσουμε ακόμη περισσότερο την απόδοση του.

4. **pthread_mutex_lock**: Τα κλειδώματα τύπου mutex της βιβλιοθήκης `pthreads` είναι από τα πιο γνωστά, καθώς η υλοποίηση τους είναι αρκετά απλή. Η λειτουργία του κλειδώματος είναι η εξής: Αν το κρίσιμο τμήμα είναι κατειλημμένο, το νήμα αναστέλλει την λειτουργία του (κοιμάται) και τίθεται υπό αναμονή, δηλαδή ο επεξεργαστής ελευθερώνεται και επιλέγεται ένα άλλο νήμα για εκτέλεση. Κατά την έξοδο από το κρίσιμο τμήμα, ενεργοποιούνται τα νήματα που βρίσκονται σε αναμονή.
   Το πρόβλημα που προκύπτει, όταν χρησιμοποιούμε κλείδωμα τύπου `mutex` είναι ότι και η διαδικασία για να αναστείλουμε ένα και να ενεργοποιήσουμε ένα νήμα είναι αρκετά απαιτητική σε κύκλους επεξεργαστή. Το πρόβλημα αυτό διογκώνεται και γίνεται ιδιαίτερα φανερό, όταν το κλείδωμα δεσμεύεται από ένα νήμα μόνο για ένα μικρό χρονικό διάστημα. Αυτό συμβαίνει, επειδή ο χρόνος που ξοδεύουμε για την αναστολή και ενεργοποίηση του νήματος, μπορεί να είναι μεγαλύτερος σε σύγκριση με το να υλοποιούσαμε ενεργό αναμονή. Τα παραπάνω μπορούν να επιβεβαιωθούν και από την γραφική αναπαράσταση, καθώς παρατηρούμε ότι όσο πληθαίνει ο αριθμός των νημάτων, ο χρόνος εκτέλεσης αυξάνεται. Στην περίπτωση μας το κρίσιμο τμήμα είναι αρκετά μικρό με αποτέλεσμα να φαίνονται ξεκάθαρα οι συνέπειες της επιλογής του `mutex` κλειδώματος. Το κλείδωμα αυτό φαίνεται να είναι κατάλληλο για μεγάλα διαστήματα αναμονής.

5. **pthread_spin_lock**: Σε αντίθεση με το παραπάνω κλείδωμα αναστολής εκτέλεσης, το κλείδωμα τύπου `spinlock`, επίσης της βιβλιοθήκης `pthreads`, λειτουργεί με ενεργό αναμονή. Αυτό σημαίνει, πως όταν το κρίσιμο τμήμα (KT) είναι κατειλημμένο, δηλαδή κάποιο άλλο νήμα βρίσκεται εκεί, το νήμα καταναλώνει υπολογιστικούς πόρους (CPU time), περιμένει, μέχρι να πάρει το κλείδωμα. Τα παραπάνω χαρακτηριστικά αναδεικνύουν το κλείδωμα αυτό, ως κατάλληλο για μικρά διαστήματα αναμονής. Ωστόσο, το busy-wait που γίνεται μέχρι να ληφθεί το κλείδωμα, που συνεπάγεται με την σπατάλη σε χρόνο CPU, το καθιστά χειρότερο σε σχέση με άλλα κλειδώματα.

6. **array_lock**: Το συγκεκριμένο κλείδωμα προσπαθεί να αντιμετωπίσει την καθυστέρηση που εμφανίζεται όταν πολλά νήματα δουλεύουν πάνω σε μία κοινή θέση μνήμης, και η οποία οφείλεται στη μεγάλη κυκλοφορία δεδομένων από το σύστημα συνάφειας κρυφής μνήμης. Η υλοποίηση αυτή εξασφαλίζει ότι κάθε νήμα θα επενεργεί πάνω σε διαφορετικά δεδομένα ως εξής: κάθε νήμα που θέλει να μπει στο κρίσιμο τμήμα πηγαίνει στην τελευταία ελεύθερη θέση ενός boolean πίνακα, και περιμένει μέχρι η τιμή στην θέση αυτή να γίνει `true`. Η τιμή αυτή θα γίνει `true` όταν το νήμα που ελέγχει την προηγούμενη θέση του πίνακα βγει από το κρίσιμο τμήμα. Αυτή η λύση αντιμετωπίζει το φαινόμενο που αναφέρθηκε παραπάνω, ωστόσο έχει μεγαλύτερες απαιτήσεις σε μνήμη (ένας πίνακας ανά κλείδωμα), και επιπλέον είναι επιρρεπής στο *false sharing*. Όπως φαίνεται και στις μετρήσεις, έχει αρκετά καλή επίδοση για μικρό αριθμό νημάτων, υστερώντας μόνο από τις υλοποιήσεις `naive` και `clh_lock`.

7. **naive**: Αυτή είναι η γνωστή υλοποίηση που χρησιμοποιεί το `#pragma omp atomic`. Όπως έχουμε ήδη αναφέρει, η χρήση του συγκεκριμένου directive είναι ιδανική για περιπτώσεις όπως αυτή που εξετάζουμε (απλά memory writes), και έχει πολύ καλή επίδοση, την οποία ξεπερνά μόνο το `clh_lock`.

8. **clh_lock**: Με βάση τις μετρήσεις που πραγματοποιήσαμε το `clh_lock` φαίνεται να είναι το κλείδωμα με την καλύτερη επίδοση. Ο αλγόριθμος αυτός εγγυάται ότι δεν θα υπάρξει *starvation* και είναι τύπου *first-come-first-served*. Το συγκεκριμένο κλείδωμα ξεπερνάει την απόδοση του *array lock*, το οποίο δεν είναι space-efficient, αφού χρειάζεται εκ των προτέρων να γνωρίζει το πλήθος των νημάτων n, ώστε για κάθε νήμα κάνει allocate έναν πίνακα μεγέθους n. Γενικότερα, το `clh_lock` χαρακτηρίζεται από μειωμένη κίνηση στο δίαυλο της μνήμης, αφού κάθε thread κάνει spin σε τοπικά αποθηκευμένες μεταβλητές, πιο συγκεκριμένα αναμένει ενεργά στο node του προγόνου του μέχρι αυτό να γίνει `False`. Από τις μετρήσεις, το κλείδωμα εμφανίζεται να υπερτερεί σχεδόν σε όλες τις περιπτώσεις, χωρίς και πάλι να αποφεύγει την πτώση της επίδοσης με την αύξηση των νημάτων. Να τονίσουμε ότι τα προτερήματα αυτά παρουσιάζονται, επειδή είμαστε σε cache aware architecture.

9. **nosync_lock**: Όπως είναι προφανές, η `nosync` υλοποίηση είναι η πιο γρήγορη απ' όλες, αφού στην πραγματικότητα δεν υπάρχει καθόλου συγχρονισμός μεταξύ των νημάτων, και άρα τα αποτελέσματα είναι φυσικά λανθασμένα. Ενδιαφέρον παρουσιάζει ότι η υλοποίηση `clh_lock` για 4 και 8 νήματα πετυχαίνει βέλτιστο χρόνο μόλις 40% πιο αργό σε σχέση με την `nosync` (0.43 sec vs. 0.3 sec). Επίσης παρατηρούμε ότι η διαφορά μεταξύ της υλοποίησης χωρίς συγχρονισμό από τις υπόλοιπες υλοποιήσεις αυξάνεται όσο αυξάνεται ο αριθμός των νημάτων, αφού στην μία περίπτωση έχουμε αυξανόμενο ανταγωνισμό στο κρίσιμο τμήμα και στην άλλη όχι.

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

2.2.1 Εισαγωγή/Σκοπός της άσκησης

Σκοπός της παρούσας άσκησης είναι η ανάπτυξη διαφορετικών παράλληλων εκδόσεων του αλγορίθμου Floyd-Warshall, η αξιολόγηση της παραγωγικότητας (productivity) ανάπτυξης παράλληλου κώδικα και της τελικής επίδοσης του παράλληλου προγράμματος. Για την ανάπτυξη των παραπάνω προγραμμάτων θα χρησιμοποιηθεί το εργαλείο OpenMP για αρχιτεκτονικές κοινής μνήμης.

2.2.2 Ο αλγόριθμος Floyd-Warshall

Ο αλγόριθμος Floyd-Warshall υπολογίζει τα ελάχιστα μονοπάτια ανάμεσα σε κάθε ζεύγος κορυφών σε έναν γράφο. Το πρόβλημα είναι να βρούμε τις μικρότερες αποστάσεις μεταξύ κάθε ζεύγους κορυφών σε ένα κατευθυνόμενο γράφημα με βάρη.

Για την ανάπτυξη του αλγόριθμου αυτού, χρησιμοποιούμε έναν πίνακα γειτνίασης A, ο οποίος αναπαριστά τον γράφο. Κάθε στοιχείο A_{ij} αντιπροσωπεύει το βάρος της ακμής μεταξύ των κορυφών i, j αν αυτή υπάρχει. Το βάρος μπορεί να είναι και αρνητικό.

Παρακάτω θα δούμε τρεις διαφορετικές υλοποιήσεις του αλγόριθμου. Αρχικά, έχουμε την βασική naive μέθοδο του αλγορίθμου με πολυπλοκότητα $O\left(N^3\right)$. Πέρα από αυτήν έχουν προταθεί άλλες δύο εκδοχές, μια αναδρομική και μια tiled, που αξιοποιούν καλύτερα την κρυφή μνήμη και είναι πιο αποδοτικές κατά την παραλληλοποίηση.

2.2.3 Δεδομένα

Αντίστοιχα με την υλοποίηση του παράλληλου προγράμματος για το *Game of Life*, στην παρούσα άσκηση θα χρησιμοποιήσουμε μεταβλητές περιβάλλοντος, την βιβλιοθήκη του OpenMP, και τις διάφορες οδηγίες προς τον μεταγλωτιστή για να παραλληλοποιήσουμε τις διάφορες εκδοχές του Floyd-Warshall. Ακολούθως, θέλουμε να αξιολογήσουμε και τις μετρήσεις επίδοσης για μεγέθη πινάκων 1024×1024, 2048×2048 και 4096×4096 για 1, 2, 4, 8, 16, 32 και 64 threads και για

διαφορετικά μεγέθη πινάκων B (για το base case scenario) στο μηχάνημα *sandman*.

2.2.4 Έκδοση 1 - standard αλγόριθμος

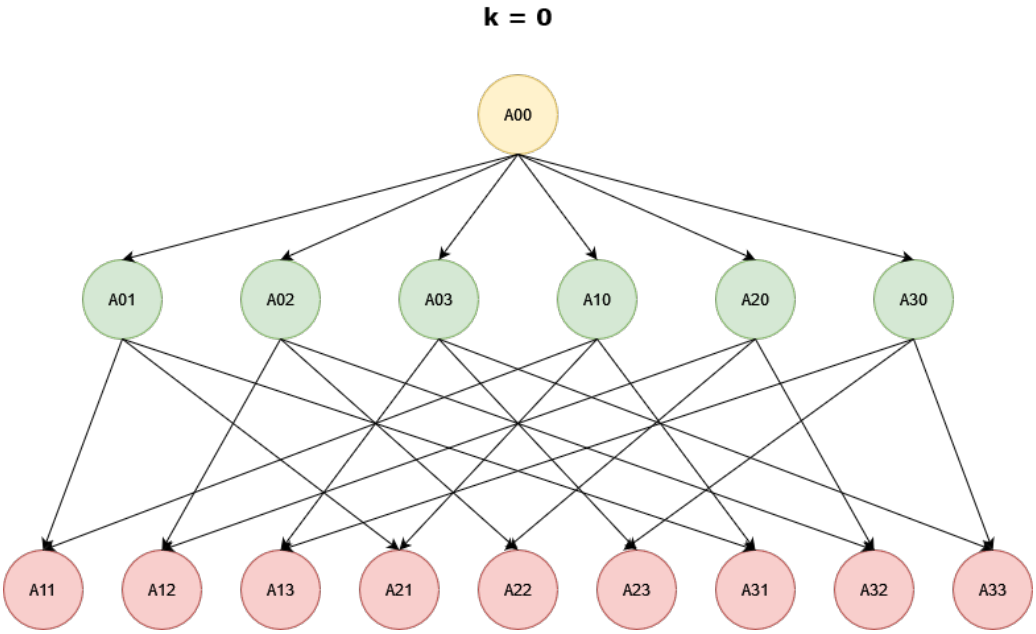
Ο naive αλγόριθμος για την υλοποίηση του Floyd-Warshall είναι ο παρακάτω:

```
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      A[i][j] = min(A[i][j], A[i][k]+A[k][j]);
```

Όπως αναφέρθηκε και παραπάνω, ο πίνακας A είναι ο πίνακας γειτνίασης του γράφου και N είναι ο αριθμός των κόμβων. Ο αλγόριθμος, με βάση τον πίνακα γειτνίασης A, αναζητά τα συντομότερα ζευγάρια κορυφών. Έστω ο παρακάτω πίνακας γειτνίασης A:

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Δημιουργείται ο παρακάτω γράφος εξαρτήσεων:



Η μεταβλητή k αναπαριστά τα χρονικά βήματα ή αλλιώς το πλήθος των ενδιάμεσων κόμβων που παρεμβάλλονται, όταν ελέγχουμε τις συντομότερες αποστάσεις μεταξύ κόμβων.

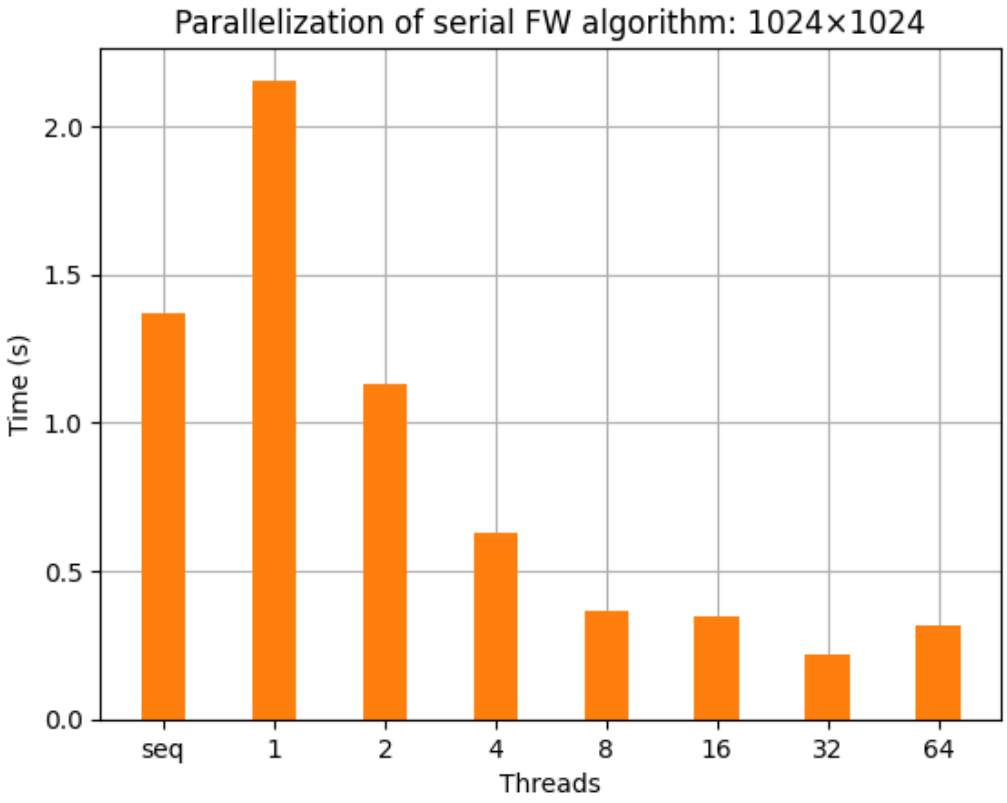
Όπως θα αποδείξουμε και παρακάτω, αυτή η standard υλοποίηση του αλγορίθμου είναι memory bound, καθώς για μεγάλα , δηλαδή για μεγάλους γράφους, ο πίνακας γειτνίασης δεν χωράει στην cache. Αυτό έχει ως αποτέλσμα να εισάγονται μεγάλες καθυστερήσεις. Οι καθυστερήσεις αυτές οφείλονται στον χρόνο που απαιτείται για την μεταφορά των στοιχείων του πίνακα από την κύρια μνήμη στην cache σε κάθε χρονική επανάληψη k.

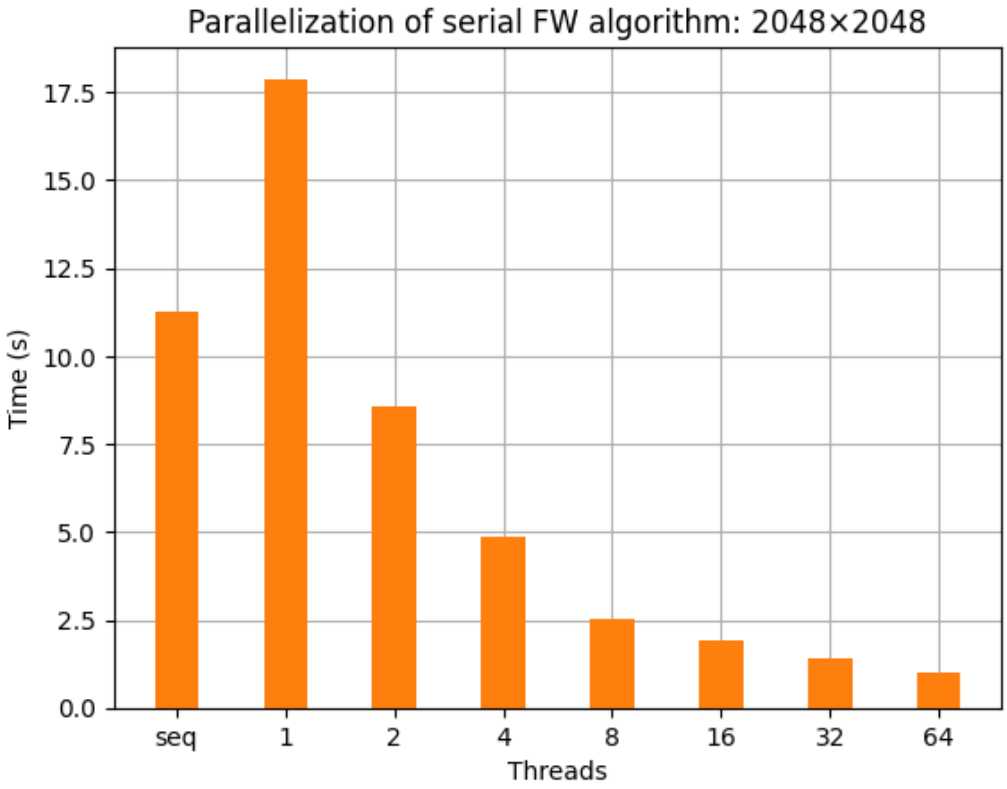
Θα μπορούσαμε να παραλληλοποιήσουμε τον αλγόριθμο αυτό, ώστε να βελτιωθεί η απόδοση του. Παρακάτω μπορούμε να δούμε μια πιθανή υλοποίηση του:

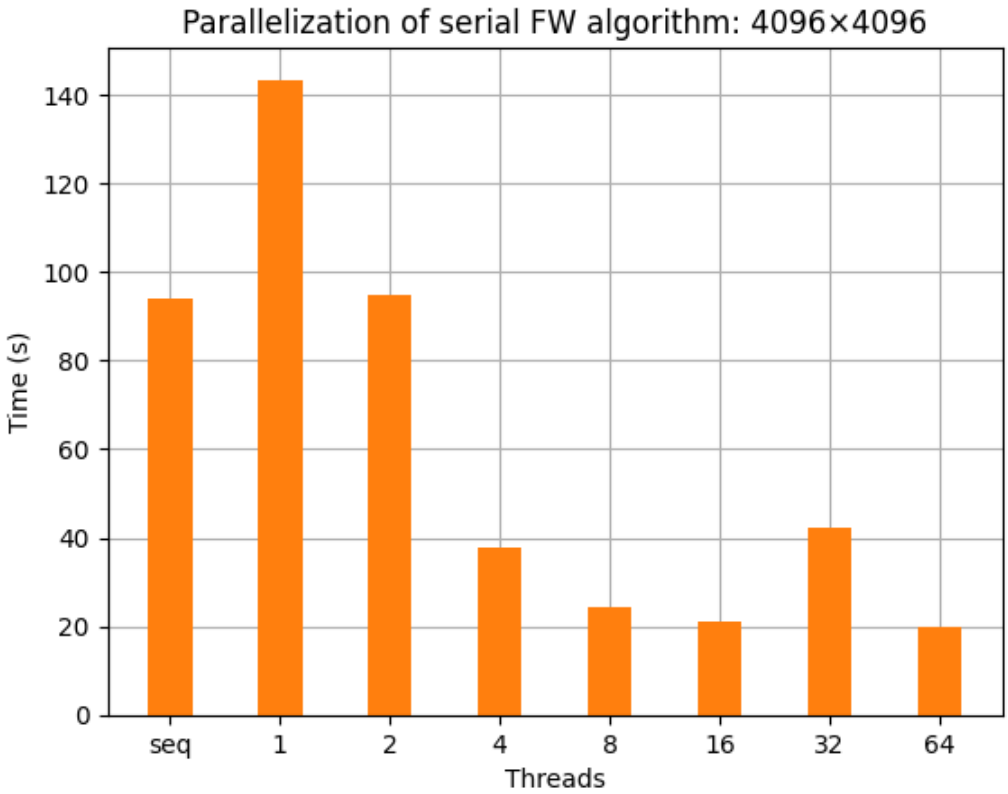
```
for (k=0; k<N; k++)
  #pragma omp parallel private(j)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      A[i][j] = min(A[i][j], A[i][k]+A[k][j]);
```

Στην παραπάνω υλοποίηση, ο εξωτερικός βρόχος αντιστοιχεί στα χρονικά βήματα της εκτέλεσης και επομένως δεν μπορεί να παραλληλοποιηθεί, όπως εξηγήθηκε και παραπάνω με τον γράφο εξαρτήσεων. Στο συγκεκριμένο τμήμα κώδικα μπορούμε όμως να παραλληλοποιήσουμε τους δύο εσωτερικούς βρόχους. Ο δείκτης `j` δηλώνεται ως private, καθώς ο `i` είναι εξ ορισμού ιδιωτικός, ώστε κάθε νήμα να έχει ένα ξεχωριστό αντίγραφο της μεταβλητής.

Παρακάτω βλέπουμε τα διαγράμματα για την χρονική απόδοση του παραπάνω αλγόριθμου για μεγέθη πινάκων 1024×1024, 2048×2048 και 4096×4096 για 1, 2, 4, 8, 16, 32 και 64 threads, αλλά και την σειριακή υλοποίηση `seq`.







Στα παραπάνω διαγράμματα παρατηρούμε ότι υπάρχει σημαντική μείωση του χρόνου με την αύξηση των threads, υποδηλώνοντας ότι ο παρών αλγόριθμος ενδείκνυται για παραλληλοποίηση. Ωστόσο, αποδεικνύεται επίσης ότι η υλοποίηση αυτή είναι memory bound, κάτι το οποίο φαίνεται στα παραπάνω διαγράμματα για μεγάλους πίνακες γειτνίασης, όπως 2048×2048 και 4096×4096. Στις περιπτώσεις αυτές που ο πίνακας δεν χωράει ολόκληρος στην μνήμη, σε κάθε χρονικό βήμα, έχουμε επιπλέον N^2 δεδομένα που πρέπει να μεταφερθούν πέρα από τις N πράξεις που πρέπει να γίνουν

(στην πραγματικότητα μόνο ένα μέρος του πίνακα μεταφέρεται). Στην περίπτωση του 1024×1024 που ο πίνακας χωράει στην L1-cache, τότε αυτή η μεταφορά δεδομένων γίνεται μόνο μια φορά στην αρχή. Οι καλύτεροι χρόνοι που μετρήθηκαν για κάθε πίνακα είναι:

- 1024×1024: With 32 threads - Execution Time: 0.2167
- 2048×2048: With 64 threads - Execution Time: 0.9806
- 4096×4096: With 64 threads - Execution Time: 19.8974

Στην συνέχεια εξετάζουμε δύο αλγορίθμους (*recursive*, *tiled*), οι οποίοι στοχεύουν σε βελτίωση της επίδοσης με καλύτερη αξιοποίηση της cache.

2.2.5 Έκδοση 2 - Recursive αλγόριθμος

Ο αναδρομικός αλγόριθμος είναι ο εξής:

```
FWI (A, B, C)
  for (k=0; k<N; k++)
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        A[i][j] = min(A[i][j], B[i][k]+C[k][j]);

FWR (A, B, C)
  if (base case)
    FWI (A, B, C)
  else
    FWR (A00, B00, C00);
    FWR (A01, B00, C01);
    FWR (A10, B10, C00);
    FWR (A11, B10, C01);
    FWR (A11, B10, C01);
    FWR (A10, B10, C00);
    FWR (A01, B00, C01);
    FWR (A00, B00, C00);
```

Παραπάνω παρατηρούμε δύο συναρτήσεις, την `FWI` και την `FWR`. Η `FWI` θα κληθεί, όταν ικανοποιείται το base case σενάριο, ενώ η `FWR` είναι ο αναδρομικός αλγόριθμος.

Η `FWR` δέχεται ως είσοδο τα εξής ορίσματα:

1. `A, B, C`: Τρεις πίνακες ίδιου μεγέθους
2. `arow, acol, brow, bcol, crow, ccol`: Η γραμμή και η στήλη του πρώτου στοιχείου κάθε πίνακα
3. `myN`: Το μέγεθος του υποπίνακα προς επεξεργασία
4. `bsize`: Το μέγεθος του υποπίνακα, για το οποίο σταματάει η αναδρομή και εκτελείται η `FWI`

Η αναδρομή σταματάει όταν $\texttt{myN} \leq \texttt{bsize}$, δηλαδή όταν το μέγεθος του υποπίνακα είναι μικρότερο από `bsize`. Η επιλογή της τιμής αυτής καθορίζει σε σημαντικό βαθμό την επίδοση του αλγορίθμου. Η επιθυμητή συνθήκη είναι η συνάρτηση `FWI` να κληθεί, όταν το μέγεθος του πίνακα είναι ακρετά

μικρό, ώστε να χωράει στην cache.

Αντίθετα, όταν myN > bsize, τότε συνεχίζεται η εκτέλεση του αναδρομικού αλγόριθμου με τις 8 αναδρομικές κλήσεις που βλέπουμε παραπάνω, οι οποίες υποδιπλασιάζουν το μέγεθος του πίνακα. Για την αναδρομική έκδοση του αλγορίθμου βασιστήκαμε στη χρήση των tasks, προκειμένου να εκφράσουμε τη ροή των εξαρτήσεων.

Αρχικά η παράλληλη περιοχή ορίστηκε στην main() συνάρτηση, ώστε όλη η αναδρομή να βρίσκεται μέσα σε ένα parallel block. Η υλοποίηση φαίνεται παρακάτω:

```
#pragma omp parallel
#pragma omp single
{
  FW_SR(A,0,0,A,0,0,A,0,0,N,B);
}
```

Στη συνέχεια, διαμορφώσαμε το σώμα της FW_SR με τέτοιο τρόπο, ώστε κάθε μία από τις εργασίες που μπορούν να εκτελεστούν παράλληλα να αποτελεί ξεχωριστό task και να οδηγείται στο pool. Παρακάτω φαίνεται το τμήμα κώδικα της συνάρτησης FW_SR:

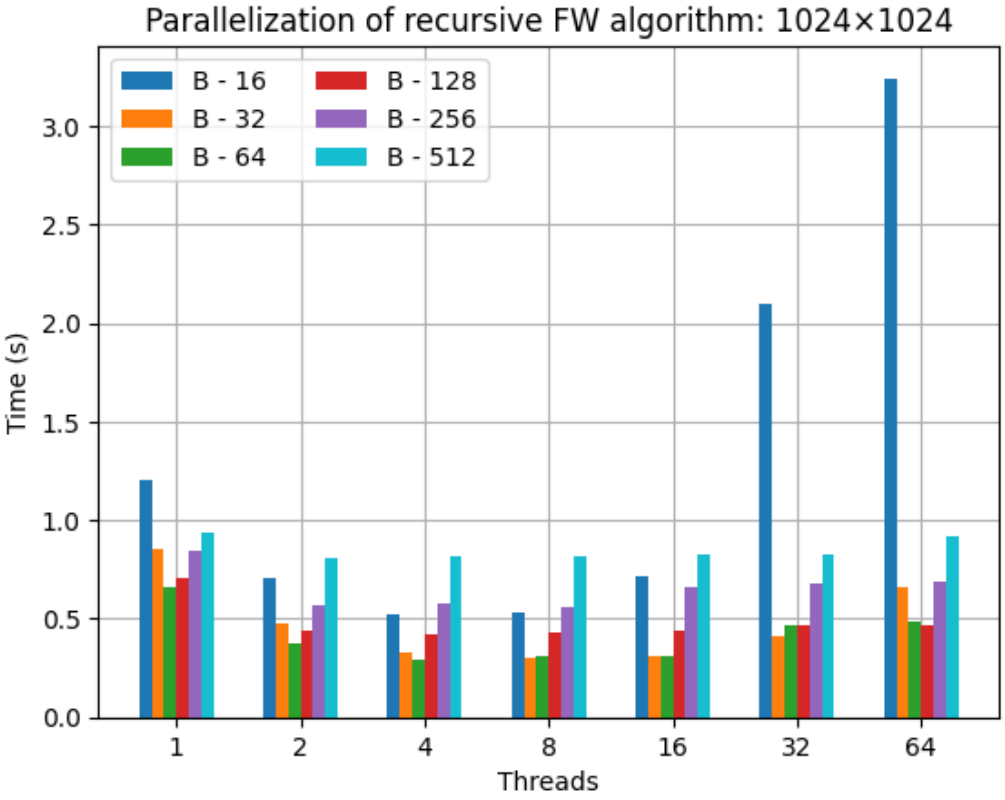
```
if(myN<=bsize)
  for(k=0; k<myN; k++)
    for(i=0; i<myN; i++)
      for(j=0; j<myN; j++)
        A[arow+i][acol+j] = min(
          A[arow+i][acol+j],
          B[brow+i][bcol+k] + C[crow+k][ccol+j]
        );
else {
  FW_SR(
      A,arow,acol,
      B,brow,bcol,
      C,crow,ccol,
      myN/2,bsize
  );

  #pragma omp task if(0)
  {
    #pragma omp task
    FW_SR(
      A,arow,acol+myN/2,
      B,brow,bcol,
      C,crow,ccol+myN/2,
      myN/2,bsize
    );
    FW_SR(
      A,arow+myN/2,acol,
      B,brow+myN/2,bcol,
      C,crow,ccol,
      myN/2,bsize
    );
```

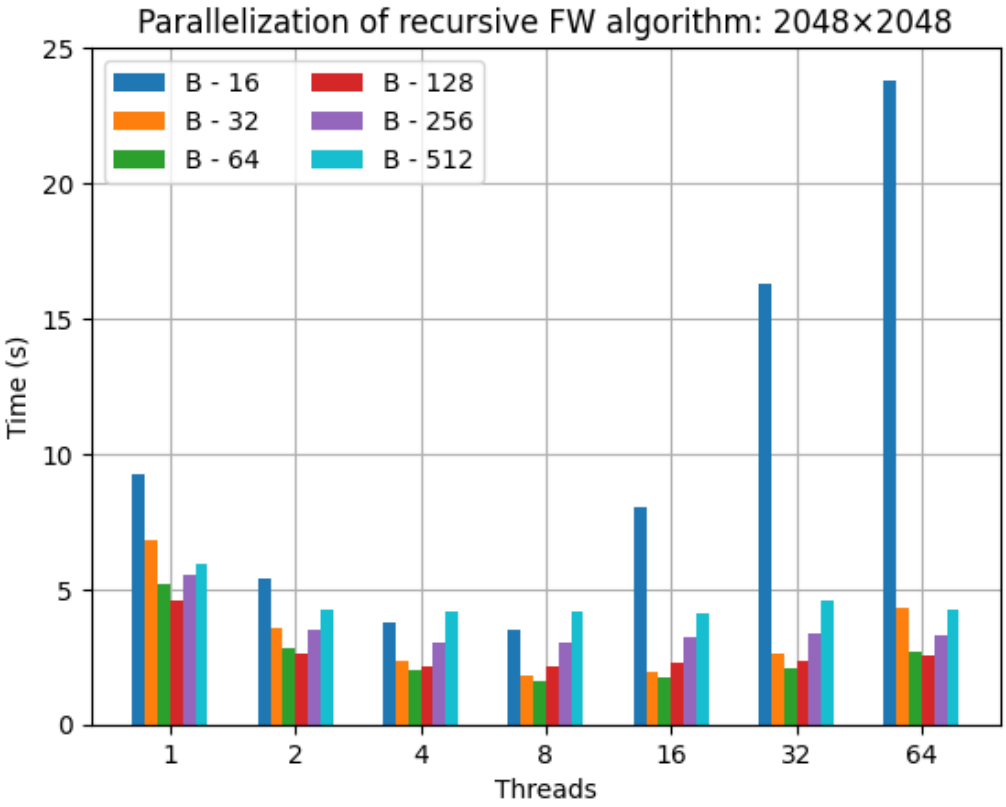
```
    #pragma omp taskwait
  }

  FW_SR(
    A,arow+myN/2,acol+myN/2,
    B,brow+myN/2,bcol,
    C,crow,ccol+myN/2,
    myN/2,bsize
  );
  FW_SR(
    A,arow+myN/2,acol+myN/2,
    B,brow+myN/2,bcol+myN/2,
    C,crow+myN/2,ccol+myN/2,
    myN/2,bsize
  );

  #pragma omp task if(0)
  {
    #pragma omp task
    FW_SR(
      A,arow+myN/2,acol,
      B,brow+myN/2,bcol+myN/2,
      C,crow+myN/2,ccol,
      myN/2,bsize
    );
    FW_SR(
      A,arow,acol+myN/2,
      B,brow,bcol+myN/2,
      C,crow+myN/2,ccol+myN/2,
      myN/2, bsize
    );

    #pragma omp taskwait
  }

  FW_SR(
    A,arow,acol,
    B,brow,bcol+myN/2,
    C,crow+myN/2,ccol,
    myN/2,bsize
  );
}
```

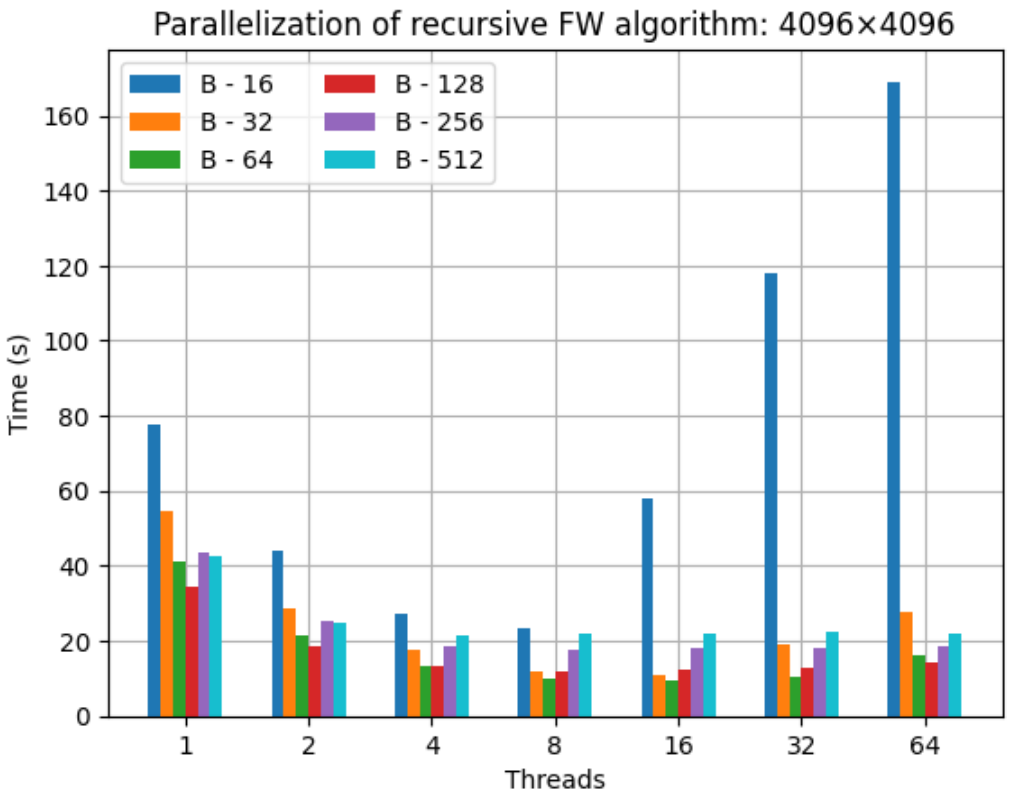
Σε μια διαφορετική εκδοχή παραλληλοποίησης του αλγορίθμου που υλοποιήσαμε, παρατηρήσαμε την εξής συμπεριφορά, ότι ο αλγόριθμος δεν κλιμάκωνε για περισσότερα από 2 νήματα, με αποτέλεσμα η υλοποίηση χωρίς παραλληλοποίηση να είναι καλύτερη. Στην υλοποίηση αυτή, έχουμε βάλει ένα parallel block μέσα στην συνάρτηση `FW_SR` στο τμήμα `else`. Ενώ περιμέναμε με κάθε επανάληψη να δημιουργούνται 2 νέα παράλληλα tasks, με αποτέλεσμα ο αριθμός των παράλληλων tasks να αυξάνεται όσο προχωράει η αναδρομή (προς όφελός μας), αυτό δεν συνέβη.

Επομένως, διερευνώντας αντιληφθήκαμε ότι η OpenMP δεν αναθέτει επιπλέον threads σε nested parallel blocks.

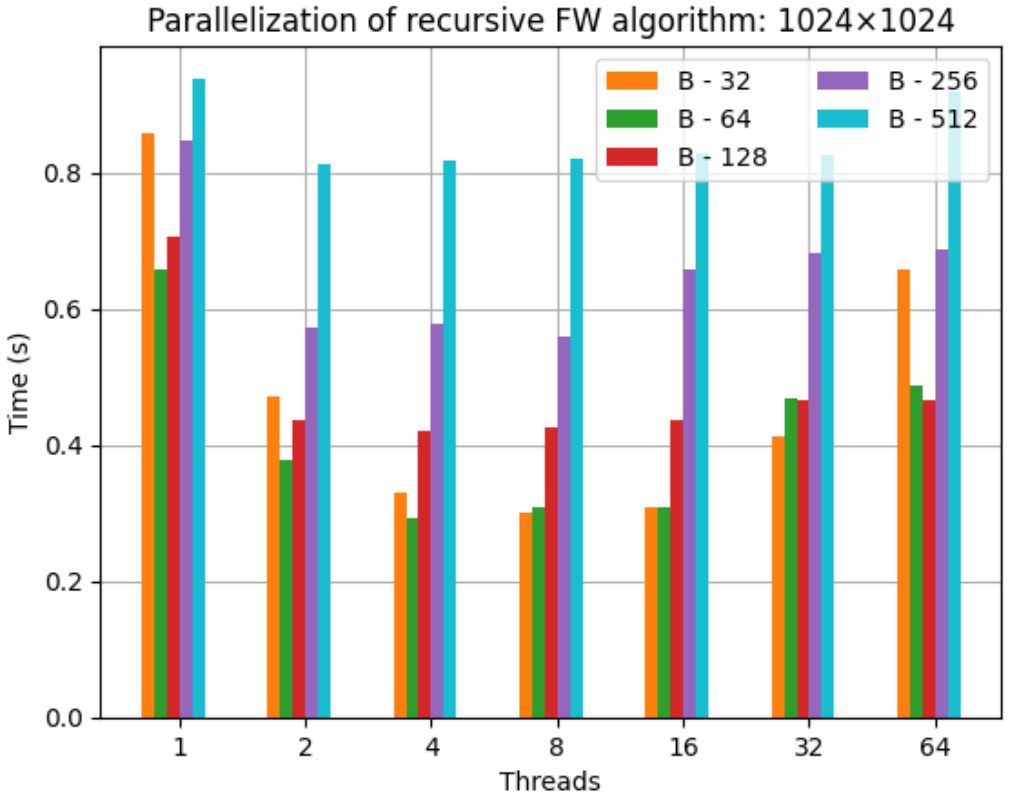
Τα παρακάτω διαγράμματα μας επιτρέπουν να συγκρίνουμε τις διαφορετικές επιδόσεις του αναδρομικού αλγόριθμου σε σχέση με το μέγεθος του πίνακα γειτνίασης A, το πλήθος των threads, και το μέγεθος του πίνακα B, δηλαδή το base case.

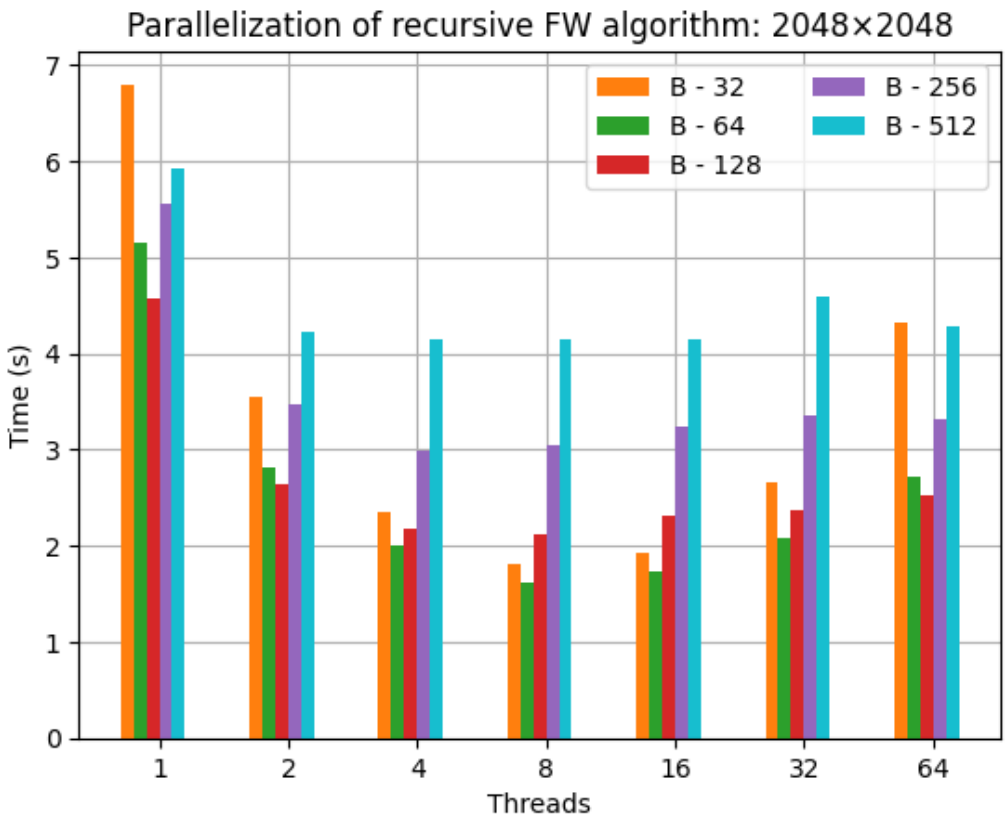


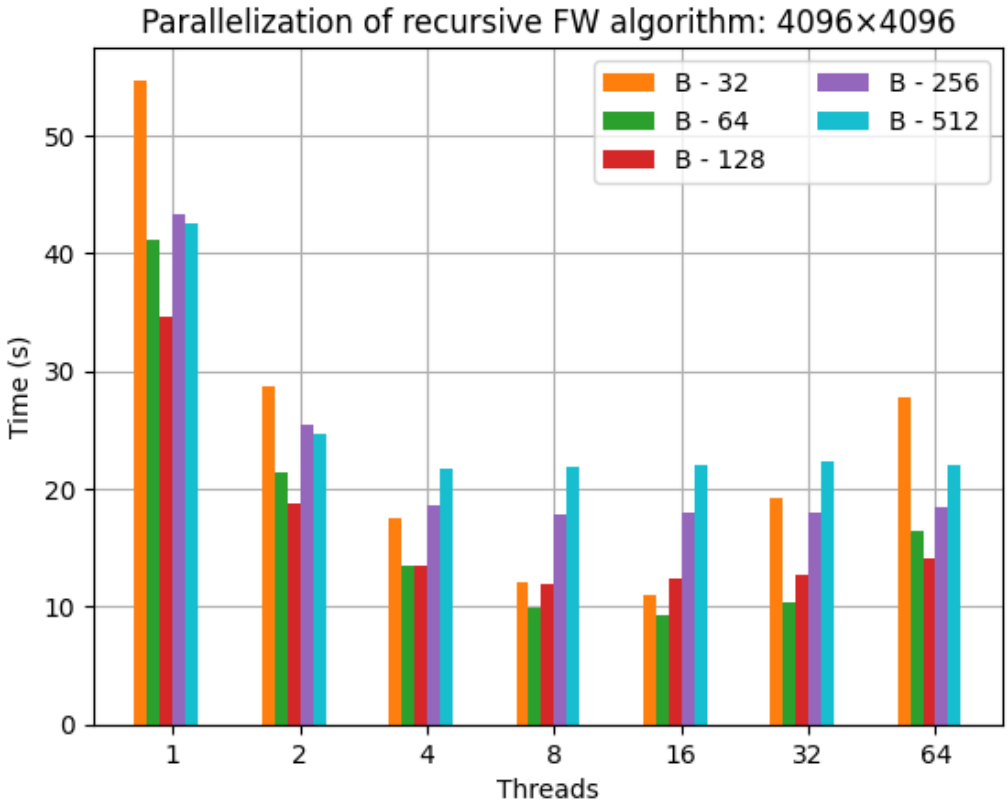




Για ευκολία στην παρατήρηση και μελέτη των αποτελεσμάτων, αφαιρούμε τις μετρήσεις για block size 16, αφού όπως φαίνεται στην περίπτωση της αναδρομικής υλοποίησης η επιλογή ενός τόσο μικρού block size δεν μας βοηθάει, καθώς χρειάζεται να γίνουν πολλές περισσότερες αναδρομές μέχρι να φτάσουμε στην συνθήκη που σταματάει την αναδρομή, το οποίο εισάγει συνολικά καθυστερήσεις που μειώνουν την επίδοση.







Ειδικότερα, παρατηρείται ότι η βέλτιστη επίδοση κάθε φορά προκύπτει από διαφορετικό πλήθος νημάτων και εμφανίζεται για μέγεθος block 64. Με δεδομένα αυτά εντοπίζουμε τους καλύτερους χρόνους που επιτεύχθηκαν για κάθε μέγεθος πίνακα:

- 1024×1024: With 4 threads and block size 64 - Execution Time: 0.2935 sec

- 2048×2048: With 8 threads and block size 64 - Execution Time: 1.6208 sec
- 4096×4096: With 16 threads and block size 64 - Execution Time: 9.3077 sec

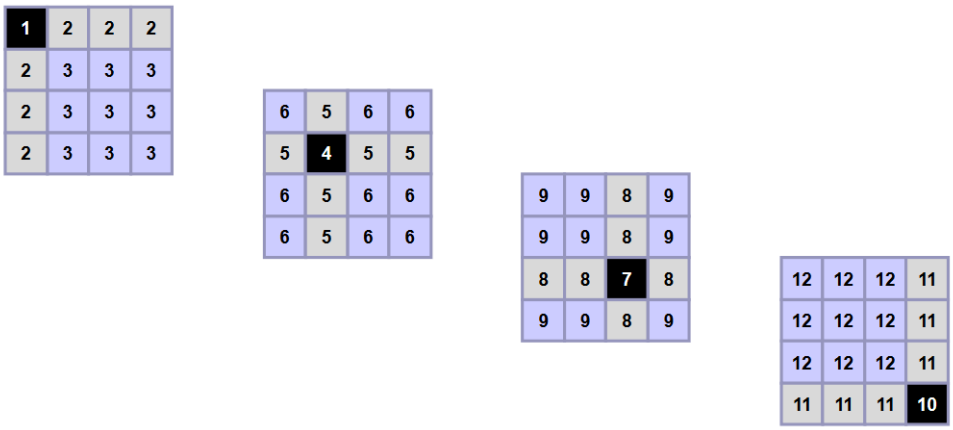
2.2.6 Έκδοση 3 - tiled αλγόριθμος

Ο tiled αλγόριθμος για την υλοποίηση του Floyd-Warshall είναι ο εξής:

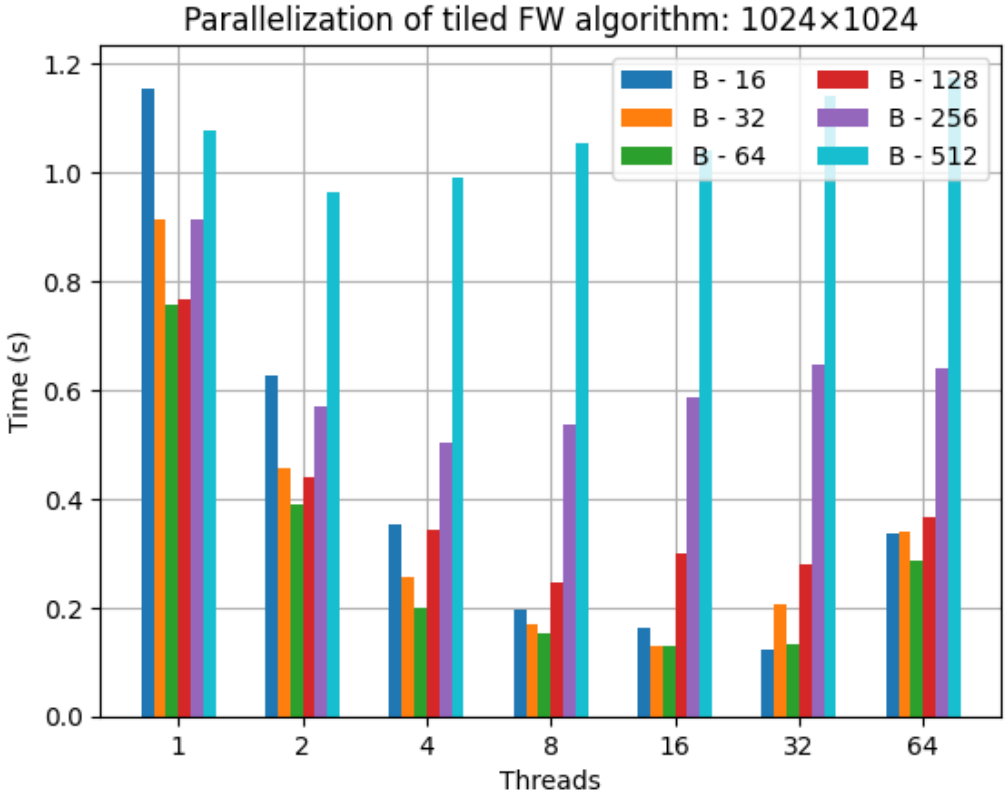
```
for(k=0;k<N;k+=B) {
  FW(A,k,k,k,B);

  #pragma omp parallel shared(A,k,B)
  {
    #pragma omp for nowait
    for(i=0; i<k; i+=B)
      FW(A,k,i,k,B);

    #pragma omp for nowait
    for(i=k+B; i<N; i+=B)
      FW(A,k,i,k,B);

    #pragma omp for nowait
    for(j=0; j<k; j+=B)
      FW(A,k,k,j,B);

    #pragma omp for nowait
    for(j=k+B; j<N; j+=B)
      FW(A,k,k,j,B);
  }

  #pragma omp parallel shared(A,k,B)
  {
    #pragma omp for collapse(2) nowait
    for(i=0; i<k; i+=B)
      for(j=0; j<k; j+=B)
        FW(A,k,i,j,B);

    #pragma omp for collapse(2) nowait
    for(i=0; i<k; i+=B)
      for(j=k+B; j<N; j+=B)
        FW(A,k,i,j,B);

    #pragma omp for collapse(2) nowait
    for(i=k+B; i<N; i+=B)
      for(j=0; j<k; j+=B)
        FW(A,k,i,j,B);

    #pragma omp for collapse(2) nowait
    for(i=k+B; i<N; i+=B)
      for(j=k+B; j<N; j+=B)
        FW(A,k,i,j,B);
```

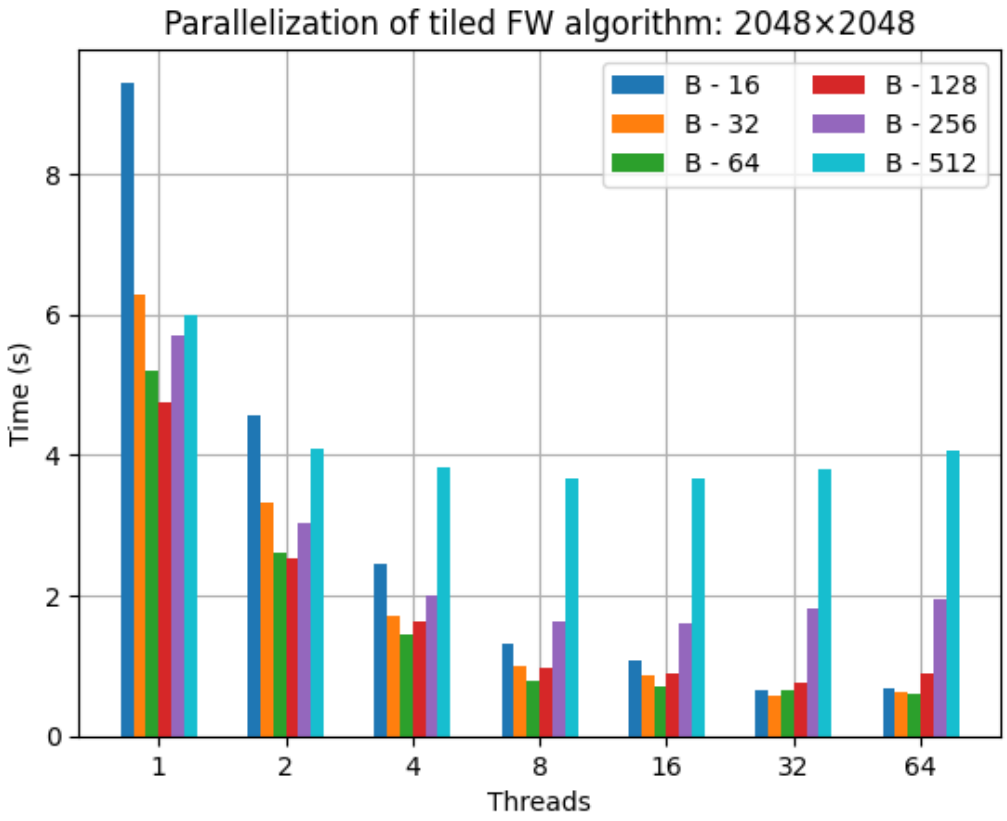
```
    }
}
```

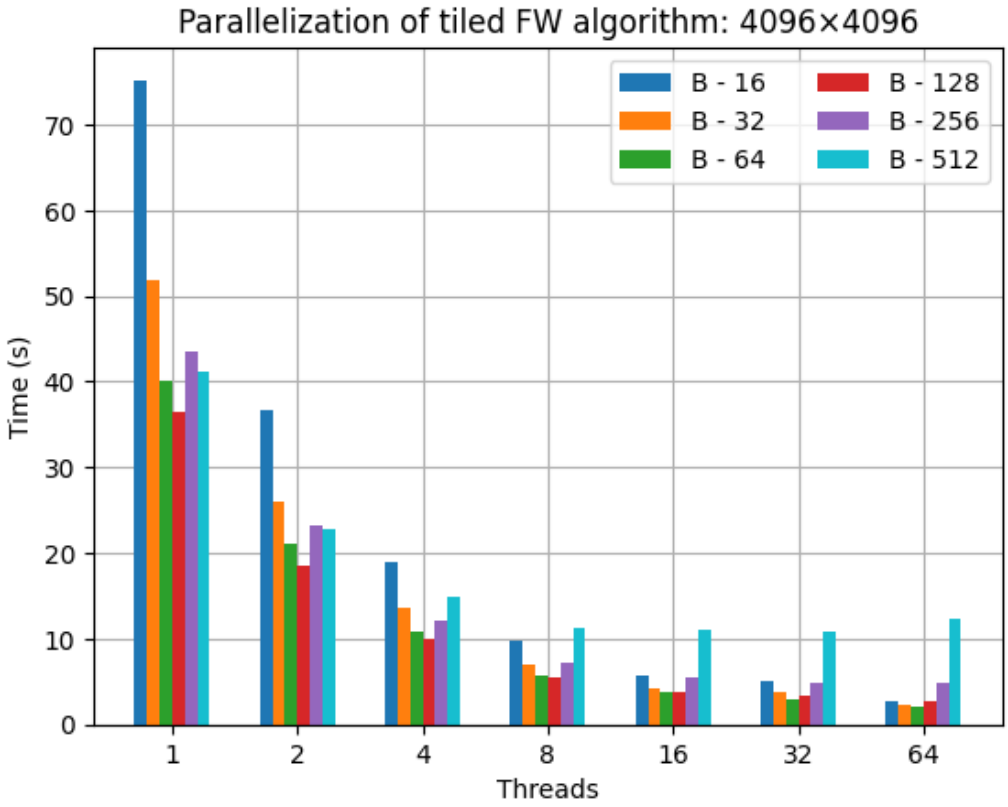
Η συγκεκριμένη εκδοχή εφαρμόζει τον κλασσικό αλγόριθμο Floyd-Warshall σε συγκεκριμένα blocks (tiles) του αρχικού πίνακα με τη σειρά που φαίνεται παρακάτω (φωτογραφία από τις διαφάνειες του μαθήματος):



Σε αυτή την εκδοχή είναι δυνατή η παράλληλη εκτέλεση των block ίδιου χρώματος και κατά αυτόν τον τρόπο οδηγηθήκαμε στην παραπάνω παράλληλη υλοποίηση. Εδώ προτιμήσαμε να παραλληλοποιήσουμε τον αλγόριθμο με χρήση parallel for. Την επίδοση της παραλληλοποίησης με *parallel for* την βελτίωσαν το *collapse* clause και *nowait* είδος χρονοδρομολόγησης. Το *collapse* βοήθησε αρκετά στην μείωση του χρόνου εκτέλεσης, καθώς μέσω αυτού καθορίσαμε πόσοι βρόχοι, δηλαδή 2 σε αυτήν την περίπτωση, σε έναν εμφωλευμένο βρόχο θα πρέπει να συμπτυχθούν σε έναν μεγάλο χώρο επανάληψης. Το *nowait* μείωσε και αυτό τον χρόνο εκτέλεσης, αφού πολλά νήματα περίμεναν να τελειώσει η επανάληψη για να μεταβούν σε κάποιο άλλο.







Ειδικότερα, παρατηρείται ότι η βέλτιστη επίδοση κάθε φορά προκύπτει από διαφορετικό πλήθος νημάτων και εμφανίζεται για μέγεθος block 64. Με δεδομένα αυτά εντοπίζουμε τους καλύτερους χρόνους που επιτεύχθηκαν για κάθε μέγεθος πίνακα:

- 1024×1024: With 16 threads and block size 16 - Execution Time: 0.1231 sec

- 2048×2048: With 32 threads and block size 32 - Execution Time: 0.5772 sec
- 4096×4096: With 64 threads and block size 64 - Execution Time: 2.0896 sec

2.2.7 Συμπεράσματα

Από τα παραπάνω συμπεραίνουμε πως την καλύτερη επίδοση την πετυχαίνουμε στην tiled έκδοση του αλγορίθμου. Το πρόβλημα που αντιμετωπίζαμε στην standard έκδοση του αλγορίθμου ήταν το γεγονός ότι η υλοποίηση ήταν memory bound με αποτέλεσμα όσο και να παραλληλοποιούσαμε να υπήρχε πάντα ένα overhead. Στην tiled εκδοχή του αλγορίθμου, όπου εφαρμόζουμε τον FW περισσότερες φορές, αλλά σε μικρότερα τμήματα του αρχικού πίνακα, αυξάνουμε το temporal locality. Δηλαδή αυξάνουμε τις πιθανότητες στοιχεία του πίνακα να βρίσκονται στην cache, όταν τα ζητήσουμε. Σε κάθε περίπτωση είναι αξιοσημείωτο, πως με μεταβολές στον αλγόριθμο, στις μεταβλητές περιβάλλοντος, στο πλήθος των νημάτων καταφέραμε να μειώσουμε τόσο πολύ τον χρόνο εκτέλεσης του προγράμματος, ακόμα και για μεγάλους πίνακες μεγέθους 4096×4096.

2.3 Ταυτόχρονες Δομές Δεδομένων

2.3.1 Εισαγωγή/Σκοπός της άσκησης

Στο συγκεκριμένο ερώτημα, δίνονται διαφορετικές ταυτόχρονες υλοποιήσεις μίας απλά συνδεδεμένης ταξινομημένης λίστας τις οποίες καλούμαστε να αξιολογήσουμε. Για τον σκοπό αυτό, εκτελούμε μετρήσεις σε διαφορετικές συνθήκες, μεταβάλλοντας:

- Τον αριθμό των νημάτων που θα χρησιμοποιηθούν.
- Το μέγεθος της λίστας.
- Το ποσοστό λειτουργιών `contains()`, `add()` και `remove()`.

2.3.2 Δεδομένα/Ζητούμενα

Στον φάκελο `/home/parallel/pps/2022-2023/a2/conc_ll/` βρίσκονται έτοιμες 5 ταυτόχρονες υλοποιήσεις απλά συνδεδεμένης ταξινομημένης λίστας:

- Coarse-grain locking
- Fine-grain locking
- Optimistic synchronization
- Lazy synchronization
- Non-blocking synchronization

Στον ίδιο φάκελο βρίσκεται και η σειριακή υλοποίηση. Αντιγράφοντας τον παραπάνω φάκελο και εκτελώντας `make`, δημιουργούνται τα (αντίστοιχα) εκτελέσιμα:

- `x.cgl`
- `x.fgl`
- `x.opt`
- `x.lazy`

- `x.nb`
- `x.serial`

Η εκτέλεση γίνεται ως εξής:

```
export MT_CONF=<mt_conf_value>
./<executable> <list_size> <contains_pct> <add_pct> <remove_pct>
```

Για παράδειγμα:

```
export MT_CONF=0,1,2,3
./x.cgl 1024 80 10 10
```

Για τιμή `MT_CONF`=$c_0, c_1, c_2, ...$, το thread 0 θα καταλάβει τον πυρήνα c_0, το thread 1 θα καταλάβει τον πυρήνα c_1 κ.ο.κ. Ορίζουμε την μεταβλητή `MT_CONF` με τέτοιο τρόπο ώστε τα νήματα να καταλαμβάνουν διαδοχικούς πυρήνες. Για παράδειγμα, αν έχουμε 8 νήματα:

```
MT_CONF=0,1,2,3,4,5,6,7
```

2.3.3 Ανάλυση των υλοποιήσεων

- **Coarse-grain locking**: Η πρώτη τεχνική συγχρονισμού που θα αναλύσουμε χρησιμοποιεί ένα κλείδωμα για όλη την δομή. Με αυτόν τον τρόπο εξασφαλίζει πως μόνο ένα νήμα θα έχει πρόσβαση στην δομή κάθε φορά, αποφεύγοντας συγκρούσεις ή/και απώλεια δεδομένων. Αυτή η υλοποίηση λειτουργεί αποδοτικά, όταν ο απαιτούμενος συγχρονισμός είναι χαμηλός. Όμως, όταν πολλά νήματα προσπαθούν να έχουν πρόσβαση στην δομή δεδομένων, τότε υπάρχει bottleneck στην υλοποίηση, καθώς τα νήματα θα χρειαστεί να περιμένουν σε ουρά για να πάρουν το κλείδωμα.
- **Fine-grain locking**: Η μέθοδος fine-grain locking βασίζεται στην τεχνική του "hand-over-hand" locking. Σε αντίθεση με την coarse-grain υλοποίηση, δεν κλειδώνουμε ολόκληρη τη λίστα, αλλά συγκεκριμένους κόμβους, επιτρέποντας έτσι σε περισσότερα από 1 νήματα να διατρέχουν τη λίστα ταυτόχρονα με μια λογική pipelining. Ειδικότερα, ένα νήμα διατρέχει τη λίστα χρησιμοποιώντας 2 κλειδώματα. Για να προχωρήσει, δηλαδή, και να κλειδώσει έναν κόμβο `B` θα πρέπει να διαθέτει το κλείδωμα του προηγούμενού του, `A`. Εάν λάβει επιτυχώς το κλείδωμα στον `B`, τότε μπορεί να απελευθερώσει τον `A` και να προχωρήσει. Θα δούμε ακολούθως την υλοποίηση των συναρτήσεων για την αναζήτηση, προσθήκη και διαγραφή κόμβου.
- **Optimistic synchronization**: Όπως γίνεται φανερό, η παραπάνω υλοποίηση έχει υψηλό κόστος συγχρονισμού λόγω του πλήθους των κλειδωμάτων που χρησιμοποιεί. Σε αυτήν την μέθοδο επιχειρούμε να ελαττώσουμε το πλήθος των κλειδωμάτων. Ένας τρόπος για να το επιτύχουμε αυτό είναι να μην χρησιμοποιούμε κλειδώματα όταν διατρέχουμε τη λίστα, αλλά να κλειδώνουμε μόνο τους κόμβους που μας ενδιαφέρουν μόλις φτάσουμε σε αυτούς. Πιο συγκεκριμένα, όταν εντοπίσουμε τον προς αναζήτηση κόμβο, τότε τον κλειδώνουμε και μετά ελέγχουμε πως ο κόμβος αυτός δεν έχει τροποποιηθεί στο ενδιάμεσο διάστημα. Αν συμβεί κάποιο λάθος, τότε απελευθερώνουμε τους πόρους και ξεκινάμε από την αρχή. Ο έλεγχος στον οποίον αναφερόμαστε γίνεται μέσω της συνάρτησης `validate()`, η οποία διατρέχει τη λίστα από την αρχή της και αν βρει τον πρώτο κόμβο που κλειδώσαμε, ελέγχει αν ο επόμενός του είναι πράγματι ο δεύτερος που κλειδώσαμε. Η τεχνική αυτή είναι αποδοτική όταν πετυχαίνει περισσότερες φορές από ότι αποτυγχάνει (γι' αυτό εξάλλου λέγεται *optimistic*).

- **Lazy synchronization**: Η συγκεκριμένη υλοποίηση αποτελεί μια βελτιστοποίηση της *optimistic* μεθόδου. Ένα πρόβλημα που θέλει να διορθώσει είναι ότι η $contains()$ στην *optimistic* υλοποίηση χρειαζόταν κλειδώματα. Σε αυτήν την μέθοδο επιδιώκουμε να την $contains()$ wait-free. Εδώ προστίθεται στη δομή ένα flag, το οποίο δηλώνει αν ένα στοιχείο υπάρχει ή όχι στην λίστα. Για τον έλεγχο των δεδομένων, δεν διατρέχει ολόκληρη τη λίστα από την αρχή της (*lazy*), αλλά ελέγχει αν οι δύο κλειδωμένοι κόμβοι ανήκουν λογικά στη λίστα και αν ο `pred` δείχνει στον `curr`.

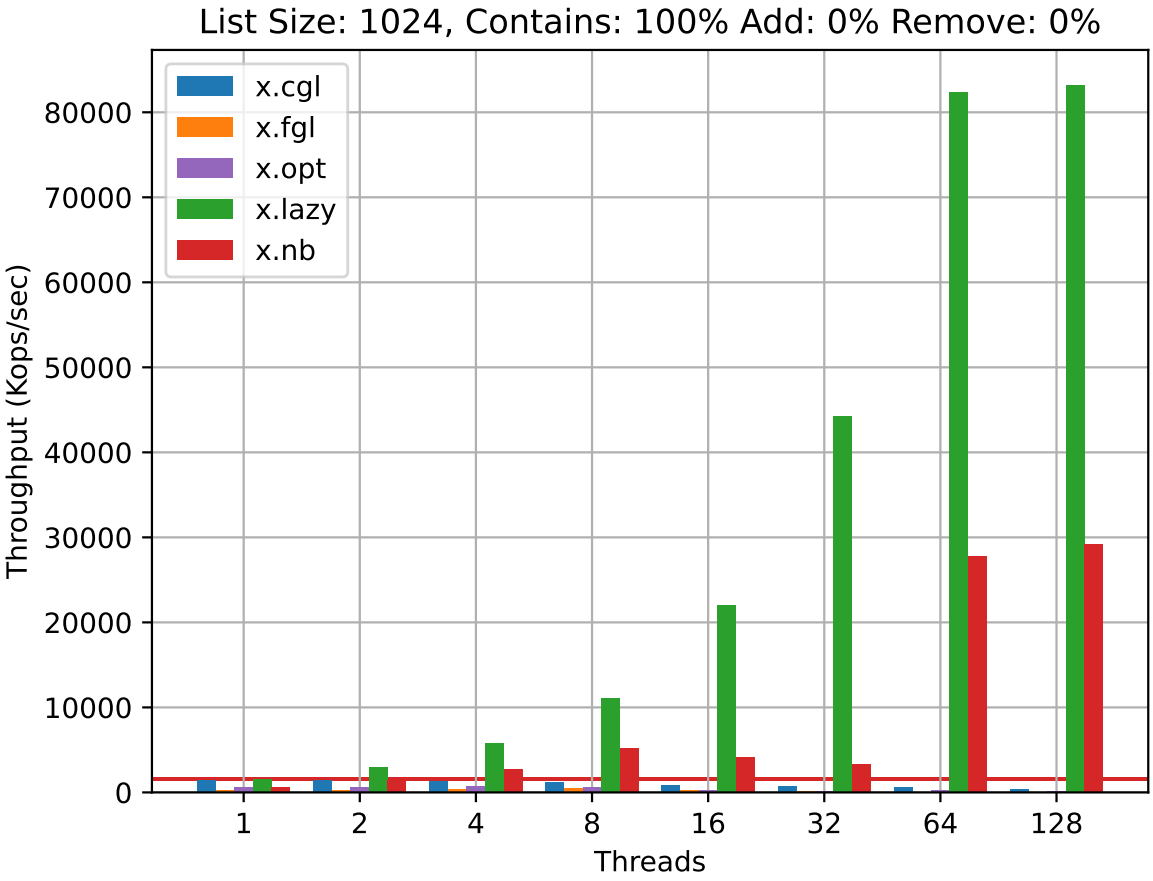
- **Non-blocking synchronization**: Όσο βελτιστοποιημένες και αν είναι οι παραπάνω υλοποιήσεις, τα κλειδώματα εισάγουν καθυστερήσεις αλλά και κινδύνους (τι θα συμβεί εάν ένα νήμα "πεθάνει" ενώ έχει το κλείδωμα;). Επομένως, ιδανικά θα θέλαμε να τα αποφύγουμε εξ'ολοκλήρου. Κάτι τέτοιο είναι εφικτό, εάν βασιστούμε περισσότερο σε ατομικές εντολές όπως η `compareAndSet()`.

2.3.4 Μετρήσεις

Εκτελούμε μετρήσεις για τις παρακάτω τιμές:

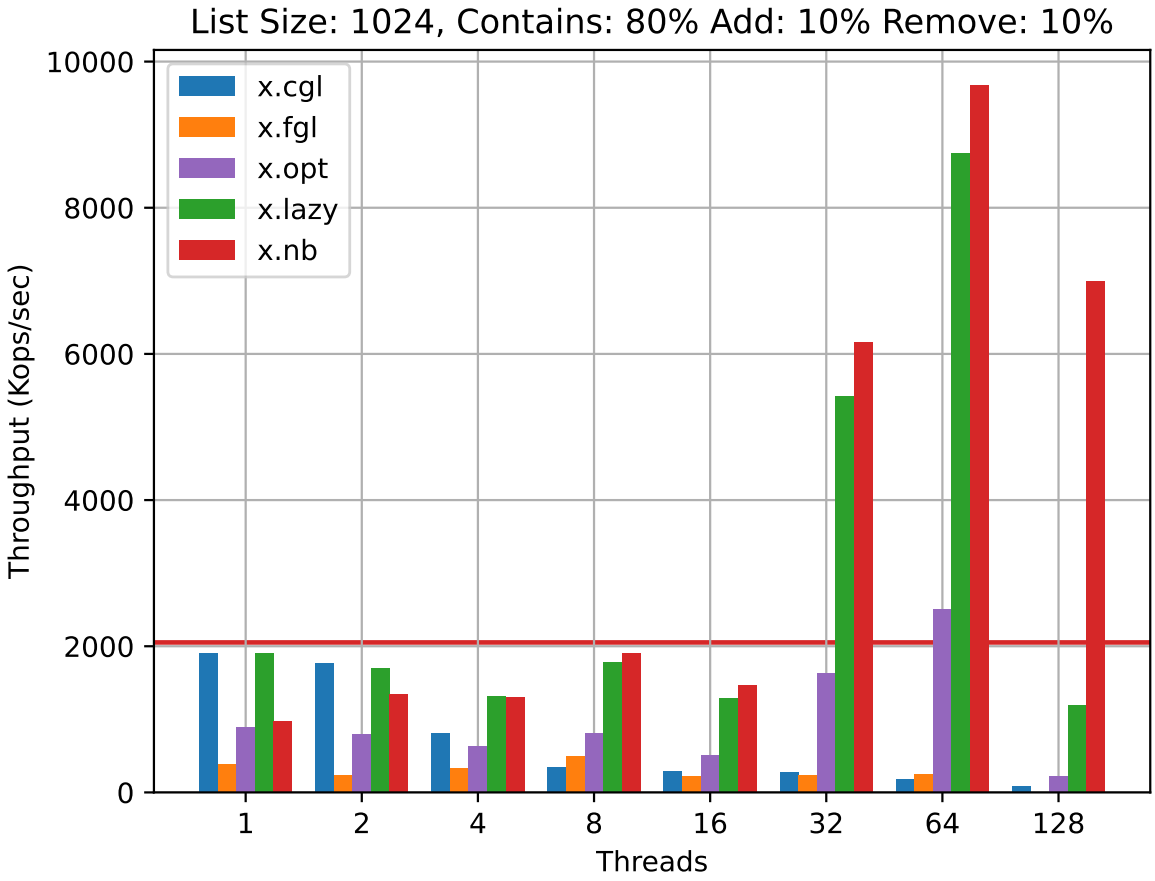
- Αριθμός νημάτων: $1, 2, 4, 8, 16, 32, 64, 128$
- Μέγεθος λίστας: $1024, 8192$
- Ποσοστό λειτουργιών (contains-add-remove): 100-0-0, 80-10-10, 20-40-40, 0-50-50

Κάθε εκτελέσιμο έχει χρόνο εκτέλεσης 10 δευτερόλεπτα (μεταβλητή `RUNTIME`), επομένως θα χρησιμοποιήσουμε σαν μετρική το *throughput* (Kops/sec). Ακολουθούν τα αποτελέσματα των μετρήσεων, στα οποία η κόκκινη οριζόντια γραμμή υποδηλώνει το throughput της σειριακής εκτέλεσης. Κάθε διάγραμμα συνοδεύεται από πίνακα με τα αντίστοιχα νούμερα για πιο λεπτομερή επισκόπηση:



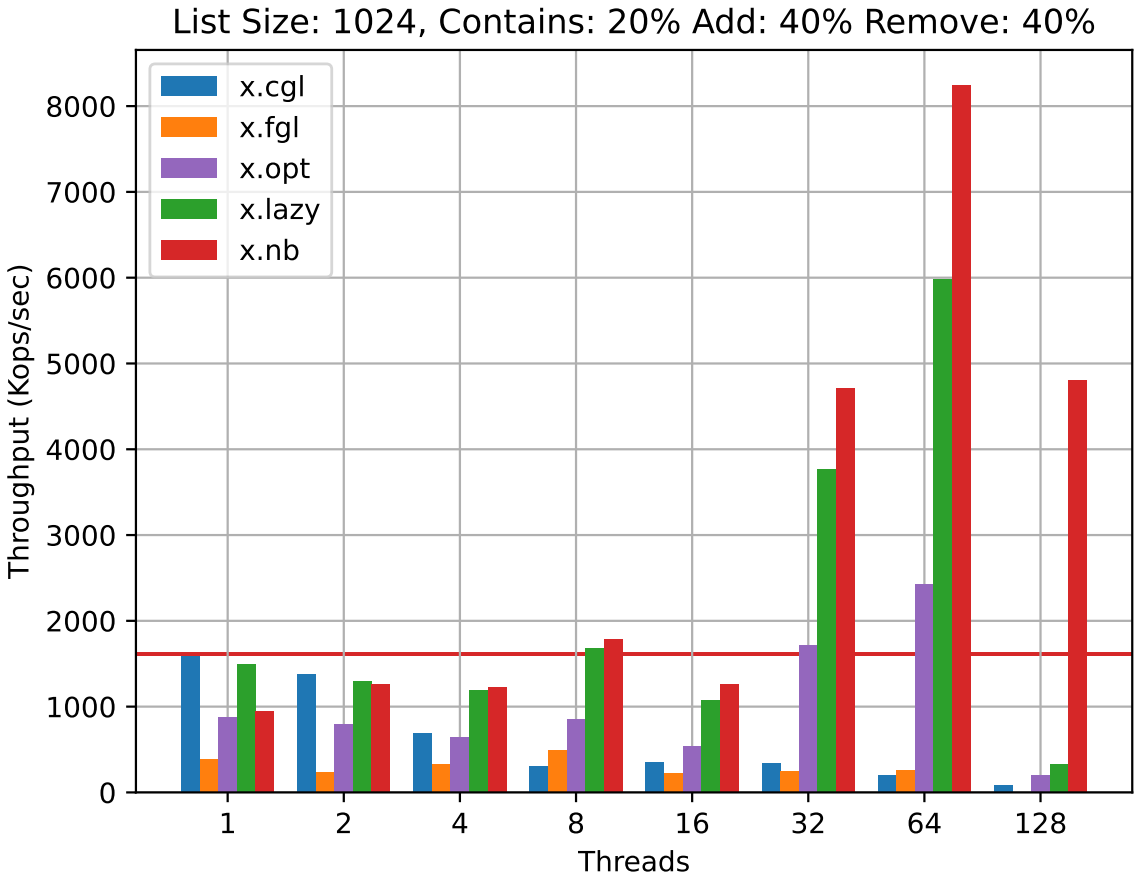
Throughput for List Size: 1024, Contains: 100%, Add: 0%, Remove: 0% (Kops/sec)

Threads \ Executable	`x.serial`	`x.cgl`	`x.fgl`	`x.opt`	`x.lazy`	`x.nb`
1	1569.33	1500.54	233.73	643.86	1561.32	678.54
2		1437.19	313.59	678.5	2991.02	1394.43
4		1343.02	449.4	756.69	5766.04	2735.76
8		1199.35	521.56	675.56	11055.49	5176.28
16		868.15	230.8	311.58	22092.83	4105.95
32		743.09	134.53	178.11	44215.49	3346.69
64		668.04	77.4	314.23	82338.19	27784.05
128		384.95	3.72	176.44	83173.8	29236.02



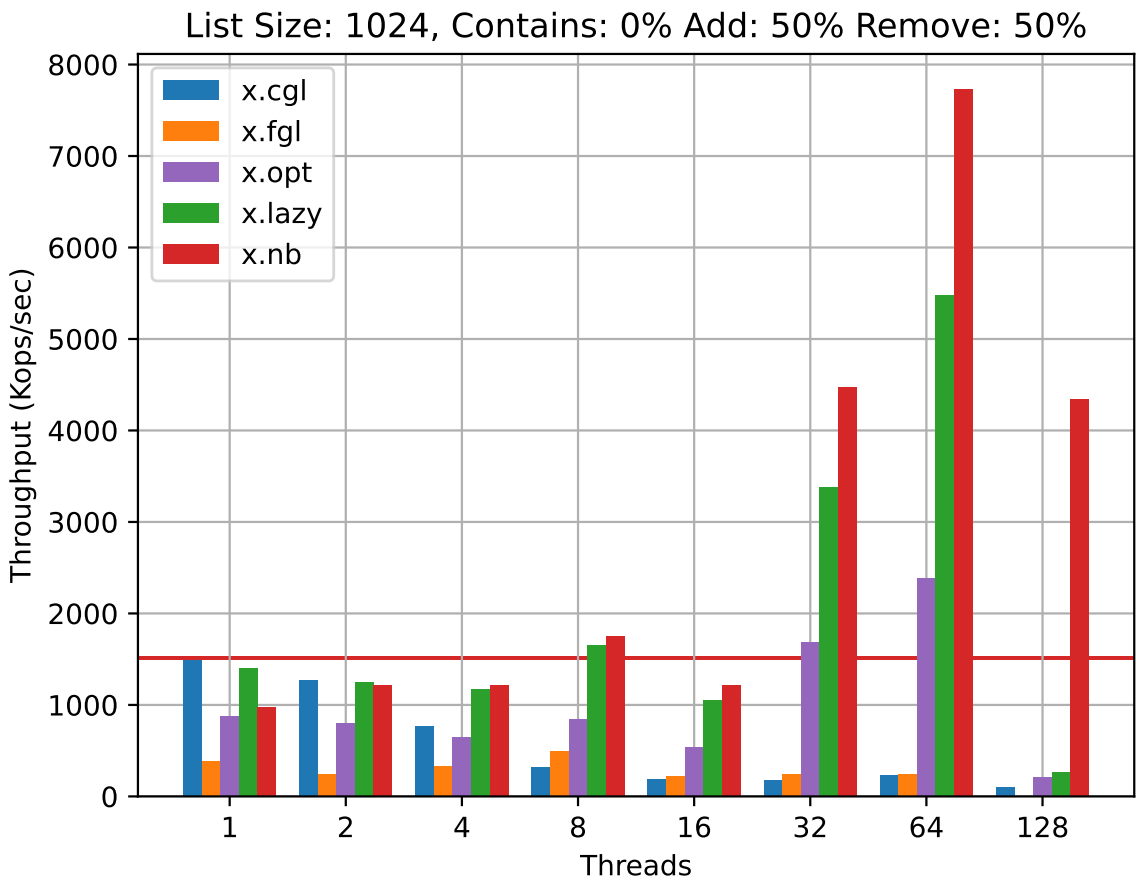
Throughput for List Size: 1024, Contains: 80%, Add: 10%, Remove: 10% (Kops/sec)

Threads \ Executable	x.serial	x.cgl	x.fgl	x.opt	x.lazy	x.nb
1	2051.98	1903.71	388.26	899.48	1908.05	977.06
2		1772.6	241.52	796.83	1704.34	1342.26
4		805.97	330.55	632.77	1318.81	1302.17
8		350.8	491.51	807.03	1788.4	1904.08
16		289.41	227.2	504.74	1297.15	1469.45
32		277.84	241.24	1639.48	5425.22	6162.14
64		183.49	247.22	2507.62	8742.26	9675.4
128		81.52	5.65	225.39	1188.77	6996.9



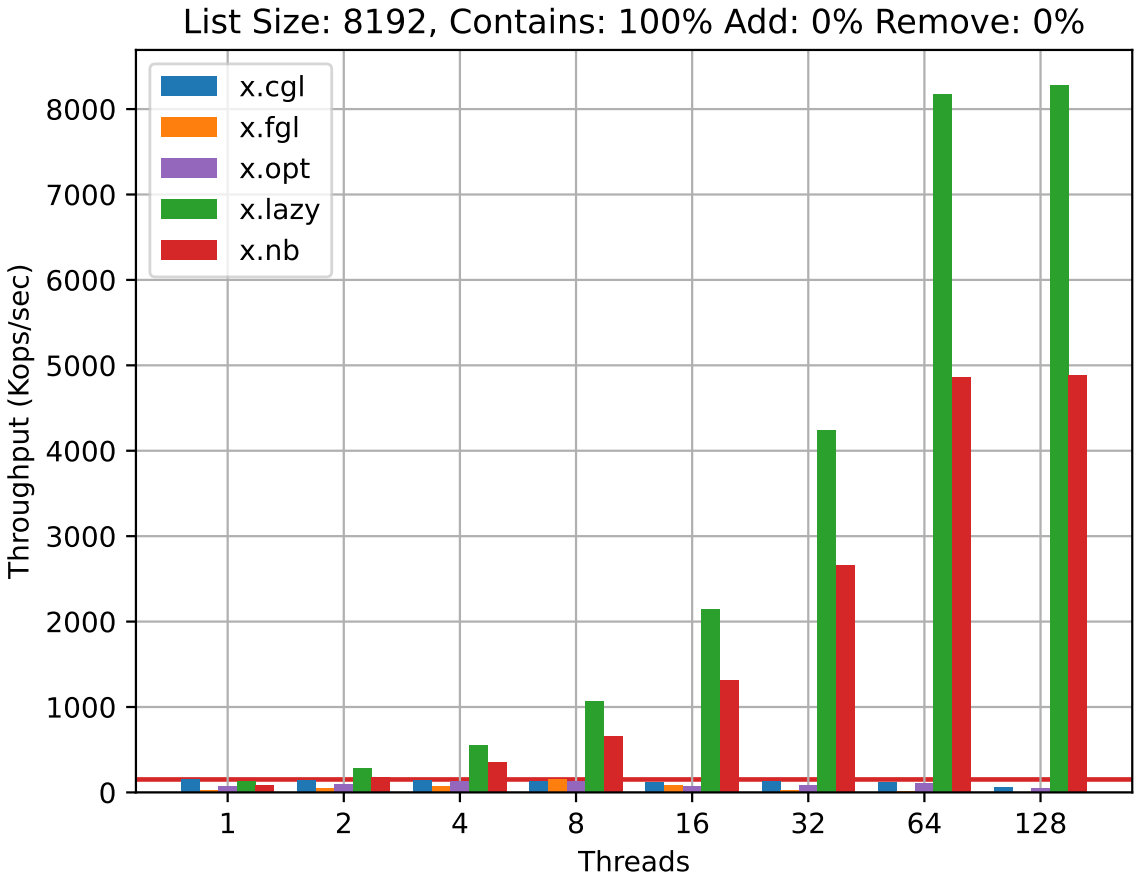
Throughput for List Size: 1024, Contains: 20%, Add: 40%, Remove: 40% (Kops/sec)

Threads \ Executable	x.serial	x.cgl	x.fgl	x.opt	x.lazy	x.nb
1	1611.25	1588.25	386.25	880.36	1499.05	943.89
2		1378.56	241.58	798.29	1293.94	1266.41
4		689.25	332.7	649.69	1191.29	1232.03
8		302.87	492.65	850.02	1680.99	1782.02
16		350.78	226.95	538.37	1080.25	1262.47
32		342.27	243.87	1718.65	3765.9	4714.85
64		199.67	255.44	2433.85	5983.67	8242.56
128		87.15	8.42	204.97	333.17	4801.78



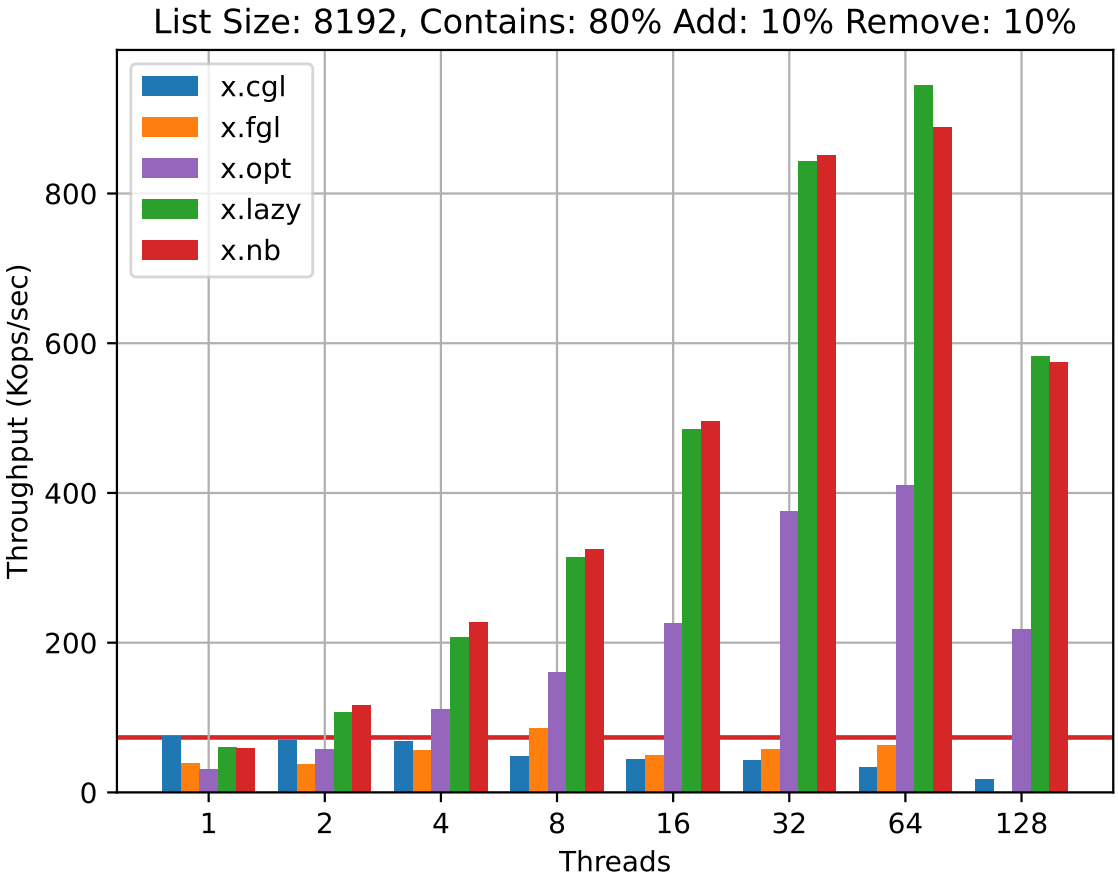
Throughput for List Size: 1024, Contains: 0%, Add: 50%, Remove: 50% (Kops/sec)

Threads \ Executable	`x.serial`	`x.cgl`	`x.fgl`	`x.opt`	`x.lazy`	`x.nb`
1	1515.06	1494.28	386.17	878.2	1399.64	972.51
2		1274.25	240.6	801.74	1246.09	1211.91
4		764.34	332.12	653.81	1177.02	1217.21
8		320.53	493.2	849.51	1649.52	1758.02
16		185.09	226.78	535.15	1047.79	1221.55
32		173.74	244.41	1691.5	3382.43	4477.97
64		236.62	246.89	2382.47	5477.71	7729.19
128		98.36	5.67	208.61	266.17	4342.16



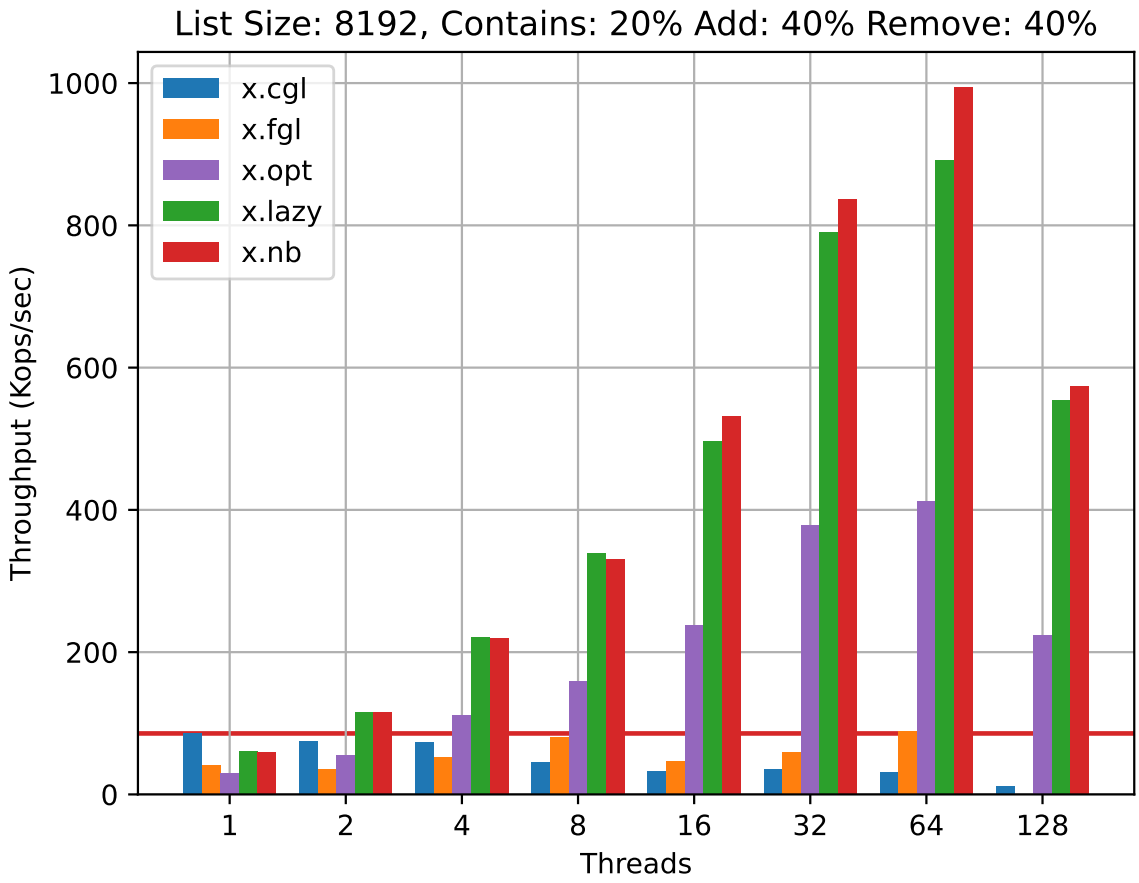
Throughput for List Size: 8192, Contains: 100%, Add: 0%, Remove: 0% (Kops/sec)

Threads \ Executable	`x.serial`	`x.cgl`	`x.fgl`	`x.opt`	`x.lazy`	`x.nb`
1	151.54	151.19	29.39	71.62	134.48	83.3
2		145.54	50.4	92.38	289.69	176.15
4		142.58	79.63	129.37	558.34	350.33
8		132.83	160.7	131.24	1074.82	656.4
16		125.74	84.01	78.86	2140.6	1318.9
32		127.39	32.33	87.88	4243.88	2662.49
64		123.74	14.42	113.45	8176.0	4859.29
128		66.44	0.38	55.26	8278.36	4890.25



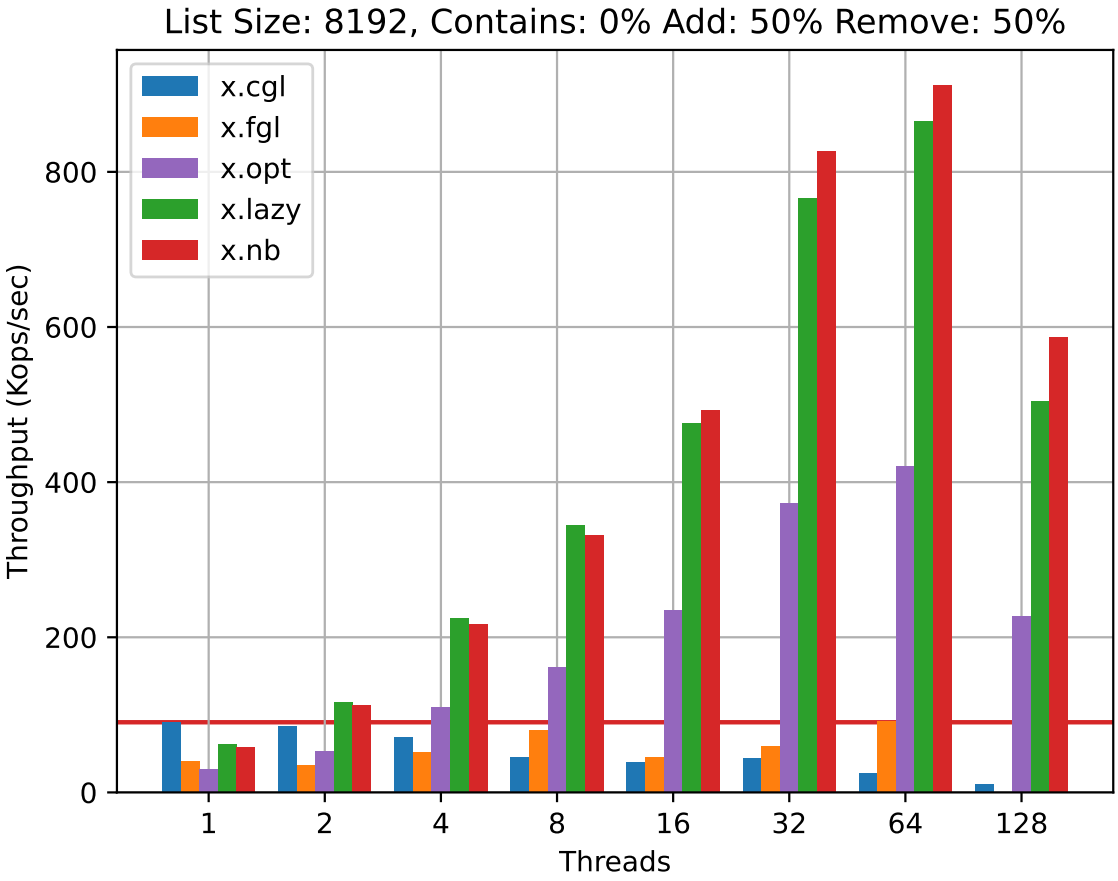
Throughput for List Size: 8192, Contains: 80%, Add: 10%, Remove: 10% (Kops/sec)

Threads \ Executable	x.serial	x.cgl	x.fgl	x.opt	x.lazy	x.nb
1	73.39	74.76	39.23	31.75	60.29	58.93
2		69.79	38.09	58.28	107.04	116.77
4		68.03	56.41	110.85	207.77	227.47
8		48.68	85.28	160.44	314.07	325.19
16		45.11	50.06	226.2	485.06	496.08
32		42.86	57.87	376.27	843.67	851.5
64		33.77	63.39	410.0	944.53	888.69
128		17.92	0.35	217.57	582.36	574.7



Throughput for List Size: 8192, Contains: 20%, Add: 40%, Remove: 40% (Kops/sec)

Threads \ Executable	`x.serial`	`x.cgl`	`x.fgl`	`x.opt`	`x.lazy`	`x.nb`
1	85.7	86.07	40.66	30.53	61.55	58.88
2		75.57	35.2	55.69	116.44	116.21
4		73.24	52.83	111.21	220.6	220.45
8		45.61	81.29	159.51	339.08	331.41
16		32.98	46.9	238.22	496.28	532.57
32		35.53	58.9	378.48	789.86	836.45
64		31.22	89.74	412.21	891.81	994.08
128		11.26	0.54	224.02	553.8	574.05



Throughput for List Size: 8192, Contains: 0%, Add: 50%, Remove: 50% (Kops/sec)

Executable / Threads	`x.serial`	`x.cgl`	`x.fgl`	`x.opt`	`x.lazy`	`x.nb`
1	90.35	90.67	40.66	29.91	62.78	58.01
2		85.95	35.11	53.66	115.93	112.63
4		70.81	52.57	110.58	224.38	217.0
8		45.82	80.83	161.24	345.28	332.13
16		38.63	45.86	235.13	476.5	493.32
32		43.89	59.22	372.47	766.6	826.17
64		25.54	91.42	420.81	865.88	911.6
128		10.73	0.29	227.56	503.93	587.31

2.3.5 Παρατηρήσεις

- Όπως είναι λογικό, όσο αυξάνεται ο αριθμός των νημάτων, τόσο περισσότερες είναι και οι καταστάσεις συναγωνισμού. Υπό αυτές τις συνθήκες βλέπουμε πως η *non-blocking* υλοποίηση σημειώνει κάθε φορά την καλύτερη επίδοση, όταν έχουμε προσθήκες/διαγραφές στοιχείων.
- Παρατηρούμε ότι με το που εμφανίζονται προσθήκες/διαγραφές στην λίστα (ακόμα και στην περίπτωση 80/10/10) η μέγιστη επίδοση πέφτει κατά μια τάξη μεγέθους περίπου.
- Παρατηρούμε ότι όταν πηγαίνουμε από μέγεθος λίστας 1024 σε 8192, το μέγιστο throughput πέφτει κατά μία τάξη μεγέθους περίπου. Αυτό είναι λογικό, διότι αυξάνεται το κόστος διάσχισης της λίστας και επιπλέον χρειάζεται να αποκτηθούν περισσότερα κλειδώματα σε περιπτώσεις όπως αυτές του *hand-over-hand-locking*.

- Από τα διαγράμματα παρατηρούμε ότι οι υλοποιήσεις *coarse-grain locking* και *fine-grain locking* έχουν συγκρίσιμη (χαμηλή) επίδοση, οι υλοποιήσεις *lazy* και *non-blocking* έχουν συγκρίσιμη (υψηλή) επίδοση, ενώ η *optimistic* βρίσκεται κάπου στη μέση. Είναι εύλογες λοιπόν οι παρακάτω συγκρίσεις:

  - **Coarse-grain vs fine-grain locking:**

    * Αρχικά, οι υλοποιήσεις *coarse-grain* και *fine-grain locking* δεν κλιμακώνουν καθόλου, και μάλιστα πετυχαίνουν επίδοση χειρότερη και από την σειριακή για όλους τους αριθμούς νημάτων. Στις περιπτώσεις όπου έχουμε μόνο αναζήτηση στοιχείων η *coarse-grain* υλοποίηση έχει σημαντικά καλύτερη επίδοση από το *fine-grain locking*, γεγονός που οφείλεται στους αυστηρούς κανόνες που επιβάλλει το *fine-grain locking*, χρησιμοποιώντας συνεχώς κλειδώματα για την υλοποίηση της `contains()`. Ωστόσο στην περίπτωση που έχουμε προσθήκες/διαγραφές, για 1-4 νήματα η *coarse-grain* υλοποίηση αποδίδει λίγο καλύτερα, ενώ για 8-128 νήματα συμβαίνει το αντίθετο.

  - **Optimistic synchronization:**

    * Η optimistic υλοποίηση, παρόλο που είναι χειρότερη από τις *lazy* και *non-blocking* μεθόδους, καταφέρνει να είναι πιο γρήγορη από την σειριακή. Κλιμακώνει μέχρι και τα 64 νήματα, τιμή για την οποία πετυχαίνει τη μέγιστή της επίδοση. Για μέγεθος λίστας 1024, ξεπερνά την σειριακή υλοποίηση μόνο για 64 νήματα (κατά $\sim$157%), ενώ για μέγεθος λίστας 8192 την ξεπερνά ήδη από τα 4 νήματα (μέγιστο για 64 νήματα, κατά $\sim$450%). Οι επιδόσεις της *optimistic synchronization* υλοποίησης είναι εμφανώς καλύτερες για μεγαλύτερο μέγεθος λίστας, διότι μειώνονται οι πιθανότητες να πληρώσουμε το τίμημα της *optimistic* προσέγγισης, δηλαδή να αποτύχει το validate και να πρέπει να ξεκινήσουμε από την αρχή.

  - **Lazy vs non-blocking synchronization:**

    * Παρατηρούμε και για τα δύο μεγέθη λίστας στην περίπτωση workload 100-0-0 κλιμάκωση της *lazy* υλοποίησης όσο αυξάνεται ο αριθμός των νημάτων. Συνολικά αυτή η υλοποίηση παρουσιάζει την καλύτερη απόδοση συγκριτικά με τις υπόλοιπες μεθόδους. Ακολουθεί με αρκετά χαμηλότερη απόδοση η *non-blocking* υλοποίηση. Στην συγκεκριμένη περίπτωση καλείται μόνο η μέθοδος `contains()` (workload 100-0-0), και επομένως είναι λογικό η *lazy* υλοποίηση να υπερισχύει, καθώς δεν χρησιμοποιεί καθόλου κλειδώματα για την υλοποίηση της μεθόδου αυτής.
    * Αντίθετα, στην περίπτωση όπου πραγματοποιούμε εισαγωγές/διαγραφές, η υλοποίηση με *non-blocking* είναι πιο αποδοτική. Ωστόσο, αξίζει να σημειωθεί ότι το προβάδισμα της σε σχέση με την *lazy* υλοποίηση είναι μικρότερο από πριν, όπου είχαμε μέγεθος λίστας 1024. Η διαφορά γίνεται ιδιαίτερα αντιληπτή για 64 νήματα.
    * Η *lazy* υλοποίηση για μέγεθος λίστας 1024 και για πλήθος νημάτων 32-64 καταφέρνει να είναι αρκετά ανταγωνιστική με την *non-blocking* υλοποίηση. Ωστόσο για 128 νήματα παρατηρούμε μια κατακόρυφη πτώση στην απόδοση, καθιστώντας την μάλιστα χειρότερη και από την σειριακή υλοποίηση. Αυτό συμβαίνει διότι αφενός λόγω του μικρού μεγέθους λίστας αυξάνεται η πιθανότητα να έχουμε *race condition*, αφετέρου πολλά νήματα μοιράζονται τους ίδιους πυρήνες.

2.3.6 Συμπεράσματα

Τα παραπάνω γραφήματα επιβεβαιώνουν τις σκέψεις και την θεωρητική μελέτη που έχουμε πραγματοποιήσει για την επίδοση των διαφορετικών υλοποιήσεων απλά συνδεδεμένης λίστας.

Σύμφωνα με την περιγραφή των υλοποιήσεων, *fine-grain locking*, *coarse-grain locking* και *optimistic locking*, είναι λογικό αυτές να έχουν σημαντικά χειρότερη επίδοση. Μάλιστα, οι υλοποιήσεις {*fine,coarse*}*-grain locking* πετυχαίνουν σταθερά επιδόσεις χειρότερες από την σειριακή υλοποίηση. Η *optimistic* υλοποίηση, παρόλο που σίγουρα δεν προσφέρει την βέλτιστη λύση, ξεπερνά αρκετές φορές την σειριακή εκτέλεση, ιδιαίτερα για μεγάλες λίστες.

Από την άλλη, η *lazy* τεχνική προορίζεται ώστε να βελτιώσει την *optimistic*, βασιζόμενη στα καλά χαρακτηριστικά της (διατρέχει τη λίστα χωρίς κλειδώματα) και διορθώνοντας παράλληλα τα ελαττώματά της. Η βελτίωση είναι εμφανής, καθώς η υλοποίηση παρουσιάζει πολύ καλή κλιμάκωση για όλα τα πειράματα.

Ενδιαφέρον είναι το γεγονός ότι η *non-blocking* υλοποίηση, η οποία αποφεύγει τη χρήση κλειδωμάτων, δεν είναι πάντα καλύτερη από την *lazy* υλοποίηση.

Φαίνεται, συγκεκριμένα, ότι η προτιμητέα υλοποίηση εξαρτάται σε μεγάλο βαθμό από τα χαρακτηριστικά του προβλήματος που καλούμαστε να λύσουμε. Πράγματι, ενώ για τη μικρή λίστα (μεγέθους 1024) η *non-blocking* υλοποίηση κυριαρχεί αδιαμφισβήτητα (στις περιπτώσεις που δεν γίνεται μόνο αναζήτηση), δε συμβαίνει το ίδιο και στην περίπτωση της μεγάλης (μεγέθους 8192), όπου η *lazy* υλοποίηση αποδεικνύεται εξίσου αποδοτική.

3 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

3.1 Άλλη μια παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

3.1.1 Σκοπός της άσκησης

Στην παρούσα άσκηση καλούμαστε να υλοποιήσουμε τον αλγόριθμο K-means για μια κάρτα γραφικών NVIDIA μέσω του CUDA. Αρχικά φτιάχνουμε μια *naive* υλοποίηση, στην οποία μετέπειτα εφαρμόζουμε ορισμένες βελτιώσεις επίδοσης.

3.1.2 Δεδομένα

Στον φάκελο `/home/parallel/pps/2022-2023/a3/kmeans/` βρίσκονται οι σκελετοί για τις υλοποιήσεις που καλούμαστε να συμπληρώσουμε, καθώς και βοηθητικά αρχεία για την μεταγλώττιση και την εκτέλεση των υλοποιήσεων αυτών.

3.1.3 Ζητούμενα

Συνολικά υπάρχουν 4 υλοποιήσεις, και κάθε μία από αυτές βασίζεται πάνω στην προηγούμενη, επομένως η σειρά των υλοποιήσεων είναι η εξής:

$$\text{Naive} \Longrightarrow \text{Transpose} \Longrightarrow \text{Shared} \Longrightarrow \text{All-GPU}$$

Naive version Σε αυτή την έκδοση αναθέτουμε στην κάρτα γραφικών τον υπολογισμό των κοντινότερων clusters ανά iteration, που είναι και το πιο υπολογιστικά βαρύ κομμάτι. Για τον σκοπό αυτό, συμπληρώνουμε τα **TODO** στο `cuda_kmeans_naive.cu`:

1. Υλοποιούμε τον πυρήνα `find_nearest_cluster()`. Ο υπολογισμός της ευκλείδειας απόστασης γίνεται όπως πριν, απλά αυτή τη φορά έχουμε προσθέσει στην `euclid_dist_2()` την παράμετρο `objectId`, η οποία αντιστοιχεί στο thread ID:

```
for (i = 0; i < numCoords; i++) {
    float a = objects[objectId * numCoords + i];
    float b = clusters[clusterId * numCoords + i];

    ans += (a-b) * (a-b);
}
```

Επίσης, προκειμένου η ανανέωση του `delta` να είναι ασφαλής (δηλαδή μόνο ένα νήμα να έχει πρόσβαση κάθε φορά) χρησιμοποιούμε την `atomicAdd()` ως εξής:

```
if (deviceMembership[tid] != index)
    atomicAdd(devdelta, 1.0);
```

2. Πραγματοποιούμε τις ζητούμενες μεταφορές δεδομένων σε κάθε επανάληψη του αλγορίθμου, ώστε να μπορεί η CPU να υπολογίσει τα νέα κέντρα των clusters. Είναι προκαθορισμένος εξαρχής ο αριθμός των επαναλήψεων και του επιθυμητού *threshold*. Οι μεταφορές αυτές γίνονται με την χρήση της συνάρτησης `cudaMemcpy`. Για την αντιγραφή του πίνακα `clusters` από την μνήμη του *host* (CPU) στον πίνακα `deviceClusters` στην μνήμη του *device* (GPU):

```
cudaMemcpy(deviceClusters, clusters,
    numClusters*numCoords*sizeof(float), cudaMemcpyHostToDevice);
```

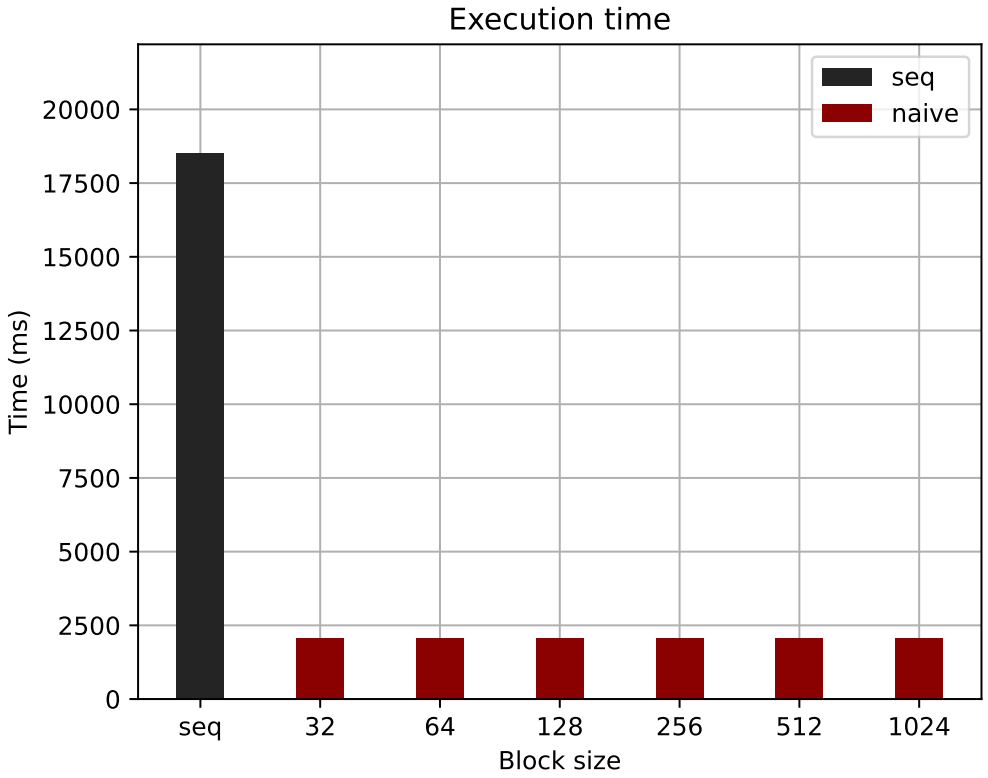
Αντίστοιχα, για την αντιγραφή του `membership` του *host* στο `deviceMembership` του *device*:

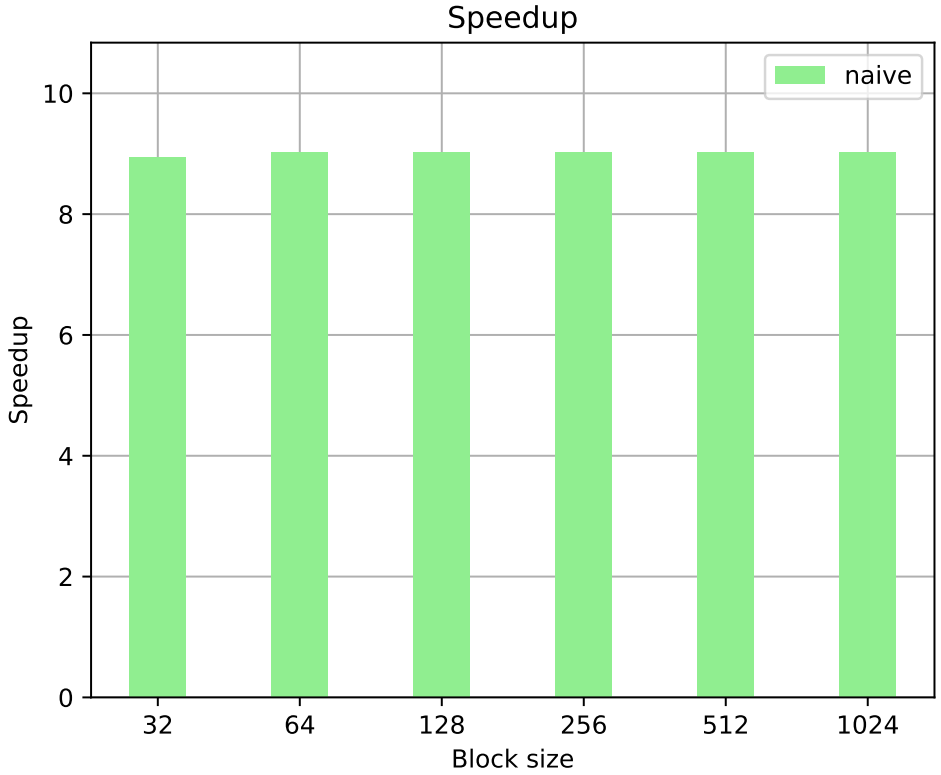
```
cudaMemcpy(membership, deviceMembership,
    numObjs*sizeof(int), cudaMemcpyDeviceToHost);
```

Η κατεύθυνση ορίζεται από τις παραμέτρους `cudaMemcpyHostToDevice` και `cudaMemcpyDeviceToHost`.

Στη συνέχεια αξιολογούμε την επίδοσή μας:

1. Πραγματοποιούμε μετρήσεις για την *naive* version και για την σειριακή έκδοση (`seq_kmeans.c`), με το configuration {Size, Coords, Clusters, Loops} = $\{256, 2, 16, 10\}$ και για `block_size` = $\{32, 64, 128, 256, 512, 1024\}$. Σημειώνεται ότι ο χρόνος που απεικονίζεται στα διαγράμματα είναι **ο χρόνος εκτέλεσης για 10 επαναλήψεις του αλγορίθμου** (Loops = 10). Δημιουργούμε barplot διαγράμματα για τον χρόνο εκτέλεσης και το speedup:





Παρατηρούμε ότι η σειριακή έκδοση χωρίς χρήση της GPU είναι περίπου μια τάξη μεγέθους πιο αργή από την *naive* εκδοχή. Λαμβάνοντας υπόψιν πως ο συνολικός αριθμός στοιχείων `numObjs` ανέρχεται στα 32M. Ο υψηλός αριθμός πυρήνων της GPU και η δυνατότητα που μας προσφέρει για πραγματοποίηση εξειδικεύμενων υπολογισμών γρήγορα είναι αυτή που μας προσφέρει την δυνατότητα, ώστε να έχουμε τα αποτελέσματα σε λίγα msec.

2. Με βάση τα παραπάνω barplot θα μελετήσουμε τώρα την συμπεριφορά της επίδοσης σε σχέση με το `block_size`.

Στην περίπτωση της *naive* υλοποίησης η επίδοση βελτιώνεται (∼20ms), όταν το `block_size` αυξάνεται από 32 σε 64, όμως αυτή η βελτίωση δεν συνεχίζεται για `block_size` μεγαλύτερο του 64, αλλά σταθεροποιείται.

Η αύξηση του `block_size` οδηγεί στην μείωση του συνολικού αριθμού των block και ο αριθμός των threads για κάθε block, καθορίζεται από το `block_size` σύμφωνα με τον κώδικα μας.

Transpose version Μία πρώτη βελτιστοποίηση που μπορούμε να κάνουμε είναι να αλλάξουμε τη δομή των δεδομένων που έχουν διαστάσεις από *row-based* σε *column-based* indexing. Συγκεκριμένα, θα κάνουμε αυτή την αλλαγή για τους πίνακες `clusters` και `objects`, δημιουργώντας τους αντίστοιχους πίνακες `dimClusters` και `dimObjects`. Για το σκοπό αυτό, συμπληρώνουμε τα **TODO** στο `cuda_kmeans_transpose.cu`, επαναχρησιμοποιώντας τμήματα κώδικα από την *naive* υλοποίηση για τα τμήματα εκείνα που δεν απαιτούν αλλαγές:

1. Υλοποιούμε την `euclid_dist_2_transpose()` για τη νέα, ανεστραμμένη δομή των δεδομένων. Πλέον το indexing γίνεται ως εξής:

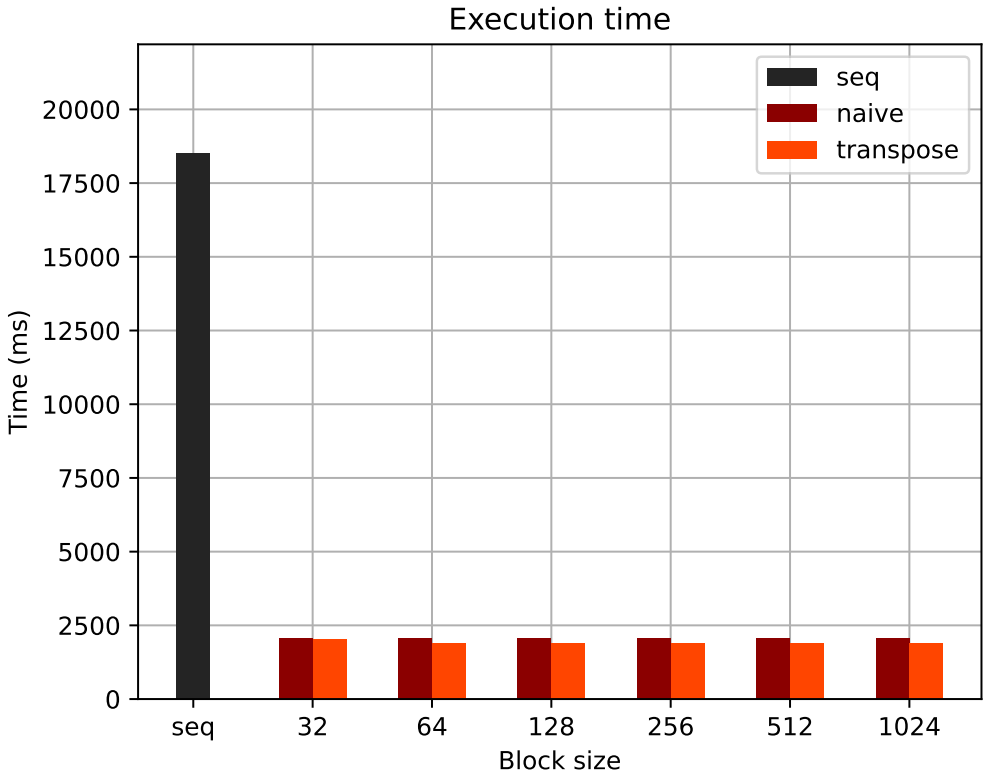
```
for (i = 0; i < numCoords; i++) {
    float a = objects[objectId * numCoords + i];
    float b = clusters[clusterId * numCoords + i];

    ans += (a-b) * (a-b);
}
```

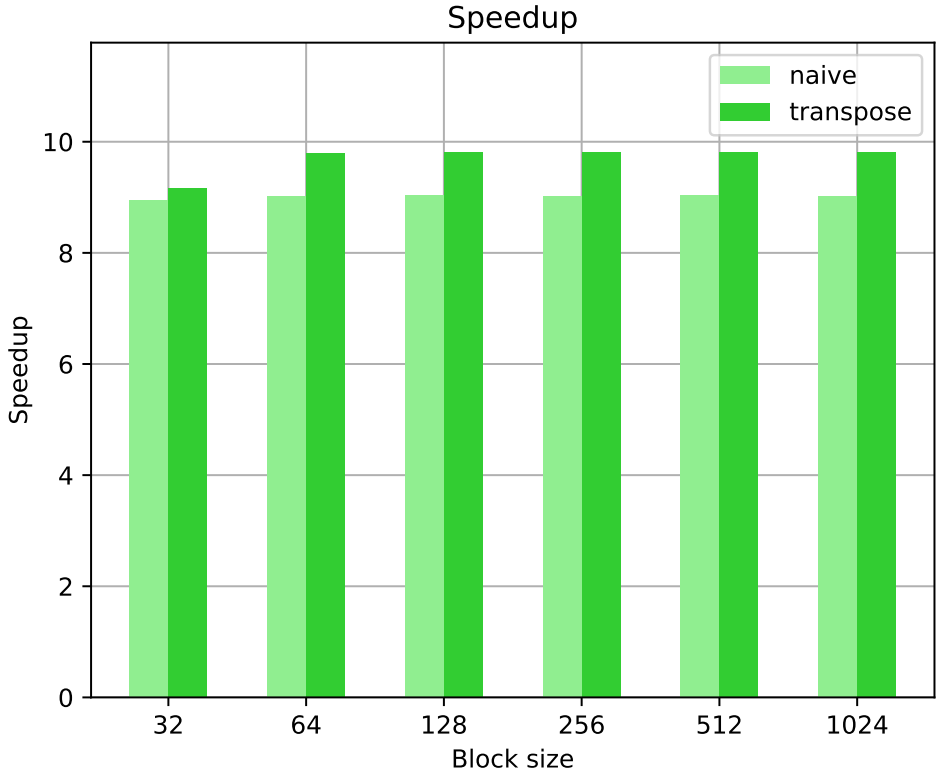
2. Για τη μετατροπή στη νέα μορφή, φροντίζουμε να κάνουμε τις σωστές αρχικοποιήσεις και μετατροπές. Αρχικοποιούμε τους πίνακες `dimObjects` και `dimClusters`, που έχουν ανάστροφες διαστάσεις από τους `objects` και `clusters` αντίστοιχα. Πριν αρχίσει το `do-while` loop, εκτελούμε τις κατάλληλες μεταφορές, της μορφής:

```
for i:
    for j:
        transposed[j][i] = original[i][j]
```

Το ίδιο φροντίζουμε να κάνουμε και αφού τελειώσει το `do-while` loop.
Στη συνέχεια αξιολογούμε την επίδοσή μας:

1. Επαναλαμβάνουμε τις παραπάνω μετρήσεις για την *transpose* υλοποίηση και την συγκρίνουμε με την *naive* τοποθετώντας τις δύο υλοποιήσεις στο ίδιο διάγραμμα:





Στην περίπτωση της *transpose* υλοποίησης η επίδοση βελτιώνεται όταν το `block_size` αυξάνεται από 32 σε 64, όμως αυτή η βελτίωση δεν συνεχίζεται για `block_size` μεγαλύτερο του 64.

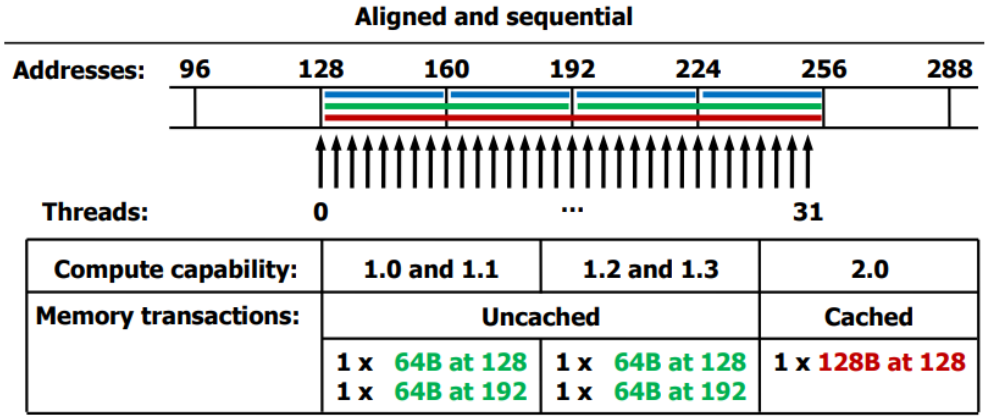
2. Παρατηρούμε μία μικρή βελτίωση της *transpose* υλοποίησης σε σχέση με την *naive*, παρόλο που φαινομενικά οι δύο υλοποιήσεις είναι ίδιες. Ωστόσο, η αναστροφή των πινάκων προσφέρει ένα

πλεονέκτημα στην εκτέλεση του πυρήνα στην GPU:

Κατά τον υπολογισμό της απόστασης, πλέον το επόμενο thread διαβάζει το διπλανό στοιχείο στη μνήμη, και όχι το στοιχείο που βρίσκεται στην από κάτω στήλη, δηλαδή αρκετές θέσεις πιο δίπλα. Αυτό φαίνεται στους δύο τρόπους δεικτοδότησης, όπου `objectId` είναι ο όρος που εξαρτάται από το thread ID:

- `objects[objectId*numCoords + i]` στην περίπτωση της *naive* υλοποίησης.
- `objects[i*numObjs + objectId]` στην περίπτωση της *transpose* υλοποίησης,

Με αυτόν τον τρόπο, εκμεταλλευόμαστε το **memory coalescing** του CUDA, χάρη στο οποίο οι αναφορές σε συνεχόμενες θέσεις μνήμης μπορούν να συνενωθούν σε μία ενιαία αναφορά, όπως φαίνεται στο παρακάτω σχήμα:



Shared version Μία ακόμη βελτιστοποίηση που μπορούμε να κάνουμε είναι να τοποθετήσουμε τα `clusters` στην shared memory της GPU ώστε τα στοιχεία κάθε block να μπορούν να τα προσπελάσουν ταχύτερα. Γι' αυτό τον σκοπό συμπληρώνουμε τα **TODO** στο `cuda_kmeans_shared.cu`, ενώ χρησιμοποιούμε την *transpose* υλοποίηση για ό,τι δεν χρειάζεται αλλαγή:

1. Ορίζουμε το μέγεθος διαμοιραζόμενης μνήμης που χρειάζεται η υλοποίηση μας, δηλαδή το μέγεθος του πίνακα `clusters`:

```
clusterBlockSharedDataSize = numCoords * numClusters * sizeof(float);
```

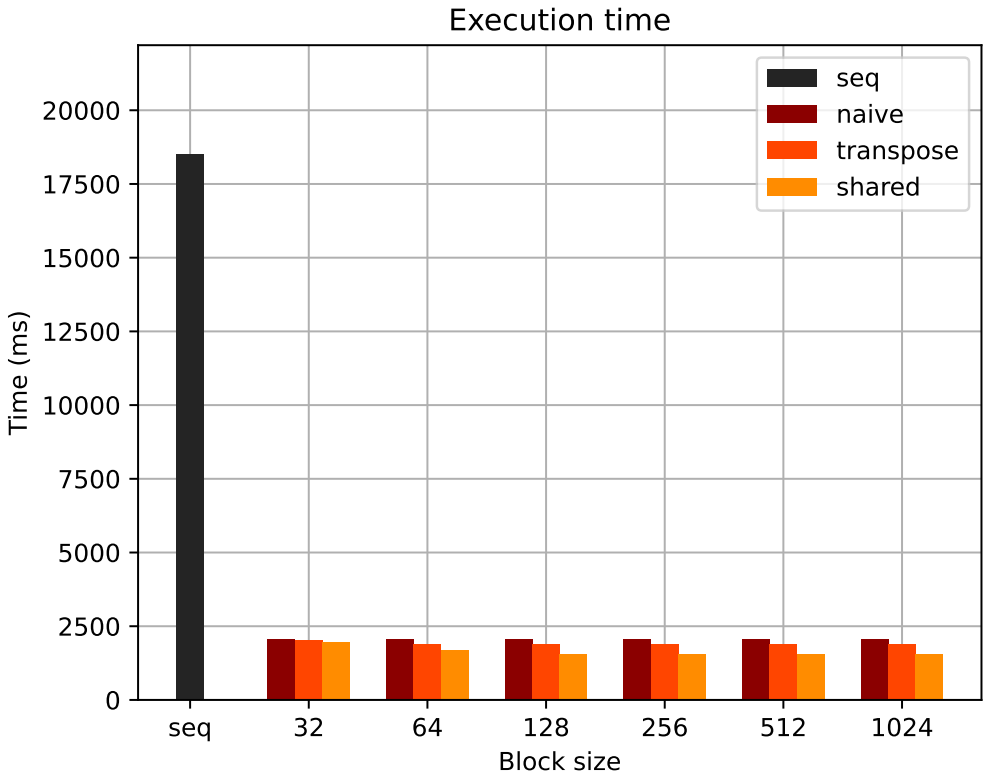
2. Προσθέτουμε στον πυρήνα `find_nearest_cluster()` την μεταφορά των clusters στην shared memory (προσέχουμε ώστε να γίνει συγχρονισμός των threads εντός του πυρήνα προτού προχωρήσουμε σε περαιτέρω ενέργειες):

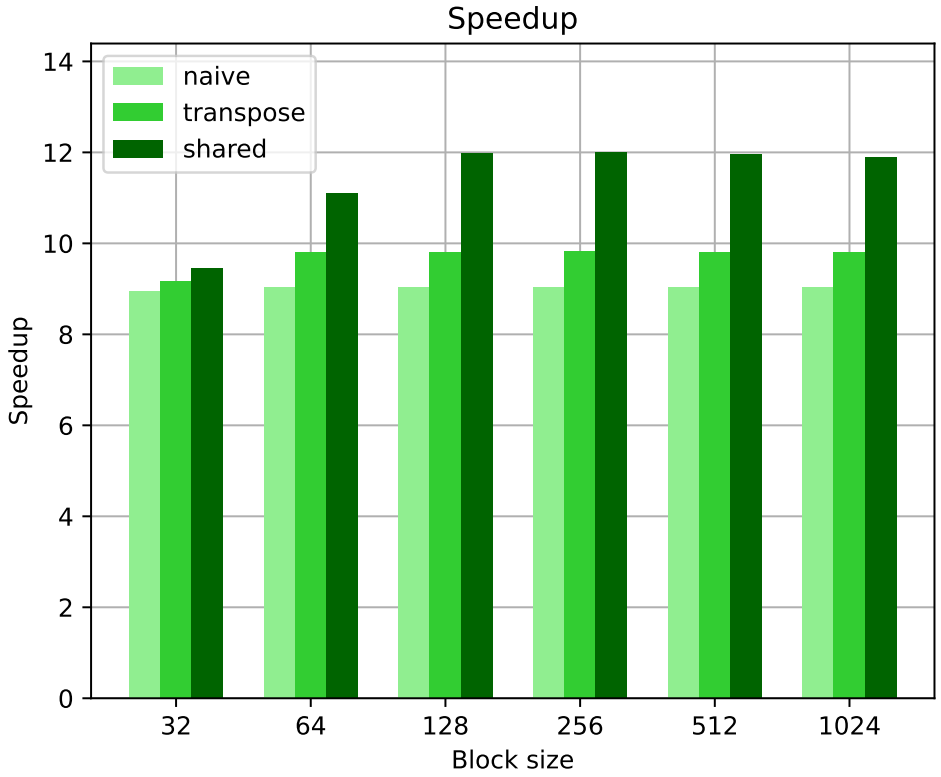
```
for (i = threadIdx.x; i < numClusters * numCoords; i += blockDim.x)
    shmemClusters[i] = deviceClusters[i];

__syncthreads();
```

Για να αποφύγουμε άσκοπες μεταφορές, αναθέτουμε στο κάθε thread ένα μέρος των μεταφορών (`i = threadIdx.x`), και μέσω του *stride* (`i += blockDim.x`) εξασφαλίζουμε ότι κανένα thread δεν έχει επικαλύψεις με τα υπόλοιπα.

Στη συνέχεια αξιολογούμε την επίδοσή μας:

1. Επαναλαμβάνουμε τις παραπάνω μετρήσεις για την *shared* υλοποίηση και την τοποθετούμε στο ίδιο διάγραμμα με τις προηγούμενες υλοποιήσεις για να συγκρίνουμε:





Παρατηρούμε ότι η επίδοση της *shared* υλοποίησης είναι αρκετά καλύτερη, αφού πετυχαίνει μέγιστο speedup 12, ενώ οι *naive* και *transpose* υλοποιήσεις πετυχαίνουν μέγιστο speedup 9 και 10 αντίστοιχα.

Επίσης παρατηρούμε ότι αυτή τη φορά η αύξηση του `block_size` αρχικά βελτιώνει την απόδοση, ενώ για τιμές μεγαλύτερες του 128 δεν έχει κάποια επίδραση. Μάλιστα, παρατηρείται μια πολύ μικρή μείωση της επίδοσης για `block_size` 512 και 1024. Η βελτίωση εξηγείται επειδή όσο μεγαλύτερο είναι το `block_size`, τόσο λιγότερες φορές χρειάζεται να γίνει "εκκίνηση" των Streaming Multiprocessors, και άρα μειώνεται το συγκεκριμένο overhead. Από την άλλη, όταν το `block_size` γίνει αρκετά μεγάλο, αυξάνεται το κόστος συγχρονισμού λόγω της κλήσης `__syncthreads()`.

Σύγκριση υλοποιήσεων/Bottleneck Analysis Έχοντας ολοκληρώσει τις τρεις παραπάνω υλοποιήσεις, προχωράμε σε μία βαθύτερη ανάλυση της επίδοσης, αλλά και πιθανών προβλημάτων που εμφανίζονται.

1. Προσθέτουμε τρεις timers για να μετρήσουμε τον χρόνο εκτέλεσης του `while` loop με περισσότερη λεπτομέρεια:

- `gpu_time`: Για τον χρόνο που χρειάζεται η GPU για την εκτέλεση του πυρήνα `find_nearest_cluster()`.
- `transfer_time`: Για τον χρόνο που απαιτούν οι μεταφορές δεδομένων από την CPU στην GPU και αντίστροφα.
- `cpu_time`: Για τον χρόνο που χρειάζεται η CPU για την ανανέωση των cluster centers.

Ακολουθούν τα αποτελέσματα των παραπάνω μετρήσεων (χρόνος σε **msec**):

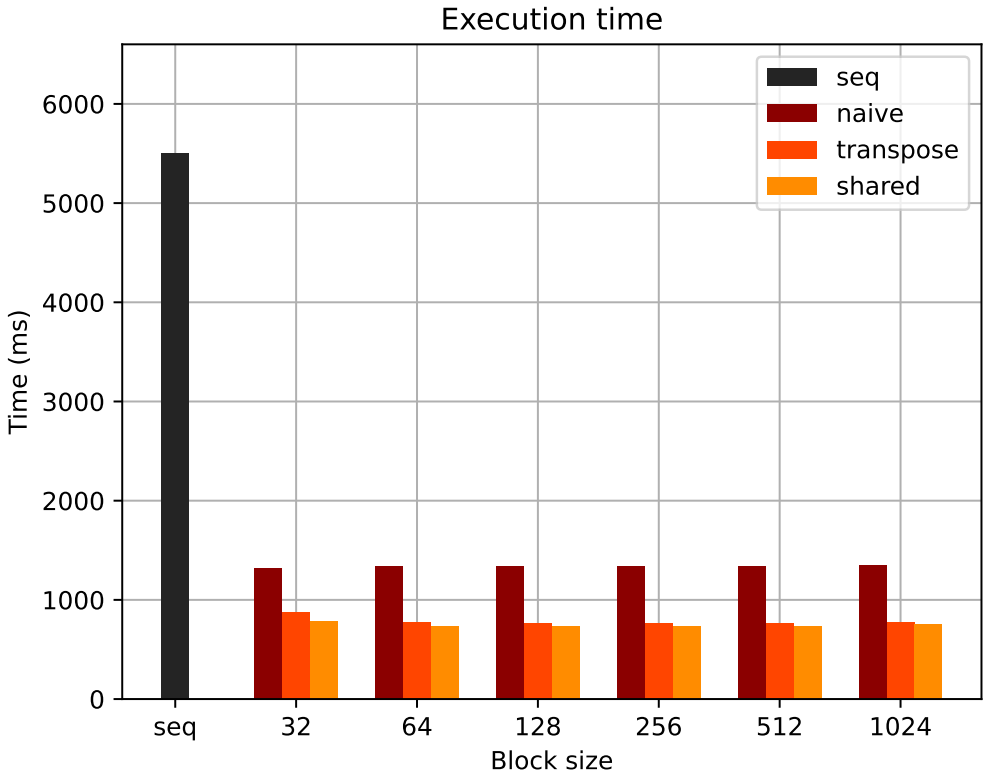
Executable	Block Size	GPU	Transfer	CPU
naive	32	781.684399	406.546831	889.051676
	64	755.886555	406.658888	887.795448
	128	755.967140	406.162024	887.254238
	256	756.000042	406.723976	889.050961
	512	755.998611	406.552315	889.670134
	1024	756.035089	405.648708	885.400057
transpose	32	815.342903	406.171322	798.816681
	64	680.954218	405.749559	796.627522
	128	680.973291	406.416416	797.651291
	256	680.988550	406.062603	797.857761
	512	681.020021	406.558037	798.206806
	1024	681.463718	406.316757	798.721075
shared	32	751.770020	406.015873	798.875093
	64	463.678837	406.492949	799.174309
	128	336.888552	406.624317	799.696684
	256	338.410854	405.832529	796.896696
	512	342.978716	405.841827	796.505690
	1024	350.111008	406.569004	798.194885

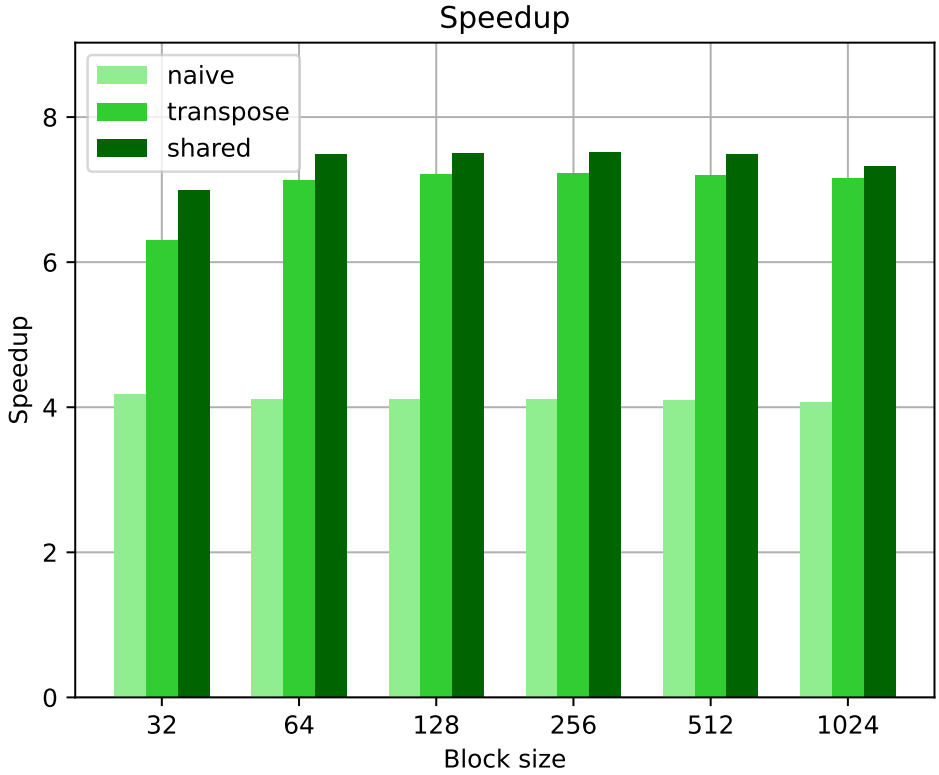
- Παρατηρούμε ότι ο χρόνος της GPU εξαρτάται από την υλοποίηση, όπως έχουμε αναφέρει και παραπάνω

- Ο χρόνος μεταφοράς των δεδομένων είναι ο ίδιος σε όλες τις υλοποιήσεις και για όλα τα block sizes, αφού εξαρτάται μόνο από το μέγεθος των δεδομένων, που παραμένει σταθερό.
- Όσον αφορά τον χρόνο της CPU, παρατηρούμε ότι κι αυτός διατηρείται σε γενικές γραμμές σταθερός μεταξύ υλοποιήσεων. Μάλιστα, για κάθε συγκεκριμένη υλοποίηση είναι ανεξάρτητος το `block_size`.

Συμπεραίνουμε επομένως ότι το μεγαλύτερο ποσοστό του χρόνου **δεν** αφιερώνεται στον υπολογισμό των nearest clusters (GPU). Αντίθετα, αφιερώνεται στις μεταφορές δεδομένων μεταξύ CPU-GPU και στην ενημέρωση των cluster centroids (CPU). Έτσι, παρόλο που έχουμε πετύχει διπλάσια επίδοση όσον αφορά το κομμάτι της GPU, η πραγματική βελτίωση είναι πολύ μικρότερη. Αυτό το κομμάτι καλείται να λύσει και η *All-GPU* υλοποίηση του αλγορίθμου που θα δούμε παρακάτω.

2. Αυτή τη φορά δοκιμάζουμε το configuration {Size, Coords, Clusters, Loops} = $\{256, 16, 16, 10\}$, πάλι για `block_size` = $\{32, 64, 128, 256, 512, 1024\}$. Ακολουθούν τα barplot διαγράμματα χρόνου εκτέλεσης και speedup:





Σχολιασμός:

- Παρατηρούμε ότι η σειριακή έκδοση χωρίς χρήση της GPU είναι περίπου μια τάξη μεγέθους πιο αργή από την *shared* εκδοχή. Αρκετά έντονη όμως είναι και η βελτίωση όλων των υλοποιήσεων έναντι της σειριακής, με την *shared* να ξεχωρίζει, αφού για όλα τα block sizes πετυχαίνει τον καλύτερο χρόνο.

- Συνεχίζοντας στην πιο προσεκτική μελέτη των υλοποιήσεων με βάση το speedup, παρατηρούμε ότι η *shared* εκδοχή είναι πάνω από 1.5 φορές ταχύτερη από την *naive*. Ταυτόχρονα, ξεπερνάει και την *transpose* εκδοχή με την διαφορά εδώ να είναι μικρότερη.
- Για όλες τις υλοποιήσεις ανεξάρτητα, ισχύει πως όταν το `block_size` μεταβαίνει από 32 σε 64 υπάρχει μια μείωση στον συνολικό χρόνο εκτέλεσης, η οποία μετά σταθεροποιείται.

Όπως και πριν παραθέτουμε και την ανάλυση του χρόνου με βάση τους GPU/transfer/CPU timers (χρόνος σε **msec**):

Executable	Block Size	GPU	Transfer	CPU
naive	32	1061.411142	53.001881	214.617968
	64	1048.543692	53.006411	214.846134
	128	1066.209316	52.986860	214.602947
	256	1054.354668	53.062201	214.432240
	512	1073.208094	52.871943	214.029551
	1024	1095.669508	52.976370	213.333130
transpose	32	260.524273	53.055525	548.460960
	64	174.290419	53.054810	548.349857
	128	171.435595	52.975655	548.605680
	256	169.901848	52.941561	546.984673
	512	171.275854	52.962303	547.971725
	1024	179.191351	53.029299	548.082352
shared	32	194.020271	52.908897	548.721790
	64	137.781858	53.027391	550.057173
	128	133.891821	53.045511	550.162792
	256	132.075071	52.941084	547.194004
	512	132.899761	52.978754	548.501015
	1024	140.088797	53.067923	550.163269

Σχολιασμός:
Στην παρούσα υλοποίηση με `numCoords=16`, ο συνολικός αριθμός των `numObjs` μειώνεται σε 4M αντί για τα 32M που είχαμε για `numCoords=2`. Αυτό συμβαίνει, επειδή η μεταβλητή `numObjs` καθορίζεται ως εξής:

```
numObjs = (dataset_size*1024*1024) / (numCoords*sizeof(float));
```

Από τα παραπάνω αξίζει να επισημάνουμε τα εξής για τους κυριότερους πίνακες που χρησιμοποιούμε στον αλγόριθμο: Ο πίνακας `clusters` διατηρεί συνολικά πάντα κοινό πλήθος δεδομένων, ανεξάρτητα από την τιμή της μεταβλητής `numCoords`, γινόμενο `numCoords*numObjs` είναι πάντα σταθερό. Το μέγεθος του πίνακα `membership` εξαρτάται αποκλειστικά από την μεταβλητή `numObjs`. Επομένως, στην περίπτωση `numCoords=16`, το πλήθος των `numObjs` μειώνεται, άρα και το μέγεθος του πίνακα `membership`.

- Στα παραπάνω διαγράμματα γίνεται ιδιαίτερα αισθητή η μείωση χρόνου εκτέλεσης της σειριακής υλοποίησης από `numCoords=2` σε `numCoords=16` κατά περίπου 30%. Η βελτίωση του χρόνου στην υλοποίηση οφείλετε στην μείωση του συνολικού αριθμού `numObjs`, όπου συνεπάγεται με λιγότερες προσβάσεις στον πίνακα `memberships`.

- Επίσης παρατηρούμε ότι η επίδοση της *transpose* υλοποίησης, σε ότι αφορά το `gpu_time`, σε σχέση με την *naive* είναι σημαντικά μεγαλύτερη για `numCoords=16` απ' ό,τι για `numCoords=2`.
- Όπως έχει ειπωθεί και παραπάνω, οι μεταφορές δεδομένων που αφορούν το `transfer_time` αναφέρονται στους πίνακες `clusters` και `membership`. Παρατηρούμε πως η τιμή του `transfer_time` κυμαίνεται για `numCoords=2` στα 406ms, ενώ για `numCoords=16` η τιμή μειώνεται στα 56ms. Επομένως, γίνεται φανερό πως η μείωση των `numObjs` από 32M σε 4M οδηγεί και σε ευθέως ανάλογη ελάττωση του χρόνου. Για την παραπάνω μείωση οφείλεται η αλλαγή στο μέγεθος του πίνακα `membership`.
- Παρατηρούμε ότι ο χρόνος που αφιερώνεται στην CPU μειώνεται για `numCoords=16`, αφού ο αριθμός πράξεων που εκτελείται στην CPU είναι ανάλογος του `numObjs`, το οποίο οπως αναφέραμε παραπάνω μειώνεται για το δεύτερο configuration.
- Για `numCoords=16` παρατηρούμε ότι το CPU time υπερδιπλασιάζεται από την *naive* στις εκδοχές *transpose* και *shared*. Στις column-based υλοποιήσεις δεν ευνοείται το locality των στοιχείων, με αποτέλεσμα να έχουμε περισσότερα cache misses σε αυτήν την περίπτωση και άρα χειρότερη επίδοση.

Από τα παραπάνω εγείρεται το ερώτημα αν η *shared* υλοποίηση με column based indexing είναι κατάλληλη για arbitrary configurations.

Με χρήση shared memory καταφέρνουμε να επιτύχουμε βελτιωμένη επίδοση, όπως είδαμε παραπάνω, αφού επιτρέπουμε στα νήματα μέσα σε ένα block να μοιράζονται και να επαναχρησιμοποιούν δεδομένα, μειώνοντας τον αριθμό των προσβάσεων καθολικής μνήμης, οι οποίες είναι πιο αργές από τις προσβάσεις κοινής μνήμης.

Ωστόσο, επειδή το μέγεθος του πίνακα `clusters` εξαρτάται από το εκάστοτε configuration (`numCoords*numClusters*sizeof(float)`), ενδεχομένως να ξεπερνά το μέγεθος της shared μνήμης, με αποτέλεσμα η *shared* υλοποίηση να μην μπορεί να εκτελεστεί. Αντίθετα, οι *naive* και *transpose* υλοποιήσεις δεν έχουν αυτόν τον περιορισμό.

Από τα παραπάνω μπορούμε να συμπεράνουμε πως η *shared* υλοποίηση με column-based indexing είναι κατάλληλη για συγκεκριμένα configurations και αρχιτεκτονικές GPU.

Bonus: Full-Offload (All-GPU) version Coming soon...

3.1.4 Σημειώσεις

- Παρατηρήσαμε ότι στα αρχεία `main_sec.c` και `main_gpu.cu`, η συνάρτηση `check_repeated_clusters()` κάνει in-place ταξινόμηση του πίνακα `clusters`. Ενώ αυτό δεν επηρεάζει την ορθότητα των αποτελεσμάτων, αλλάζει την σειρά με την οποία τυπώνονται τα τελικά κέντρα των clusters στην *transpose* υλοποίηση, με αποτέλεσμα να μην είναι εύκολη η επαλήθευση της λύσης, είτε εποπτικά είτε με χρήση του `VALIDATE` flag, παρόλο που η λύση τελικά είναι σωστή. Αυτό συμβαίνει διότι η αντιγραφή των δεδομένων από τον πίνακα `clusters` στον `dimClusters` γίνεται αφού ο `clusters` έχει ταξινομηθεί. Γι' αυτό, έχουμε αλλάξει τον κώδικα ώστε η `check_repeated_clusters()` να καλείται πάνω σε ένα **αντίγραφο** του `clusters` (`copyClusters` στον κώδικα).
- Ο αλγόριθμος K-means δεν είναι ο πιο ιδανικός για την ανάδειξη των δυνατοτήτων της παραλληλοποίησης σε GPU, αφού δεν έχει υψηλό computational intensity.

4 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

4.1 Σκοπός της άσκησης

Σκοπός της παρούσας άσκησης είναι η μελέτη της κατανομής της θερμότητας σε μια δοσμένη περιοχή, καθώς μεταβάλλεται ο χρόνος. Για την επίλυση του προβλήματος της διάδοσης θερμότητας σε δύο διαστάσεις θα χρησιμοποιήσουμε τρεις υπολογιστικούς πυρήνες.
Η παραλληλοποίηση της εξίσωσης θερμότητας γίνεται με 3 διαφορετικές μεθόδους (Jacobi, Gauss-Seidel SOR, Red-Black SOR) σε μοντέλο ανταλλαγής μηνυμάτων πάνω από κατανεμημένη αρχιτεκτονική με χρήση της βιβλιοθήκης MPI.

4.2 Ζητούμενα

Ζητούμενα της άσκησης είναι να ανακαλύψουμε τον παραλληλισμό του κάθε αλγορίθμου και να σχεδιάσουμε την παραλληλοποίησή του σε αρχιτεκτονικές κατανεμημένης μνήμης με μοντέλο ανταλλαγής μηνυμάτων. Στην συνέχεια αναπτύσουμε παράλληλο πρόγραμμα για κάθε μέθοδο με χρήση της βιβλιοθήκης MPI. Τέλος, πραγματοποιούμε μετρήσεις τόσο με αλλά και χωρίς έλεγχο σύγκλισης και σχολιάζουμε τα αποτελέσματα.

4.3 Μέθοδος Jacobi

Με χρήση της μεθόδου Jacobi σε κάθε χρονικό βήμα μπορούμε να υπολογίσουμε την τιμή ενός στοιχείου υπολογίζοντας τον μέσο όρο των τιμών των τεσσάρων γειτονικών στοιχείων του (βόρειος, δυτικός, ανατολικός, νότιος γείτονας) που γνωρίζουμε από το προηγούμενο χρονικά βήμα.

$$u_{x,y}^{t+1} = \frac{u_{x-1,y}^{t} + u_{x,y-1}^{t} + u_{x+1,y}^{t} + u_{x,y+1}^{t}}{4}$$

Code Παρακάτω φαίνεται η υλοποίηση της παραπάνω μεθόδου:

```
for (t = 0; t < T && ! converged ; t ++)
    for (i = 1; i < X - 1; i ++)
        for (j = 1; j < Y - 1; j ++)
            U[t+1][i][j]=(1/4)*(U[t][i-1][j]+U[t][i][j-1]+U[t][i+1][j]
            +U[t][i][j+1]);
    converged = check_convergence(U[t+1], U[t]);
```

Task graph Παρακάτω βλέπουμε το task graph της υλοποίησης, όπου φαίνονται πιο ξεκάθαρα οι εξαρτήσεις των στοιχείων του χρονικού βήματος t+1 από το βήμα t. Για ευκολία απεικονίζουμε το task graph μόνο για δύο χρονικές στιγμές :

Παρατηρούμε τόσο από τον κώδικα και επιβεβαιώνουμε και από την υλοποίηση του task graph, πως τα δύο εσωτερικά for-loops είναι παραλληλοποιήσιμα, αφού είναι ανεξάρτητα.

4.4 Μέθοδος Gauss-Seidel SOR

Η μέθοδος Jacobi αν και έχει υψηλή δυνατότητα παραλληλοποίησης, παρουσιάζει αρκετά αργό ρυθμό σύγκλισης. Το πρόβλημα αυτό επιλύει η μέθοδος Gauss-Seidel, όπου για τον υπολογισμό ενός

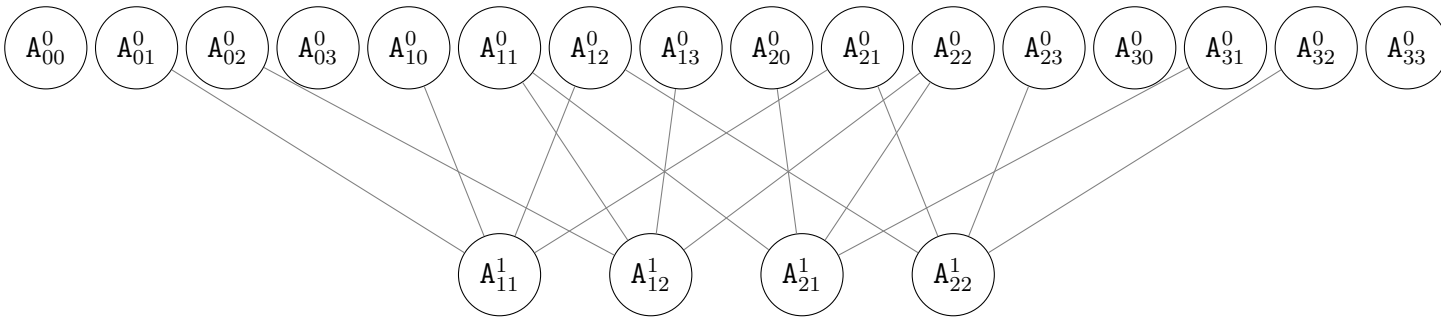


Figure 1: Jacobi task graph

στοιχείου χρησιμοποιεί πληροφορίες τόσο από την τωρινή χρονική στιγμή t+1 όσο και από την προηγούμενη t. Πιο συγκεκριμένα για τον υπολογισμό ενός στοιχείου χρησιμοποιεί τις τιμές της ίδιας χρονικής στιγμής του βόρειου και ανατολικού γείτονα και τις τιμές του νότιου και δυτικού γείτονα προηγούμενης χρονικής στιγμής. Από αυτές εξάγουμε πάλι τον μέσο όρο τους, όπως φαίνεται παρακάτω:

$$u^{t+1}_{x,y} = \frac{u^{t+1}_{x+1,y} + u^{t+1}_{x,y-1} + u^{t}_{x+1,y} + u^{t}_{x,y+1}}{4}$$

Στην SOR (Successive Over-Relaxation) υλοποίηση του Gauss-Seidel αξιοποιούμε για τον υπολογισμό της τιμής ενός στοιχείου στο χρονικό βήμα t+1 πέρα από τα προαναφερθέντα και την τιμή του ίδιου στοιχείου που γνωρίζουμε από την χρονική στιγμή t. Με αυτόν τον τρόπο καταφέρνουμε να επιταχύνουμε ακόμα περισσότερο τον ρυθμό σύγκλισης του αλγορίθμου. Η νέα υλοποίηση φαίνεται παρακάτω:

$$u^{t+1}_{x,y} = u^{t}_{x,y} + \omega \cdot \frac{u^{t+1}_{x+1,y} + u^{t+1}_{x,y-1} + u^{t}_{x+1,y} + u^{t}_{x,y+1}}{4}$$

Code Με χρήση της μεθόδου Gauss-Seidel σε κάθε χρονικό βήμα

```
for (t=0; t<T && ! converged; t++)
    for (i=1; i<X - 1; i++)
        for (j=1; j<Y-1; j++)
            if ((i+j)%2==0)
                U[t+1][i][j]=U[t][i][j]+(omega/4)*(U[t][i-1][j]+
                U[t][i][j-1]+U[t][i+1][j]+U[t][i][j+1]-4*U[t][i][j]);
```

Task graph

4.5 Μέθοδος Red-Black

Η μέθοδος Red-Black παρουσιάζει αρκετά κοινά σημεία με την μέθοδο Gauss-Seidel που μελετήσαμε παραπάνω, με την διαφορά ότι εδώ τα στοιχεία του πίνακα χωρίζονται σε δύο ομάδες (κόκκινη, μαύρη). Ο διαχωρισμός αυτός γίνεται με βάση την θέση του κάθε στοιχείου. Ο υπολογισμός ενός στοιχείου (`current`) πραγματοποιείται σε δύο φάσεις, πρώτα γίνεται ο υπολογισμός της κόκκινης ομάδας (κόκκινα κελιά) και μετά της μαύρης (μαύρα κελιά). Παρακάτω φαίνονται οι βασικές συναρτήσεις που χρησιμοποιούμε για τον υπολογισμό:

$$u^{t+1}_{x,y} = u^{t}_{x,y} + \omega \cdot \frac{u^{t}_{x-1,y} + u^{t}_{x,y-1} + u^{t}_{x,y+1} - 4u^{t}_{x,y}}{4}, when(x+2)\%2 == 0$$

$$u^{t+1}_{x,y} = u^{t}_{x,y} + \omega \cdot \frac{u^{t+1}_{x-1,y} + u^{t+1}_{x,y-1} + u^{t+1}_{x+1,y} + u^{t+1}_{x,y+1} - 4u^{t}_{x,y}}{4}, when(x+2)\%2 == 1$$

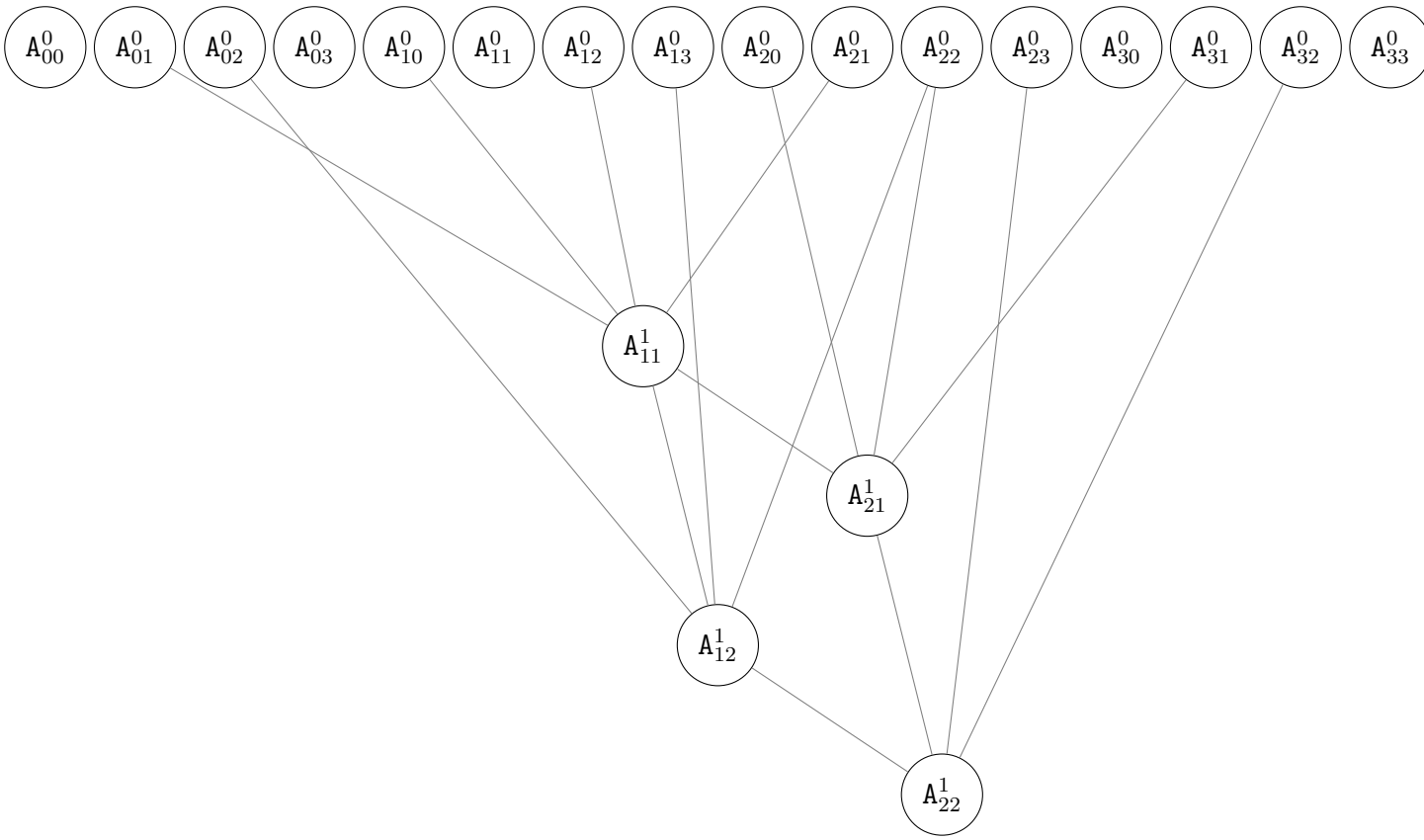


Figure 2: Gauss-Seidel SOR task graph

Code

```
for (t=0; t<T && ! converged ; t++)
    for (i=1; i<X-1; i++)
        for (j=1; j<Y-1; j++)
            if ((i+j)%2 == 0)
                U[t+1][i][j]=U[t][i][j]+(omega/4)*(U[t][i-1][j]+
                U[t][i][j-1]+U[t][i+1][j]+U[t][i][j+1]-4*U[t][i][j ]);

    for (i=1; i<X-1; i++)
        for (j=1; j<Y-1; j++)
            if ((i+j)%2 == 1)
                U[t+1][i][j]=U[t][i][j]+(omega/4)*(U[t+1][i-1][j]+U[t+1][i][j-1]
                +U[t+1][i+1][j]+U[t+1][i][j+1]-4*U[t][i][j]);
    converged = check_convergence(U[t+1],U[t]);
```

Μεταξύ των ”μαύρων” και ”κόκκινων” κελιών που υπολογίζουμε υπάρχουν αρκετές εξαρτήσεις. Στην αρχική φάση υπολογισμού των κόκκινων στοιχείων εντοπίζεται εξάρτηση από τα μαύρα στοιχεία του `previous` πίνακα (γείτονες και το ίδιο το στοιχείο). Στην δεύτερη φάση, ο υπολογισμός των μαύρων στοιχείων εξαρτάται από τα κόκκινα στοιχεία του πίνακα `current`. Επομένως, η πρώτη φάση μπορεί να παραλληλοποιηθεί, δηλαδή τα κόκκινα στοιχεία μπορούν να υπολογιστούν παράλληλα, αφού εξαρτώνται μόνο από τον πίνακα `previous`, ενώ η δεύτερη φάση δεν μπορεί να παραλληλοποιηθεί. Για οπτικοποίηση των παραπάνω προσφέρεται το παρακάτω task graph, όπου φαίνονται πιο καθαρά οι εξαρτήσεις:

Task Graph

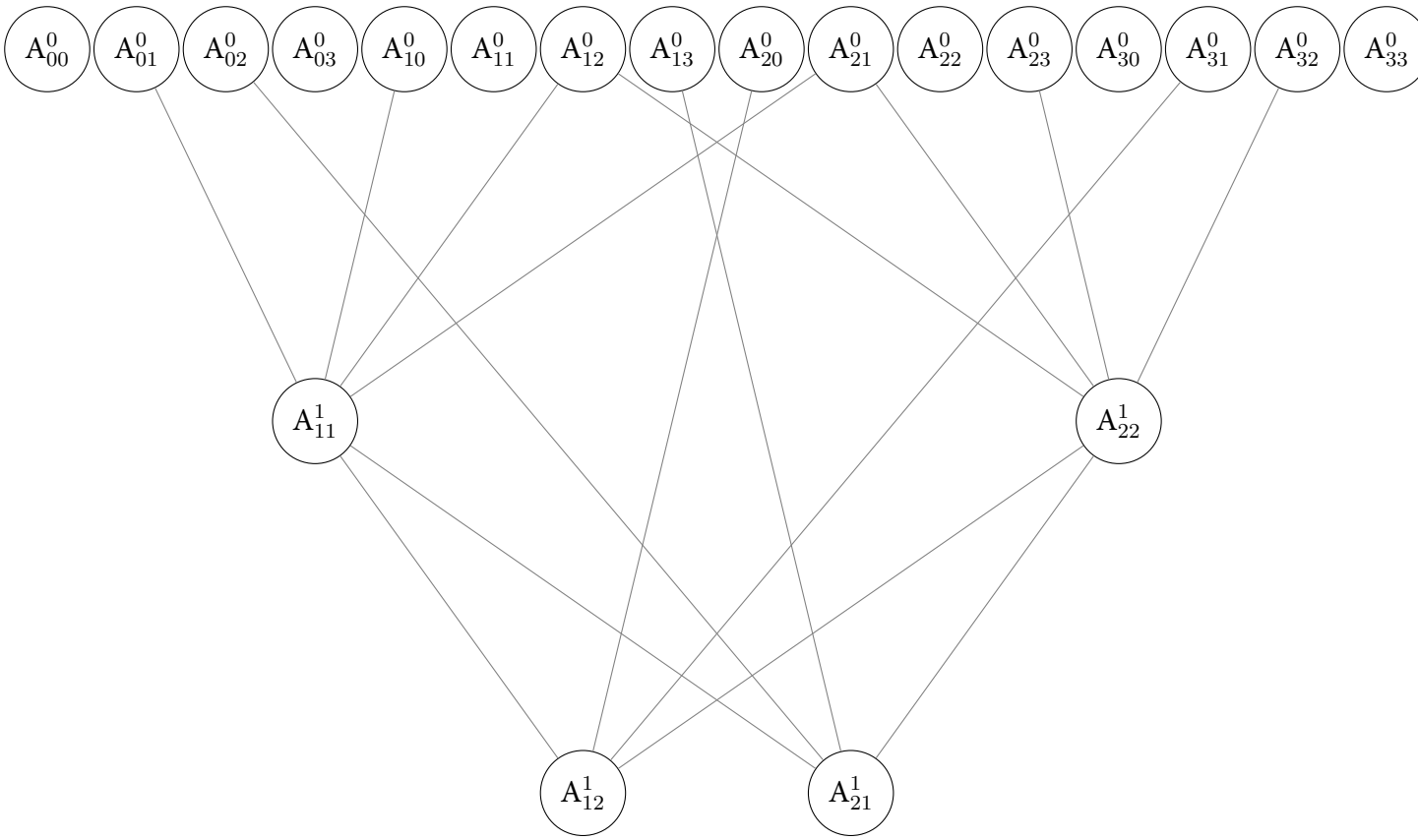


Figure 3: Red-Black SOR task graph

4.6 Μετρήσεις επίδοσης

4.6.1 Μετρήσεις με έλεγχο σύγκλισης

Grid Size	Algorithm	Iterations	tcomp (s)	tconv (s)	tcomm (s)	ttotal (s)
1024	Jacobi	798201	39.790418	0.416205	182.141695	222.348318
	Gauss-Seidel	3201	0.607223	0.006321	1.493841	2.107385
	Red-Black	2501	0.299372	0.001866	1.12454	1.425778

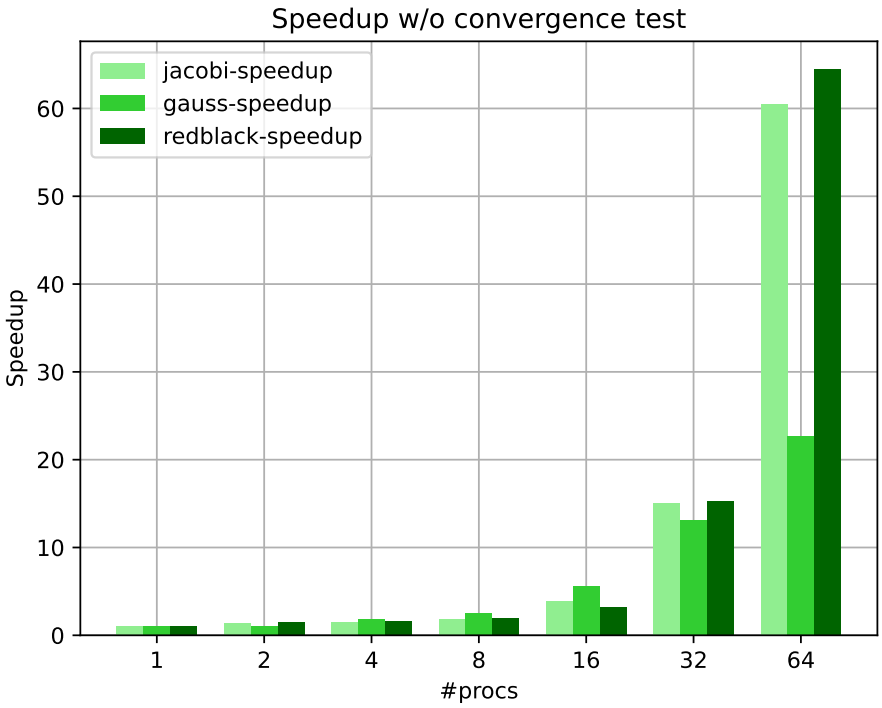
- Παρατηρούμε πως η μέθοδος Jacobi έχει τον πιο αργό ρυθμό σύγκλισης σε σχέση με τις υπόλοιπες μεθόδους. Συγκεκριμένα χρειάζεται περίπου 800.000 επανάληψεις για να συγκλίνει, δηλαδή χρειάζεται 200 φορές περισσότερες επαναλήψεις από τις άλλες δύο μεθόδους.
- Συνολικά, τον ταχύτερο έλεγχο σύγκλισης τον πετυχαίνει η μέθοδος Red-Black SOR, με μικρή διαφορά ωστόσο από την μέθοδο Gauss-Seidel SOR, χρειάζεται περίπου 700 επαναλήψεις λιγότερες.

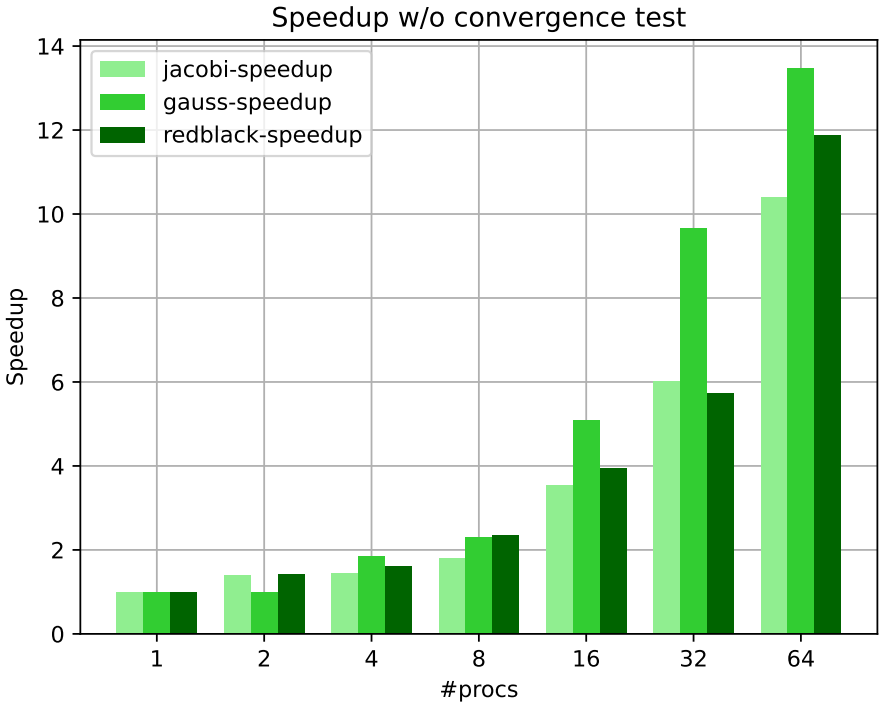
Για την επίλυση του συγκεκριμένου προβλήματος στο συγκεκριμένο σύστημα κατανεμημένης μνήμης θα επιλέγαμε την μέθοδο Red-Black SOR. Ωστόσο στην γενική περίπτωση θα έπρεπε να μελετήσουμε το δίκτυο διασύνδεσης. Σε περίπτωση υψηλής κίνησης στο δίκτυο, θα προτιμούσαμε την μέθοδο Gauss-Seidel.

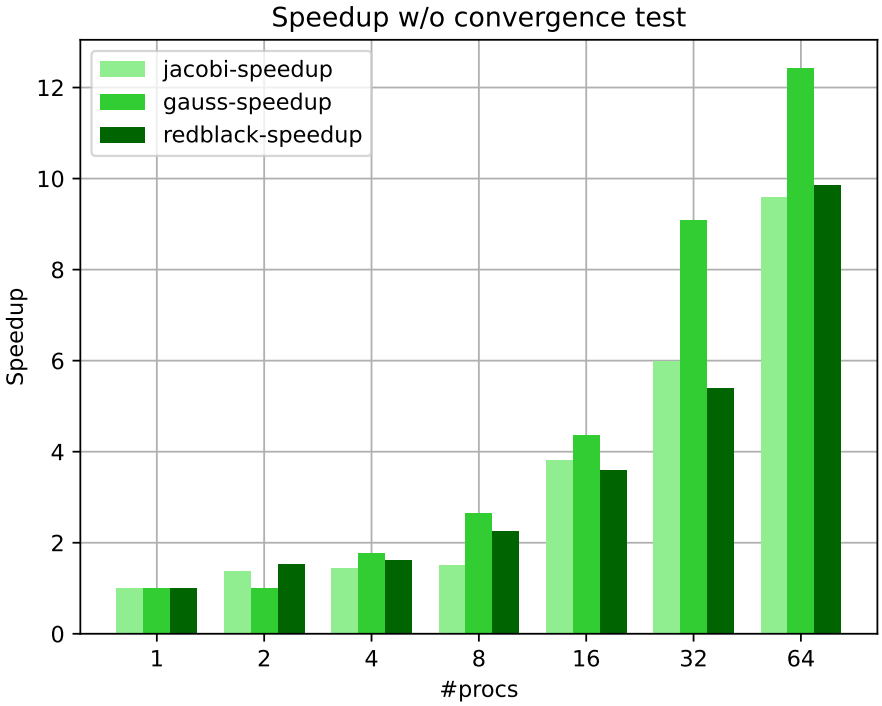
4.6.2 Μετρήσεις χωρίς έλεγχο σύγκλισης

Παρακάτω παρουσιάζονται οι μετρήσεις, αφού έχουμε απενεργοποιήσεις τον έλεγχο σύγκλισης για σταθερό αριθμό επαναλήψεων (T=256), για μεγέθη πίνακα 2048x2048, 4096x4096, 6144x6144, για 1, 2, 4, 8, 16, 32, 64 MPI διεργασίες.

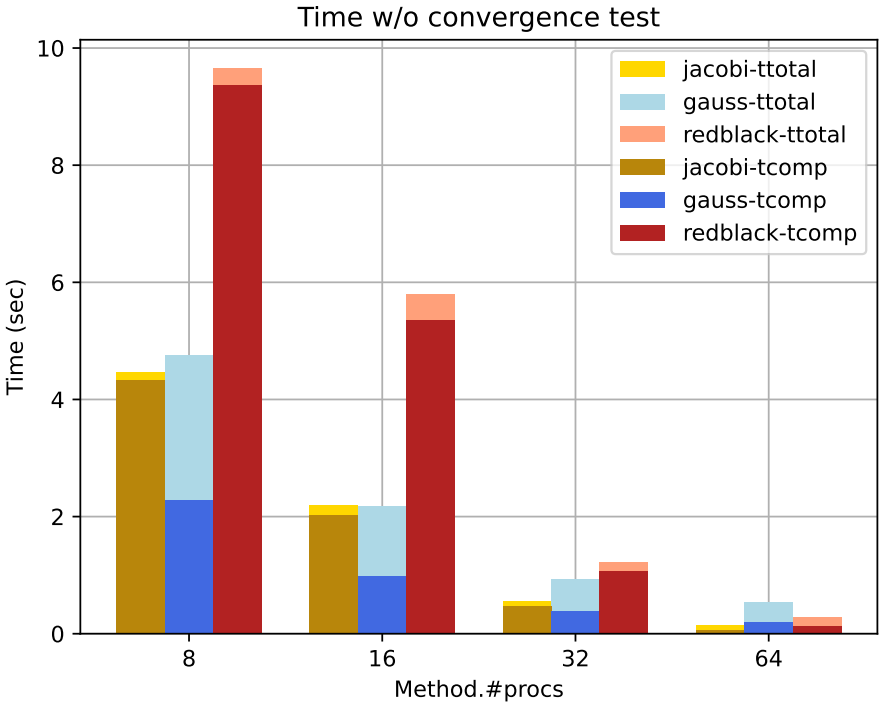
Speedup

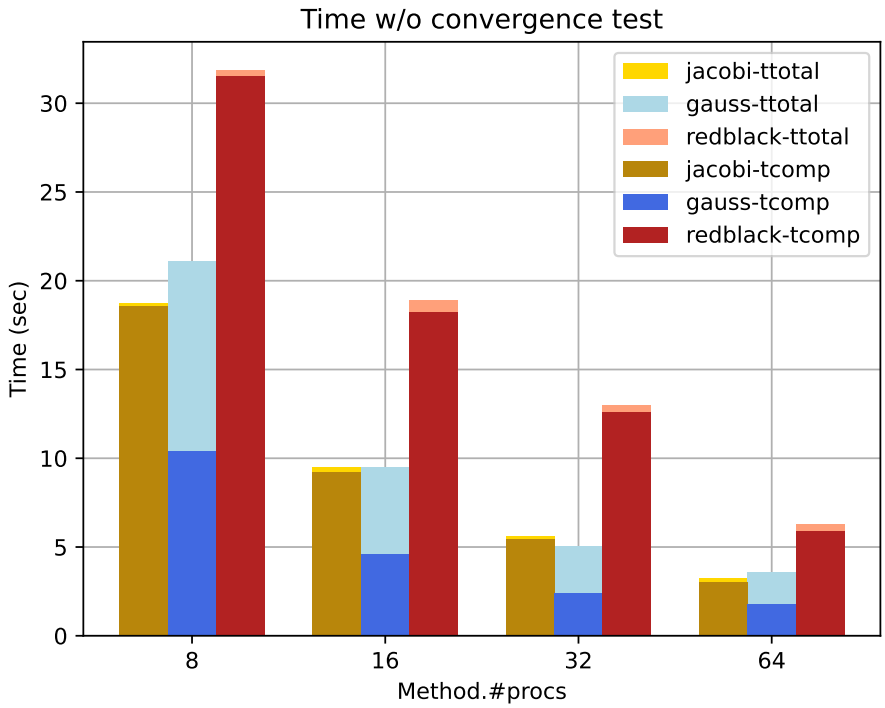


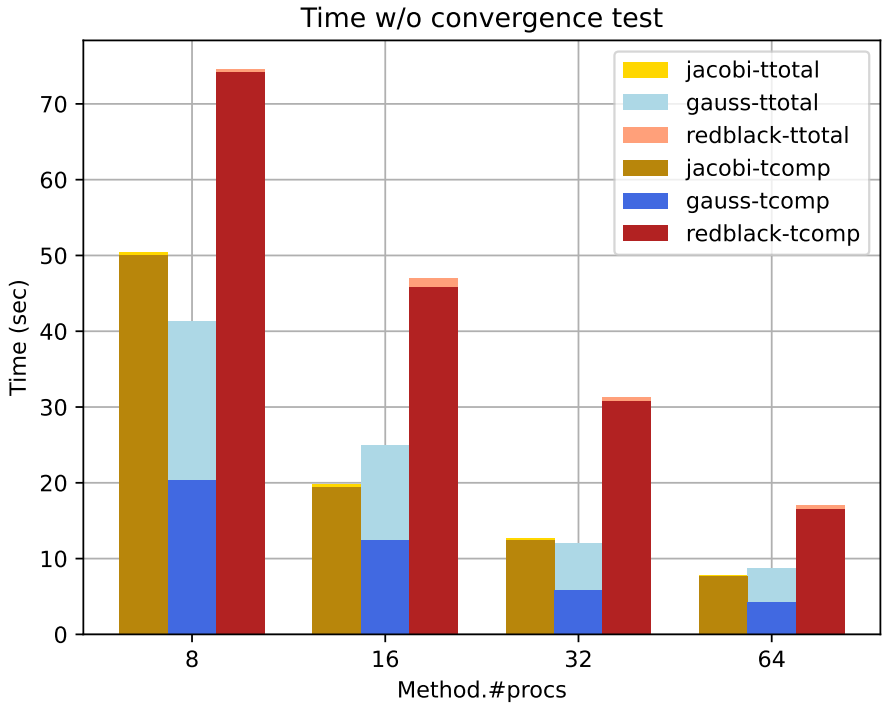



Speedup w/o convergence test
jacobi-speedup
gauss-speedup
redblack-speedup
Speedup
0
2
4
6
8
10
12
1
2
4
8
16
32
64
#procs

Time


Time w/o convergence test
jacobi-ttotal
gauss-ttotal
redblack-ttotal
jacobi-tcomp
gauss-tcomp
redblack-tcomp
Time (sec)
0
2
4
6
8
10
8
16
32
64
Method.#procs





4.7 Συμπεράσματα

- Από τα παραπάνω γίνεται ξεκάθαρη η αξία και η επιρροή που έχει ένας αποδοτικός αλγόριθμος στην ταχύτητα σύγκλισης ενός προγράμματος, αλλά και το πως η παραλληλοποίηση, όσο σημαντική και αν είναι, μπορεί να επιταχύνει έναν αλγόριθμο μέχρι το σημείο εκείνο που της ”επιτρέπει” ο αλγόριθμος. Με λίγα λόγια, όσο υψηλή και αν είναι η δυνατότητα παραλληλοποίησης ενός προγράμματος, αν ο αλγόριθμος δεν είναι τόσο καλός, δηλαδή δεν συγκλίνει γρήγορα, η επίδοση του προγράμματος δεν είναι απαραίτητο ότι θα βελτιωθεί.

Εδω χαρακτηριστικά φαίνεται πως αν και η μέθοδος Jacobi παρουσιάζει σημαντική δυνατότητα παραλληλοποίησης σε αντίθεση με την μέθοδο Gauss-Seidel υστερεί κατά πολύ χρονικά λόγω αλγορίθμου, επειδή δεν αξιοποιεί καθόλου τις υπολογισμένες τιμές του χρονικού βήματος t+1 και έτσι συγκλίνει πιο αργά.