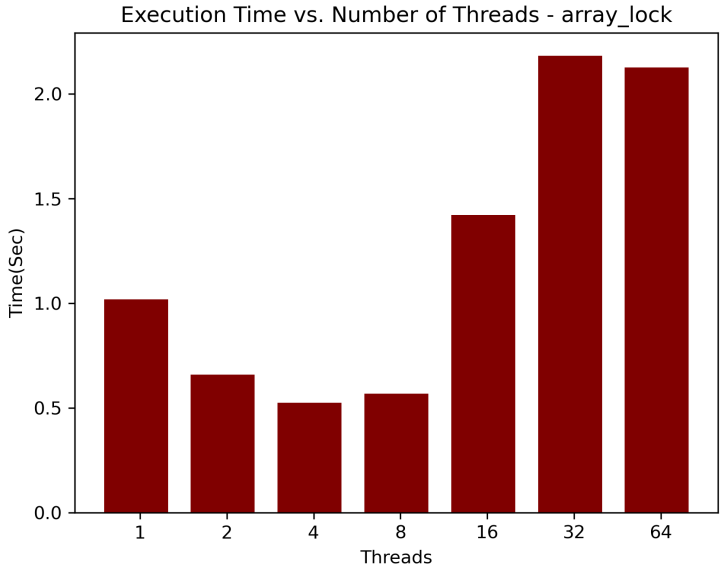
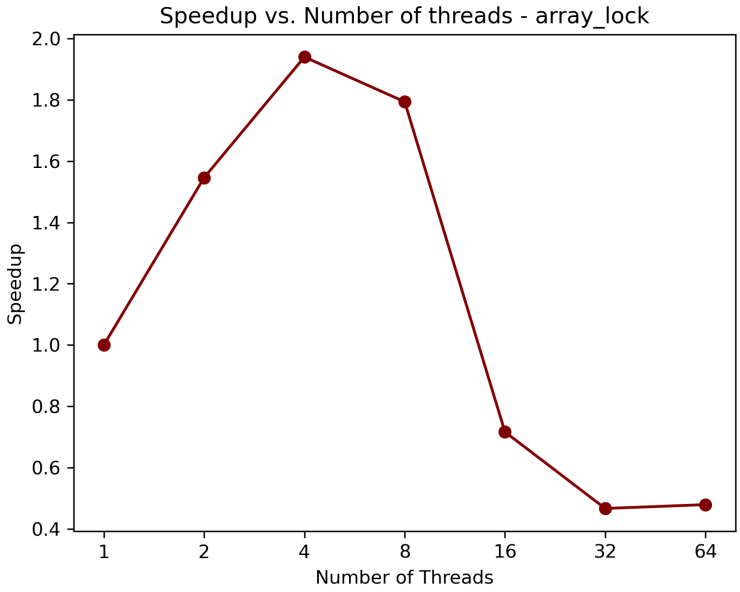
3.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means - Locks

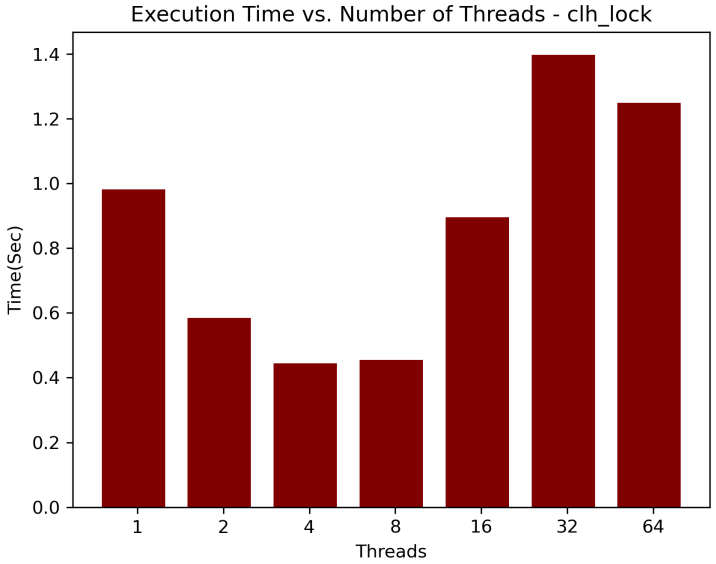
- Array Lock:

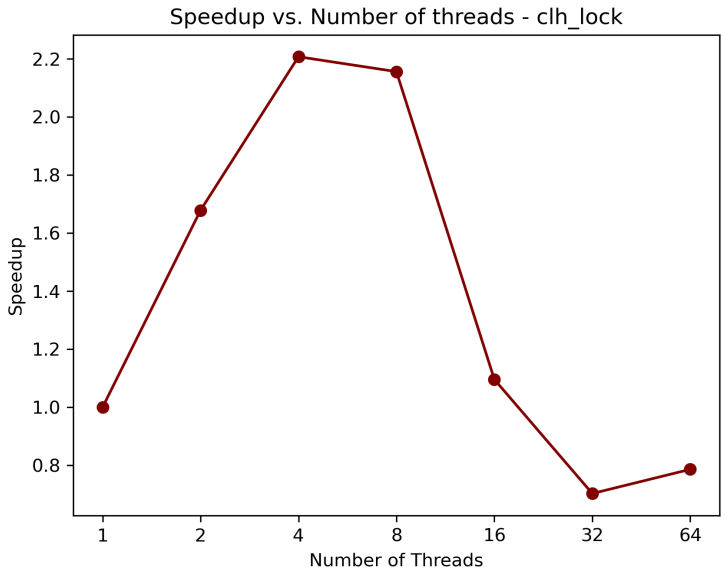




Με το array lock δημιουργείται μια λίστα με μέγεθος ίσο με τον αριθμό των threads. Στο τέλος της λίστας υπάρχει ένας counter, ο οποίος αρχικά έχει τιμή μηδέν και ορίζει το τέλος της λίστας. Επιπλέον, το στοιχείο lock[0] αρχικά έχει την τιμή true, ώστε να επιτρέπεται στο πρώτο thread που θα προσπαθήσει να εκτελέσει να τα καταφέρει. Κάθε φορά που ένα thread θέλει να εκτελέσει, λαμβάνει την τιμή του μετρητή και τον αυξάνει κατά ένα. Αναμένει μέχρι το αντίστοιχο στοιχείο του στον πίνακα lock (έστω i) να γίνει lock[i] = true. Όταν αυτό συμβεί, εκτελεί το κρίσιμο τμήμα του κώδικα. Μετά την ολοκλήρωση του κρίσιμου τμήματος, θέτει το lock[i] = false και το lock[(i+1)%size] = true, επιτρέποντας έτσι στην επόμενη διαδικασία να προχωρήσει. Έτσι μειώνεται το contention στο bus γιατι κάθε thread διαβάζει μία ξεχωριστή διεύθυνση μνήμης. Αναμενόμενο λοιπόν είναι και το αποτέλεσμα του χρόνου εκτέλεσης που παρατηρούμε. Απο τα διαγράμματα παρατηρούμε πως το array lock δίνει αθροιστικά λιγότερο χρόνο. Μετά τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ απο το contention για το lock).

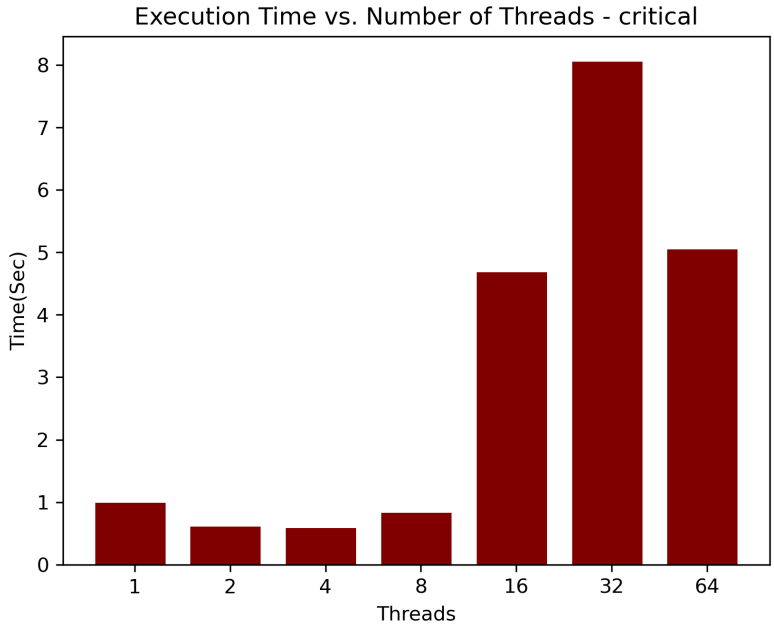
- CLH Lock:

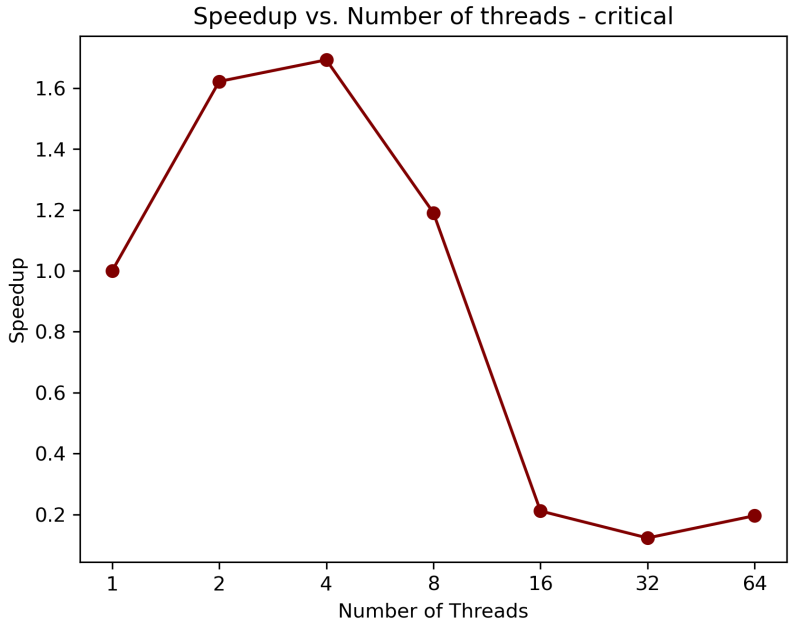




Το clh lock υλοποιεί μια συνδεδεμένη λίστα στην οποία κάθε νήμα που περιμένει να μπεί στο κρίσιμο τμήμα δημιουργεί ένα νέο κόμβο στο τέλος της λίστας και θέτει το flag του σε false (δείχνοντας πως θέλει να πάρει το lock). Έτσι κάθε νήμα κάνει spin το flag του προγόνου του. Το clh lock είναι αποδοτικότερο κλείδωμα ιδιαίτερα για critical sections που δε διαρκούν μεγάλο χρονικό διάστημα διότι είναι τύπου spin lock και σε αυτές τις περιπτώσεις είναι αποδοτικότερο να κάνει spin σε μία μεταβλητή απο το να βάζουμε σε sleep τα νήματα και να ξανα ξυπνάμε. Το clh lock μας δίνει τους καλύτερους χρόνους εκτέλεσης, λόγω του spin σε δικιά του διεύθυνση για κάθε νήμα και της μείωσης των Cache Invalidations καθώς γίνεται invalidate μόνο η cache line του επόμενου στη σειράς νήματος πετυχαίνουμε τους καλύτερους χρόνους.

- Critical:



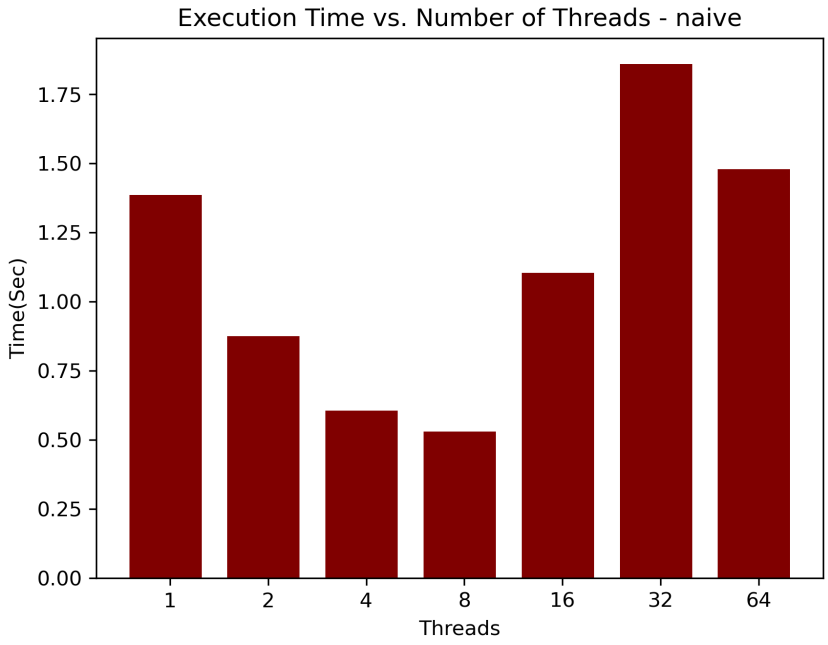


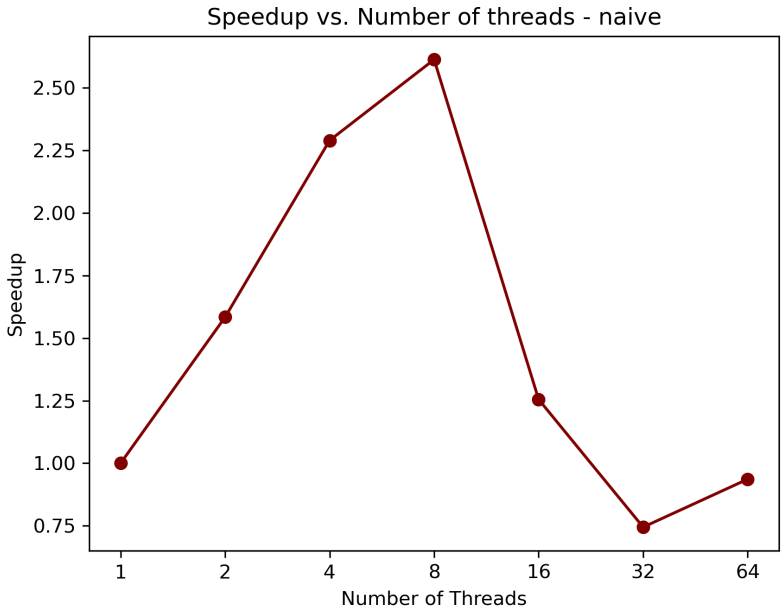
Στην critical υλοποίηση ο συγχρονισμός γίνεται απο το API της OpenMP, συγκεκριμένα παρεμβάλει πριν και μετά από το critical section την εξής “εντολή”:

```
#pragma omp critical
```

```
    #pragma omp critical
    {
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++){
            newClusters[index*numCoords + j] += objects[i*numCoords + j];
        }
    }
}
```

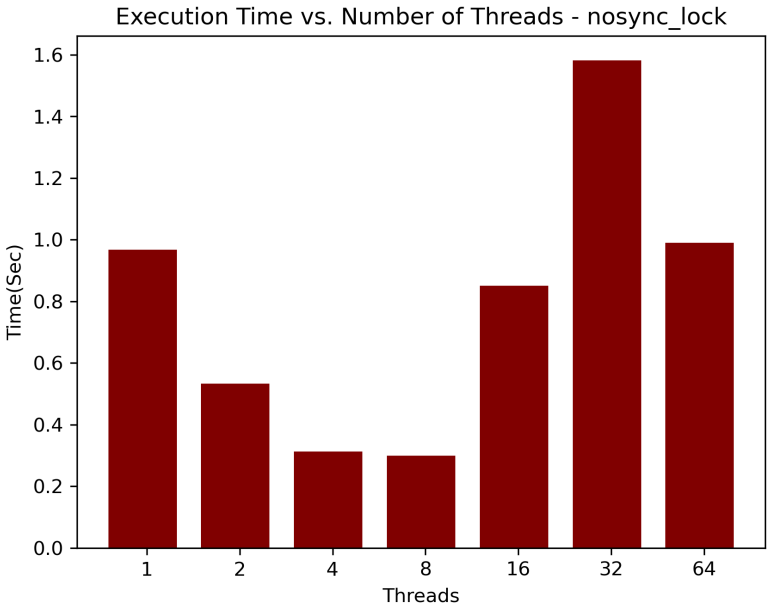
- Naive:

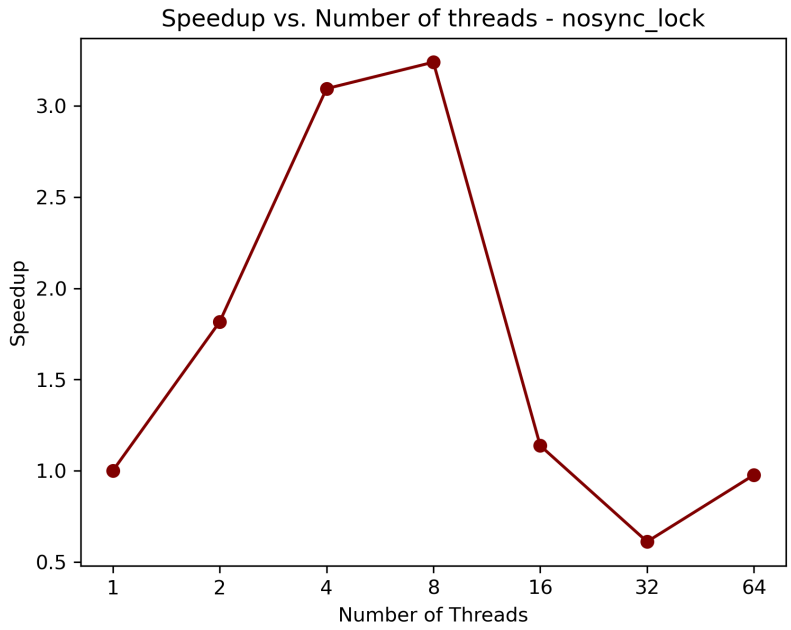




Στη Naive υλοποίηση χρησιμοποιούμε #pragma omp atomic, πριν τη πρόσβαση στο newClusters και newClusterSize ώστε να εξασφαλίσουμε atomic access σε αυτα.

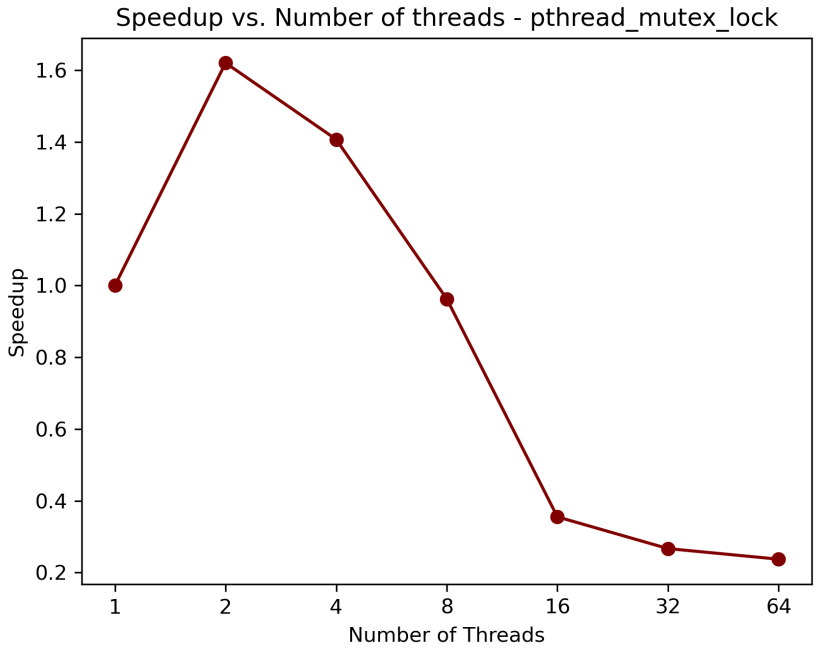
- NoSync Lock:





Το nosync lock όπως είναι αναμενόμενο έχει το λιγότερο συνολικό χρόνο για κάθε configuration νημάτων, δε φροντίζει για ατομικό access στα newClusters και newClusterSize και επομένως το clustering που κάνει είναι λάθος.

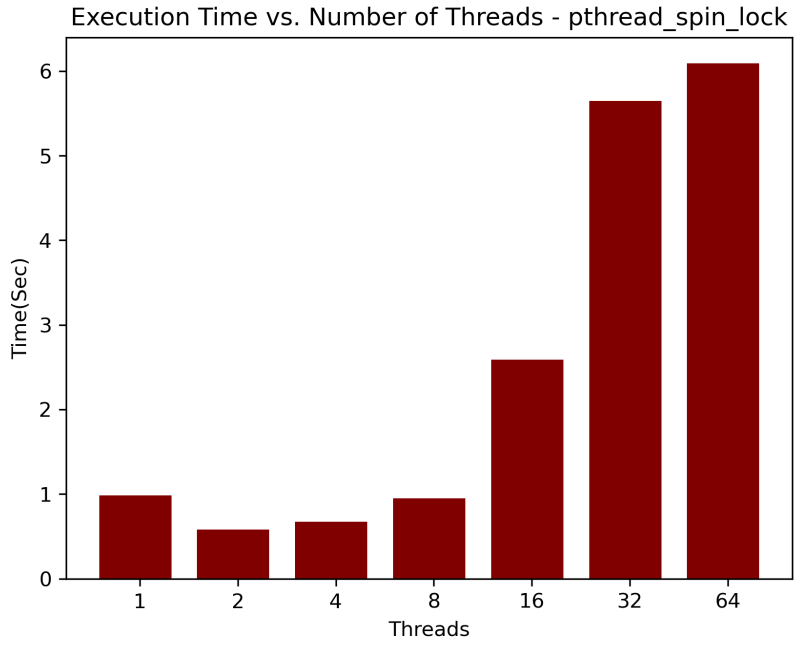
- Pthread Mutex Lock:

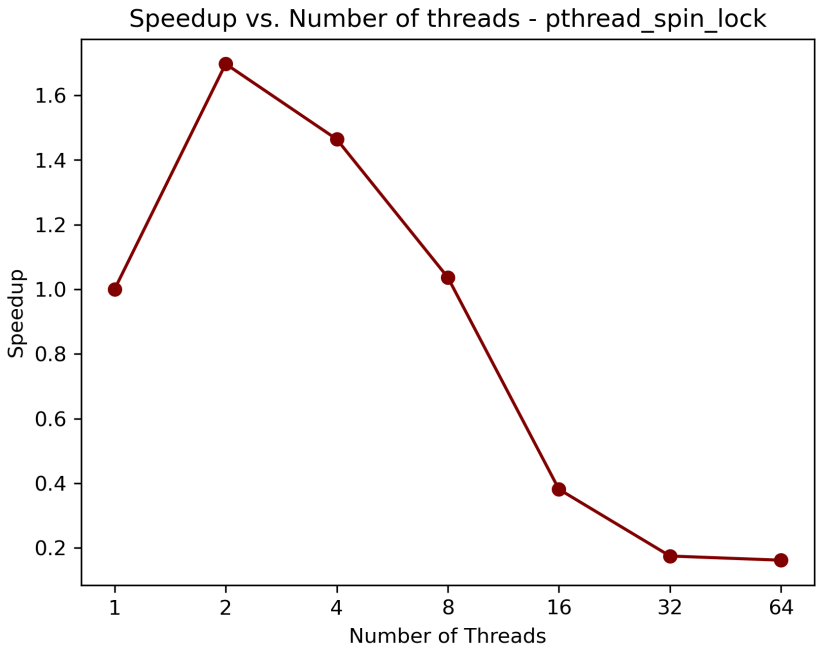


Στην υλοποίηση αυτή κάθε νήμα περιμένει να πάρει το lock κάνοντας κλήση στη συνάρτηση acquire_lock(), αν το mutex δεν είναι ελεύθερο το νήμα περιμένει (sleep) μέχρι να είναι ελεύθερο. Ξανα όπως και στις άλλες υλοποιήσεις μετα τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του

πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ απο το contention για το lock).

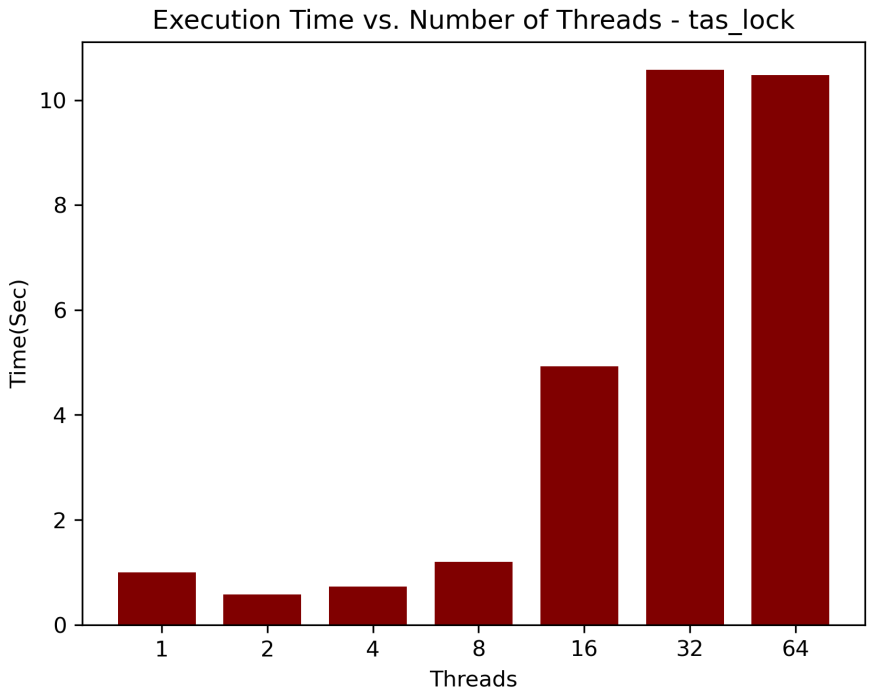
- Pthread Spin Lock:

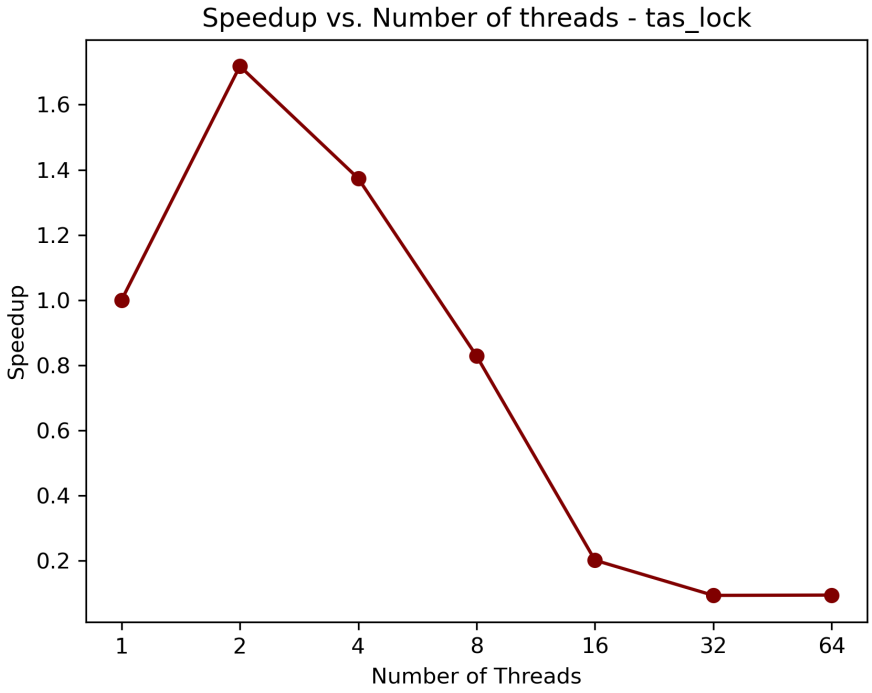




Σε αυτή την υλοποίηση το thread προσπαθεί να πάρει το lock κάνοντας συνέχεια spin (τσεκάρει συνεχώς τη διαθεσιμότητα του lock) μέχρι να καταφέρει να το πάρει. Δεν εξασφαλίζεται κάποια σειριοποίηση αν και στη προκειμένη δε μας ενδιαφέρει. Ξανα όπως και στις άλλες υλοποιήσεις μετα τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ απο το contention για το lock).

- TAS Lock





Η υλοποίηση αυτή βασίζεται στη λογική Test and Set, το κάθε νήμα με χρήση ατομικής εντολής __sync_lock_test_and_set γράφει locked στο state του lock και επιστρέφει πίσω την παλια τιμή του state, αν δεν ηταν unlocked συνεχίζει το while loop.

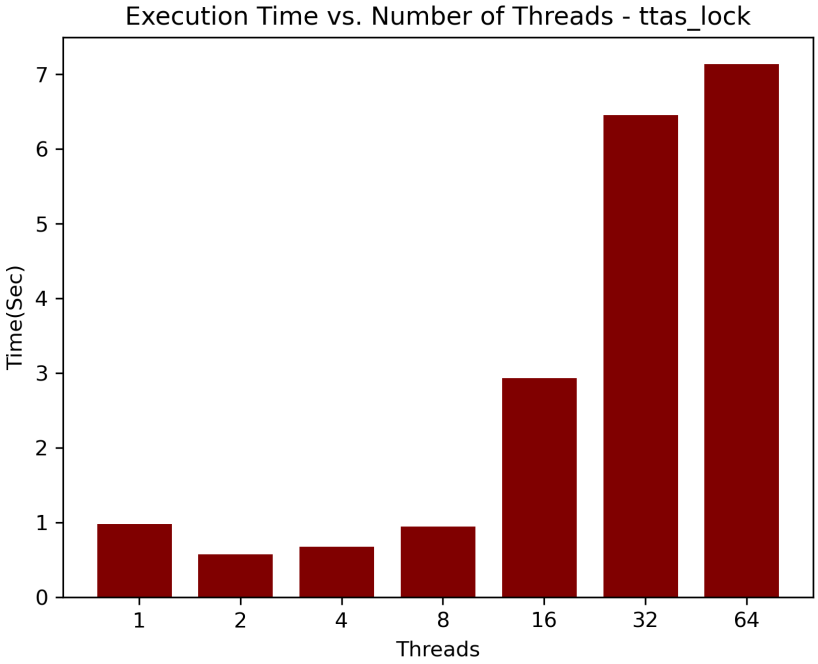
The `while` loop uses the `__sync_lock_test_and_set` function to attempt to acquire the lock. This function is a built-in function provided by GCC for atomic operations. It takes two arguments: a pointer to a memory location and a new value. The function atomically sets the value at the given memory location to the new value and returns the old value.

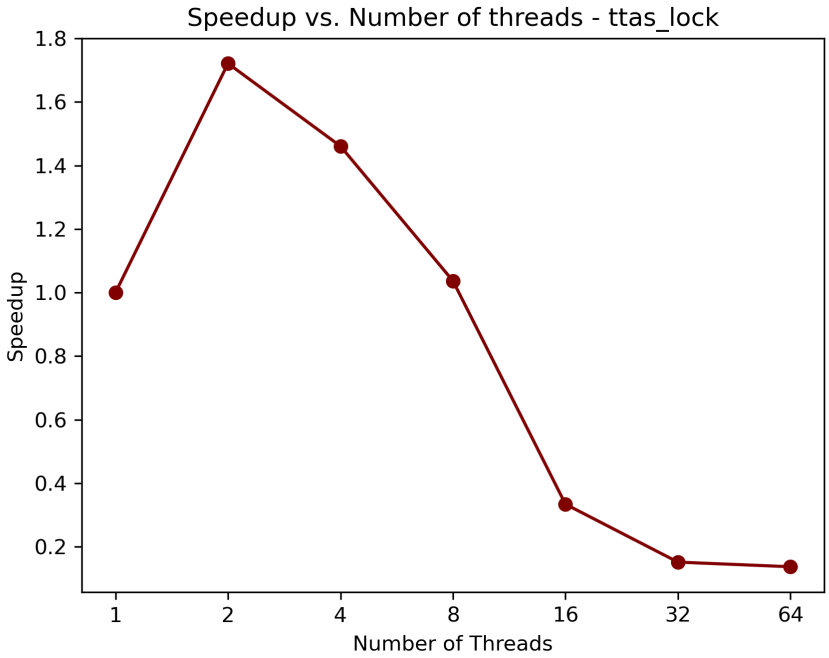
```
void lock_acquire(lock_t *lock)
{
    lock_t *l = lock;

    while (__sync_lock_test_and_set(&l->state, LOCKED) == LOCKED)
        /* do nothing */ ;
}
```

Ξανα όπως και στις άλλες υλοποιήσεις μετα τα 8 threads έχουμε χειρότερη απόδοση καθώς είναι πιο εμφανές το contention για το lock του πίνακα των clusters (οι υπολογισμοί για το κοντινότερο cluster γίνονται μεν γρηγορότερα αλλα αντισταθμίζεται κατα πολύ απο το contention για το lock).

- TTAS Lock:


Execution Time vs. Number of Threads - ttas_lock
Time(Sec)
0
1
2
3
4
5
6
7
1
2
4
8
16
32
64
Threads


Speedup vs. Number of threads - ttas_lock
Speedup
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8
1
2
4
8
16
32
64
Number of Threads

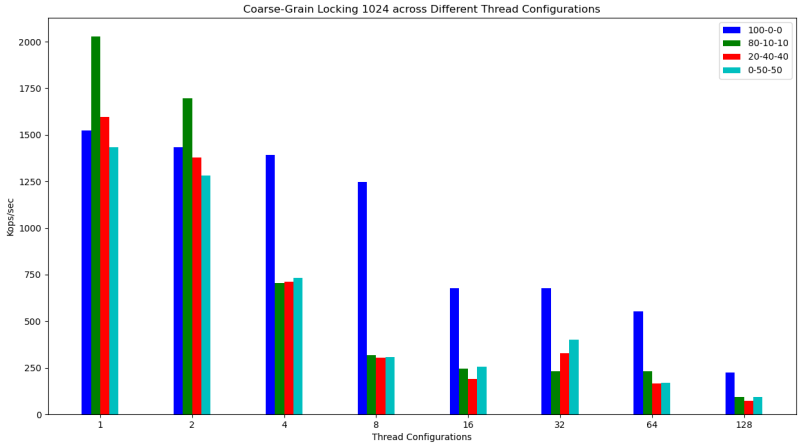
Σε αντίθεση με το tas_lock, το ttas_lock προσπαθεί να αποκτήσει το κλείδωμα μόνο όταν διαπιστώσει ότι το κλείδωμα είναι ελεύθερο. Αυτό σημαίνει ότι η ατομική εντολή που γράφει στην κοινόχρηστη μνήμη εκτελείται λιγότερες φορές, άρα έχουμε και λιγότερη κίνηση στο bus. Επομένως, αναμένουμε να δούμε βελτίωση της απόδοσης σε σχέση με πριν. Κάτι που παρατηρούμε καθώς σε όλα τα threads βλέπουμε μειωμένο χρόνο εκτέλεσης.

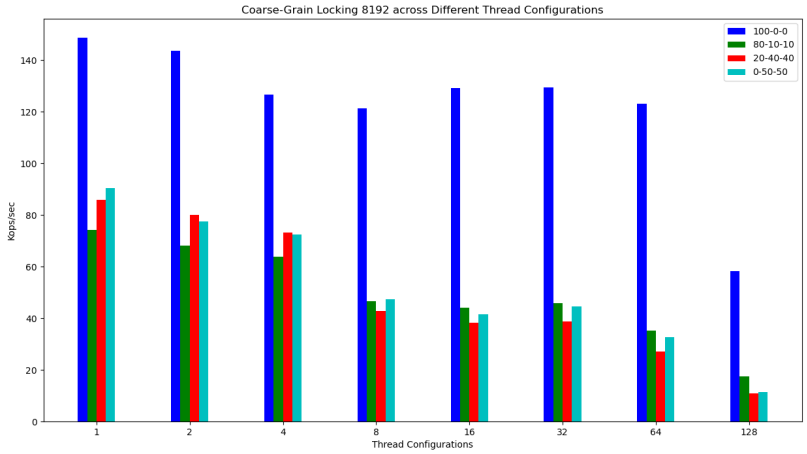
```
void lock_acquire(lock_t *lock)
{
    lock_t *l = lock;

    do {
        while (l->state == LOCKED)
            /* do nothing */ ;
    } while (__sync_lock_test_and_set(&l->state, LOCKED) == LOCKED);
}
```

2.3 Ταυτόχρονες Δομές Δεδομένων

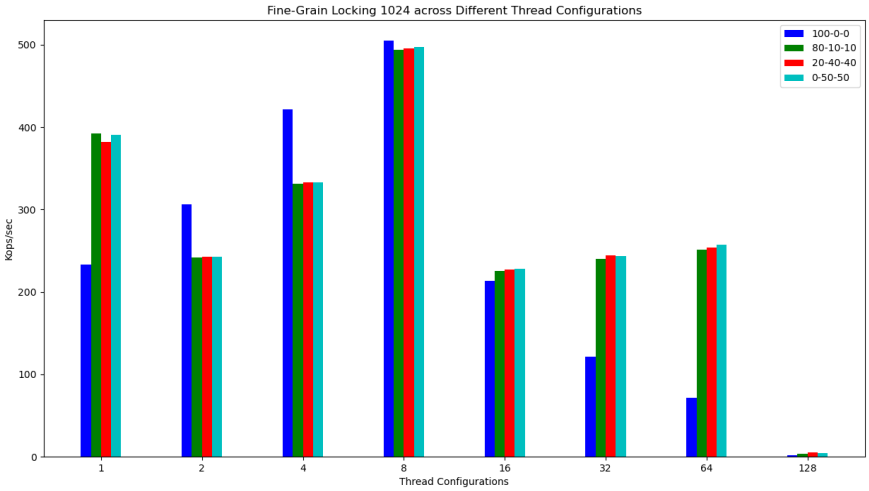
- **Coarse-Grain Locking**

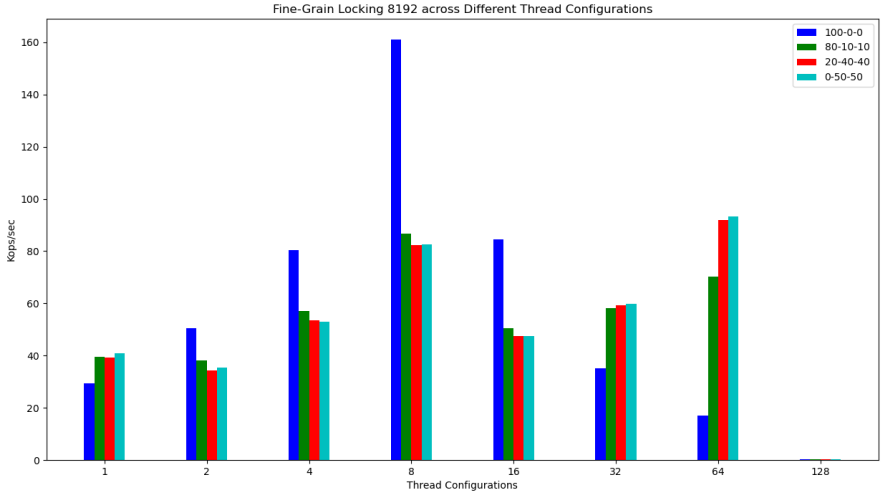




Στο coarse-grain locking για καθε operation στη λίστα κλειδώνεται όλη η λίστα. Αυτό οδηγεί ,όπως είναι αναμενόμενο, στο να μειώνονται τα operations ανα second όσο περισσότερα threads εμπλέκουμε, τόσο στο μέγεθος 1024 όσο και στο 8192.

- **Fine-Grain Locking**

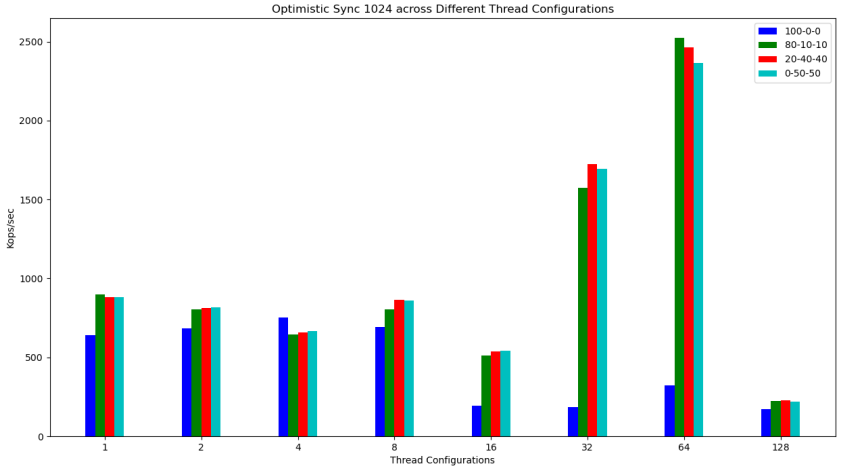


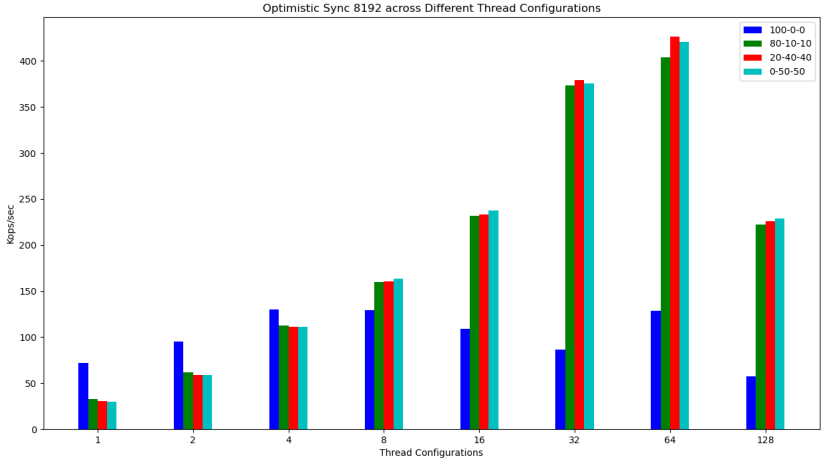


Στο fine grain locking δε κλειδώνει όλη τη λίστα καθε thread που θέλει να πραγματοποιήσει ένα operation, αλλα κλειδώνει σε επίπεδο node. Ένα νήμα πρέπει να πάρει το lock του πριν μπορέσει να το διαβάσει ή να εκτελέσει μια πράξη όπως εισαγωγή ή διαγραφή. Όπως είναι αναμενόμενο η απόδοση είναι ανοδική με αύξηση των threads. Ωστόσο χειροτερεύει όταν έχουμε μεγάλο ποσοστό αναζητήσεων, λόγω του hand-over-hand locking τα thread που θα

θέλουν να διαβάσουν πέρα από το locked node απο ένα άλλο thread θα δημιουργούν “κίνηση” πίσω από αυτό, με αποτέλεσμα να σπαταλούν ώρα σε αναμονή παρά στη πράξη τους.

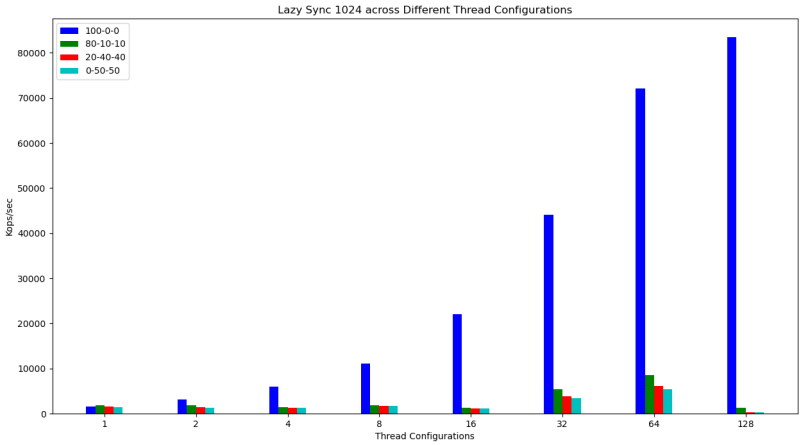
- **Optimistic Synchronization**

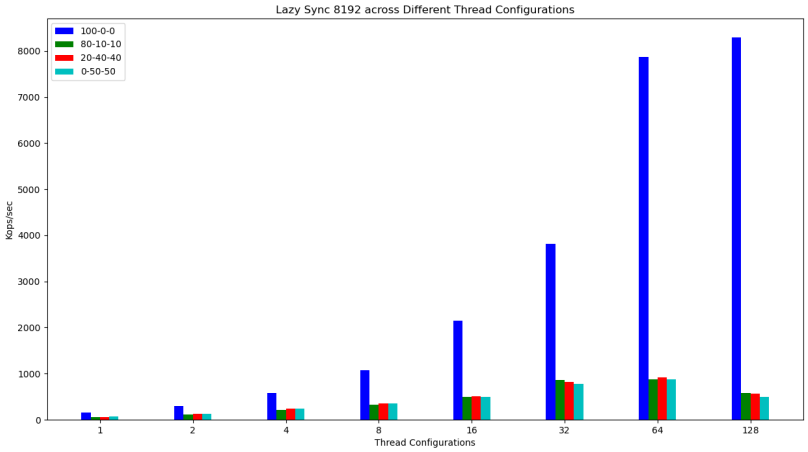




Το optimistic synchronization βασίζεται στην ιδέα οτι η σύγκρουση μεταξύ νημάτων δεν είναι συχνή, με αυτό το σκεπτικό κάθε νήμα αρχικά δουλεύει χωρίς locks και εκτελεί τα operations και μόνο στο τέλος κανει validate οτι δεν υπήρξε σύγκρουση με operation άλλου thread. Η διαφορά στην απόδοση συγκριτικά με άλλα locking schemas είναι πιο εμφανής στα configurations που περιλαμβάνουν εισαγωγές/διαγραφές. Ένα σημαντικό μειονέκτημα είναι ότι είναι πιθανό το starvation καθώς σε συστήματα που έχουν μεγάλη κίνηση είναι πιθανό τα νήματα να αποτυγχάνουν να κάνουν validate και να ξαναπροσπαθούν ξανα και ξανα.

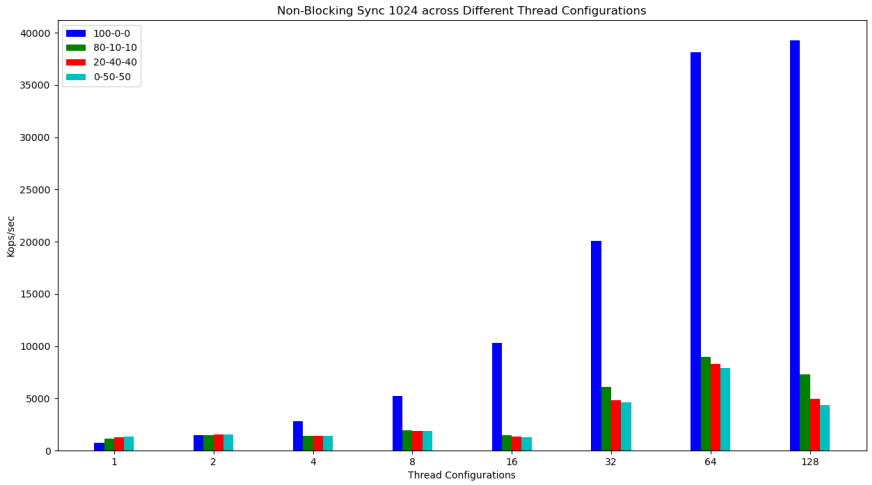
- **Lazy Synchronization**

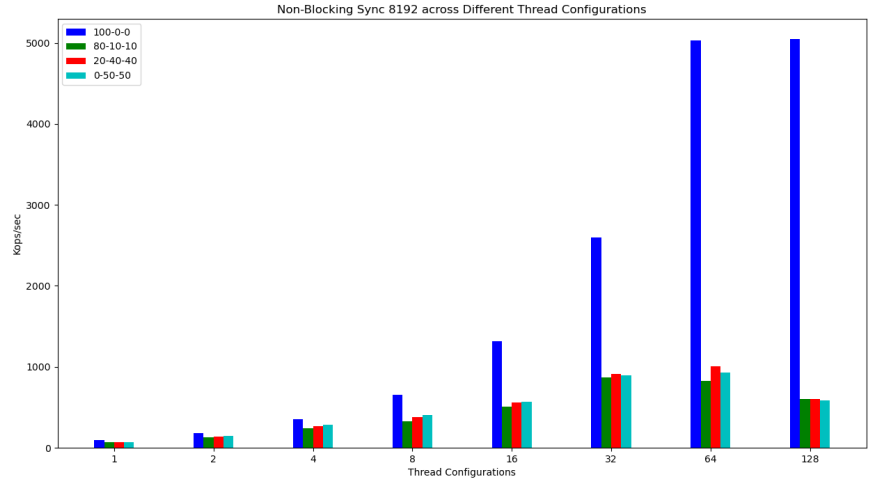




Στο Lazy synchronization προσθέτουμε στη δομή μία boolean μεταβλητή που δείχνει αν ο κόμβος βρίσκεται στη λίστα ή έχει διαγραφεί από αυτή. Η κύρια διαφορά σε σχέση με το optimistic synchronization είναι οτι η μέθοδος contains δε χρειάζεται να διατρέξει την λίστα απο την αρχή σε περίπτωση που το validate αποτύχει. Γενικά παρατηρούμε πως πετυχαίνουμε “θετικό” scaling καθώς ανεβαίνουμε τα thread configurations.

- **Non-Blocking Synchronization**





Το non-blocking synchronization αποτελεί τη λύση στην χρήση lock της lazy synchronization στα insert και delete tasks. Το overhead των locks απαλείφεται και αντικαθιστάται με reruns στην περίπτωση που δεν ισχύει η τωρινή κατάσταση κάποιου εκ των εξεταζόμενων κόμβων. Το μεγαλύτερο μειονέκτημα αυτής της στρατηγικής είναι η επιβάρυνση της find. Αυτή, κατά τη διάσχιση της λίστας, κάνει έναν έλεγχο σε κάθε κόμβο, ώστε να δει αν είναι marked. Αυτό επιτυγχάνεται συγχωνεύοντας την boolean που αναφέραμε στην Lazy υλοποίηση με το πεδίο next του κόμβου, το οποίο δείχνει στο επόμενο στοιχείο της λίστας. Έτσι, η boolean αυτή λειτουργεί τώρα ως flag που επιτρέπει την ενημέρωση του πεδίου next του κόμβου, εφόσον αυτός υπάρχει λογικά. Έχουμε πάλι “θετικο” scaling μεχρι και τα 64 threads.