Συστήματα Παράλληλης Επεξεργασίας

Άσκηση 2

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

Ομάδα	parlab04
Θεόδωρος Αράπης	el18028
Εμμανουήλ Βλάσσης	el18086
Παναγιώτης Παπαδέας	el18039

Αμοιβαίος Αποκλεισμός - Κλειδώματα

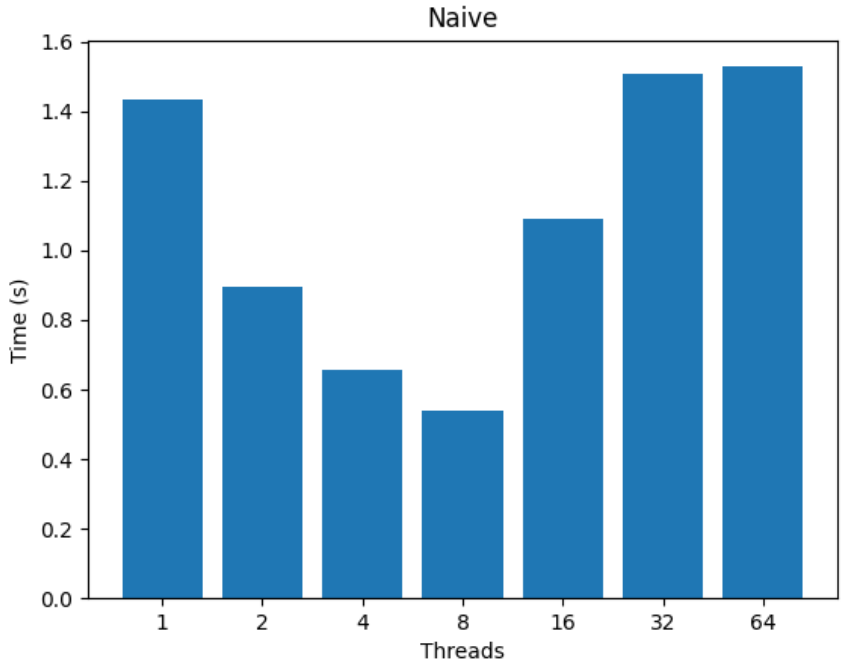
Ζητούμενα

Ζητείται η πραγματοποίηση μετρήσεων με το configuration {Size, Coords, Clusters, Loops} = {16, 16, 16, 10} για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχάνημα sandman για την υλοποίηση του αλγορίθμου K-means με χρήση διαφόρων ειδών κλειδωμάτων, καθώς και η σύγκριση των αποτελεσμάτων.

Μετρήσεις

naive

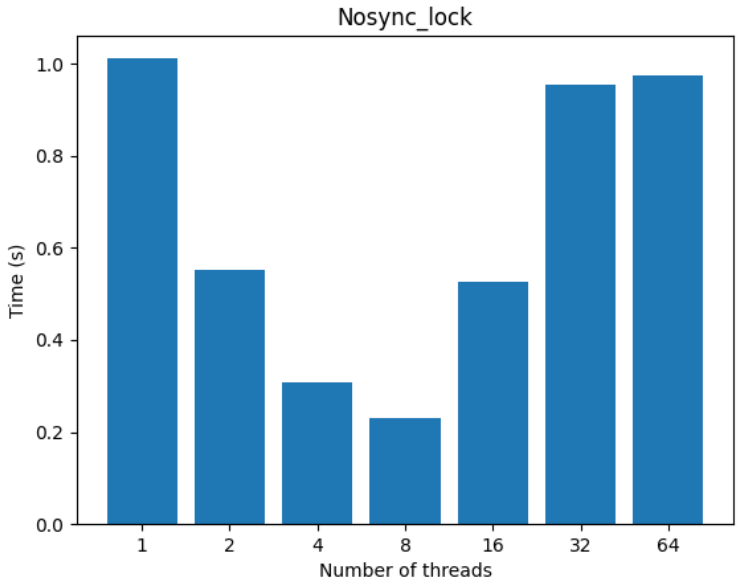
Αρχικά παραθέτουμε τις μετρήσεις για την shared-clusters (naive) υλοποίηση του αλγορίθμου K-means, με την μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY απενεργοποιημένη και ενεργοποιημένη, οι οποίες θα χρησιμοποιηθούν ως μέτρο σύγκρισης των χρόνων που επιτυγχάνουμε με τα διάφορα κλειδώματα:



Εδώ είναι φανερή η αύξηση της ταχύτητας μέχρι τα 8 threads και η μείωσή της για 16 threads και πάνω. Αυτό οφείλεται σε καθυστερήσεις στην δημιουργία και επικοινωνία των threads (ατομική εκτέλεση με χρήση του #pragma omp atomic), οι οποίες επισκιάζουν χρονικά τα οφέλη της παραλληλοποίησης.

nosync_lock

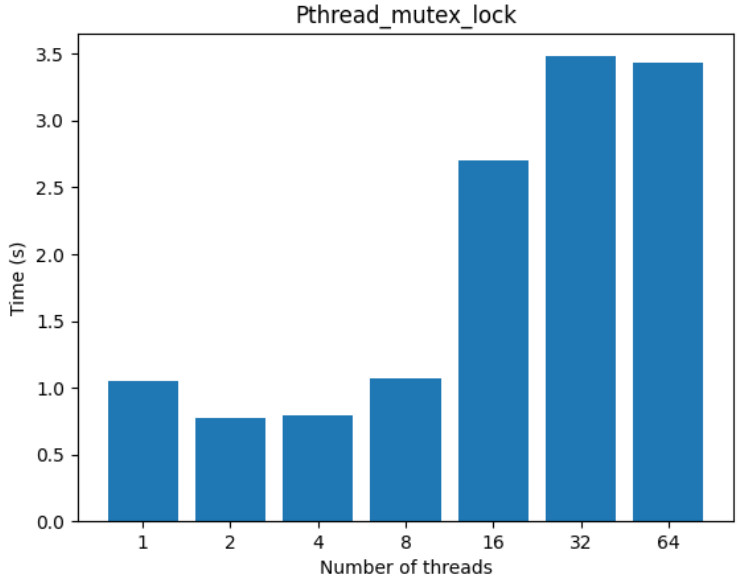
Η υλοποίηση αυτή δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα, επιτυγχάνει όμως τις καλύτερες ταχύτητες. Ωστόσο, θα χρησιμοποιηθεί ως άνω όριο (ιδανικό κλείδωμα) για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων. Οι χρόνοι που επιτυγχάνει είναι οι ακόλουθοι:



Παρατηρούμε ότι για κάθε πλήθος thread η ταχύτητα είναι μεγαλύτερη από την naive υλοποίηση, παρόλ' αυτά έχουμε το ίδιο scaling. Πιο συγκεκριμένα, μέχρι τα 8 threads υπάρχει αύξηση της ταχύτητας, στη συνέχεια, όμως, η ταχύτητα ελαττώνεται. Η διαφορά αυτή οφείλεται στο γεγονός ότι δεν έχουμε atomic operations, οπότε δεν υπάρχει καθυστέρηση στην εκτέλεση των threads, καθώς δεν περιμένουν.

pthread_mutex_lock

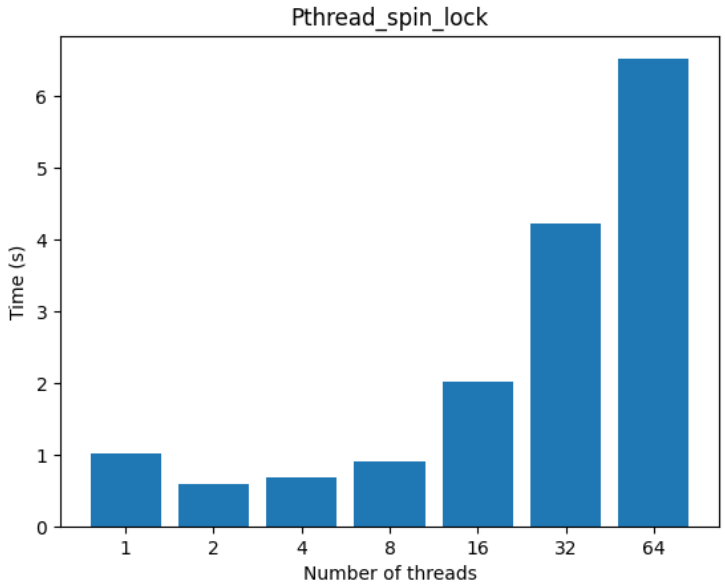
Το mutex λειτουργεί ως ένα flag το οποίο επιχειρούν να ενεργοποιήσουν τα threads και το επιτυγχάνουν μόνο όταν δεν το έχει ενεργοποιήσει άλλο thread προηγουμένως, αλλιώς περιμένουν (sleep) απελευθερώνοντας τους πόρους της CPU. Τα αποτελέσματα των μετρήσεων είναι:



Βλέπουμε ότι οι χρόνοι μέχρι τα 8 threads είναι αρκετά καλύτεροι από την naive υλοποίηση, επιτυγχάνοντας τώρα την μεγαλύτερη επίδοση στα 2 και 4 threads, πλησιάζοντας τους χρόνους της nosync υλοποίησης, ενώ για περισσότερα η ταχύτητα εκτέλεσης μειώνεται αρκετά και είναι χειρότερη από τη naive υλοποίηση. Αυτό πιθανόν συμβαίνει διότι με περισσότερα threads έχουμε περισσότερα context switching, τα οποία μπορεί να επιφέρουν μεγάλες καθυστερήσεις.

pthread_spin_lock

Το κλείδωμα με spin lock κάθε φορά ελέγχει αν το κρίσιμο τμήμα είναι κλειδωμένο. Σε περίπτωση που είναι, το thread μπαίνει σε ένα busy-wait loop, ελέγχοντας συνεχώς το κρίσιμο τμήμα μέχρι να γίνει Σημειώνουμε ότι το phthread_spin_lock εξασφαλίζει ότι δεν θα υπάρχουν ποτέ 2 threads στο κρίσιμο νήμα, αλλά δεν εξασφαλίζει ότι η εκτέλεση των εντολών θα γίνεται κάθε φορά με συγκεκριμένη σειρά, δηλαδή δεν εξασφαλίζεται η σειριοποίηση, αν και στην περίπτωσή μας δεν έχουμε σειριοποίηση. Ακολουθούν οι μετρήσεις:



Όπως και προηγουμένως, έχουμε σαφώς καλύτερους χρόνους από την naïve υλοποίηση μέχρι τα 8 threads, αλλά πάλι χειρότερους χρόνους από την nonsync υλοποίηση, ειδικά για μεγαλύτερα πλήθη επίδοση. Παρατηρούμε, ακόμη, ότι για πάνω από 8 threads ο χρόνος εκτέλεσης αυξάνεται εκθετικά και είναι χειρότερος σε σχέση με την naïve υλοποίηση. Συμπεραίνουμε ότι τα spinlocks γίνονται λιγότερο αποδοτικά όσο παραπάνω πυρήνες έχουμε, δηλαδή υπάρχει κακό scaling. Ο λόγος γι' αυτό είναι πιθανότατα ο πολύ χρόνος που ξοδεύεται στο busy waiting από την υπερβολική χρήση του διαδρόμου (bus).

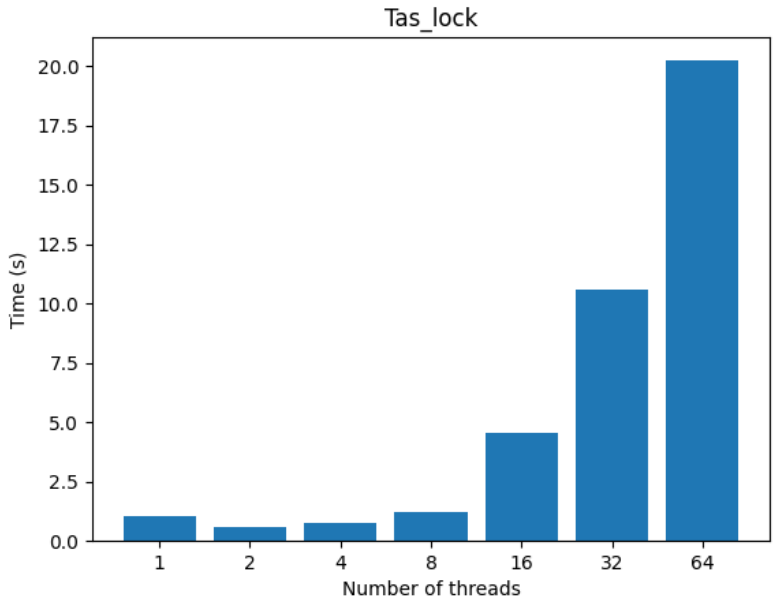
tas_lock

Γενικά, το Test-And-Set (TAS) είναι η πιο απλή μορφή spinlock. Το tas_lock χρησιμοποιεί το Test-And-Set με while loop προκειμένου να ελέγχει συνεχώς αν το lock είναι ελεύθερο:

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

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

Οι μετρήσεις που λάβαμε:



Παρατηρούμε ότι η δική μας υλοποίηση του tas lock είναι ταχύτερη σε σχέση με την naive υλοποίηση μέχρι τα 8 threads. Όπως και στο pthread_spin_lock, έχουμε εκθετική αύξηση του χρόνου μετά τα 8 thread, ενώ πετυχαίνουμε σχεδόν ίδιους χρόνους για threads 1, 2, 4 και 8. Γενικότερα, ισχύουν οι ίδιες παρατηρήσεις.

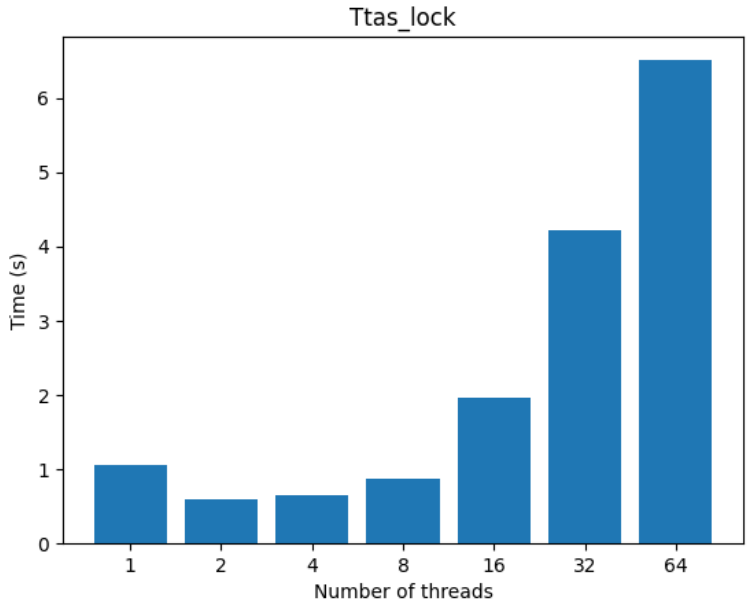
ttas_lock

Σε αντίθεση με το tas_lock, το ttas_lock κάνει διεκδίκηση του κλειδώματος μόνο όταν βλέπει ότι το lock είναι ξεκλείδωτο. Αυτό σημαίνει ότι εκτελείται λιγότερες φορές η ατομική εντολή που γράφει στη κοινή θέση μνήμης. Επομένως περιμένουμε να έχουμε αύξηση της επίδοσης σε σχέση με πριν.

```
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);
}
```

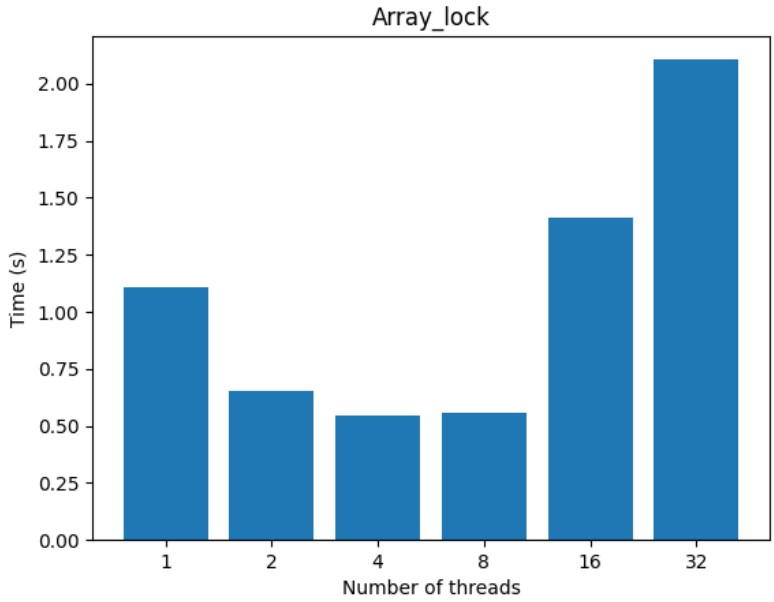
Τα αποτελέσματα των μετρήσεων:



Μέχρι τα 8 threads έχουμε χρόνους εκτέλεσης περίπου όπως στην περίπτωση του tas. Για περισσότερα threads έχουμε πάλι μεγάλη εκθετική αύξηση, απλά με μικρότερο λόγο (από 32 σε 64 πυρήνες αύξηση του χρόνου x1.5). Συνεπώς, ισχύουν οι ίδιες παρατηρήσεις με την περίπτωση του tas.

array_lock

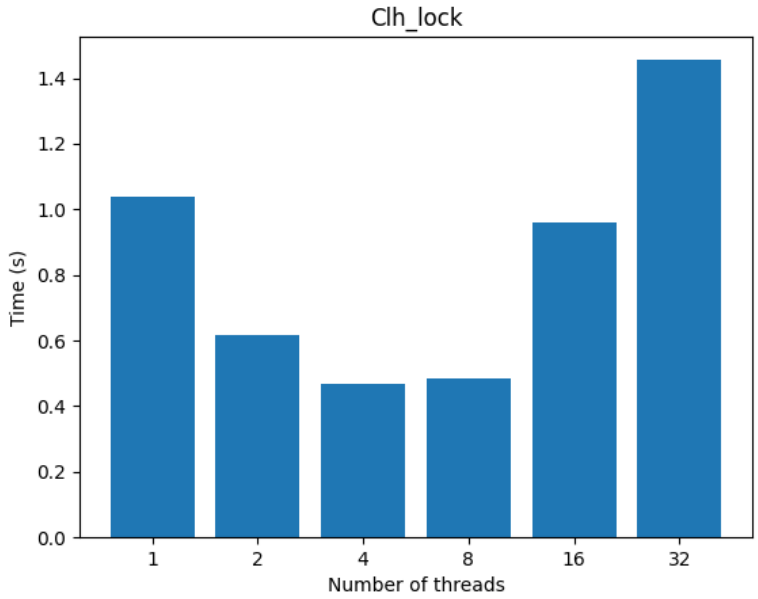
Σε αυτήν την υλοποίηση χρησιμοποιούμε έναν πίνακα μεγέθους όσο το πλήθος των threads, προκειμένου το καθένα από αυτά να κάνει spin σε μια μοναδική θέση μνήμης. Πιο συγκεκριμένα, όταν κάνει lock το πρώτο thread της ομάδας (καθώς το η τοπική του μεταβλητή flag είναι ενεργοποιημένη από την αρχικοποίηση), μπαίνει στην κρίσιμη περιοχή, την ίδια στιγμή που τα υπόλοιπα threads κάνουν spin στην μεταβλητή flag τους. Μόλις βγει από την κρίσιμη περιοχή, απενεργοποιεί το flag του και ενεργοποιεί το flag του επόμενου thread εν σειρά (FIFO) και επαναλαμβάνεται η ίδια διαδικασία. Οι χρόνοι που πέτυχε η υλοποίηση είναι:



Όπως είναι αναμενόμενο, παρατηρούμε συνολικά τις καλύτερες επιδόσεις συγκριτικά με όλα τα προηγούμενα locks. Ειδικότερα, βλέπουμε ότι οι επιδόσεις για πλήθη threads 1, 2, 4 και 8 είναι περίπου ίδιες με όλες τις υλοποιήσεις lock μέχρι τώρα, ωστόσο έχει βελτιωθεί σημαντικά ο χρόνος για 16 και 32. Στην περίπτωση των 64 threads δεν λάβαμε αποτέλεσμα καθώς το πρόγραμμα δεν τερμάτισε ποτέ. Πιθανότατα αυτό είναι αποτέλεσμα deadlock.

clh_lock

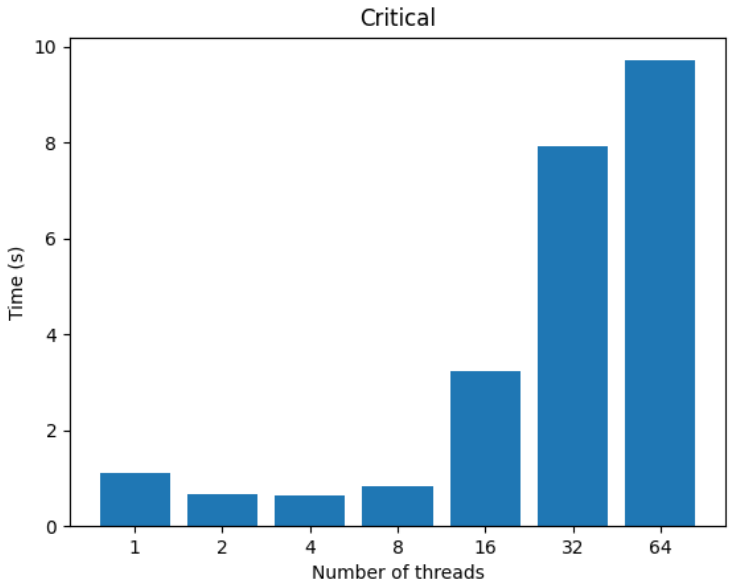
Εδώ χρησιμοποιούμε locks σε συνδεδεμένη λίστα και παίρνουμε τις παρακάτω μετρήσεις:



Παρατηρούμε ότι όλοι οι χρόνοι για κάθε αριθμό thread έχουν βελτιωθεί σε σχέση με όλες τις άλλες υλοποιήσεις που έχουμε δει ως τώρα, πλησιάζοντας σημαντικά την nosync υλοποίηση. Μάλιστα εμφανίζει καλύτερη επίδοση από την naive υλοποίηση ακόμα και για 16 ή 32 threads. Παρατηρούμε και εδώ ότι δεν τερματίζει το πρόγραμμα για 64 threads και υποπτευόμαστε ότι οφείλεται σε deadlock.

critical

Η υλοποίηση αυτή κάνει χρήση του directive ***#pragma omp critical*** στην κρίσιμη περιοχή, αντί του ***#pragma omp atomic*** που είχαμε στην naïve υλοποίηση. Οι μετρήσεις που λαμβάνουμε είναι:



Όπως βλέπουμε έχουμε καλούς χρόνους και scaling μέχρι τα 8 threads και πάλι. Ωστόσο για 16 threads και πάνω οι χρόνοι αυξάνονται σημαντικά και δεν παρουσιάζει καλή επίδοση.