Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Συστήματα Παράλληλης Επεξεργασίας

9ο εξάμηνο

<u>Άσκηση 3:</u> Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα

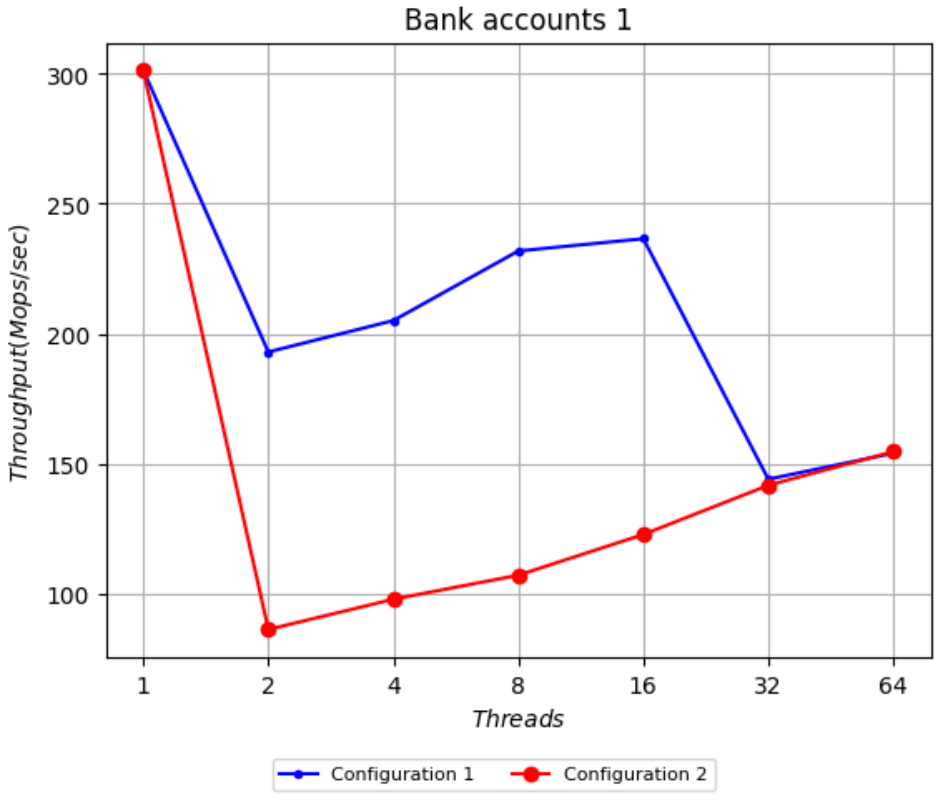
Δαζέα Ελένη 03114060
Καναβάκης Ελευθέριος 03114180

Μέρος 1 : Λογαριασμοί τράπεζας

Στο πρώτο μέρος της άσκησης μας δίνεται ένα πολυνηματικό πρόγραμμα όπου κάθε νήμα εκτελεί ένα σύνολο πράξεων πάνω σε συγκεκριμένο στοιχείο ενός πίνακα που αντιπροσωπεύει τους λογαριασμούς των πελατών μίας τράπεζας . Φυσικά , το γεγονός ότι κάθε νήμα εκτελεί πράξεις σε διαφορετικό στοιχείο του πίνακα μας καθιστά ξεκάθαρο πως δεν υπάρχει καμία ανάγκη για συγχρονισμό ανάμεσα στα νήματα της εφαρμογής μας.

Στην άσκηση αυτή μας ζητείται να πάρουμε μετρήσεις για διαφορετικό αριθμό νημάτων για δύο διαφορετικά configurations. Κατά τις μετρήσεις αυτές αυξάνουμε σταδιακά τα νήματα από 1 σε 64 . Παρατηρούμε ότι η εφαρμογή μας χρησιμοποιεί το throughput (Mops/sec) για την αξιολόγηση της επίδοσης. Η αναμενόμενη συμπεριφορά είναι πως αύξηση του αριθμού των νημάτων συνεπάγεται και αύξηση του throughput . Αυτό θα πρέπει να συμβαίνει γιατί ο χρόνος εκτέλεσης διατηρείται σταθερός και ίσος με περίπου 10 sec . Έτσι , το μόνο που μεταβάλλεται είναι ο αριθμός των πράξεων που εκτελούνται συνολικά . Φυσικά , μεγαλύτερος αριθμός νημάτων συνεπάγεται περισσότερους εργάτες οι οποίοι μάλιστα δρουν εντελώς ανεξάρτητα . Ως εκ τούτου , ο αριθμός των συνολικών πράξεων (ops) πρέπει να παρουσιάζει αύξηση όταν αυξάνεται ο αριθμός των νημάτων .

Στην συνέχεια παραθέτουμε διάγραμμα με τα αποτελέσματα που εξάγαμε από τις προσομοιώσεις μας.



Πριν προχωρήσουμε σε σχολιασμό των παραπάνω γραφικών παραστάσεων θεωρούμε σκόπιμο να σχολιάσουμε την διαφορά ανάμεσα στα δύο configurations που χρησιμοποιήσαμε . Παρατηρούμε ότι το πρώτο configuration προσπαθεί να βάλλει όσα περισσότερα νήματα μπορεί στον ίδιο επεξεργαστή . Δηλαδή , για παράδειγμα όταν έχουμε αριθμό νημάτων ίσο με 16 τοποθετεί και τα 16 νήματα στον πρώτο επεξεργαστή (16 είναι και η μέγιστη χωρητικότητα του). Αντίθετα , το δεύτερο configuration προσπαθεί να μοιράσει τα νήματα σε όσο δυνατόν περισσότερους επεξεργαστές . Δηλαδή , για αριθμό νημάτων ίσο με 16 μοιράζει 4 νήματα σε κάθε επεξεργαστή . Από τα παραπάνω καταλαβαίνουμε ότι το πρώτο configuration επιτυγχάνει σίγουρα καλύτερο cache locality . Επιπρόσθετα, δεν πρέπει να ξεχνάμε ότι βρισκόμαστε σε ένα NUMA σύστημα . Το γεγονός αυτό σημαίνει ότι η απόσταση που έχουν νήματα που τρέχουν σε κοντινές θέσεις μνήμης θέλουμε να είναι όσο γίνεται πιο κοντά για να περιορίσουμε το κόστος επικοινωνίας που έχουμε σε ένα τέτοιο σύστημα . Φυσικά , το πρώτο configuration είναι και πάλι αυτό που αξιοποιεί την ιδιαιτερότητα αυτή του συστήματος μας. Βέβαια , το γεγονός ότι το ένα από τα δύο configurations αξιοποιεί κάποιες ιδιαιτερότητες δεν συνεπάγεται απαραίτητα ότι αυτό είναι καλύτερο. Όπως θα δούμε παρακάτω η επιλογή του βέλτιστου configuration εξαρτάται από τον τρόπο με τον οποίο έχουμε υλοποιήσει τον κώδικα μας.

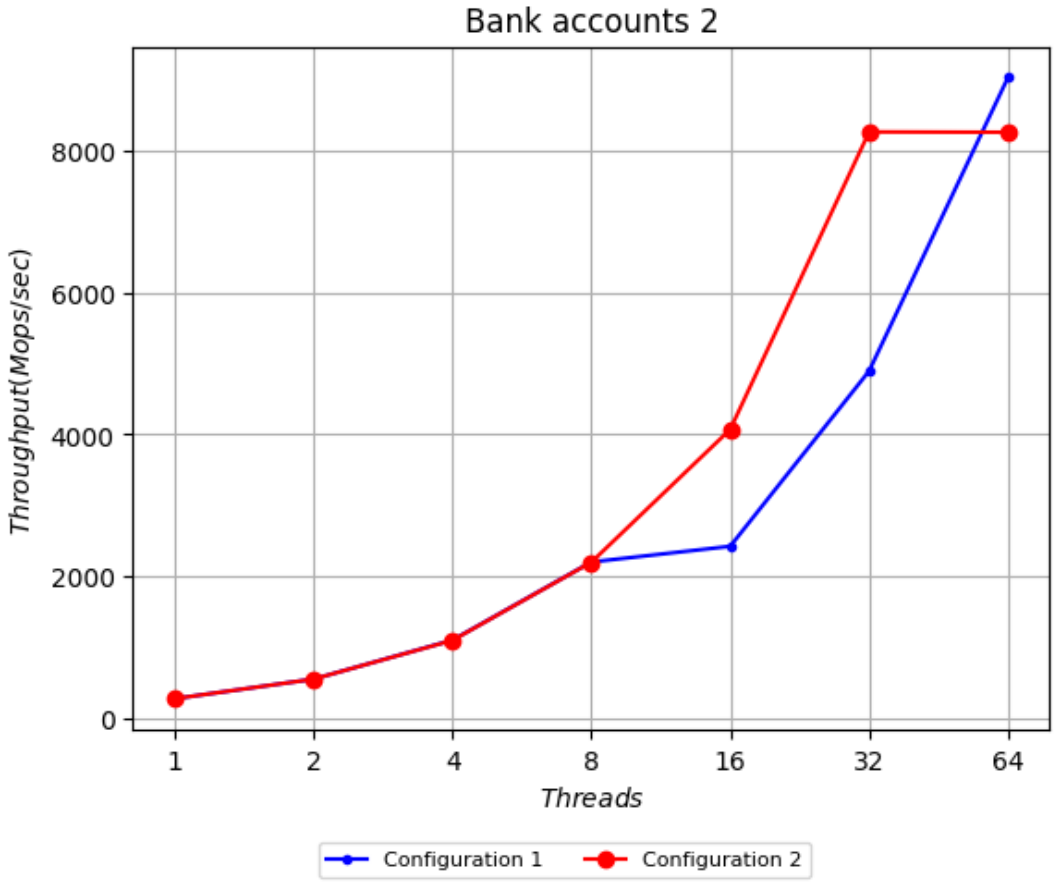
Στην συνέχεια θα προχωρήσουμε στον σχολιασμό της παραπάνω γραφικής παράστασης. Αρχικά , παρατηρούμε ότι στην γενική περίπτωση , αύξηση του αριθμού των νημάτων επιφέρει μείωση του throughput . Η συμπεριφορά αυτή φυσικά είναι μη αναμενόμενη αλλά θα σχολιαστεί αναλυτικά παρακάτω όταν και θα προσπαθήσουμε να την επιλύσουμε . Επιπρόσθετα , παρατηρούμε ότι οι καμπύλες μας έχουν ακριβώς την ίδια συμπεριφορά για αριθμό νημάτων ίσο με 1 και αριθμό νημάτων ίσο με 64 . Η εξήγηση είναι προφανής καθώς σε αυτές τις περιπτώσεις ουσιαστικά πρόκειται για το ίδιο configuration . Για τις υπόλοιπες δοκιμές (2,4,8,16,32 νήματα) παρατηρούμε ότι από 1 σε 2 νήματα υπάρχει μία ραγδαία πτώση (η συμπεριφορά αυτή είναι κοινή και για τα δύο configuration). Στην συνέχεια το πρώτο configuration παρουσιάζει μια αύξηση του throughput μέχρι τα 16 νήματα όπου και ξεκινάει να πέφτει και πάλι . Την ίδια συμπεριφορά παρουσιάζει και το configuration 2 με την διαφορά ότι αυτό αυξάνεται αργά μέχρι τα 32 νήματα . Σε γενικές γραμμές παρατηρούμε ότι το throughput είναι χειρότερο για παραπάνω από 1 νήματα και ότι το configuration 1 παρουσιάζει μια overall καλύτερη συμπεριφορά από το configuration 2.

Στο σημείο αυτό μας δημιουργείται ένας προβληματισμός καθώς όπως προείπαμε αύξηση του αριθμού νημάτων επιφέρει μείωση του throughput αντί για αύξηση του . Το γεγονός αυτό μας οδήγησε στο να αναζητήσουμε το πρόβλημα στον κώδικα μας . Φυσικά , με μια δεύτερη ματιά το πρόβλημα είναι εμφανές . Παρατηρούμε , ότι ενώ στο struct του που περιλαμβάνει τις πληροφορίες του κάθε νήματος γίνεται padding , στο struct που αφορά τον πίνακα με τους λογαριασμούς δεν γίνεται . Το γεγονός αυτό είναι καταστροφικό καθώς έτσι γίνονται overlap στην cache τιμές του πίνακα accounts που επεξεργάζονται διαφορετικοί πυρήνες . Έτσι , σε κάθε write που κάνει ένας επεξεργαστείς σε μία θέση μνήμης του πίνακα όλοι οι υπόλοιποι πυρήνες που έχουν την θέση μνήμης αυτής στην cache τους (χωρίς να την χρειάζονται) δέχονται coherence miss . Ως επακόλουθο , έχουμε συμφόρηση του διαύλου και πολλούς κύκλους οι οποίοι καταναλώνονται σε coherency misses . Το γεγονός αυτός εξηγεί και τον λόγο που το configuration 1 ήταν καλύτερο από το 2 . Όπως προείπαμε στο configuration 1 τα δεδομένα διατηρούνται όσο γίνεται πιο κοντά (στον ίδιο επεξεργαστή) και έτσι εφόσον το σύστημα είναι NUMA γλυτώνουμε το κόστος της επικοινωνίας που πληρώνουμε για το δεύτερο configuration .

Η λύση του παραπάνω προβλήματος είναι απλή (padding στο struct) και ο κώδικας φαίνεται παρακάτω :

```
struct {
        unsigned int value;
        char padding2[64 - sizeof(unsigned int)];
} accounts[MAX_THREADS];
```

Στην συνέχεια επαναλάβαμε τις παραπάνω μετρήσεις και παρουσιάζουμε τα αποτελέσματα στην παρακάτω γραφική παράσταση :

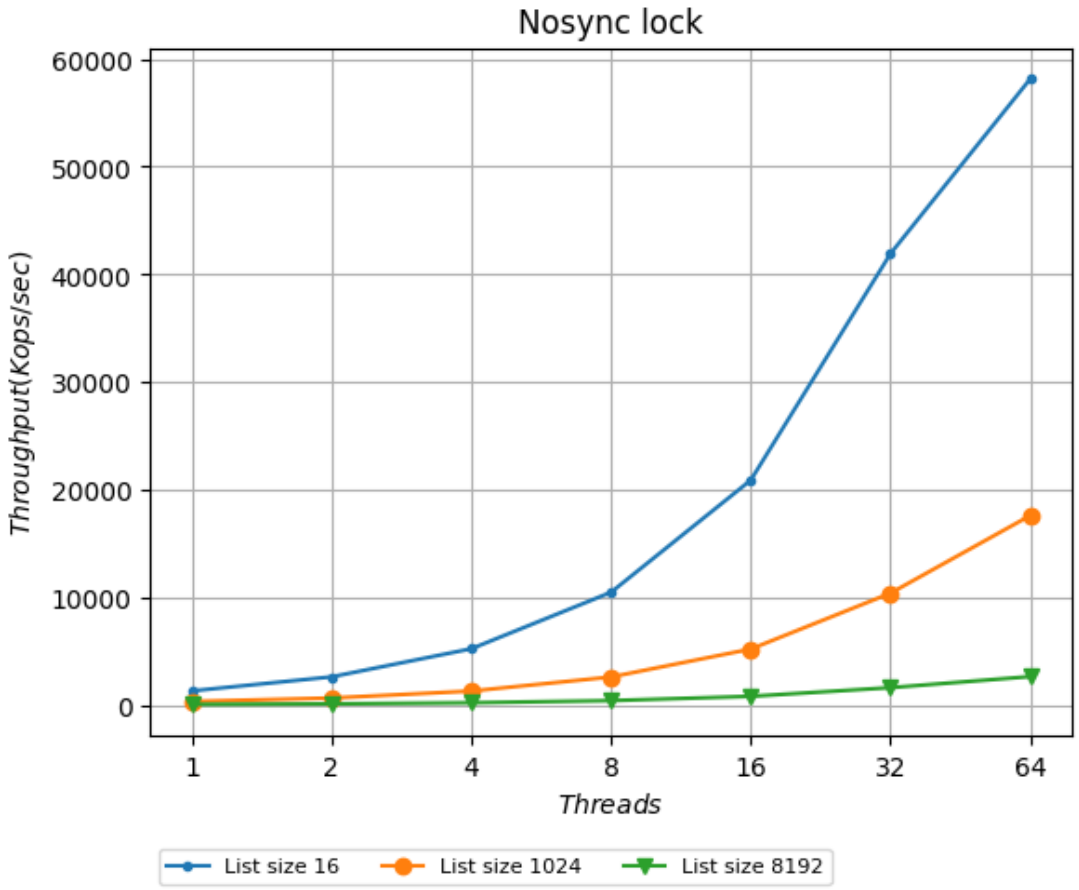


Παρατηρούμε ότι με την λύση που προτείναμε η συμπεριφορά είναι πλέον η αναμενόμενη καθώς η αύξηση του αριθμού των νημάτων συνεπάγεται και αύξηση του throughput . Επιπρόσθετα , παρατηρούμε ότι για αριθμό νημάτων ίσο με 16 και 32 το configuration 2 είναι αρκετά καλύτερο από το 1 ενώ εμείς θα περιμέναμε παρόμοια συμπεριφορά . Αυτό πιθανώς να οφείλεται στο ότι στο πρώτο configuration χρησιμοποιούμε την τεχνική του hyperthreading ενώ στο δεύτερο χρησιμοποιούμε μόνος αληθινούς πυρήνες για να τρέξουμε τα νήματα μας . Ο λόγος για τον οποίο το hyperthreading είναι μάλλον πιο αργό ίσως έχει να κάνει με το γεγονός ότι η cache μοιράζεται κατά την τεχνική αυτή. Τέλος , παρατηρούμε ότι για αριθμό νημάτων ίσο με 64 παίρνουμε λίγο διαφορετική μέτρηση για τα δύο configuration ενώ θα έπρεπε να ήταν ίδια . Αυτό οφείλεται στο γεγονός ότι ο χρόνος με τον οποίο διαιρούμε για να υπολογίσουμε το throughput είναι περίπου ο ίδιος κάθε φορά (περίπου 10 sec) και όχι ακριβώς ο ίδιος.

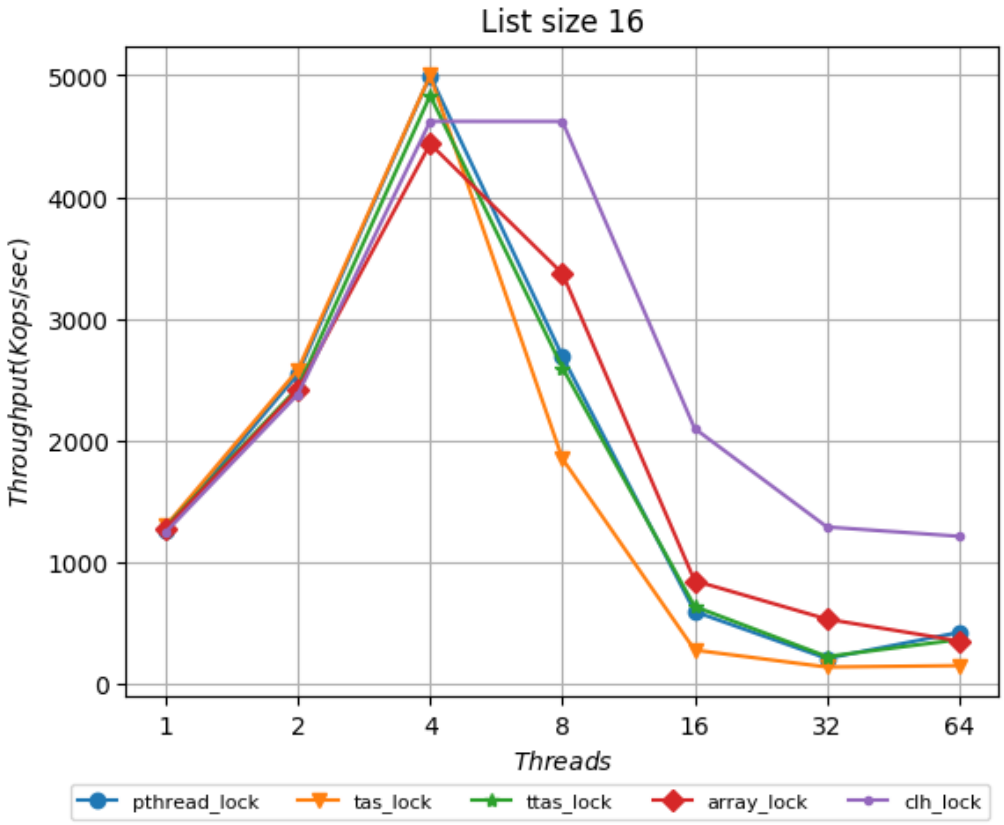
Μέρος 2 : Αμοιβαίος Αποκλεισμός – Κλειδώματα

Στο δεύτερο μέρος της άσκησης αυτής μελετήσαμε διαφορετικά είδη κλειδωμάτων τα οποία εφαρμόζονται στο ίδιο κρίσιμο τμήμα . Το critical section αυτό περιλαμβάνει την αναζήτηση τυχαίων στοιχείων σε μία ταξινομημένη συνδεδεμένη λίστα . Στην συνέχεια , παραθέτουμε γραφικές παραστάσεις στις οποίες συγκρίνουμε την επίδοση των διαφορετικών κλειδωμάτων που μελετήσαμε . Για την καλύτερη αναπαράσταση των αποτελεσμάτων μας επιλέξαμε να παραθέσουμε το nosync_lock σε διαφορετική γραφική παράσταση καθώς όπως ήταν αναμενόμενο οι τιμές του throughput του ήταν πολύ υψηλές και έτσι η (οπτική) σύγκριση μεταξύ των υπολοίπων κλειδωμάτων γινόταν δυσκολότερη .

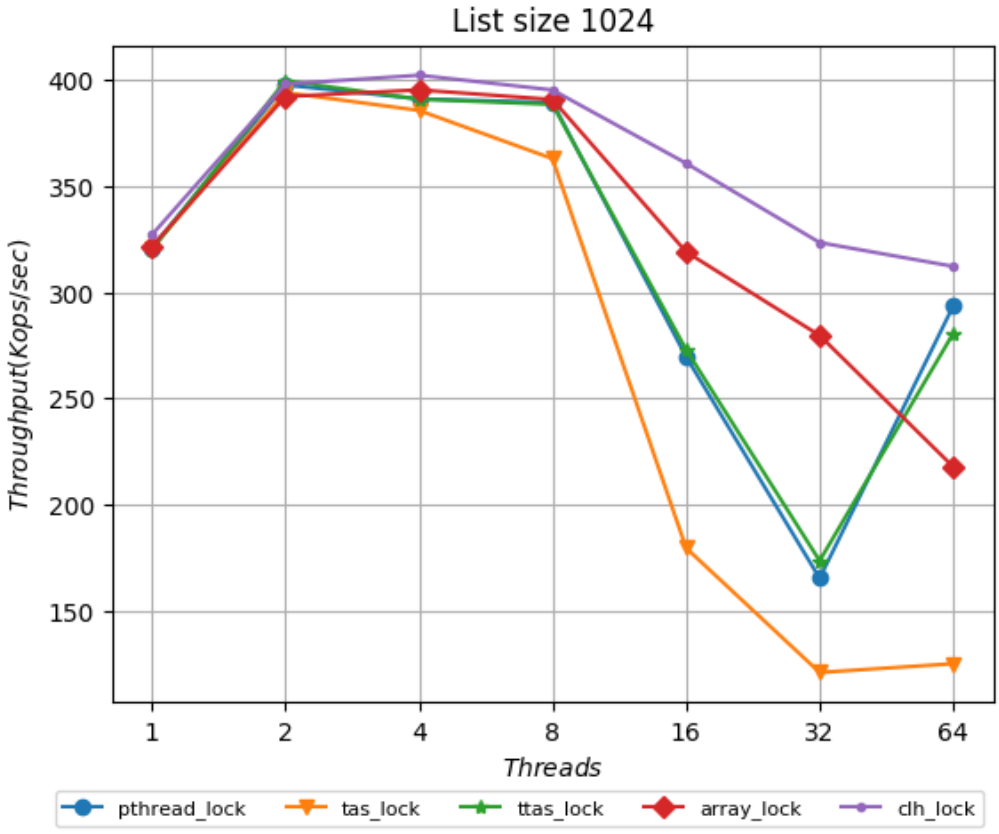
Nosync_lock :



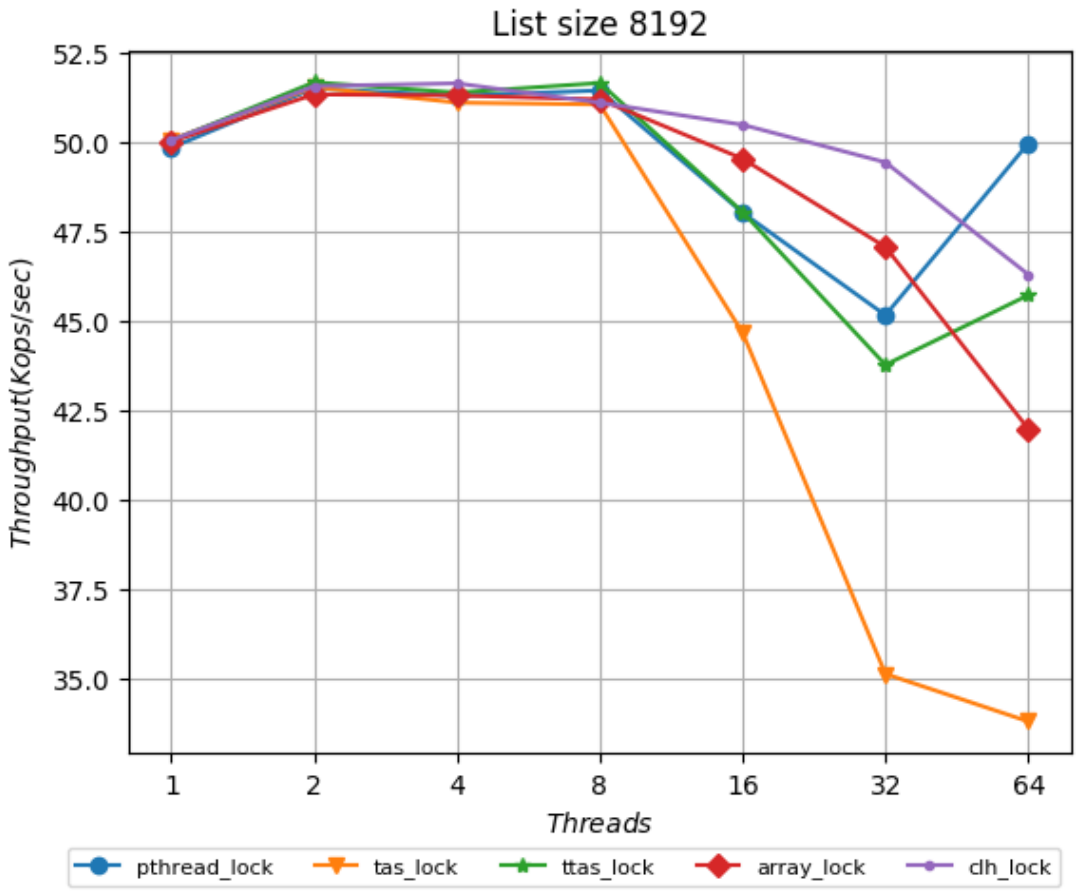
Μέγεθος λίστας 16 :



Μέγεθος λίστας 1024 :



Μέγεθος λίστας 8192 :



Το πρώτο πράγμα που παρατηρούμε είναι ότι στην γενική περίπτωση αύξηση του μεγέθους της λίστας συνεπάγεται και μείωση του throughput . Το γεγονός αυτό οφείλεται στο ότι η contains που εκτελεί το thread μας αφού κάνει acquire το lock διατρέχει όλη την λίστα από την αρχή . Έτσι , αύξηση του μεγέθους της λίστας συνεπάγεται πολύ περισσότερα accesses για την contains . Επιπρόσθετα , παρατηρούμε ότι κανένα κλείδωμα δεν πλησιάζει την επίδοση της nosync αλλά αυτό είναι αναμενόμενο καθώς η nosync δεν κλειδώνει και δεν εξασφαλίζει την σωστή λειτουργία του προγράμματος μας .Στην συνέχεια , παρατηρούμε ότι overall την χειρότερη επίδοση παρουσιάζει η Test And Set (TAS) . Πιο συγκεκριμένα , παρατηρούμε ότι για αριθμό νημάτων μεγαλύτερο του 4 το throughput του κλειδώματος αυτού παρουσιάζει δραματική πτώση . Η συμπεριφορά αυτή οφείλεται στο γεγονός ότι η TAS προκαλεί συμφόρηση στον δίαυλο αφού δημιουργεί πολλά coherency misses ! Στην συνέχεια παρατηρούμε ότι η pthread_lock καθώς και η TTAS παρουσιάζουν παρόμοιες συμπεριφορές , οι οποίες είναι σταθερά

καλύτερες από την TAS . Το γεγονός αυτό συμβαίνει γιατί τα κλειδώματα αυτά δεν προκαλούν την συμφόρηση του διαύλου που προκαλεί η TAS. Επιπρόσθετα, παρατηρούμε ότι η array lock παρουσιάζει σε γενικές γραμμές καλύτερη συμπεριφορά από τις υπόλοιπες (εκτός της clh) . Αυτό συμβαίνει γιατί με την array_lock το κάθε νήμα δουλεύει στην δική του θέση μνήμης όσο αναφορά το lock και έτσι μειώνονται τα coherency misses . Τέλος , παρατηρούμε ότι η clh_lock είναι σε γενικές γραμμές η καλύτερη καθώς αντιμετωπίζει όλα τα προβλήματα που είχαν οι προηγούμενες .

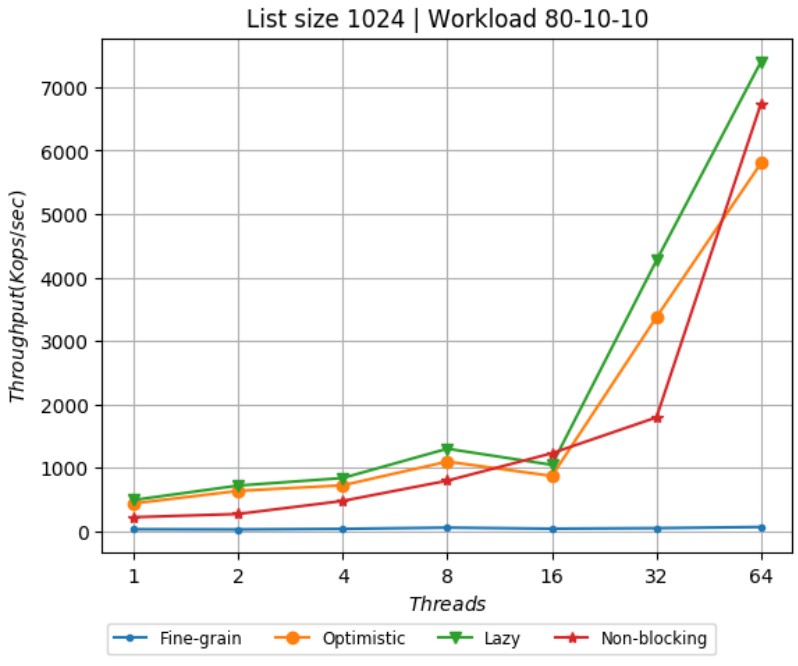
Μέρος 3 : Τακτικές συγχρονισμού για δομές δεδομένων

Στο τρίτο μέρος της άσκησης αυτής θα ασχοληθούμε με τους εναλλακτικούς τρόπους κλειδωμάτων σε δομές δεδομένων όπως παρουσιάστηκαν στις διαλέξεις του μαθήματος . Σαν δομή δεδομένων θα χρησιμοποιήσουμε μια ταξινομημένη συνδεδεμένη λίστα για την αξιολόγηση των κλειδωμάτων μας ! Οι τακτικές συγχρονισμού οι οποίες υλοποιήσαμε είναι οι παρακάτω :

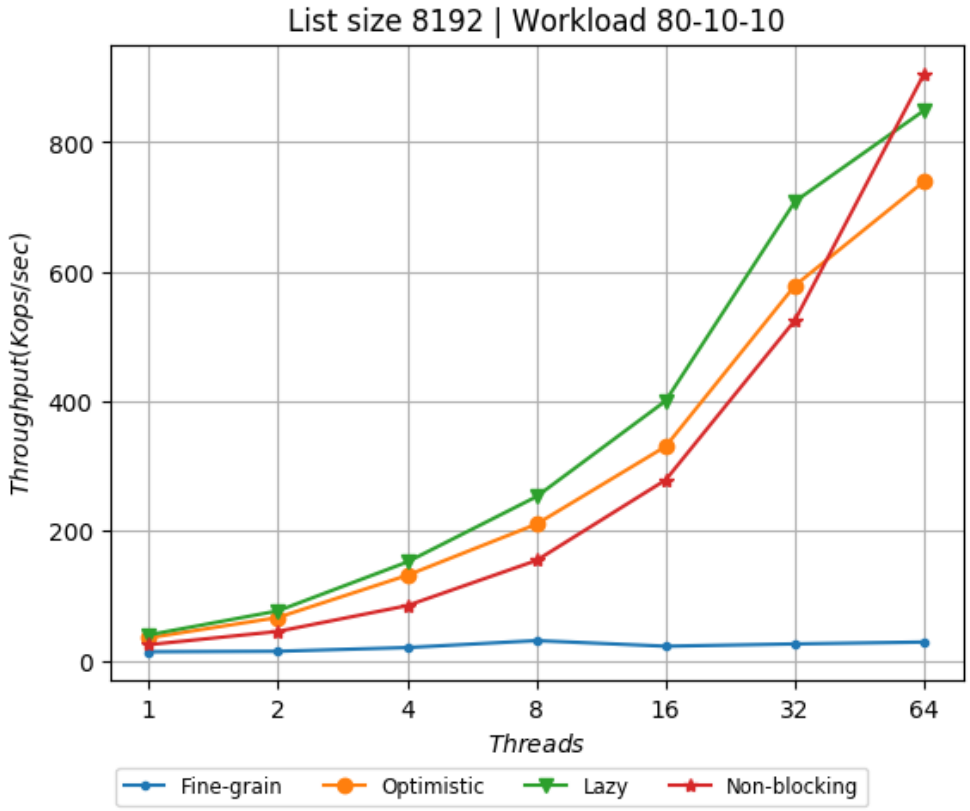
- ✓ Fine-grain Locking
- ✓ Optimistic synchronization
- ✓ Lazy synchronization
- ✓ Non-blocking synchronization

Στην συνέχεια , θα παραθέσουμε γραφικές παραστάσεις στις οποίες θα συγκρίνουμε τις παραπάνω υλοποιήσεις . Για να εξάγουμε τις γραφικές παραστάσεις αυτές πήραμε μετρήσεις για δύο διαφορετικά μεγέθη λίστας (1024 και 8192) και δύο διαφορετικά workloads(80-10-10 και 20-40-40).

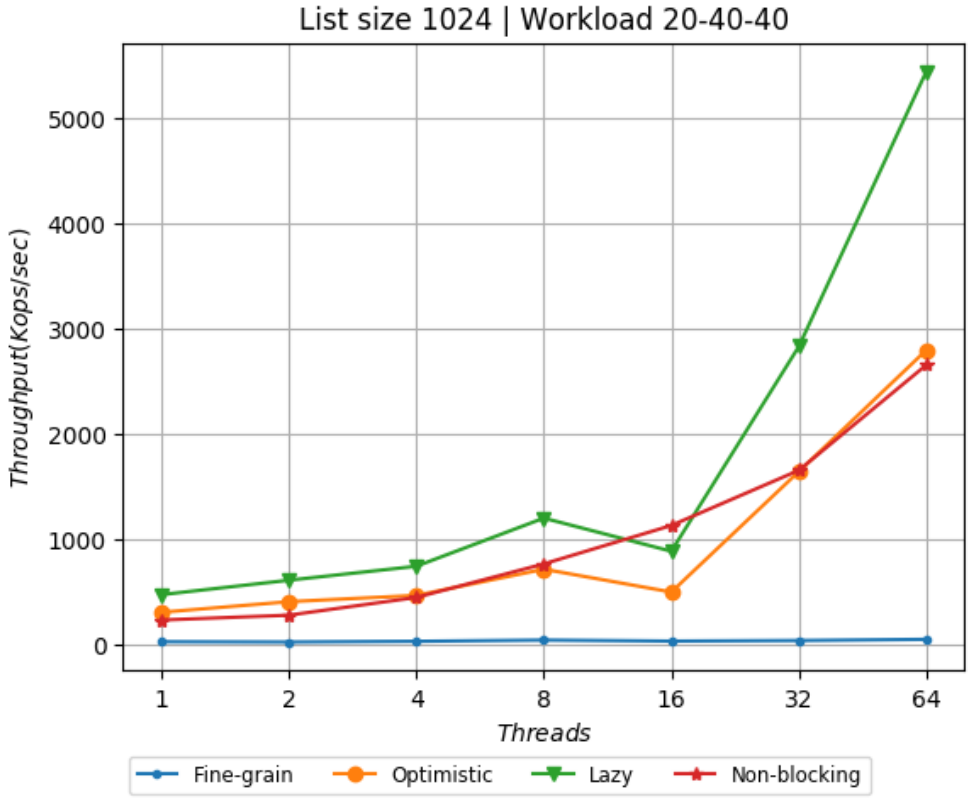
Μέγεθος λίστας 1024 | Workload 80-10-10 :



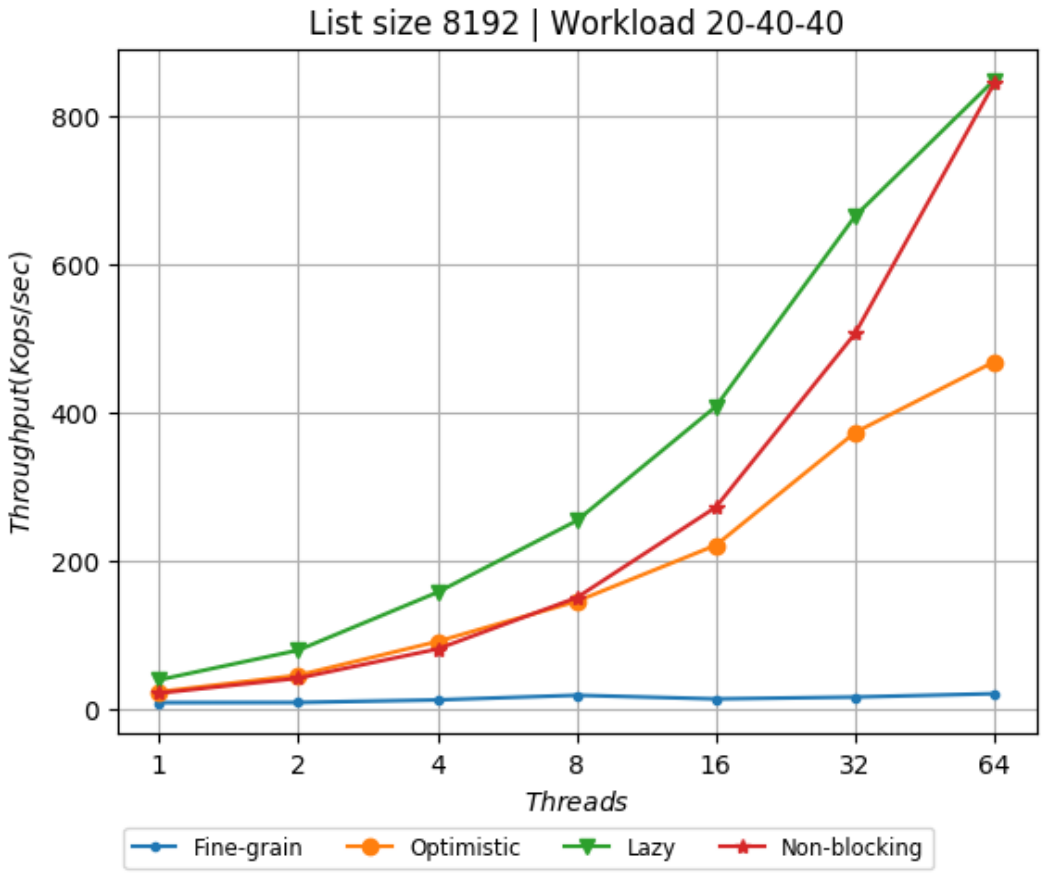
Μέγεθος λίστας 8192 | Workload 80-10-10 :



Μέγεθος λίστας 1024 | Workload 20-40-40 :



Μέγεθος λίστας 8192 | Workload 20-40-40 :



Παρατηρούμε ότι σε γενικές γραμμές η fine-grain είναι η χειρότερη από όλες. Το γεγονός αυτό είναι απόλυτα λογικό καθώς η υλοποίηση αυτή αποτρέπει νήματα τα οποία πιθανώς να εργάζονται σε διαφορετικά μέρη της λίστας να δουλέψουν παράλληλα . Παράλληλα , η μέθοδος contains της fine-grain περιέχει κλειδώματα γεγονός που οδηγεί σε αρκετά χειρότερη επίδοση . Στην συνέχεια , παρατηρούμε ότι η optimistic είναι καλύτερη από την fine-grain αλλά χειρότερη από τις lazy και non-blocking . Η optimistic είναι καλύτερη από την fine-grain καθώς όχι μόνο επιτρέπει στα νήματα να εργάζονται ταυτόχρονα σε διαφορετικά κομμάτια της λίστας αλλά και η μέθοδος contains που διαθέτει δεν περιέχει κλειδώματα . Στην συνέχεια , παρατηρούμε ότι οι lazy και non-blocking παρουσιάζουν αρκετά καλύτερη επίδοση από την optimistic καθώς πλέον η μέθοδος contains δεν χρειάζεται να διατρέχει ολόκληρη την λίστα αλλά κάνει τοπικούς ελέγχους (η διαφορά αυτή προφανώς φαίνεται περισσότερο όταν το μέγεθος της λίστας είναι αρκετά μεγάλο , ενώ για μικρά μεγέθη λιστών το impact είναι σχετικά μικρό). Τέλος , παρατηρούμε ότι σε κάποιες περιπτώσεις η lazy έχει καλύτερη επίδοση από την non-blocking . Η παρατήρηση αυτή είναι μη αναμενόμενη

και αποκλίνει από την θεωρία . Η συμπεριφορά αυτή πιθανώς να οφείλεται στο persistent delete που προσπαθεί να κάνει η find ή σε χειρότερη απόδοση των ατομικών εντολών έναντι στα spinlocks.

Κώδικες :

accounts.c :

```
/**
 * A very simple parallel application
 * that handles an array of bank accounts.
 **/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#include "timer.h"
#include "aff.h"

#define MAX_THREADS 64
#define RUNTIME 10

#define print_error_and_exit(format...) \
    do { \
        fprintf(stderr, format); \
        exit(EXIT_FAILURE); \
    } while (0)

/**
 * Global data.
**/
pthread_barrier_t start_barrier;
int time_to_leave;

/**
 * The accounts' array.
 **/
struct {
    unsigned int value;

    char padding2[64 - sizeof(unsigned int)];
} accounts[MAX_THREADS];

/**
 * The struct that is passed as an argument to each thread.
**/
typedef struct {
    int tid,
        cpu;
    unsigned long long ops;

    /* Why do we use padding here? */
    char padding[64 - 2*sizeof(int) - sizeof(unsigned long long)];
} tdata_t;
```

```
void *thread_fn(void *targ);

int main()
{
    timer_tt *wall_timer;
    pthread_t threads[MAX_THREADS];
    tdata_t threads_data[MAX_THREADS];
    unsigned int nthreads = 0, *cpus;
    unsigned int i;

    //> Initializations.
    get_mtconf_options(&nthreads, &cpus);
    mt_conf_print(nthreads, cpus);
    if (pthread_barrier_init(&start_barrier, NULL, nthreads+1))
        print_error_and_exit("Failed to initialize
start_barrier.\n");
    wall_timer = timer_init();

    //> Spawn threads.
    for (i=0; i < nthreads; i++) {
        threads_data[i].tid = i;
        threads_data[i].cpu = cpus[i];
        threads_data[i].ops = 0;
        if (pthread_create(&threads[i], NULL, thread_fn,
&threads_data[i]))
            print_error_and_exit("Error creating thread %d.\n", i);
    }

    //> Signal threads to start computation.
    pthread_barrier_wait(&start_barrier);
    timer_start(wall_timer);

    sleep(RUNTIME);
    time_to_leave = 1;

    //> Wait for threads to complete their execution.
    for (i=0; i < nthreads; i++) {
        if (pthread_join(threads[i], NULL))
            print_error_and_exit("Failure on pthread_join for thread
%d.\n", i);
    }

    timer_stop(wall_timer);

    //> How many operations have been performed by all threads?
    unsigned long long total_ops = 0;
    for (i=0; i < nthreads; i++)
        total_ops += threads_data[i].ops;

    //> Print results.
    double secs = timer_report_sec(wall_timer);
    double throughout = (double)total_ops / secs / 1000000.0;
    printf("Nthreads: %d  Runtime(sec): %d  Throughput(Mops/sec):
%5.2lf\n",
            nthreads, RUNTIME, throughout);

    return EXIT_SUCCESS;
}

/**
```

```
 * This is the function that is executed by each spawned thread.
 **/
void *thread_fn(void *targ)
{
    tdata_t *mydata = targ;
    int i;

    //> Pin thread to the specified cpu.
    setaffinity_oncpu(mydata->cpu);

    //> Wait until master gives the green light!
    pthread_barrier_wait(&start_barrier);

    while (!time_to_leave) {
        for (i=0; i < 1000; i++)
            accounts[mydata->tid].value++;
        mydata->ops += 1000;
    }

    return NULL;
}
```

pthread_lock.c :

```
#include "lock.h"
#include "alloc.h"
#include <pthread.h>

struct lock_struct {
    pthread_spinlock_t mylock ;
};

lock_t *lock_init(int nthreads)
{
    lock_t *lock;

    XMALLOC(lock, 1);
    /* other initializations here. */
    pthread_spin_init(&lock->mylock,1);
    return lock;
}

void lock_free(lock_t *lock)
{
    XFREE(lock);
}

void lock_acquire(lock_t *lock)
{
    pthread_spin_lock(&lock->mylock);
}

void lock_release(lock_t *lock)
{
    pthread_spin_unlock(&lock->mylock);
}
```

ttas_lock.c :

```
#include "lock.h"
#include "alloc.h"

typedef enum {
        UNLOCKED = 0,
        LOCKED
} lock_state_t;


struct lock_struct {

    lock_state_t state;
};

lock_t *lock_init(int nthreads)
{
    lock_t *lock;

    XMALLOC(lock, 1);
    /* other initializations here. */
    lock->state = UNLOCKED;
    return lock;
}

void lock_free(lock_t *lock)
{
    XFREE(lock);
}

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

        while(1) {
        while((l->state)==LOCKED) ;
        if(__sync_lock_test_and_set(&l->state, LOCKED) == UNLOCKED)
                return;

    }
                /* do nothing */ ;
}

void lock_release(lock_t *lock)
{
        lock_t *l = lock;


        __sync_lock_release(&l->state);
}
```

array_lock.c :

```
#include "lock.h"
#include "alloc.h"
#include <pthread.h>

static __thread int index;

struct lock_struct {

    int* flag;
    int tail, size;

};

lock_t *lock_init(int nthreads)
{
    lock_t *lock;

    XMALLOC(lock, 1);
    /* other initializations here. */
    lock->size = nthreads;
    XMALLOC(lock->flag, lock->size);
    lock->flag[0] = 1;
    lock->tail = 0;
    return lock;
}

void lock_free(lock_t *lock)
{
    XFREE(lock->flag);
    XFREE(lock);
    return;
}

void lock_acquire(lock_t *lock)
{
    index = __sync_fetch_and_add(&(lock->tail),1)%(lock->size);
    while(!lock->flag[index]) ;
    return;

}

void lock_release(lock_t *lock)
{
    lock->flag[index] = 0;
    lock->flag[(index+1) % (lock->size)] = 1;
    return;

}
```

ll_fgl.c :

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "alloc.h"
#include "ll.h"

typedef struct ll_node {
    int key;
    pthread_spinlock_t state;
    struct ll_node *next;

    /* other fields here? */
} ll_node_t;

struct linked_list {
    ll_node_t *head;
    /* other fields here? */
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;
    XMALLOC(ret, 1);
    ret->key = key;
    ret->next = NULL;
    /* Other initializations here? */
    pthread_spin_init(&ret->state,1);
    return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
    XFREE(ll_node);
}

/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
    ll_t *ret;

    XMALLOC(ret, 1);
    ret->head = ll_node_new(-1);
    ret->head->next = ll_node_new(INT_MAX);
    ret->head->next->next = NULL;

    return ret;
}

/**
```

```
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

int ll_contains(ll_t *ll, int key)
{

    ll_node_t *prev,*curr;
        pthread_spin_lock(&ll->head->state);
        prev=ll->head;
        curr = prev->next;
        pthread_spin_lock(&curr->state);
        while(curr->key<=key) {
                        if(curr->key==key) {
                                pthread_spin_unlock(&prev->state);
                                pthread_spin_unlock(&curr->state);
                                return 1;
                        }
                        pthread_spin_unlock(&prev->state);
                        prev=curr;
                        curr = curr->next;
                        pthread_spin_lock(&curr->state);
                }
                pthread_spin_unlock(&curr->state);
                pthread_spin_unlock(&prev->state);



        return 0;}

int ll_add(ll_t *ll, int key)
{
    ll_node_t *prev,*curr;
        pthread_spin_lock(&ll->head->state);
    prev = ll->head;
        if(prev->key>key) {
                ll->head = ll_node_new(key);
                ll->head->next=prev;
                pthread_spin_unlock(&prev->state);
                 }
        else if(prev->next==NULL) {
                curr = ll_node_new(key);
                prev->next=curr;
                pthread_spin_unlock(&prev->state);

        }
    else {
        curr=prev->next;
        pthread_spin_lock(&curr->state);
        while(curr->key<key) {
                pthread_spin_unlock(&prev->state);
                prev = curr;
```

```
                curr = curr->next;
                if(curr==NULL) break;
                pthread_spin_lock(&curr->state);
                }
        prev->next=ll_node_new(key);
        prev->next->next=curr;
        pthread_spin_unlock(&prev->state);
        if(curr!=NULL) pthread_spin_unlock(&curr->state);
}
        return key;
}

int ll_remove(ll_t *ll, int key)
{
    int ret=0;
    ll_node_t *prev,*curr;
        pthread_spin_lock(&ll->head->state);
    prev = ll->head;
        curr = prev->next;
        pthread_spin_lock(&curr->state);
        while (curr->key <= key) {
                if(curr->key==key) {
            prev->next=curr->next;
                        pthread_spin_unlock(&prev->state);
                        ll_node_free(curr);
                        ret= 1;
            break;}
        else {
                pthread_spin_unlock(&prev->state);
                prev = curr;
                curr = curr->next;
                pthread_spin_lock(&curr->state);
        }}
    if(ret==0) {
        pthread_spin_unlock(&prev->state);
        pthread_spin_unlock(&curr->state);}


        return ret;
}

/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
    ll_node_t *curr = ll->head;
    printf("LIST [");
    while (curr) {
        if (curr->key == INT_MAX)
            printf(" -> MAX");
        else
            printf(" -> %d", curr->key);
        curr = curr->next;
    }
    printf(" ]\n");
}
```

ll_lazy.c :

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "alloc.h"
#include "ll.h"

typedef struct ll_node {
    int key;
    int tag;
    struct ll_node *next;

    pthread_spinlock_t state ;

} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;

    XMALLOC(ret, 1);
    ret->key = key;
    ret->next = NULL;
    ret->tag = 0;
    pthread_spin_init(&ret->state,PTHREAD_PROCESS_PRIVATE);

    return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
    pthread_spin_destroy(&ll_node->state);
    XFREE(ll_node);
}

/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
    ll_t *ret;

    XMALLOC(ret, 1);
```

```
    ret->head = ll_node_new(-1);
    ret->head->next = ll_node_new(INT_MAX);
    ret->head->next->next = NULL;

    return ret;
}

/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

int ll_contains(ll_t *ll, int key)
{
    int ret = 0;
    ll_node_t *curr;
    curr=ll->head ;

    while(curr->key < key) {
        curr=curr->next;
    }

    ret = ((key==curr->key) && (!curr->tag));
    return ret;
}


int validate(ll_node_t *prev, ll_node_t *curr)
{
    int ret = 0;

    ret = ((!prev->tag) && (!curr->tag) && (prev->next==curr));
        return ret;
}

int ll_add(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *new_node;
        ll_node_t *prev,*curr;

        while(1) {

        prev = ll->head;
        curr = prev->next;

        while( curr->key <= key) {
            if(curr->key == key) break;
            prev = curr;
            curr = curr->next;
        }
        pthread_spin_lock(&(prev->state));
```

```
        pthread_spin_lock(&(curr->state));

        if(validate(prev,curr)){
            if(curr->key != key){
                new_node=ll_node_new(key);
                new_node->next = curr;
                prev->next = new_node;
                ret = 1;
            }
            pthread_spin_unlock(&(prev->state));
            pthread_spin_unlock(&(curr->state));
            break;
        }
        pthread_spin_unlock(&(prev->state));
        pthread_spin_unlock(&(curr->state));
        }
        return ret;
}


int ll_remove(ll_t *ll, int key)
{

        int ret=0;
        ll_node_t *prev,*curr;

        while(1) {

        prev=ll->head;
        curr=prev->next;

        while(curr->key <= key) {
            if(key == curr->key) break;
            prev=curr;
            curr=curr->next;
        }

        pthread_spin_lock(&(prev->state));
        pthread_spin_lock(&(curr->state));

        if(validate(prev,curr)) {
            if(curr->key == key) {
                curr->tag = 1;
                prev->next=curr->next;
                ret=1;
            }
            pthread_spin_unlock(&(prev->state));
            pthread_spin_unlock(&(curr->state));
            break;
        }
        pthread_spin_unlock(&(prev->state));
        pthread_spin_unlock(&(curr->state));
        }

        return ret;
}
/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
```

```
    ll_node_t *curr = ll->head;
    printf("LIST [");
    while (curr) {
        if (curr->key == INT_MAX)
            printf(" -> MAX");
        else
            printf(" -> %d", curr->key);
        curr = curr->next;
    }
    printf(" ]\n");
}
```

ll_opt.c :

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "alloc.h"
#include "ll.h"

typedef struct ll_node {
    int key;
    struct ll_node *next;

    pthread_spinlock_t state ;

} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;

    XMALLOC(ret, 1);
    ret->key = key;
    ret->next = NULL;
    pthread_spin_init(&ret->state,PTHREAD_PROCESS_PRIVATE);

    return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
    pthread_spin_destroy(&ll_node->state);
    XFREE(ll_node);
}

/**
 * Create a new empty linked list.
```

```
 **/
ll_t *ll_new()
{
    ll_t *ret;

    XMALLOC(ret, 1);
    ret->head = ll_node_new(-1);
    ret->head->next = ll_node_new(INT_MAX);
    ret->head->next->next = NULL;

    return ret;
}

/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

int ll_contains(ll_t *ll, int key)
{
    int ret = 0;
    ll_node_t *curr;
    curr=ll->head ;

    while(curr->key < key) {
        curr=curr->next;
    }

    ret = (key==curr->key);
    return ret;
}


int validate(ll_t *ll, ll_node_t *prev, ll_node_t *curr)
{
        ll_node_t *current;
        current = ll->head;

        while(current->key <= prev->key) {
                if (current == prev) return current->next == curr;
                current = current->next;
        }

        return 0;
}

int ll_add(ll_t *ll, int key)
{
        int ret = 0;
        ll_node_t *new_node;
        ll_node_t *prev,*curr;
```

```
        while(1) {

        prev = ll->head;
        curr = prev->next;

        while( curr->key <= key) {
            if(curr->key == key) break;
            prev = curr;
            curr = curr->next;
        }
        pthread_spin_lock(&(prev->state));
        pthread_spin_lock(&(curr->state));

        if(validate(ll,prev,curr)){
            if(curr->key != key){
                new_node=ll_node_new(key);
                new_node->next = curr;
                prev->next = new_node;
                ret = 1;
            }
            pthread_spin_unlock(&(prev->state));
            pthread_spin_unlock(&(curr->state));
            break;
        }
        pthread_spin_unlock(&(prev->state));
        pthread_spin_unlock(&(curr->state));
        }
        return ret;
}


int ll_remove(ll_t *ll, int key)
{

        int ret=0;
        ll_node_t *prev,*curr;

        while(1) {

        prev=ll->head;
        curr=prev->next;

        while(curr->key <= key) {
            if(key == curr->key) break;
            prev=curr;
            curr=curr->next;
        }

        pthread_spin_lock(&(prev->state));
        pthread_spin_lock(&(curr->state));

        if(validate(ll,prev,curr)) {
            if(curr->key == key) {
                prev->next=curr->next;
                ret=1;
            }
            pthread_spin_unlock(&(prev->state));
            pthread_spin_unlock(&(curr->state));
            break;
        }
        pthread_spin_unlock(&(prev->state));
```

```
        pthread_spin_unlock(&(curr->state));
        }

        return ret;
}
/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
    ll_node_t *curr = ll->head;
    printf("LIST [");
    while (curr) {
        if (curr->key == INT_MAX)
            printf(" -> MAX");
        else
            printf(" -> %d", curr->key);
        curr = curr->next;
    }
    printf(" ]\n");
}
```

ll_nb.c :

```
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <pthread.h> /* for pthread_spinlock_t */

#include "alloc.h"
#include "ll.h"

typedef struct ll_node {
    int key;
    struct ll_node *next;
    char padding2[64-sizeof(int)-sizeof(struct ll_node*)];

} ll_node_t;

struct linked_list {
    ll_node_t *head;
};

/**
 * Create a new linked list node.
 **/
static ll_node_t *ll_node_new(int key)
{
    ll_node_t *ret;

    XMALLOC(ret, 1);
    ret->key = key;
    ret->next = NULL;

    return ret;
}

/**
 * Free a linked list node.
 **/
```

```
static void ll_node_free(ll_node_t *ll_node)
{
    XFREE(ll_node);
}

/**
 * Create a new empty linked list.
 **/
ll_t *ll_new()
{
    ll_t *ret;

    XMALLOC(ret, 1);
    ret->head = ll_node_new(-1);
    ret->head->next = ll_node_new(INT_MAX);
    ret->head->next->next = NULL;

    return ret;
}

/**
 * Free a linked list and all its contained nodes.
 **/
void ll_free(ll_t *ll)
{
    ll_node_t *next, *curr = ll->head;
    while (curr) {
        next = curr->next;
        ll_node_free(curr);
        curr = next;
    }
    XFREE(ll);
}

int ll_contains(ll_t *ll, int key)
{
    int ret = 0;
    ll_node_t *curr;
    curr=ll->head ;
    //curr=curr->next;

    while(curr->key < key) {
        curr=curr->next;
    }

    ret = ((key==curr->key) && (!(long long)curr)&1);
    return ret;
}

int find(ll_t *ll, int key, ll_node_t **prev, ll_node_t **curr) {

    int snip = 0;
    int flag = 0;
    ll_node_t *previous,*current;

    while(1) {

        snip=0;
        previous = ll->head;
        current = previous->next;
```

```
        while(1) {

            if(((long long)current)&1) {
                snip = __sync_bool_compare_and_swap(&previous->next,current,current->next);
                break;
            }

            if(current->key >= key) {
                flag=1;
                break;
            }
            previous = current;
            current = current->next;
        }
        if(!snip && !flag) continue;
        *prev = previous;
        *curr = current;
        return 1;
    }
}


int ll_add(ll_t *ll, int key)
{
        int ret = 0;
    int snip = 0;
        ll_node_t *new_node;
        ll_node_t *prev,*curr;

        while(1) {

        find(ll,key,&prev,&curr);

        if(curr->key == key) break;

        new_node = ll_node_new(key);
        __sync_bool_compare_and_swap(&new_node->next,new_node->next,curr);
        snip = __sync_bool_compare_and_swap(&prev->next,curr,new_node);
        if(!snip) break;
        ret = 1;
        break;
    }

        return ret;
}


int ll_remove(ll_t *ll, int key)
{

        int ret=0;
    int snip=0;
        ll_node_t *prev,*curr;

        while(1) {

        find(ll,key,&prev,&curr);
```

```
        if(curr->key!=key) break;

        snip = __sync_bool_compare_and_swap(&curr,(long
long)curr&(~1),(long long)curr|1);
        if(!snip) continue;
        __sync_bool_compare_and_swap(&prev->next,curr,curr->next);
        ret=1;
        break;
    }

        return ret;
}
/**
 * Print a linked list.
 **/
void ll_print(ll_t *ll)
{
    ll_node_t *curr = ll->head;
    printf("LIST [");
    while (curr) {
        if (curr->key == INT_MAX)
            printf(" -> MAX");
        else
            printf(" -> %d", curr->key);
        curr = curr->next;
    }
    printf(" ]\n");
}
```