9ο Εξάμηνο, Ακαδημαϊκό Έτος 2021-2022

Συστήματα Παράλληλης Επεξεργασίας

Αναφορά 5ης Εργαστηριακής Άσκησης:
Θέματα Συγχρονισμού σε Σύγχρονα Πολυπύρηνα Συστήματα

parlab20

Ζάρα Στέλλα, 03117154
Λιάγκα Αικατερίνη, 03117208

2. Λογαριασμοί Τράπεζας

2.1 Ερωτήσεις - Ζητούμενα

1. Όχι, δεν υπάρχει ανάγκη συγχρονισμού για τα νήματα της συγκεκριμένης εφαρμογής. Κάθε νήμα της εφαρμογής εκτελεί πράξεις σε δικά του, συγκεκριμένα στοιχεία του πίνακα accounts, χωρίς να υπάρχει σε αυτά πρόσβαση από άλλο νήμα. Εφόσον, λοιπόν, δεν υπάρχει ταυτόχρονη πρόσβαση στα ίδια δεδομένα από διαφορετικά νήματα και οι πράξεις σε ένα στοιχείο του πίνακα γίνονται ανεξάρτητα με το τι συμβαίνει στον υπόλοιπο πίνακα, δεν χρειάζεται να γίνει συγχρονισμός των νημάτων.

2. Αφού τα νήματα λειτουργούν ανεξάρτητα μεταξύ τους περιμένουμε καθώς αυξάνουμε τον αριθμό των νημάτων να αυξάνεται και το throughput της εφαρμογής. Πιο συγκεκριμένα όσο περισσότερα νήματα έχουμε τόσα περισσότερα στοιχεία του πίνακα θα υπολογίζονται ταυτόχρονα και εφόσον κάθε νήμα έχει τον ίδιο αριθμό λειτουργιών, στον ίδιο χρόνο θα εκτελούνται περισσότερες λειτουργίες (operations), άρα θα αυξάνεται και το $throughput = \frac{\#operations}{time}$.

3. Παρακάτω φαίνεται το αρχείο *make_on_queue.sh* που υποβλήθηκε στο sandman για την μεταγλώττιση του δοσμένου προγράμματος *accounts.c*. Σημειώνεται ότι χρησιμοποιήθηκε το Makefile που δόθηκε έτοιμο.

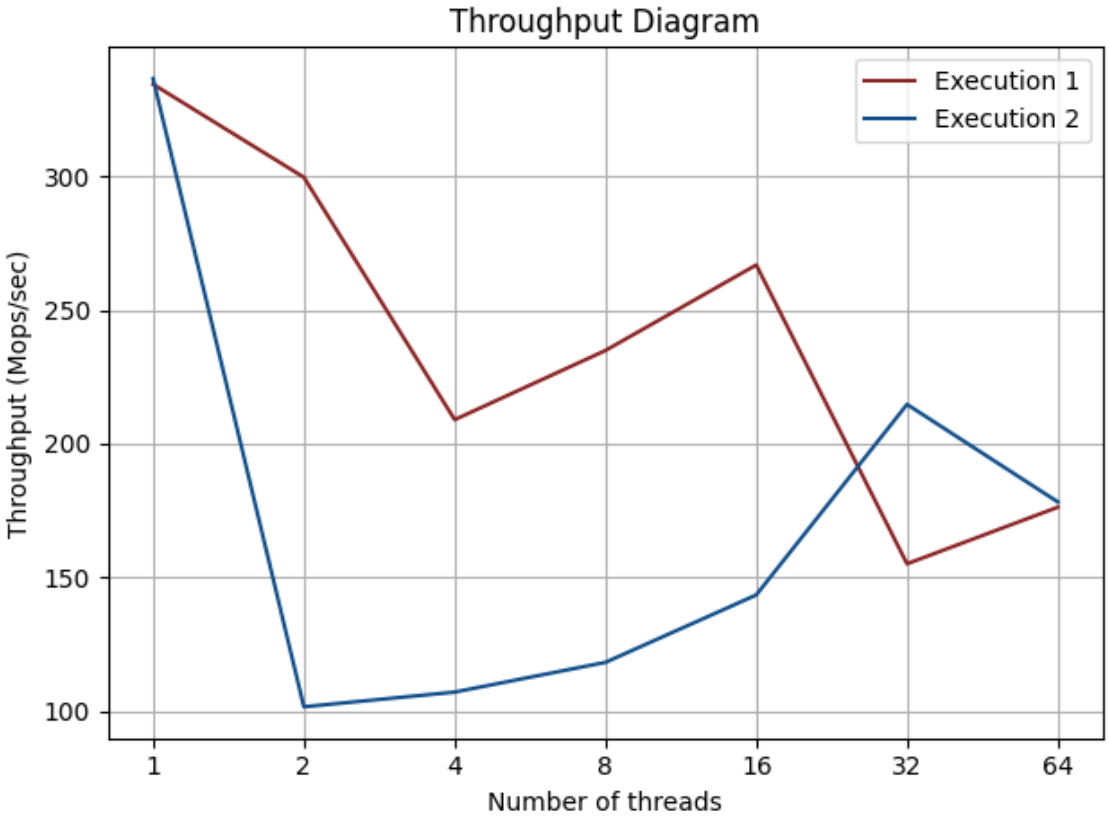
```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_accounts

## Output and error files
#PBS -o make_accounts.out
#PBS -e make_accounts.err

## How many machines should we get?

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab20/a5/z1
make
```

Για την εκτέλεση του accounts από το sandman χρησιμοποιήθηκε το αρχείο *run_on_queue.sh* που παρατίθεται παρακάτω.

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_accounts

## Output and error files
#PBS -o run_accounts.out
#PBS -e run_accounts.err

## How many machines should we get?

##How long should the job run for?
#PBS -l walltime=00:30:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab20/a5/z1

echo "Implementation 1"

MT_CONF=0 ./accounts

MT_CONF=0,1 ./accounts

MT_CONF=0,1,2,3 ./accounts

MT_CONF=0,1,2,3,4,5,6,7 ./accounts

MT_CONF=0,1,2,3,4,5,6,7,32,33,34,35,36,37,38,39 ./accounts

MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,32,33,34,35,36,37,38,3
9,40,41,42,43,44,45,46,47 ./accounts

MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,2
3,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,
46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63 ./accounts
```

```
echo "Implementation 2"

MT_CONF=0 ./accounts

MT_CONF=0,8 ./accounts

MT_CONF=0,8,16,24 ./accounts

MT_CONF=0,1,8,9,16,17,24,25 ./accounts

MT_CONF=0,1,2,3,8,9,10,11,16,17,18,19,24,25,26,27 ./accounts

MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31 ./accounts

MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63 ./accounts
```

Το διάγραμμα των throughputs που προκύπτει από την εκτέλεση του προγράμματος είναι το ακόλουθο



Παρατηρήσεις:

Παρατηρείται λοιπόν ότι και στις 2 εκτελέσεις αντί να αυξάνεται το throughput όσο αυξάνεται και ο αριθμος των νημάτων αυτό αντίθετα μειώνεται, κάτι το οποίο παρατηρείται πιο εντονα στην δεύτερη εκτέλεση που παρουσιάζει γενικά μικρότερα throughput.

Η διαφορά ανάμεσα στις 2 εκτελέσεις έγκειται στον τρόπο με τον οποίο έχουν επιλεγεί οι πυρήνες των νημάτων. Οι διαφορετικοί συνδυασμοί πυρήνων που επιλέγονται δίνουν διαφορετικές επιδόσεις εξαιτίας της αρχιτεκτονικής μνήμης του sandman. Το μηχάνημα sandman αποτελείται από 4 NUMA Nodes, καθένας από τους οποίους εμπεριέχει 8 πυρήνες, ενώ κάθε πυρήνας έχει δυνατότητα να τρέξει 2 threads. Οι πυρήνες που βρίσκονται στον ίδιο NUMA Node μοιράζονται μνήμη L3 cache με αποτέλεσμα οι πυρήνες αυτοί να έχουν πιο γρήγορη πρόσβαση σε δεδομένα που έχουν ήδη φορτωθεί στην cache (λόγω locality). Τα threads χωρίζονται στα NUMA Nodes έτσι:

socket0: 0-7, 32-39	socket1: 8-15, 40-47	socket2: 16-23, 48-55	socket3: 24-31, 56-63

Παρατηρείται, λοιπόν, ότι στην 1η εκτελεση επιλέγουμε πυρήνες που είναι κοντά ο ένας στον άλλον, εκμεταλλευόμενοι το memory locality της NUMA αρχιτεκτονικής του sandman. Από 1 έως 16 threads η 1η εκτέλεση κάνει χρήση ενός NUMA Node, σε αντίθεση με την 2η εκτέλεση που επιλέγει τους πιο απομακρυσμενους πυρήνες και καταλήγει να κάνει χρήση και των 4 NUMA Nodes, για αυτό και μέχρι τα 16 threads παρατηρείται ότι η 1η εκτέλεση δίνει σαφώς μεγαλύτερα throughput. Για 64 threads έχουμε την ίδια συμπεριφορά ανάμεσα στις δύο εκτελέσεις αφού γίνεται χρήση όλων των πυρήνων, ενώ για 32 threads παρατηρείται ότι το throughput της 2ης εκτέλεσης παίρνει την καλύτερη τιμή του (πέραν του 1 thread) και μάλιστα καταφέρνει να ξεπεράσει το throughput της 1ης εκτέλεσης για 32 threads.

Από την άλλη πλευρά πρέπει να αναφερθεί ότι για την 1η εκτέλεση από 8 threads και πάνω χρησιμοποιείται Hyperthreading, δηλαδή στους ίδιους πυρήνες χρησιμοποιούνται 2 threads. Ωστόσο το hyperthreading δεν είναι καθαρά παράλληλος άλλα concurrent τρόπος εκτέλεσης. Επομένως θα περίμενε κανείς για 8 εως και 32 threads η 2η εκτέλεση να έχει καλύτερη συμπεριφορά από την 1η. Ενδεχομένως για αυτό και στα 32 threads το throughput της 2ης να είναι καλύτερο από της 1ης, αφού όλα τα Threads της 1ης εκτέλεσης λειτουργούν concurrently και έχουμε πιο έντονο hyperthreading.

4. Η εφαρμογή δεν έχει την αναμενόμενη συμπεριφορά, αφού περιμέναμε με αύξηση των threads να αυξάνεται και το αντίστοιχο Throughput. Ωστόσο, παραπάνω παρατηρήθηκε η αντίθετη συμπεριφορά και ενώ εξηγήθηκε η σχέση μεταξύ των δύο γραφικών, η παραπάνω εξήγηση δεν δικαιολογεί γιατί και οι 2 εκτελέσεις δεν κλιμακώνουν και έχουν διαρκώς μειούμενο throughput.

Η διαρκής μείωση του throughput έχει να κάνει ξανά με την ιεραρχία μνήμης και την λειτουργία των caches. Πιο συγκεκριμένα κάθε thread φορτώνει στην cache του πυρήνα του, πέραν του στοιχείου που χρειάζεται, και άλλα γειτονικά στοιχεία του πίνακα accounts, διότι το cache block size είναι 64, ενώ εμείς χρειαζόμαστε 4 μόνο bytes για την αποθήκευση μιας θέσης του πίνακα accounts. Αν, λοιπόν, ένα από τα υπόλοιπα στοιχεία έχει τροποποιηθεί από κάποιο άλλο thread η cache line θα δηλωθεί ως dirty και λόγω

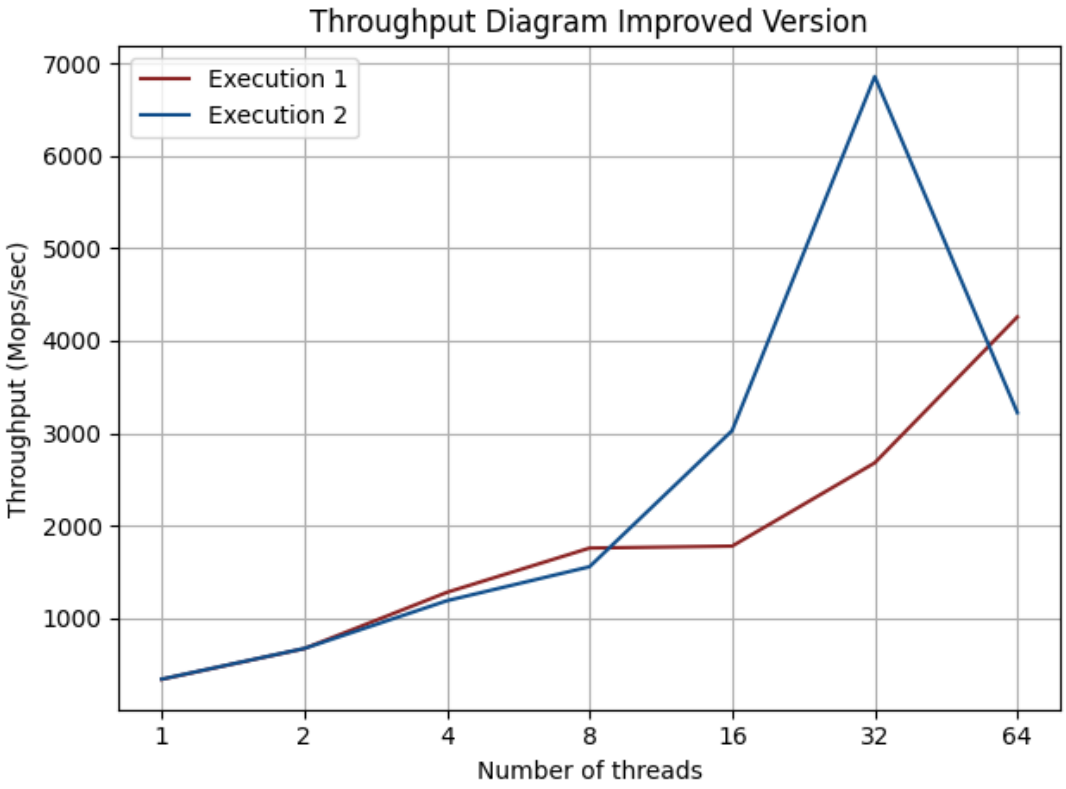
του cache coherence (MESI protocol) θα πρέπει να ενημερωθεί πρώτα η κύρια μνήμη και ύστερα η cache του πυρήνα του thread μου. Επομένως, όσο περισσότερα threads υπάρχουν, τόσες περισσότερες ταυτόχρονες τροποποιήσεις γίνονται στον πίνακα accounts, άρα και τόσες περισσότερες ενημερώσεις μνήμης θα πρέπει να γίνουν, για αυτό και παρατηρούμε ότι με την αύξηση των threads τελικά το throughput μειώνεται.
Αξίζει να σημειωθεί ότι πιθανόν αυτός είναι ακόμα ένας λόγος που παραπάνω στα 32 threads η 2η εκτέλεση έχει καλύτερο throughput από την 1η. Στην 1η εκτέλεση όλα τα threads βρίσκονται σε 2 μόνο NUMA Nodes με αποτέλεσμα προφανώς να υπάρχουν περισσότερα conflicts μεταξύ των cache lines και το overhead της ενημέρωσης της μνήμης να ξεπερνά το πλεονέκτημα που δίνει το memory locality στην 1η εκτέλεση.

Μια λύση στο πρόβλημα είναι να παίρνουμε από τον πίνακα accounts μόνο το στοιχείο που θέλουμε κάθε φορα να επεξεργαστούμε. Εφόσον γνωρίζουμε ότι κάθε στοιχείο τροποποιείται από ένα μόνο συγκεκριμένο thread δεν θα υπάρχουν πλέον dirty cache lines και πλέον το πρόγραμμα θα κλιμακώνει όπως περιμέναμε.
Για να επιτευχθεί ο παραπάνω στόχος προσθέτουμε σε κάθε στοιχείο του πίνακα accounts περιττά bytes ως padding, ώστε κάθε στοιχείο να έχει συνολικά μέγεθος 64 bytes, ίσο με το μέγεθος της cache line μας. Τροποποιείται, λοιπόν, το αρχείο accounts.c και συγκεκριμένα γίνεται η αλλαγή που φαίνεται παρακάτω στο struct του πίνακα accounts[ ].

```
struct {
        unsigned int value;
        /*The code we added is the padding*/
        char padding_accounts[64 - sizeof(unsigned int)];
} accounts[MAX_THREADS];
```

Σημειώνεται ότι χρησιμοποιήθηκε το ίδιο *make_on_queue.sh* και καινούριο αρχείο *run_on_queue_improved.sh,* στο οποίο η μόνη διαφορά με το *run_on_queue.sh* του προηγούμενου ερωτήματος είναι πως έχουμε διαφορετικά αρχεία error και out (run_accounts_improved.err και run_accounts_improved.out) για μην πανωγράψουμε τα αποτελέσματα του ερωτήματος 3.

Το διάγραμμα των throughput για τις 2 εκτελέσεις της βελτιωμένης έκδοσης του κώδικα φαίνεται παρακάτω. Όπως περιμέναμε αυτήν την φορά υπάρχει η αναμενόμενη κλιμάκωση. Μάλιστα η 2η εκτέλεση έχει όντως καλύτερη απόδοση από την 1η λόγω του hyperthreading, όπως αναφέρθηκε και στο 3ο ερώτημα.



3. Αμοιβαίος Αποκλεισμός - Κλειδώματα

3.1 Ερωτήσεις - Ζητούμενα

1. Τροποποιήθηκε το αρχείο *Makefile*, το οποίο παρατίθεται παρακάτω

```
CC = gcc
CFLAGS = -Wall -Wextra -pthread -O0

## Remove some warnings.
CFLAGS += -Wno-unused-parameter -Wno-unused-variable -Wno-unused-function

all:  linked_list_nosync  test_nosync  linked_list_pthread  test_pthread
linked_list_tas  test_tas  linked_list_ttas  test_ttas  linked_list_array
test_array linked_list_clh test_clh

## Additional source files
SRC_FILES = ../common/aff.c
```

```
linked_list_nosync: main.c nosync_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@
test_nosync: test.c nosync_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@

linked_list_pthread: main.c pthread_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@
test_pthread: test.c pthread_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@

linked_list_tas: main.c tas_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@
test_tas: test.c tas_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@

linked_list_ttas: main.c ttas_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@
test_ttas: test.c ttas_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@

linked_list_array: main.c array_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@
test_array: test.c array_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@

linked_list_clh: main.c clh_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@
test_clh: test.c clh_lock.c $(SRC_FILES)
	$(CC) $(CFLAGS) $^ -o $@



clean:
	rm -f linked_list_nosync test_nosync linked_list_pthread
test_pthread linked_list_tas test_tas linked_list_ttas test_ttas
linked_list_array test_array linked_list_clh test_clh
```

Το παρακάτω είναι το αρχείο *make_on_queue.sh* που υποβλήθηκε στο sandman για την μεταγλώττιση του δοσμένου κώδικα με όλα τα διαφορετικά locks που περιέχονται στα αρχεία <lock_type>_lock.c.

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make
```

```
## Output and error files
#PBS -o make.out
#PBS -e make.err

## How many machines should we get?

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab20/a5/z2
make
```

Το παρακάτω αρχείο είναι το *run_on_queue.sh* που χρησιμοποιήθηκε για την εκτέλεση του κώδικα με όλα τα διαφορετικά locks στο μηχάνημα sandman.

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N locks

## Output and error files
#PBS -o locks.out
#PBS -e locks.err

##How long should the job run for?
#PBS -l walltime=00:30:00

## Start
## Run make in the src folder

cd /home/parallel/parlab20/a5/z2

lockfiles=( "linked_list_nosync" "linked_list_pthread" "linked_list_tas"
"linked_list_ttas" "linked_list_array" "linked_list_clh" )
for size in 16 1024 8192
do
   echo "${size}"
   for i in "${lockfiles[@]}"
   do
      echo "${i}"

      MT_CONF=0 ./${i} ${size}
```

```
      MT_CONF=0,1 ./${i} ${size}

      MT_CONF=0,1,2,3 ./${i} ${size}

      MT_CONF=0,1,2,3,4,5,6,7 ./${i} ${size}

      MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ./${i} ${size}


      MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31 ./${i} ${size}


      MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,4
7,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63 ./${i} ${size}

   done
done
```

Παρακάτω παρουσιάζονται τα αρχεία pthread_lock.c, ttas_lock.c και array_lock.c που ζητήθηκε να τροποποιήσουμε.

- **pthread_lock.c**

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

struct lock_struct {
        pthread_spinlock_t pthread_lock;
};

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

        XMALLOC(lock, 1);
        pthread_spin_init(&lock->pthread_lock, PTHREAD_PROCESS_SHARED);

        return lock;
}
```

```
void lock_free(lock_t *lock)
{
        pthread_spin_destroy(&lock->pthread_lock);
        XFREE(lock);
}

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

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

- **ttas_lock.c**

```
#include "lock.h"
#include "../common/alloc.h"
#include <stdbool.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);
        lock->state = UNLOCKED;
        return lock;
}

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

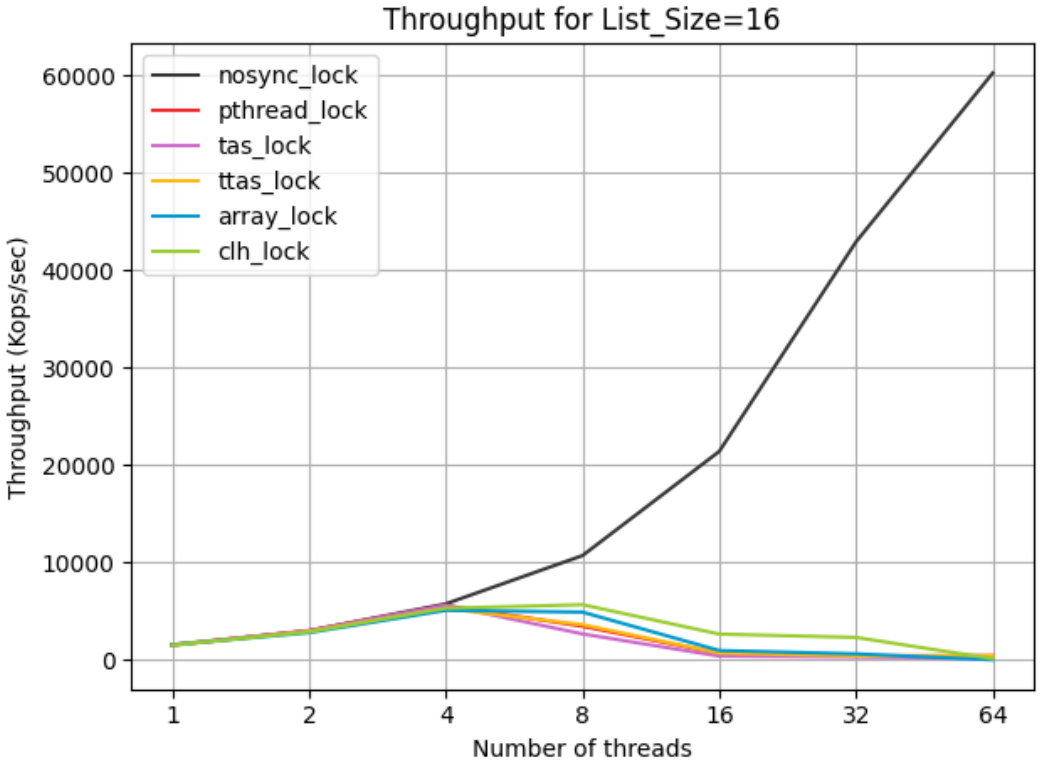
```
void lock_acquire(lock_t *lock)
{
        lock_t *l = lock;

        while(true) {
              while(l->state == LOCKED)
                    /* do nothing */ ;

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

        }
}

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

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

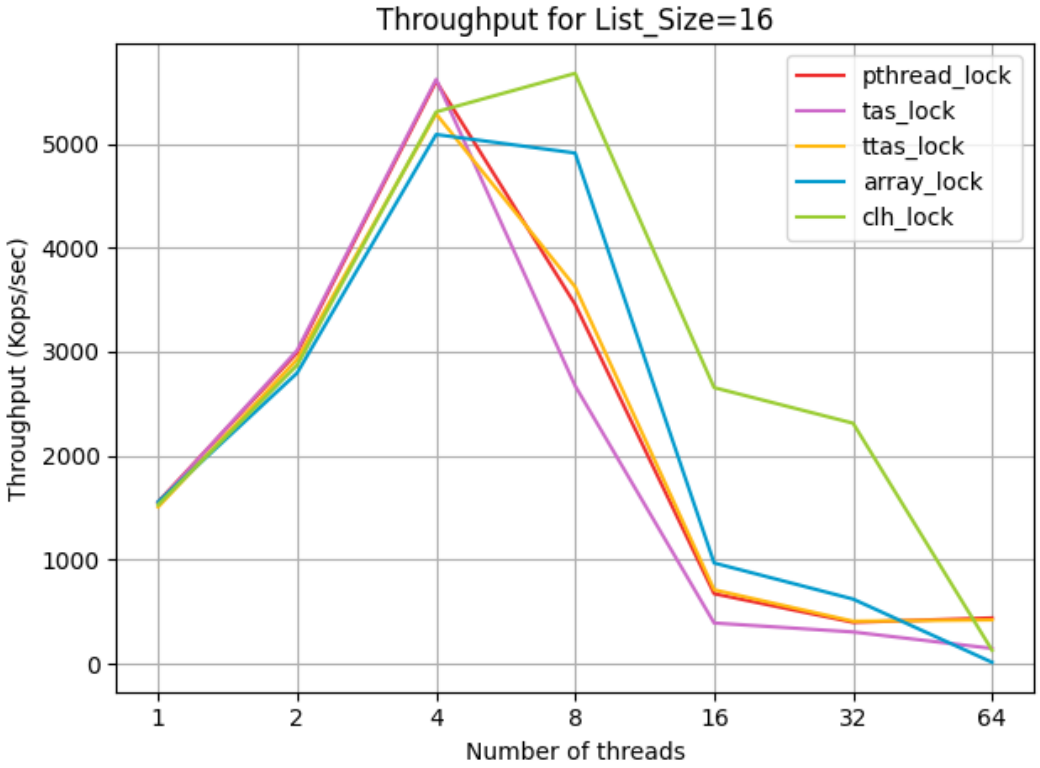
- **array_lock.c**

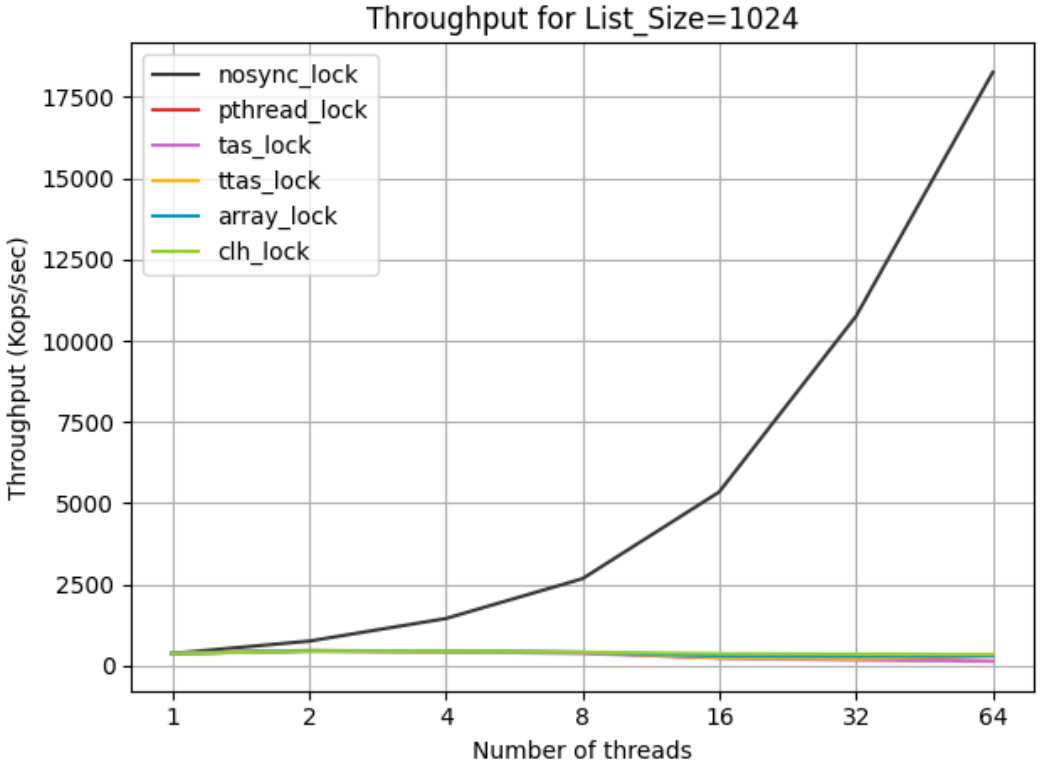
```
#include "lock.h"
#include "../common/alloc.h"
#include <stdbool.h>

struct lock_struct {
        int size;
        int tail;
        bool *flag;
};

__thread int mySlotIndex;

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

        XMALLOC(lock, 1);
        XMALLOC(lock->flag, nthreads);
```

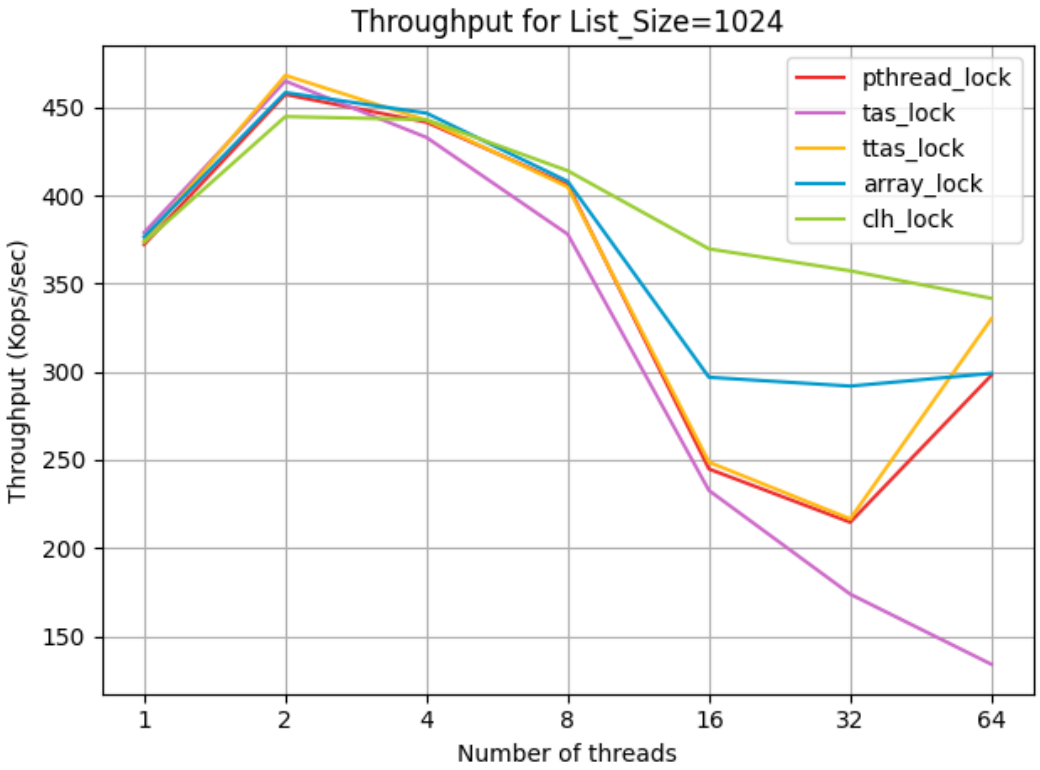
```
        lock->size = nthreads;
        lock->tail = 0;
        for (int i=1; i<nthreads; i++)
                lock->flag[i] = false;
        lock->flag[0] = true;

        return lock;
}

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

void lock_acquire(lock_t *lock)
{
        int slot = __sync_fetch_and_add(&lock->tail, 1) % (lock->size);
        mySlotIndex = slot;
        while(!lock->flag[slot])
                /* do nothing */ ;
}

void lock_release(lock_t *lock)
{
        int slot = mySlotIndex;
        lock->flag[slot] = false;
        lock->flag[(slot + 1) % (lock->size)] = true;
}
```

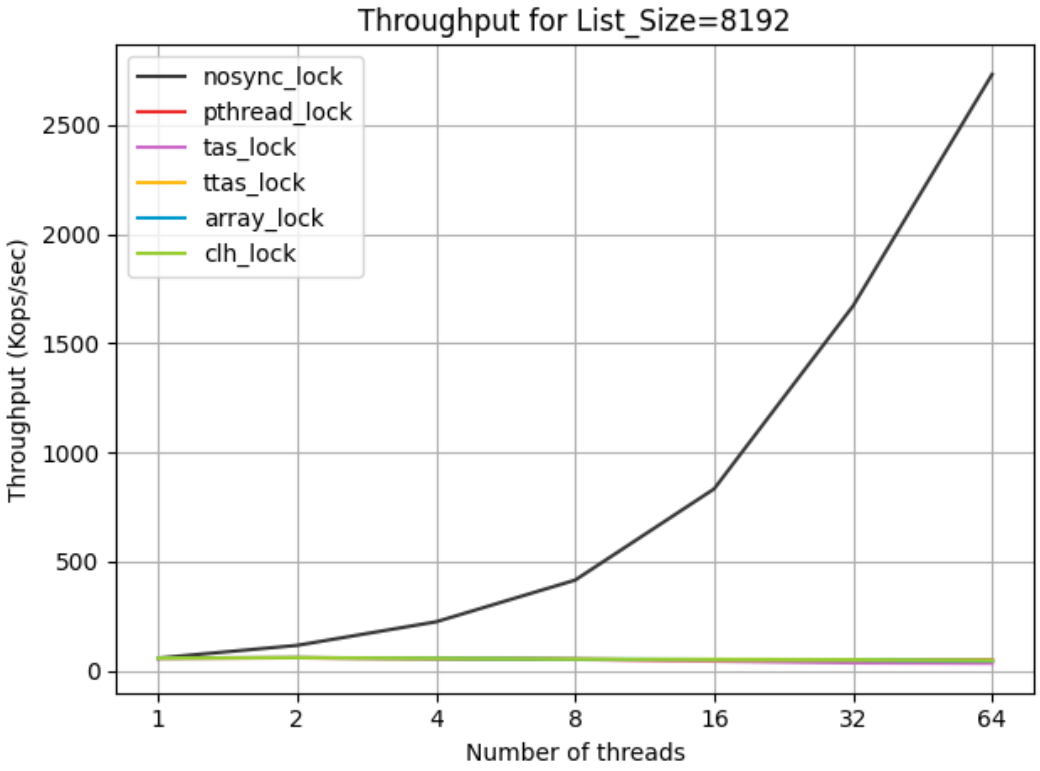
2. Ακολουθούν τα διαγράμματα που παρουσιάζουν τα throughput που προέκυψαν μετά την εκτέλεση της εφαρμογής για όλα τα υπάρχοντα διαφορετικά locks για μεγέθη λίστας 16, 1024 και 8192. Σημειώνεται ότι για κάθε μέγεθος λίστας δίνονται δύο διαγράμματα, ένα με το nosync_lock και ένα χωρίς αυτό, για να μπορεί να γίνει πιο εύκολα σύγκριση μεταξύ των υπόλοιπων locks, αφού το nosync_lock έχει πολύ μεγαλύτερες τιμές throughput και χάνεται η ακρίβεια των υπόλοιπων.

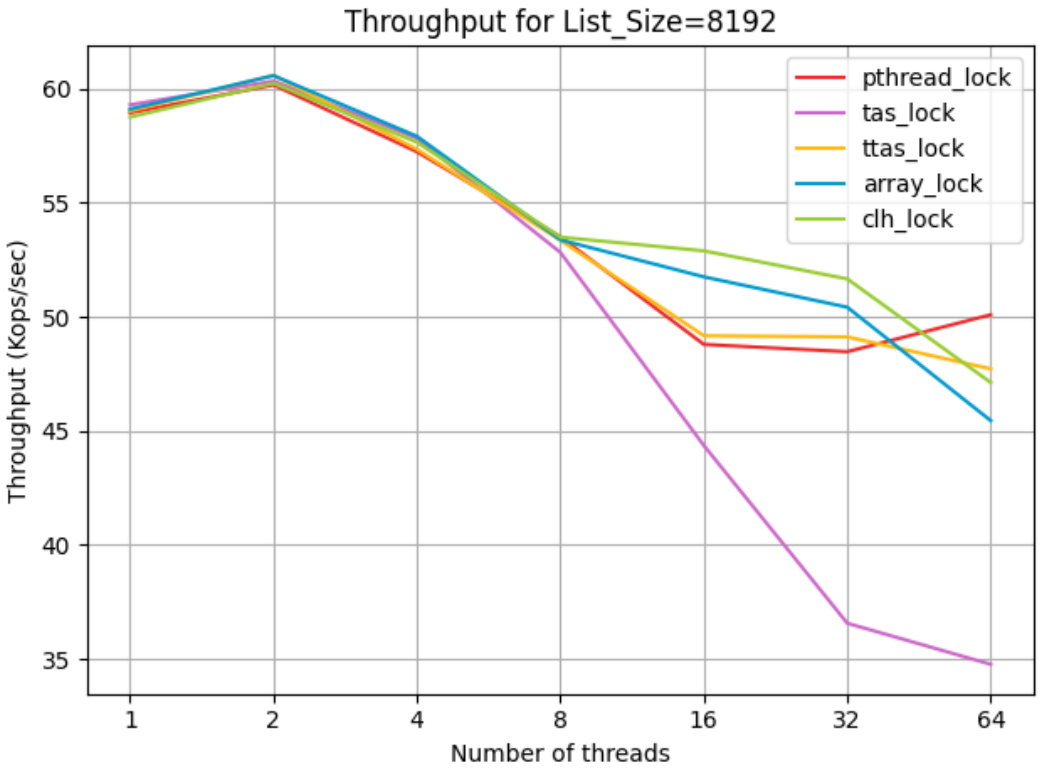

Throughput for List_Size=16
nosync_lock
pthread_lock
tas_lock
ttas_lock
array_lock
clh_lock
Throughput (Kops/sec)
60000
50000
40000
30000
20000
10000
0
1
2
4
8
16
32
64
Number of threads


Throughput for List_Size=16
pthread_lock
tas_lock
ttas_lock
array_lock
clh_lock
Throughput (Kops/sec)
5000
4000
3000
2000
1000
0
1
2
4
8
16
32
64
Number of threads


Throughput for List_Size=1024
nosync_lock
pthread_lock
tas_lock
ttas_lock
array_lock
clh_lock
Throughput (Kops/sec)
17500
15000
12500
10000
7500
5000
2500
0
1
2
4
8
16
32
64
Number of threads


Throughput for List_Size=1024
pthread_lock
tas_lock
ttas_lock
array_lock
clh_lock
Throughput (Kops/sec)
450
400
350
300
250
200
150
1
2
4
8
16
32
64
Number of threads


Throughput for List_Size=8192
nosync_lock
pthread_lock
tas_lock
ttas_lock
array_lock
clh_lock
Throughput (Kops/sec)
0
500
1000
1500
2000
2500
1
2
4
8
16
32
64
Number of threads


Throughput for List_Size=8192
pthread_lock
tas_lock
ttas_lock
array_lock
clh_lock
Throughput (Kops/sec)
35
40
45
50
55
60
1
2
4
8
16
32
64
Number of threads

Παρατηρήσεις:

Καταρχάς πρέπει να αναφέρουμε ότι στην υλοποίηση χωρίς κλειδώματα, δηλαδή με το **nosync_lock**, έχουμε ραγδαία αύξηση του throughput όσο αυξάνονται τα threads, σε αντίθεση με τα υπόλοιπα είδη κλειδώματος που παρατηρούμε ότι το throughput παραμένει σχετικά σταθερό και σε κάποιες περιπτώσεις ακόμα και μειώνεται με την αύξηση των threads. Μια υλοποίηση χωρίς κλειδώματα δεν είναι εφικτή, παρόλο που φαίνεται να δίνει καλύτερη επίδοση, διότι η εφαρμογή θα υπέκυπτε σε λάθη, αφού πολλά threads θα επιχειρούσαν να τροποποιήσουν ίδια κομμάτια της λίστας, με αποτέλεσμα να δημιουργηθούν ασυνέπειες και να προκύψουν errors. Ωστόσο δίνεται το throughput της υλοποίησης αυτής για να έχουμε αντίληψη του πόσο επιβαρύνουν τα κλειδώματα την επίδοση της εφαρμογής μας.

Όσον αφορά το κλειδωμα **tas_lock** φαίνεται να παρουσιάζει σε όλες τις περιπτώσεις το μικρότερο throughput σε σχέση με όλα τα υπόλοιπα κλειδώματα. Το γεγονός αυτό οφείλεται στις συνεχείς κλήσεις που κάνει το tas στην testAndSet, που δημιουργεί συμφόρηση στο bus (εξαιτίας του cache coherence) και καθυστερεί όλα τα υπόλοιπα threads. Μάλιστα, κάθε κλήση της testAndSet ακυρώνει τα αντίγραφα του lock για όλα τα υπόλοιπα threads, τα οποία αναγκάζονται να ζητήσουν ξανά από την μνήμη αντίγραφο του lock, με αποτέλεσμα να επιβαρύνεται περαιτέρω η απόδοση.

Το **ttas_lock** είναι σαφώς βελτιωμένη εκδοχή του tas, αφού παρατηρούμε ότι σε κάθε περίπτωση είναι αποδοτικότερο από το tas, όπως προβλέπει και η θεωρία, αφού εδώ η testAndSet καλείται μόνο αν το lock είναι ξεκλείδωτο, δημιουργώντας αισθητά μικρότερη συμφορηση του bus. Σε αντίθεση, λοιπόν, με το tas κλείδωμα που έχει συνεχώς μειούμενο throughput όσο αυξάνεται ο αριθμός των threads, το ttas lock φαίνεται να παρουσιάζει πιο σταθερή συμπεριφορα για πάνω από 16 Threads και ειδικότερα σε μεγαλύτερα μεγέθη λιστών. Παρατηρείται, μάλιστα, ότι για 64 threads προσεγγίζει και κάποιες φορές ακόμα και ξεπερνά την απόδοση καλύτερων κλειδωμάτων όπως το array_lock ακόμα και του βέλτιστου clh_lock.

Το **pthread_lock** είναι παρεμφερές σε απόδοση με το ttas, αφού οι τιμές των throughput για τα 2 κλειδώματα είναι πολύ κοντά. Το pthread είναι από τα πιο απλά κλειδώματα, αφού κάθε thread απλά περιμένει να ξεκλειδωθεί το εκάστοτε lock για να το χρησιμοποιήσει. Γενικά οι αποδόσεις του είναι αρκετά ικανοποιητικές, αν και γενικά ενδείκνυται για μικρό μέγεθος critical section, αφού αν το critical section είναι πολύ μεγάλο, θα είναι μεγάλος και ο χρόνος που θα περιμένουν τα Threads για να χρησιμοποιήσουν το lock, άρα η αποδοτικότητα της εφαρμογής μας θα μειώνεται.

Καλύτερο από το ttas και pthread είναι το **array_lock**, ιδιαίτερα για μεγαλύτερες τιμές threads (από 8 και πάνω, με εξαίρεση ίσως τα 64 threads), αφού το κάθε thread εξαρτάται μόνο από αυτό που εκτελέστηκε ακριβώς πριν από αυτό. Ωστόσο σε αυτό το κλείδωμα μπορεί να έχουμε εμφάνιση false sharing, όπως και στο tas, που δημιουργεί συμφόρηση στο bus λόγω του cache coherence πρωτοκόλλου και κατα συνέπεια μειώνει την απόδοση (throughput).

Αδιαμφισβήτητα καλύτερο από όλα τα κλειδώματα είναι το **clh_lock**, που καταφέρνει να απομονώσει τα threads, αποφεύγοντας συνωστισμό στο bus, για αυτό και έχει αισθητά καλύτερα throughput από όλα τα υπόλοιπα κλειδώματα, γεγονός που φαίνεται πιο αισθητά για μεγαλύτερο αριθμό threads (από 8 και πάνω).

4. Τακτικές συγχρονισμού για δομές δεδομένων

4.1 Ερωτήσεις - Ζητούμενα

1. Τροποποιήθηκε το αρχείο *Makefile*, το οποίο παρατίθεται παρακάτω ώστε να μεταγλωττιστεί το πρόγραμμα και με τα 4 είδη συγχρονισμών που ζητούνται.

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_sync

## Output and error files
#PBS -o make_sync.out
#PBS -e make_sync.err

## How many machines should we get?

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab20/a5/z3
make
parlab20@scirouter:~/a5/z3$ cat Makefile
CC = gcc
CFLAGS = -Wall -Wextra -pthread
## Remove some warnings.
CFLAGS += -Wno-unused-parameter -Wno-unused-variable -Wno-unused-function

all: serial fine_grain optimistic lazy non_blocking

## Additional source files
SRC_FILES = ../common/aff.c

serial: main.c ll_serial.c ${SRC_FILES}
        $(CC) $(CFLAGS) $^ -o $@

fine_grain: main.c ll_fgl.c ${SRC_FILES}
        $(CC) $(CFLAGS) $^ -o $@

optimistic: main.c ll_opt.c ${SRC_FILES}
        $(CC) $(CFLAGS) $^ -o $@
```

```
lazy: main.c ll_lazy.c ${SRC_FILES}
	$(CC) $(CFLAGS) $^ -o $@

non_blocking: main.c ll_nb.c ${SRC_FILES}
	$(CC) $(CFLAGS) $^ -o $@

clean:
	rm -f serial fine_grain optimistic lazy non_blocking
```

Το παρακάτω είναι το αρχείο *make_on_queue.sh* που υποβλήθηκε στο sandman για την μεταγλώττιση του δοσμένου κώδικα *main.c* και με τα 4 διαφορετικά είδη συγχρονισμού που ζητούνται

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_sync

## Output and error files
#PBS -o make_sync.out
#PBS -e make_sync.err

## How many machines should we get?

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

cd /home/parallel/parlab20/a5/z3
make
```

Χρησιμοποιήθηκαν 4 αρχεία για την εκτέλεση του κώδικα, ένα για κάθε διαφορετικό είδος συγχρονισμού. Τα αρχεία αυτά είναι τα: *run_on_queue_fgl.sh, run_on_queue_lazy.sh, run_on_queue_nb.sh, run_on_queue_opt.sh.* Παρακάτω παρουσιάζεται ένα από αυτά. Σημειώνεται ότι η μόνη διαφορά των υπολοιπων είναι μόνη είναι τα διαφορετικά .out και .err, καθώς αλλάζει και το όνομα του εκτελέσιμου (όπως το έχουμε ορίσει στο Makefile).

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_sync_fgl
## Output and error files
#PBS -o run_sync_fgl.out
```

```
#PBS -e run_sync_fgl.err

##How long should the job run for?
#PBS -l walltime=01:00:00

## Start
## Run make in the src folder

cd /home/parallel/parlab20/a5/z3

operation_stats=( "80 10 10" "20 40 40" )

for size in 1024 8192
do
   echo "${size}"
   for op in "${operation_stats[@]}"
   do
         MT_CONF=0 ./fine_grain ${size} ${op}

         MT_CONF=0,1 ./fine_grain ${size} ${op}

         MT_CONF=0,1,2,3 ./fine_grain ${size} ${op}

         MT_CONF=0,1,2,3,4,5,6,7 ./fine_grain ${size} ${op}

       MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 ./fine_grain ${size}
${op}



MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31 ./fine_grain ${size} ${op}



MT_CONF=0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,4
7,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63   ./fine_grain   ${size}
${op}

   done
done
```

Παρακάτω παρατίθενται τα αρχεία των τακτικών συγχρονισμών που τροποποιήθηκαν. Συγκεκριμένα σε όλα τα παρακάτω αρχεία έχουμε συμπληρώσει εξ'ολοκλήρου τις συναρτήσεις ll_contains(), ll_add() και ll_remove(), ενώ περαιτέρω τροποποιήσεις που έγιναν αναφέρονται ατομικά σε κάθε αρχείο. Σημειώνεται και οι 3 παραπάνω, καθώς και όλες οι άλλες συναρτήσεις που συμπληρώθηκαν από εμάς, ακολουθούν σε κάθε περίπτωση τους αντίστοιχους αλγόριθμους που δίνονται στις διαφάνειες του μαθήματος.

- **ll_fgl.c**
  Πέραν των 3 συναρτήσεων που προαναφέραμε γίνονται οι εξής τροποποιήσεις κώδικα που μας δόθηκε σε αυτό το αρχείο:
  1. Στο struct ll_node συμπληρώθηκε το πεδίο του pthread_spinlock_t lock;
  2. Στην συνάρτηση ll_node_new() αρχικοποιείται το lock με την pthread_spin_init(&ret->lock, PTHREAD_PROCESS_SHARED);
  3. Στην συνάρτηση ll_node_free() καταστρέφουμε το lock που αρχικοποιήσαμε με χρήση της pthread_spin_destroy(&ll_node->lock);

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

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        pthread_spinlock_t lock;
} 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->lock, PTHREAD_PROCESS_SHARED);

        return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        pthread_spin_destroy(&ll_node->lock);
        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 *prev, *curr;

        prev = ll->head;
        pthread_spin_lock(&prev->lock);
        curr = prev->next;
        pthread_spin_lock(&curr->lock);

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

        ret = (curr->key == key);
        pthread_spin_unlock(&curr->lock);
        pthread_spin_unlock(&prev->lock);
        return ret;
}

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

        prev = ll->head;
        pthread_spin_lock(&prev->lock);
        curr = prev->next;
        pthread_spin_lock(&curr->lock);

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

        if (curr->key != key) {
                new_node = ll_node_new(key);
                new_node->next = curr;
```

```
                prev->next = new_node;
                pthread_spin_unlock(&curr->lock);
                pthread_spin_unlock(&prev->lock);
                return 1;
        }

        pthread_spin_unlock(&curr->lock);
        pthread_spin_unlock(&prev->lock);
        return 0;
}

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

        prev = ll->head;
        pthread_spin_lock(&prev->lock);
        curr = prev->next;
        pthread_spin_lock(&curr->lock);

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

        if (curr->key == key) {
                prev->next = curr->next;
                pthread_spin_unlock(&curr->lock);
                pthread_spin_unlock(&prev->lock);
                ll_node_free(curr);
                return 1;
        }

        pthread_spin_unlock(&curr->lock);
        pthread_spin_unlock(&prev->lock);
        return 0;
}

/**
 * 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**
  Κάνουμε τις ίδιες τροποποιήσεις που κάναμε και στο αρχείο ll_fgl.c και επιπλέον:
  4. Ορίζουμε μια συνάρτηση ll_validate(ll_t *ll, ll_node_t *prev, ll_node_t *curr), η οποία ελέγχει, με διάσχιση της λίστας, αν όντως η δομή είναι συνεπής, δηλαδή αν τα prev και curr είναι ακόμα μέρος της δομής και αν αυτά είναι ακόμα διαδοχικά.

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

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        pthread_spinlock_t lock;
} 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->lock, PTHREAD_PROCESS_SHARED);

        return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        pthread_spin_destroy(&ll_node->lock);
        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_validate(ll_t *ll, ll_node_t *prev, ll_node_t *curr)
{
        ll_node_t *node;
        node = ll->head;

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

        return 0;
}

int ll_contains(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 (curr->key == key)
                                break;
                        prev = curr;
                        curr = curr->next;
                }
                pthread_spin_lock(&prev->lock);
                pthread_spin_lock(&curr->lock);

                if (ll_validate(ll, prev, curr)) {
                        ret = (curr->key == key);
                        pthread_spin_unlock(&prev->lock);
                        pthread_spin_unlock(&curr->lock);
                        return ret;
                }

                pthread_spin_unlock(&prev->lock);
                pthread_spin_unlock(&curr->lock);
        }
}
```

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

        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->lock);
                pthread_spin_lock(&curr->lock);

                if (ll_validate(ll, prev, curr)) {
                        if (curr->key != key) {
                                new_node = ll_node_new(key);
                                new_node->next = curr;
                                prev->next = new_node;
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 1;
                        }
                        else {
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 0;
                        }
                }

                pthread_spin_unlock(&prev->lock);
                pthread_spin_unlock(&curr->lock);
        }
}

int ll_remove(ll_t *ll, int key)
{
```

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

                if (ll_validate(ll, prev, curr)) {
                        if (curr->key == key) {
                                prev->next = curr->next;
                                pthread_spin_unlock(&curr->lock);
                                pthread_spin_unlock(&prev->lock);
                                return 1;
                        }
                        else {
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 0;
                        }
                }

                pthread_spin_unlock(&prev->lock);
                pthread_spin_unlock(&curr->lock);
        }
}

/**
 * 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**
  Κάνουμε τις ίδιες τροποποιήσεις που κάναμε και στο αρχείο ll_fgl.c και επιπλέον:
  4. Προσθέτουμε στο struct ll_node το πεδίο int marked που είναι μια δομή boolean που σηματοδοτεί λογικά αν ο κόμβος βρίσκεται στη λίστα ή έχει διαγραφεί.
  5. Αρχικοποιούμε στην συνάρτηση ll_node_new() την ret->marked = 0;
  6. Ορίζουμε μια διαφορετική συνάρτηση ll_validate(ll_node_t *prev, ll_node_t *curr), η οποία ελέγχει λογικά αν όντως η δομή είναι συνεπής, δηλαδή αν τα prev και curr είναι ακόμα μέρος της δομής και αν αυτά είναι ακόμα διαδοχικά.

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

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        int marked;
        struct ll_node *next;
        pthread_spinlock_t lock;
} 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->marked = 0;
        ret->next = NULL;
        pthread_spin_init(&ret->lock, PTHREAD_PROCESS_SHARED);

        return ret;
}

/**
 * Free a linked list node.
 **/
static void ll_node_free(ll_node_t *ll_node)
{
        pthread_spin_destroy(&ll_node->lock);
        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_validate(ll_node_t *prev, ll_node_t *curr)
{
        return ( !(prev->marked) && !(curr->marked) && prev->next == curr
);
}

int ll_contains(ll_t *ll, int key)
{
        ll_node_t *curr;

        curr = ll->head;

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

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

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

        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->lock);
                pthread_spin_lock(&curr->lock);

                if (ll_validate(prev, curr)) {
                        if (curr->key != key) {
                                new_node = ll_node_new(key);
                                new_node->next = curr;
                                prev->next = new_node;
```

```
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 1;
                        }
                        else {
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 0;
                        }
                }

                pthread_spin_unlock(&prev->lock);
                pthread_spin_unlock(&curr->lock);
        }

}

int ll_remove(ll_t *ll, int key)
{
        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->lock);
                pthread_spin_lock(&curr->lock);

                if (ll_validate(prev, curr)) {
                        if (curr->key == key) {
                                curr->marked = 1;
                                prev->next = curr->next;
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 1;
                        }
                        else {
```

```
                                pthread_spin_unlock(&prev->lock);
                                pthread_spin_unlock(&curr->lock);
                                return 0;
                        }
                }

                pthread_spin_unlock(&prev->lock);
                pthread_spin_unlock(&curr->lock);
        }
}

/**
 * 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**
  Όπως αναφέρεται και στις διαφάνειες σε αυτή την τακτική συγχρονισμού ενσωματώνουμε την boolean marked μέσα στην διεύθυνση του *next του κάθε ll_node. Αναλυτικότερα εκμεταλλευόμαστε το γεγονός ότι, λόγω alignment, το τελευταίο bit μιας διεύθυνσης είναι πάντα 0, οπότε ενσωματώνουμε στο LSB της διεύθυνσης του κόμβου next την boolean τιμή του marked, η οποία όταν είναι 0 ο κάμβος υπάρχει, ενώ όταν είναι 1 ο κόμβος έχει διαγραφεί λογικά. Με αυτόν τον τρόπο αντιμετωπίζουμε το next και το marked σαν μια ατομική μονάδα, την οποία διαχειριζόμαστε με τις συναρτήσεις getReference(), get() και την compareAndSet(). Συγκεκριμένα προστίθενται:
  1. Η συνάρτηση get(ll_node_t *node, bool *marked) που αποθηκεύει την τιμή του marked bit της παραμέτρου node στην παράμετρο marked και επιστρέφει ως αποτέλεσμα την διεύθυνση του node (πεδίο next χωρις το τελευταίο bit, λογικό ΚΑΙ με το αντίστροφο του δυαδικού 1).

2. Η συνάρτηση getReference(ll_node_t *node) που απλώς επιστρέφει ως αποτέλεσμα την διεύθυνση του node.
3. Η συνάρτηση compareAndSet(ll_node_t *node, ll_node_t *expectedRef, ll_node_t *updateRef, bool expectedMark, bool updateMark) που με την βοήθεια της ατομικής συνάρτησης __sync_bool_compare_and_swap ελέγχει αν ο node έχει τις expected τιμές και στην συνέχεια ενημερώνει την τιμή και στα δύο πεδία marked και next με τις τιμές των update παραμέτρων. Επιστρέφει True όταν επιτύχει όλα τα παραπάνω.
4. Ορίζουμε καινούριο struct ll_window {ll_node_t*prev; ll_node_t*curr;} ll_window_t; το οποίο είναι βοηθητικό
5. Ορίζουμε την συνάρτηση ll_window_new(ll_node_t *myPrev, ll_node_t *myCurr) που θα χρησιμοποιηθεί για να αρχικοποιηθεί ένα καινούριο ll_window_t.
6. Στην συνέχεια φτιάχνουμε με βάση τον αλγόριθμο των διαφανειών την συνάρτηση ll_find(ll_t *ll, int key) που επιστρέφει ένα δείκτη στο ακριβώς μικρότερο στοιχείο και ένα δείκτη στο αμέσως μεγαλύτερο ή ίσο στοιχείο του key, ενώ αν συναντήσει στοιχεία που έχουν σβηστεί λογικά τα σβήνει και φυσικά.

```c
#include <stdio.h>
#include <stdlib.h> /* rand() */
#include <limits.h>
#include <stdbool.h>

#include "../common/alloc.h"
#include "ll.h"

typedef struct ll_node {
        int key;
        struct ll_node *next;
        /* other fields here? */
} ll_node_t;

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

typedef struct ll_window {
        ll_node_t *prev;
        ll_node_t *curr;
} ll_window_t;

ll_window_t *ll_window_new(ll_node_t *myPrev, ll_node_t *myCurr)
{
        ll_window_t *ret;
```

```
        XMALLOC(ret, 1);
        ret->prev = myPrev;
        ret->curr = myCurr;

        return ret;
}

/**
 * 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? */

        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);
}

ll_node_t *get(ll_node_t *node, bool *marked)
{
        ll_node_t *res;

        *marked = (bool) ( (unsigned long long)node & 1 );
        res = (ll_node_t *) ( (unsigned long long)node & (~1) );

        return res;
}

ll_node_t *getReference(ll_node_t *node)
{
        ll_node_t *res;

        res = (ll_node_t *) ( (unsigned long long)node & (~1) );

        return res;
}

bool  compareAndSet(ll_node_t  *node,  ll_node_t  *expectedRef,  ll_node_t
*updateRef, bool expectedMark, bool updateMark)
{
        bool res;
        ll_node_t *expectedNode, *updateNode;

        expectedNode=(ll_node_t *)((unsigned long long)expectedRef | expectedMark);
        updateNode = (ll_node_t *) ((unsigned long long)updateRef | updateMark);

        res = __sync_bool_compare_and_swap(&node, expectedNode, updateNode);
```

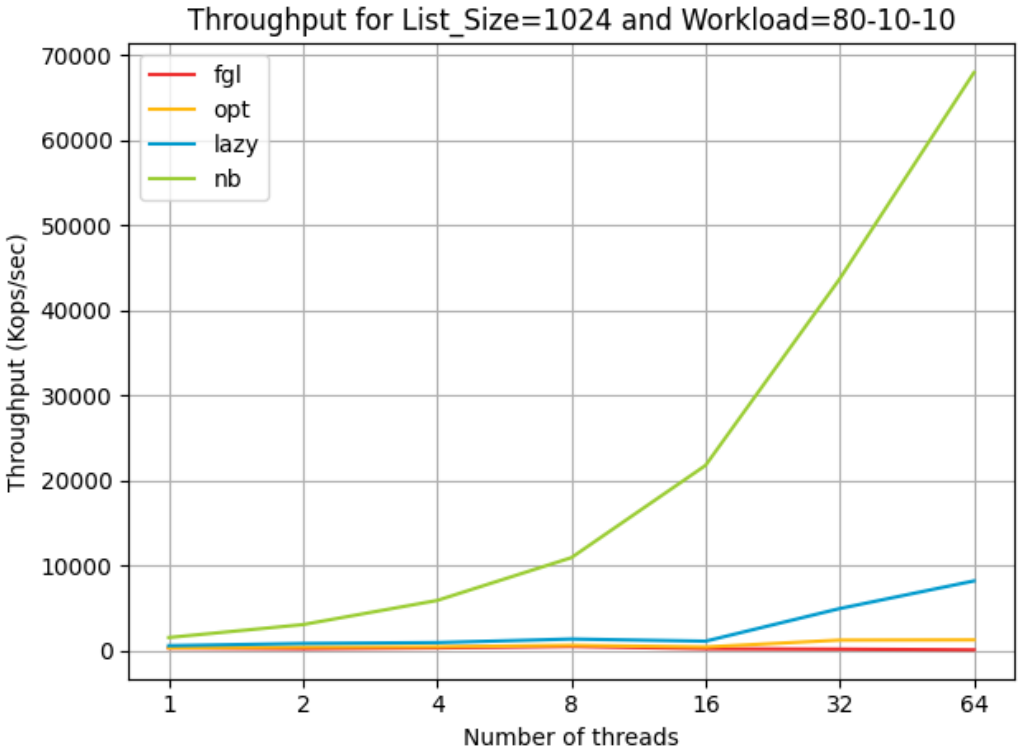
```
        return res;
}

ll_window_t *ll_find(ll_t *ll, int key)
{
        ll_node_t *prev, *curr, *succ;
        bool marked, snip;
        ll_window_t *res;

        prev = NULL;
        curr = NULL;
        succ = NULL;
        marked = false;

retry:
        while (true) {
                prev = ll->head;
                curr = getReference(prev->next);

                while (true) {
                        succ = get(curr->next, &marked);

                        while (marked) {
                                snip = compareAndSet(prev->next, curr, succ, false,
false);
                             if (!snip)
                                     goto retry;
                             curr = succ;
                             succ = get(curr->next, &marked);
                        }

                        if (curr->key >= key) {
                             res = ll_window_new(prev, curr);
                             return res;
                        }
                        prev = curr;
                        curr = succ;
                }
        }
}


int ll_contains(ll_t *ll, int key)
```

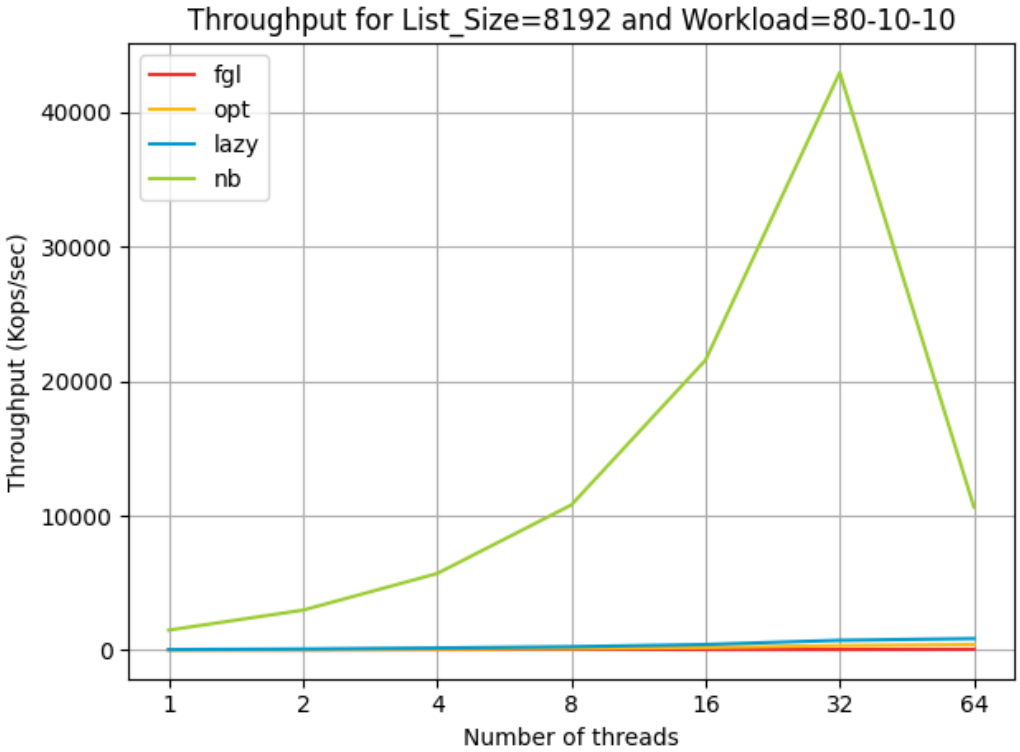
```
{
        bool marked;
        ll_node_t *curr;

        curr = ll->head;
        while (curr->key < key) {
                curr = getReference(curr->next);
        }
        get(curr->next, &marked);

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

int ll_add(ll_t *ll, int key)
{
        bool splice;
        ll_window_t *myWindow;
        ll_node_t *prev, *curr, *new_node;

        while (true) {
                myWindow = ll_find(ll, key);
                prev = myWindow->prev;
                curr = myWindow->curr;
                XFREE(myWindow);

                if (curr->key == key)
                        return false;
                else {
                        new_node = ll_node_new(key);
                        new_node->next = curr;
                         if ( compareAndSet(prev->next, curr, new_node, false, false) )
                                return true;
                }
        }
}


int ll_remove(ll_t *ll, int key)
{
        bool snip;
        ll_window_t *myWindow;
        ll_node_t *prev, *curr, *succ;

        while (true) {
```

```
                myWindow = ll_find(ll, key);
                prev = myWindow->prev;
                curr = myWindow->curr;
                XFREE(myWindow);

                if (curr->key != key)
                        return false;
                else {
                        succ = getReference(curr->next);
                        snip = compareAndSet(curr->next, succ, succ, false, true);
                        if (!snip)
                                continue;
                        compareAndSet(prev->next, curr, succ, false, false);
                        return true;
                }
        }
}

/**
 * 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");
}
```

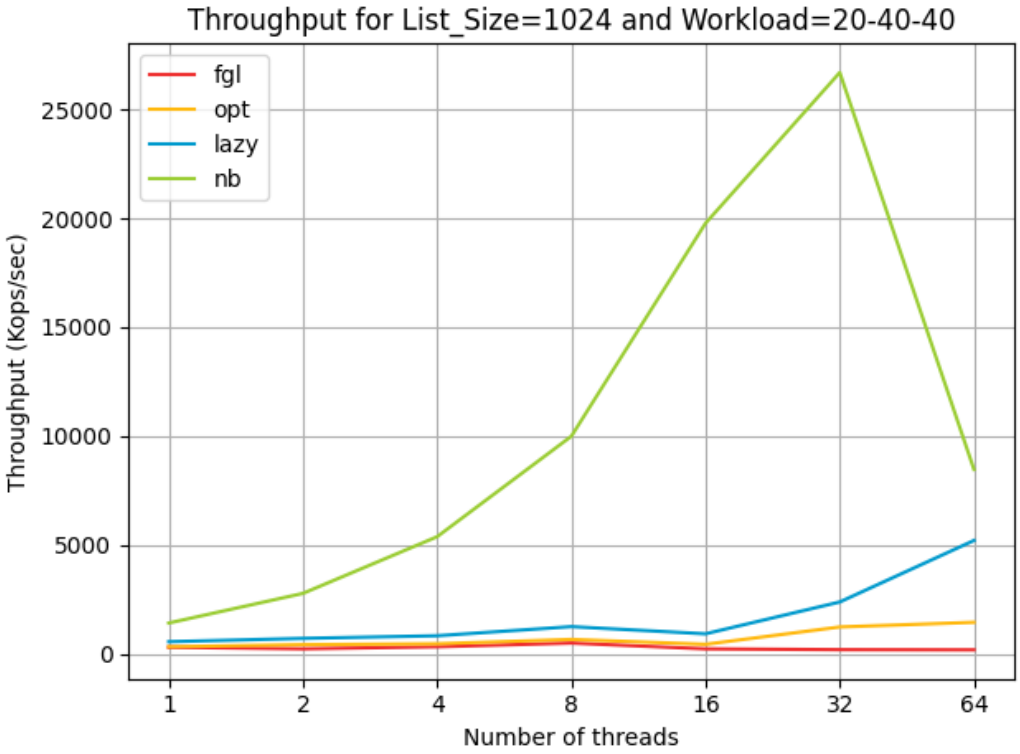
2. Τα διαγράμματα των Throughput που πήραμε ως αποτελέσματα μετα τις εκτελέσεις της εφαρμογής για τα διαφορετικά workloads παρουσιάζονται στις επόμενες σελίδες.

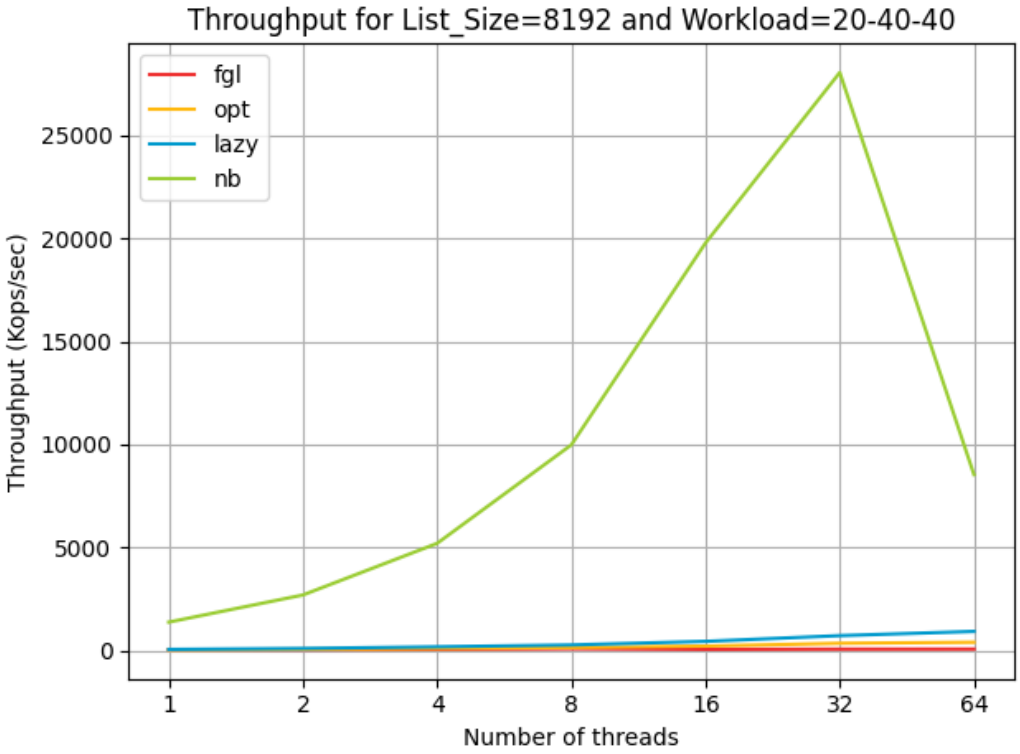
<u>Συνδυασμός λειτουργιών 80-10-10</u>
(80% contains - 10% add - 10% remove)





Συνδυασμός λειτουργιών 20-40-40
(20% contains - 40% add - 40% remove)





Παρατηρήσεις:

Αρχικά παρατηρεί κανείς ότι σε κάθε περίπτωση μεγαλύτερο Throughput εχει κατα πολυ η τακτική non_blocking, ακολουθεί η lazy, στην συνέχεια η optimistic και τέλος η fine_grain locking. Επομένως η κατάταξη των τακτικών φαίνεται να ικανοποιεί αυτό που αναμέναμε από την θεωρία.

Σε γενικές γραμμές όσο αυξάνεται ο αριθμός των threads τόσο εμφανίζονται περισσότερες λειτουργίες και καταστάσεις συναγωνισμού. Επομένως, όπως και όντως παρατηρείται, για αύξηση των threads έχουμε μεγαλύτερη διαφοροποίηση μεταξύ των throughput όλων των τακτικών συγχρονισμού.

Όσον αφορά ατομικά τώρα την κάθε τακτική συγχρονισμού παρατηρούμε αρχικά την **fine_grain locking**, η οποία παραμένει σχεδόν σταθερή για όλα τα μεγέθη λίστας, χωρίς να κλιμακώνει. Αυτό οφείλεται στο γεγονός ότι χρησιμοποιούνται πολλά locks σε αρκετά σημεία και στις 3 συναρτήσεις contains, add, remove (hand-over-locking), με αποτέλεσμα τα threads να βρίσκονται συνεχώς σε μια νεκρή κατάσταση αναμονής και να μην μπορούν να εκτελέσουν λειτουργίες ταυτόχρονα σε διαφορετικά σημεία.

Η αμέσως καλύτερη υλοποίηση είναι η **optimistic**, στην οποία χρησιμοποιούνται και πάλι κλειδώματα σε όλες τις συναρτήσεις contains, add, remove, αλλά αυτά έχουν μειωθεί σημαντικά, αφού διατρέχουμε την λίστα χωρίς κλειδώματα και ελέγχουμε την ορθότητα της με την βοηθητική συνάρτηση validate. Για αυτό και παρατηρούμε καλύτερη κλιμάκωση ειδικά για μεγαλύτερους αριθμούς threads.

Ακόμα καλύτερα αποτελέσματα δίνει η **lazy** τακτική συγχρονισμού όπου καταφέρνουμε με την χρήση της boolean τιμής marked να εξαλείψουμε εντελώς τα locks από την μέθοδο contains.

Και στις 3 παραπάνω μεθόδους παρατηρείται ότι με την αύξηση του μεγέθους της λίστας το throughput μειώνεται, γεγονός λογικό αφού θα πρέπει να γίνει διάσχιση μεγαλύτερης λίστας κάθε φορα.

Ωστόσο το μέγεθος της λίστας δεν φαίνεται να επηρεάζει το throughput της τακτικής συγχρονισμού **non_blocking** η οποία ξεπερνά κατά πολύ σε απόδοση τις 3 προαναφερθείσες τακτικές και παρατηρούμε ότι επηρεάζεται κυρίως από τον συνδυασμό των λειτουργιών. Συγκεκριμένα η τακτική αυτή εφαρμόζει αναζήτηση (contains) με Wait-Free τρόπο, επομένως και αναμένεται για περισσότερες αναζητήσεις να έχουμε καλύτερα throughput, αφού η contains δεν περιορίζει με κάποιο τρόπο την απόδοση. Από την άλλη, στις συναρτήσεις add και remove, ενώ αυτές είναι lock-free, γίνονται reties σε περίπτωση που αυτές κάνουν προσπάθεια να τροποποιήσουν κόμβο που έχει διαγραφεί λογικά αλλά όχι φυσικά (marked_bit=1). Επομένως, ειδικά για μεγάλο αριθμό threads, που πολλά νήματα τροποποιούν ταυτόχρονα την λίστα, είναι μεγαλύτερη η πιθανότητα να γίνουν πολλαπλά retries που θα μειώσουν την απόδοση, άρα και το throughput, αν το ποσοστό των add και remove είναι μεγάλο. Για αυτόν ακριβώς τον λόγο παρατηρούμε ότι για Workload 80% αναζητήσεις, 10% προσθήκες και 10% διαγραφές έχουμε τόσο μεγαλύτερες τιμές throughput (σχεδόν διπλάσιες), όσο και καλύτερη κλιμάκωση για μεγάλο αριθμό threads σε σχέση με την εκτέλεση για Workload 20% αναζητήσεις, 40% προσθήκες και 40% διαγραφές στην τακτική non_blocking.

Από την άλλη, οι 3 πρώτες τακτικές συγχρονισμού (fine_grain locking, optimistic και lazy) παρουσιάζουν παρόμοια συμπεριφορά κλιμάκωσης για διαφορετικό συνδυασμό λειτουργιών (workload). Η μόνη διαφορά που παρουσιάζεται με την αλλαγή workload είναι ότι για 80% αναζητήσεις 10% εισαγωγές και 10% διαγραφές έχουμε και εδώ μεγαλύτερες τιμές throughput σε σχέση με την εκτέλεση για workload 20% αναζητήσεις, 40% προσθήκες και 40% διαγραφές, γεγονός που εξηγήθηκε αναλυτικά παραπάνω.