



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ  
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

## Συστήματα Παράλληλης Επεξεργασίας

**Αναφορά 1ης Εργαστηριακής Άσκησης**

**Ραπτόπουλος Πέτρος (el19145)  
Κόγιος Δημήτριος (el19220)  
Καπετανάκης Αναστάσιος (el19048)**

1. Συνδεόμαστε στο cluster του εργαστηρίου και λύνουμε πιθανά προβλήματα σύνδεσης:

```
parlab10@scirouter:~  
(base) petrosrapto@petrosraptoAssistant:-$ ssh parlab10@orion.cslab.ece.ntua.gr  
The authenticity of host 'orion.cslab.ece.ntua.gr (147.102.3.236)' can't be established.  
ED25519 key fingerprint is SHA256:bA9q2rUkw1LvYf8uC1uaZTgL8DjYW+NzvP7zrrsbdI.  
This key is not known by any other names  
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes  
Warning: Permanently added 'orion.cslab.ece.ntua.gr' (ED25519) to the list of known hosts.  
parlab10@orion.cslab.ece.ntua.gr's password:  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
Last login: Wed Oct 11 12:19:49 2023 from 37.6.86.151  
parlab10@orion:~$ ssh scirouter.cslab.ece.ntua.gr  
Linux scirouter 2.6.26-2-amd64 #1 SMP Sun Mar 4 21:48:06 UTC 2012 x86_64  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
  
=====  
||   \_\_\_\_O_____ /____/ | /____/ | /____/  
| / \_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/  
| / \_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/  
| \_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/  
| \_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/\_\_\_\_/  
=====  
If you have any problems please contact admins<at>cslab.ece.ntua.gr  
=====  
Last login: Wed Oct 11 12:22:00 2023 from orion.cslab.ece.ntua.gr  
parlab10@scirouter:~$
```

2. Γινόμαστε οικείοι με την μεταγλώττιση και την υποβολή jobs στα cluster queues. Για παράδειγμα μεταγλωτίζουμε και υποβάλλουμε την δοθείσα σειριακή εκδοχή του Game Of Love:

```
parlab10@scirouter:~/ex1/serial$ ls  
gameOfLife.c Makefile make_on_queue.sh run_on_queue.sh  
parlab10@scirouter:~/ex1/serial$ qsub -q parlab make_on_queue.sh  
516046.localhost  
parlab10@scirouter:~/ex1/serial$ ls  
gameOfLife gameOfLife.c Makefile make_gameOfLife.err make_gameOfLife.out make_on_queue.sh run_on_queue.sh  
parlab10@scirouter:~/ex1/serial$ cat make_gameOfLife.out  
gcc -O3 -fopenmp -o gameOfLife gameOfLife.c  
parlab10@scirouter:~/ex1/serial$ cat make_gameOfLife.err  
parlab10@scirouter:~/ex1/serial$  
  
parlab10@scirouter:~/ex1/serial$ qsub -q parlab run_on_queue.sh  
516046.localhost  
parlab10@scirouter:~/ex1/serial$ qstat -f 516046.localhost  
qstat: Unknown Job Id 516046.localhost  
parlab10@scirouter:~/ex1/serial$ ls  
gameOfLife Makefile make_gameOfLife.out run_gameOfLife.err run_on_queue.sh  
gameOfLife.c make_gameOfLife.err make_on_queue.sh run_gameOfLife.out  
parlab10@scirouter:~/ex1/serial$ cat run_gameOfLife.out  
GameOfLife: Size 64 Steps 1000 Time 0.020344  
parlab10@scirouter:~/ex1/serial$ cat run_gameOfLife.err  
parlab10@scirouter:~/ex1/serial$
```

Τα περιεχόμενα των Makefile, make\_on\_queue.sh και run\_on\_queue.sh είναι τα εξής:

```
parlab10@scirouter:~/ex1/serial$ cat Makefile  
all: gameOfLife  
  
gameOfLife: gameOfLife.c  
        gcc -O3 -fopenmp -o gameOfLife gameOfLife.c  
  
clean:  
        rm gameOfLife
```

```

parlab10@scirouter:~/ex1/serial$ cat make_on_queue.sh
#!/bin/bash

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

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

## How many machines should we get?
#PBS -l nodes=1:ppn=1

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

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

module load openmp
cd /home/parallel/parlab10/ex1/serial
make

```

```

parlab10@scirouter:~/ex1/serial$ cat run_on_queue.sh
#!/bin/bash

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

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

## How many machines should we get?
#PBS -l nodes=1:ppn=8

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

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

module load openmp
cd /home/parallel/parlab10/ex1/serial
export OMP_NUM_THREADS=8
./gameOfLife 64 1000

```

3. Παρατηρούμε ότι η κατάσταση κάθε χρονικής στιγμής εξαρτάται από αυτή της προηγούμενης χρονικής στιγμής και συνεπώς δεν μπορούμε να παραλληλοποιήσουμε την εργασία ως προς τον χρόνο. Όμως η κατάσταση κάθε κελιού την χρονική στιγμή t εξαρτάται μόνο από την κατάσταση των γειτονικών του κελιών για τη στιγμή t-1. Υπό αυτό το πρίσμα μπορούμε να βρούμε τη καθολική κατάσταση της χρονικής στιγμής t παράλληλα:

Προσθέτουμε τη γραμμή #pragma omp parallel for shared(N, previous, current) private(i, j, nbrs) πριν το δεύτερο εμφωλιασμένο βρόγχο ώστε να παραλληλοποιήσουμε τον δοθέντα κώδικα υποθέτοντας shared address space και χρησιμοποιώντας το OpenMP.

Με την εντολή αυτή ζητάμε να παραλληλοποιηθεί ο βρόγχος που ακολουθεί ανάμεσα σε OMP\_NUM\_THREADS (μεταβλητή περιβάλλοντος) νήματα. Δηλώνουμε ότι η μεταβλητή N καθώς και οι δείκτες previous/current μοιράζονται μεταξύ των νημάτων, ενώ οι μεταβλητές i,j, nbrs είναι ιδιωτικά για το εκάστοτε νήμα.

Άρα ο κώδικας γίνεται:

```

parlab10@scirouter:~/ex1/parallel$ cat gameOfLife.c
/****************************************************************************
 **** Conway's game of life ****
 ****
 Usage: ./exec ArraySize TimeSteps

 Compile with -DOUTPUT to print output in output.gif
 (You will need ImageMagick for that - Install with
 sudo apt-get install imagemagick)
 WARNING: Do not print output for large array sizes!
 or multiple time steps!
 ****

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define FINALIZE \
convert -delay 20 `ls -1 out*.pgm | sort -V` output.gif\n\
rm *pgm\n\
"

int ** allocate_array(int N);
void free_array(int ** array, int N);
void init_random(int ** array1, int ** array2, int N);
void print_to_pgm( int ** array, int N, int t );

int main (int argc, char * argv[]) {
    int N;                                //array dimensions
    int T;                                //time steps
    int ** current, ** previous;           //arrays - one for current timestep, one for previous timestep
    int ** swap;                           //array pointer
    int t, i, j, nbrs;                     //helper variables

    double time;                           //variables for timing
    struct timeval ts,tf;

    /*Read input arguments*/
    if ( argc != 3 ) {
        fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
    }
}
```

```

        fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
        exit(-1);
    }
    else {
        N = atoi(argv[1]);
        T = atoi(argv[2]);
    }

    /*Allocate and initialize matrices*/
    current = allocate_array(N);                                //allocate array for current time step
    previous = allocate_array(N);                             //allocate array for previous time step

    init_random(previous, current, N);           //initialize previous array with pattern

    #ifdef OUTPUT
    print_to_pgm(previous, N, 0);
    #endif

    /*Game of Life*/

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

        #ifdef OUTPUT
        print_to_pgm(current, N, t+1);
        #endif
        //Swap current array with previous array
        swap=current;
        current=previous;
        previous=swap;
    }

    gettimeofday(&tf,NULL);
    time=(tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec)*0.000001;
}

```

```

    free_array(current, N);
    free_array(previous, N);
    printf("GameOfLife: Size %d Steps %d Time %lf\n", N, T, time);
    #ifdef OUTPUT
    system(FINALIZE);
    #endif
}

int ** allocate_array(int N) {
    int ** array;
    int i,j;
    array = malloc(N * sizeof(int*));
    for ( i = 0; i < N ; i++ )
        array[i] = malloc( N * sizeof(int));
    for ( i = 0; i < N ; i++ )
        for ( j = 0; j < N ; j++ )
            array[i][j] = 0;
    return array;
}

void free_array(int ** array, int N) {
    int i;
    for ( i = 0 ; i < N ; i++ )
        free(array[i]);
    free(array);
}

void init_random(int ** array1, int ** array2, int N) {
    int i,pos,x,y;

    for ( i = 0 ; i < (N * N)/10 ; i++ ) {
        pos = rand() % ((N-2)*(N-2));
        array1[pos%(N-2)+1][pos/(N-2)+1] = 1;
        array2[pos%(N-2)+1][pos/(N-2)+1] = 1;
    }
}

void print_to_pgm(int ** array, int N, int t) {
    int i,j;

```

```

void print_to_pgm(int ** array, int N, int t) {
    int i,j;
    char * s = malloc(30*sizeof(char));
    sprintf(s,"out%d.pgm",t);
    FILE * f = fopen(s,"wb");
    fprintf(f, "P5\n%d %d 1\n", N,N);
    for ( i = 0; i < N ; i++ )
        for ( j = 0; j < N ; j++ )
            if ( array[i][j]==1 )
                fputc(1,f);
            else
                fputc(0,f);
    fclose(f);
    free(s);
}

```

Στο script run\_on\_queue.sh προσθέτουμε το παρακάτω:

```

for thr in 1 2 4 6 8
do
    export OMP_NUM_THREADS=$thr
    ./gameOfLife 64 1000
    ./gameOfLife 1024 1000
    ./gameOfLife 4096 1000
done

```

4. Λαμβάνουμε μετρήσεις επίδοσης (Συνολικό χρόνο εκτέλεσης σε sec) για έναν κλώνο για 1,2,4,6,8 πυρήνες και για μεγέθη πίνακα 64 x 64, 1024 x 1024, 4096 x 4096 έχοντας θέσει 1000 γενιές/time-steps για όλες τις περιπτώσεις.

Cores/Table Size	64 x 64	1024 x 1024	4096 x 4096
1	0.023132	10.970660	175.946833
2	0.013687	5.457147	88.263236
4	0.010124	2.723553	44.536678
6	0.009446	1.830368	37.184969
8	0.009352	1.376726	36.517382

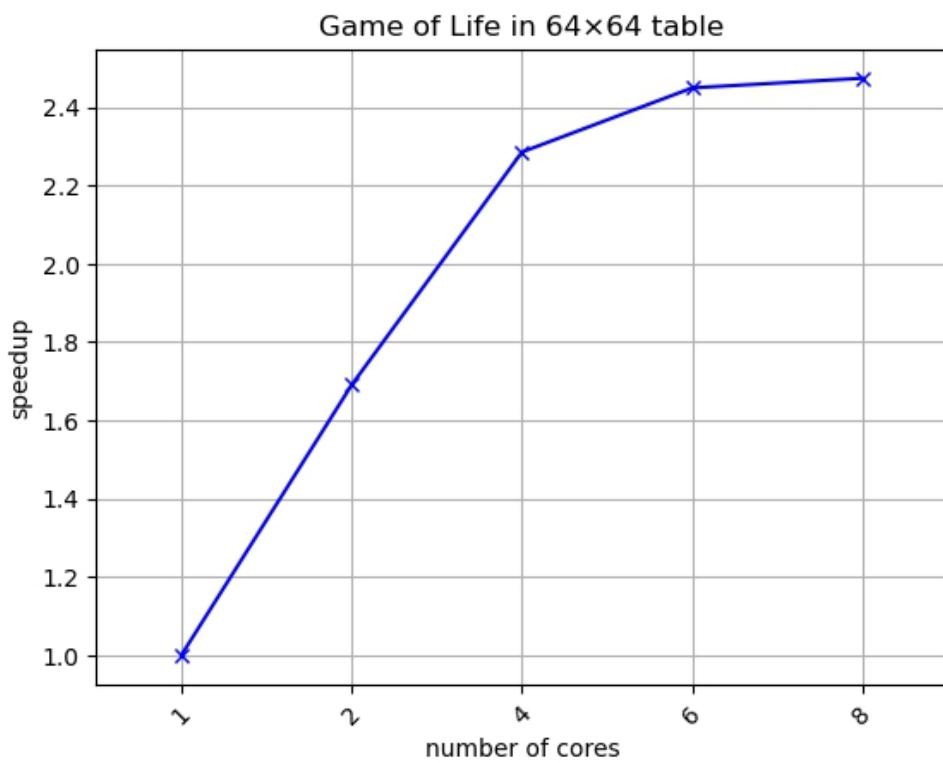
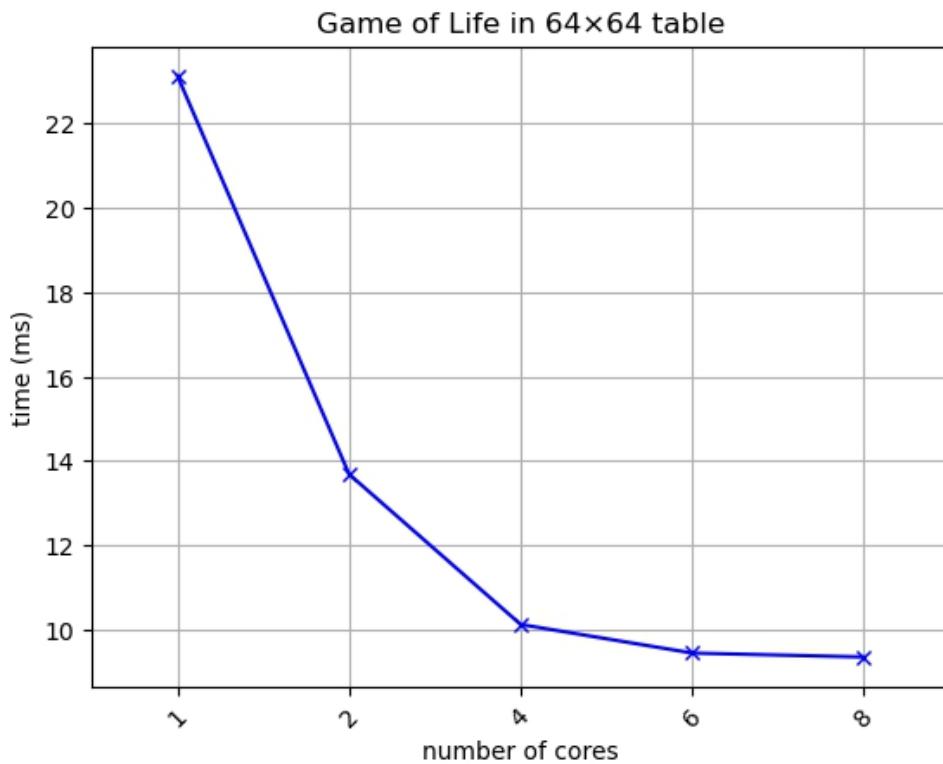
5. Προκειμένου να έχουμε εποπτική εικόνα της επίδρασης της αύξησης των cores στην επίδοση κατασκευάζουμε για κάθε μέγεθος ταμπλό διάγραμμα μεταβολής χρόνου και speedup συναρτήσει του αριθμού των πυρήνων.

Σκοπός της παραλληλοποίησης είναι η μείωση του χρόνου εκτέλεσης, ιδανικά ανάλογα με τον αριθμό των πυρήνων.

Ωστόσο αυτό δεν συμβαίνει πάντα λόγω:

1. Κόστος δημιουργίας και τερματισμού νημάτων: Καθώς αυξάνεται ο αριθμός των νημάτων, το overhead που σχετίζεται με τη δημιουργία, διαχείριση και τερματισμό των νημάτων μπορεί να γίνει σημαντικό, ειδικά αν τα threads ζουν για λίγο.
2. Κόστος συγχρονισμού: Εάν ένα thread απαιτεί συχνό συγχρονισμό (χρησιμοποιώντας locks, semaphores...) το overhead αυξάνεται καθώς περισσότερα νήματα διαγωνίζονται για τους ίδιους πόρους.
3. Ανταγωνισμός πόρων: Πολλαπλά threads μπορεί να διαγωνίζονται για κοινούς πόρους, όπως μνήμη ή I/Os. Πάλι θα χρειαστεί να καταναλωθεί χρόνος στον συγχρονισμό τους.
4. Συνάφεια κρυψής μνήμης: Σε multicore συστήματα, κάθε πυρήνας έχει την δική του cache. Όταν ένας πυρήνας τροποποιεί τα δεδομένα στην δική του cache, το σύστημα πρέπει να ανανεώσει ή να κάνει invalidate τα αντίστοιχα entries στις υπόλοιπες caches ώστε να διατηρηθεί η συνάφεια cache. Αυτό προσθέτει περαιτέρω αναμονή αν τα threads επεξεργάζονται τα ίδια blocks.
5. Περιορισμό στο Bandwidth μνήμης: Καθώς αυξάνεται ο αριθμός των πυρήνων, η ζήτηση για πόρους από την μνήμη αυξάνεται οδηγώντας πολλές φορές σε κορεσμό του bandwidth, ειδικά αν όλοι οι πυρήνες προσπαθούν να διαβάσουν/γράψουν στην μνήμη ταυτόχρονα.
6. False Sharing: Περισσότερα νήματα σημαίνει μεγαλύτερη επεξεργασία δεδομένων. Πολλές φορές τα δεδομένα αυτά που χρησιμοποιούνται από διαφορετικά threads βρίσκονται στο ίδιο cache line προκαλώντας updates/invalidations και άρα επιπρόσθετες καθυστερήσεις.
7. Νόμος του Amdahl: Υπάρχουν όρια στο speedup από την παραλληλοποίηση. Το μέγιστο δυνατό speedup καθορίζεται από το ποσοστό του προγράμματος που είναι σειριακό και δεν μπορεί να παραλληλοποιηθεί. Καθώς προσθέτουμε πυρήνες, το σειριακό μέρος του προγράμματος κυριαρχεί στον χρόνο εκτέλεσης, περιορίζοντας το κατορθωτό speedup.

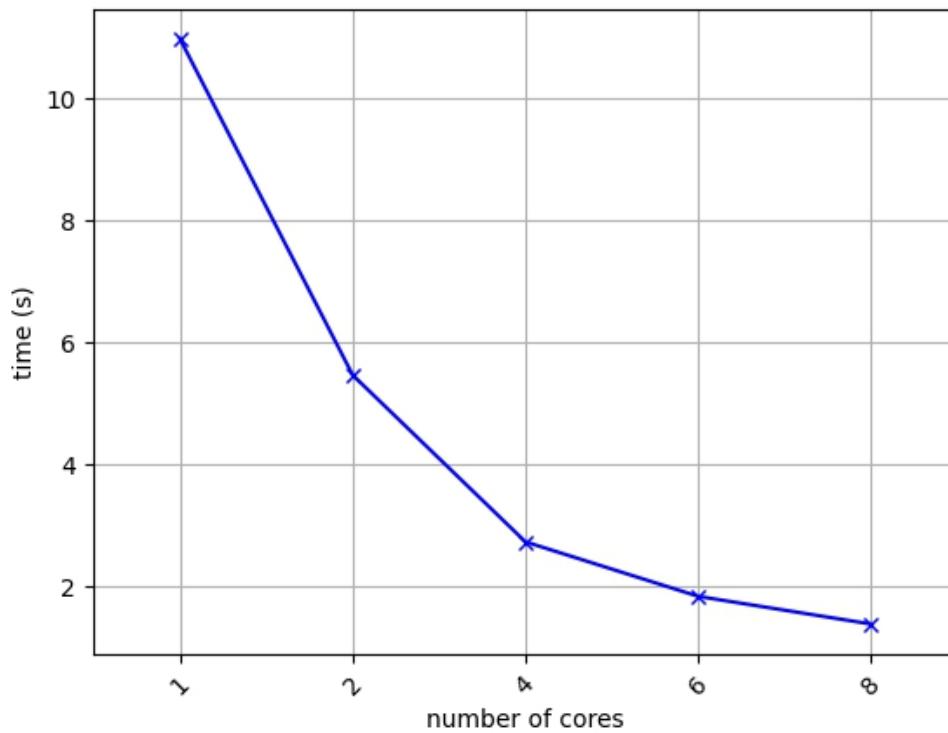
8. Ανισορροπίες φόρτου: Το workload μπορεί να μην γίνεται να κατανεμηθεί ομοιόμορφα μεταξύ των νημάτων, συνεπώς κάποια νήματα θα τελειώσουν την εργασία τους πολύ νωρίς και να παραμείνουν αδρανή ενώ άλλα δουλεύουν.



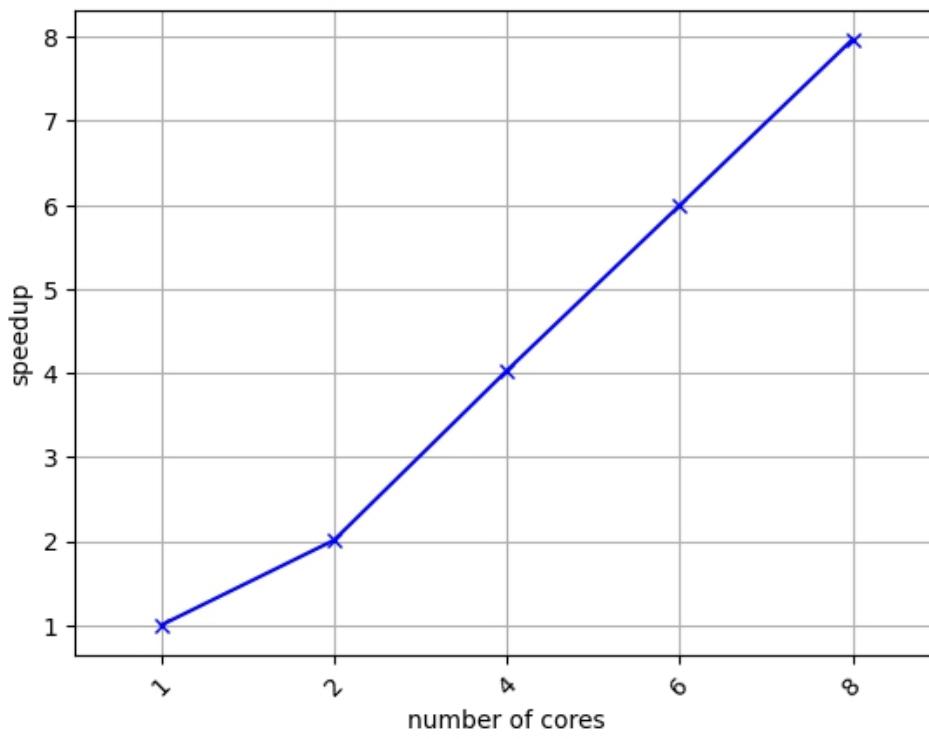
Παρατηρούμε ότι εν γένει με την αύξηση των πυρήνων έχουμε βελτίωση της επίδοσης του προγράμματος καθώς ελαττώνεται ο χρόνος εκτέλεσης. Ωστόσο η μείωση αυτή δεν είναι ανάλογη με τον αριθμό των πυρήνων που προσθέτουμε. Παρατηρούμε μεγάλη βελτίωση της επίδοσης από τον 1 πυρήνα στους 2, μικρότερη από τους 4 στους 6 και αμελητέα από τους 6 πυρήνες στους 8. Αφού το ταμπλό είναι σχετικά μικρό, υπάρχει περιορισμένη εργασία που μπορεί να παραλληλοποιηθεί και να κατανεμηθεί ανάμεσα σε περισσότερους "εργάτες". Υπό αυτό το πλαίσιο, από ένα σημείο και μετά καταναλώνεται περισσότερος χρόνος για επικοινωνία και συγχρονισμό μεταξύ νημάτων παρά για αφέλιμη εργασία.

Η καμπύλη συνεπώς έρχεται σε "κορεσμό" και φαίνεται ότι περαιτέρω αύξηση του αριθμού των πυρήνων όχι μόνο δεν θα αυξήσει την απόδοση αλλά ίσως και να τη χειροτερεύσει.

Game of Life 1024×1024 table



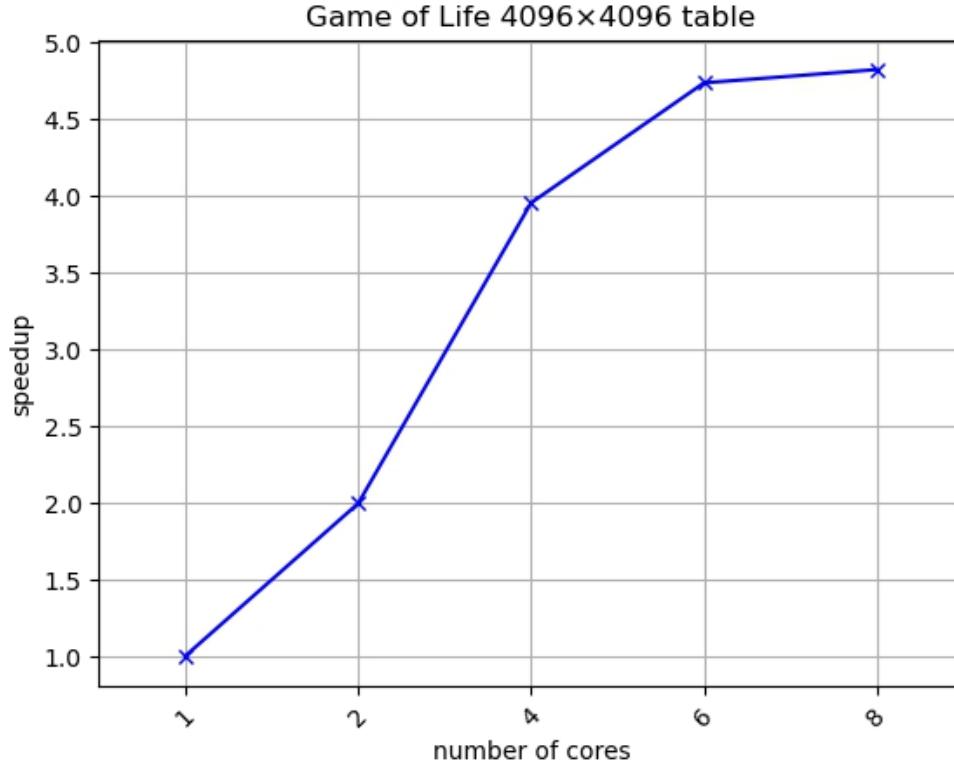
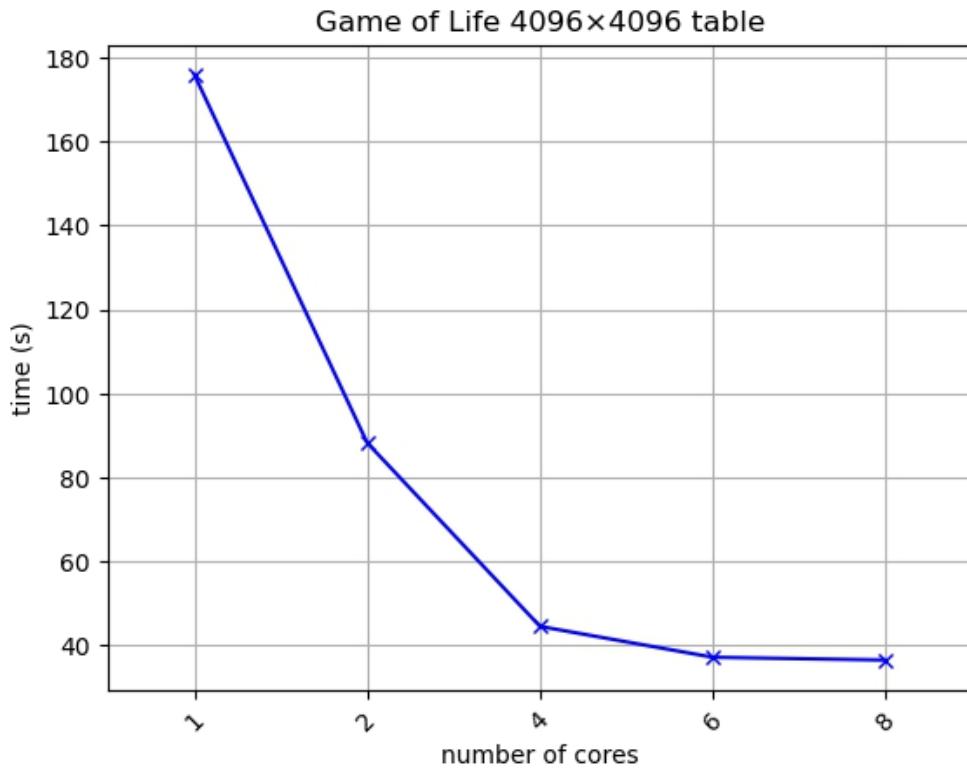
Game of Life 1024×1024 table



Παρατηρούμε ότι εν γένει με την αύξηση των πυρήνων έχουμε βελτίωση της επίδοσης του προγράμματος καθώς ελαττώνεται ο χρόνος εκτέλεσης. Σε σύγκριση με πριν, πλέον η μείωση του χρόνου (και άρα το speedup) είναι ανάλογη με τον αριθμό των πυρήνων που προσθέτουμε. Παρατηρούμε μεγάλη (καθολική) μείωση του χρόνου εκτέλεσης από τον 1 πυρήνα στους 2, μικρότερη από τους 4 στους 6 και ακόμα μικρότερη από τους 6 πυρήνες στους 8 αλλά (όπως φαίνεται στο speedup όπου έχουμε 2πλάσιο για 2 πυρήνες, 4πλάσιο για 4 πυρήνες ...) η αναλογική μείωση (υποδιπλασιάζεται ο χρόνος εκτέλεσης) είναι η ίδια καθώς προσθέτουμε πυρήνες.

Συνεπώς το ταμπλό με μέγεθος 1024 x 1024 αποτελεί καλή περίπτωση παραλληλισμού. Δεν παρατηρείται "κορεσμός" και άρα προβλέπουμε ότι περαιτέρω αύξηση των πυρήνων θα οδηγήσει σε σημαντική μείωση του χρόνου εκτέλεσης. Αυτό συμβαίνει διότι στο εν λόγω ταμπλό ο χρόνος που χάνεται για την επικοινωνία των νημάτων σε σύγκριση με τον χρόνο που κερδίσουμε από την παραλληλοποίηση είναι μικρός.

Note: Το speedup ορίζεται ως ο λόγος του χρόνου που έκανε το σειριακό πρόγραμμα να εκτελεστεί προς τον χρόνο που έκανε το παράλληλο πρόγραμμα να εκτελεστεί.



Παρατηρούμε ότι εν γένει με την αύξηση των πυρήνων έχουμε βελτίωση της επίδοσης του προγράμματος καθώς ελαττώνεται ο χρόνος εκτέλεσης. Σε σύγκριση με πριν, πλέον η μείωση του χρόνου (και άρα το speedup) είναι ανάλογη με τον αριθμό των πυρήνων που προσθέτουμε μέχρι και τους 4 πυρήνες. Στη συνέχεια έχουμε μικρή αύξηση του speedup. Παρατηρούμε μεγάλη (καθολική) μείωση του χρόνου εκτέλεσης από τον 1 πυρήνα στους 2, μικρότερη από τους 4 στους 6 και αμελητέα από τους 6 πυρήνες στους 8.

Η καμπύλη συνεπώς έρχεται σε “κορεσμό” και φαίνεται ότι περαιτέρω αύξηση του αριθμού των πυρήνων όχι μόνο δεν θα αυξήσει την απόδοση αλλά ίσως και να τη χειροτερεύσει.

Υποθέτουμε ότι ο παραλληλισμός είναι χειρότερος από την περίπτωση μεγέθους 1024 x 1024 διότι το ταμπλό χώραγε στις caches των επεξεργαστών ενώ για το ταμπλό 4096 x 4096 έχουμε περισσότερες προσβάσεις στην γενική μνήμη.



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ  
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

## Συστήματα Παράλληλης Επεξεργασίας

Αναφορά 2ης Εργαστηριακής Άσκησης  
Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε  
αρχιτεκτονικές κοινής μνήμης

Ραπτόπουλος Πέτρος (el19145)  
Κόγιος Δημήτριος (el19220)  
Καπετανάκης Αναστάσιος (el19048)

## 2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

**Σκοπός της άσκησης:** Ανάπτυξη δύο παράλληλων εκδόσεων του αλγορίθμου K-means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με τη χρήση του προγραμματιστικού εργαλείου OpenMP.

Δοκιμή βελτιώσεων υλοποίησης παράλληλου κώδικα και αξιολόγηση της τελικής επίδοσης του παράλληλου προγράμματος.

### Ο αλγόριθμος K-means

Σκοπός του αλγορίθμου είναι ο διαχωρισμός  $N$  αντικειμένων σε  $k$  μη επικαλυπτόμενες ομάδες (συστάδες - clusters). Ο αλγόριθμος λειτουργεί ως εξής:

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

Δίνεται ο κώδικας του σειριακού προγράμματος (seq\_kmeans).

Η περιοχή ενδιαφέροντος αποτελείται από τις παρακάτω γραμμές κώδικα:

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

    // if membership changes, increase delta by 1
    if (membership[i] != index)
        delta += 1.0;

    // assign the membership to object i
    membership[i] = index;

    // update new cluster centers : sum of objects located within
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
}

// average the sum and replace old cluster centers with newClusters
for (i=0; i<numClusters; i++) {
    if (newClusterSize[i] > 0) {
        for (j=0; j<numCoords; j++) {
            clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
        }
    }
}
```

Οι εκδόσεις παραλληλοποίησης κώδικα που καλούμαστε να αναπτύξουμε είναι δύο:

### 1. shared clusters

Μια βασική έκδοση που παραλληλοποιεί τον σειριακό αλγόριθμο με προσθήκη κατάλληλων εντολών συγχρονισμού ώστε να γίνει ορθά η παράλληλη ενημέρωση του πίνακα newClusters με τις νέες υπολογιζόμενες συντεταγμένες των κέντρων των συστάδων. (omp\_naive\_kmeans)

**1.** Παρατηρούμε ότι οι υπολογισμοί για κάθε object (σημείο στο επίπεδο) είναι ανεξάρτητοι μεταξύ τους και μπορούν να παραλληλοποιηθούν με τη χρήση #pragma omp parallel for. Οι υπολογισμοί γίνονται πάνω σε κοινή μνήμη με τη χρήση #pragma omp atomic για να επιτευχθεί ο συγχρονισμός. Ο κώδικας για την περιοχή ενδιαφέροντος διαμορφώνεται ως εξής:

```

/*
 * TODO: Detect parallelizable region and use appropriate OpenMP pragmas
 */
#pragma omp parallel for shared(numObjs, numClusters, numCoords, clusters, objects,
membership, newClusters, newClusterSize, delta) private(i, j, index)
for (i=0; i<numObjs; i++) {
    // find the array index of nearest cluster center
    index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);

    // if membership changes, increase delta by 1
    if (membership[i] != index)
        #pragma omp atomic
        delta += 1.0;

    // assign the membership to object i
    // no need for synchronization, as only one thread will modify membership[i]
    membership[i] = index;

    // update new cluster centers : sum of objects located within
    /*
     * TODO: protect update on shared "newClusterSize" array
     * DONE: We use the atomic pragma. Its more efficient than critical.
     * Critical is used for more complex (multiple lined) statements
     */
    #pragma omp atomic
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++) {
        /*
         * TODO: protect update on shared "newClusters" array
         * DONE: Usage of atomic pragma
         */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }
}

// average the sum and replace old cluster centers with newClusters
for (i=0; i<numClusters; i++) {
    if (newClusterSize[i] > 0) {
        for (j=0; j<numCoords; j++) {
            clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
        }
    }
}
}

```

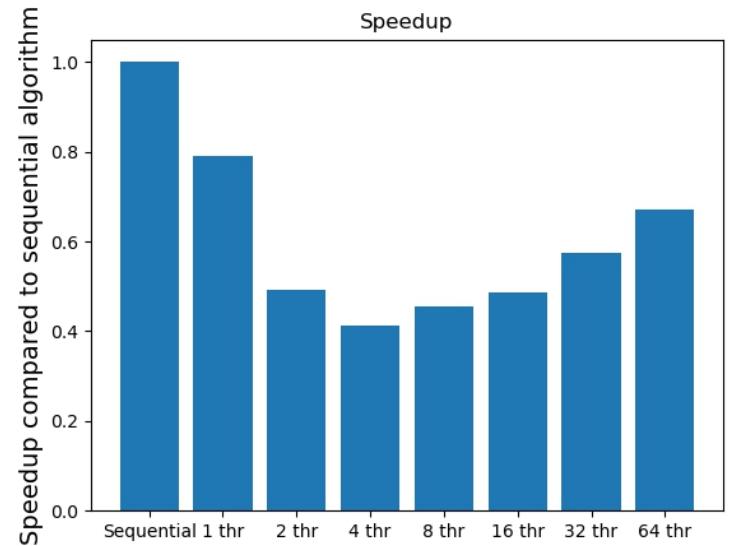
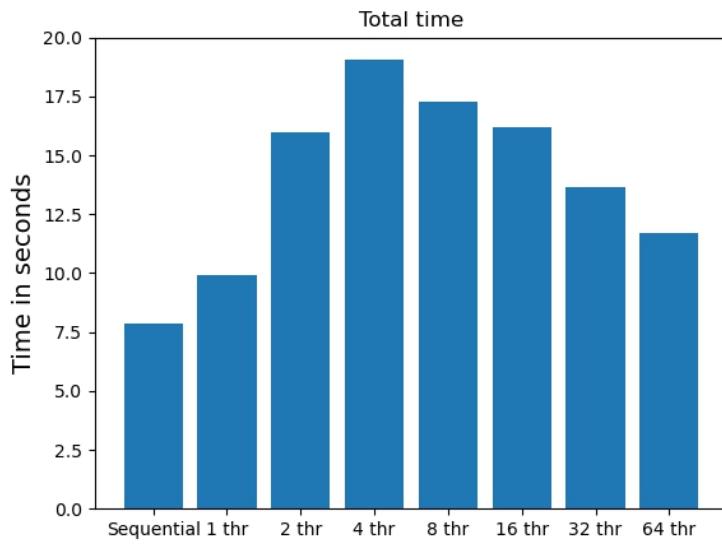
Για να μεταγλωτίσουμε τον κώδικα τροποποιούμε κατάλληλα το Makefile.

Τροποποιούμε επίσης και τα make\_on\_queue.sh/run\_on\_queue.sh ώστε να πραγματοποιήσουμε μετρήσεις για το configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχάνημα sandman.

**Σημείωση:** Η ακριβής σύνταξη του κώδικα καθώς και των scripts/makefiles κ.ά. φαίνεται στο παρακάτω github repository: [https://github.com/petrosrapto/ParallelProcessingSystems\\_23-24/](https://github.com/petrosrapto/ParallelProcessingSystems_23-24/)

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_naive_kmeans	9.9265s	15.9681s	19.0513s	17.2718s	16.1752s	13.6271s	11.6955s

Με τη βοήθεια python script δημιουργούμε το barplot διάγραμμα χρόνου εκτέλεσης που προκύπτει (x-axis = sequential και threads, y-axis = time), και το αντίστοιχο speedup plot (x-axis = sequential και threads, y-axis = seq\_time/time).



Παρατηρούμε ότι ο χρόνος του προγράμματος `omp_naive_kmeans` ακόμη και για ένα thread είναι μεγαλύτερος από τον χρόνο του σειριακού προγράμματος. Αυτό οφείλεται στους εξής παράγοντες:

#### -Overhead των OMP pragmas

Περιλαμβάνει την διαχείριση των parallel regions, την δημιουργία/τερματισμό/διαχείριση των threads.

#### -Κόστος συγχρονισμού κοινών πόρων

Οι ατομικές εντολές επιδρούν αρνητικά στην επίδοση.

#### -Cache Coherence Protocol (+ False Sharing)

Η τροποποίηση ίδιων θέσεων μνήμης πυροδοτεί block invalidations στις διαφορετικές caches των CPUs και επιπλέον καθυστέρηση για την ενημέρωση των εν λόγω cache blocks.

#### -Non-Uniform Memory Access (NUMA) architecture and First-Touch Policy

Τα threads που τρέχουν σε απομακρυσμένα cores σε σχέση με τη RAM όπου έχουν αποθηκευτεί οι κοινοί πόροι, κάνουν περισσότερο χρόνο να προσπελάσουν την μνήμη.

#### -Compiler optimizations

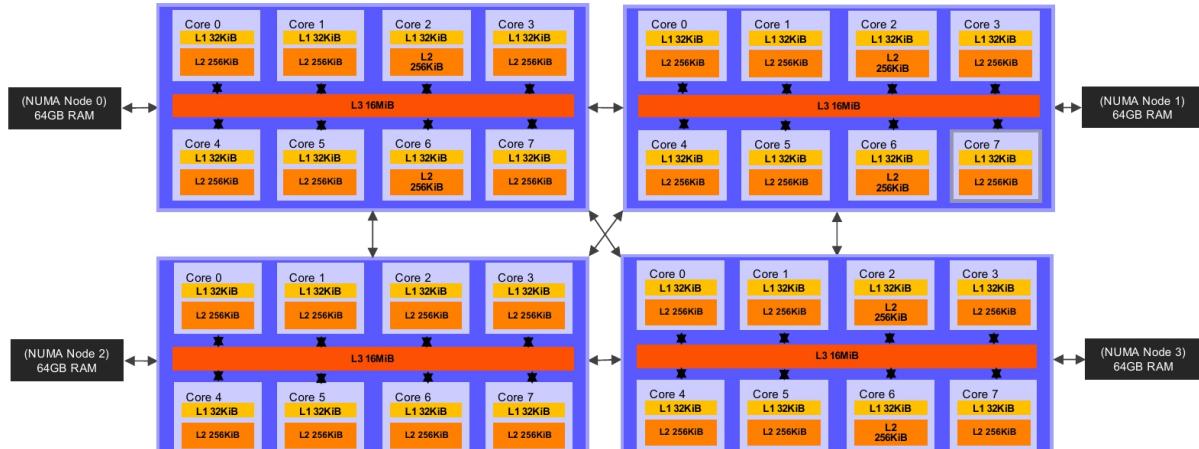
Ο μεταγλωτιστής μπορεί να εφαρμόζει βελτιώσεις στον σειριακό κώδικα, οι οποίες για τον παράλληλο κώδικα δεν είναι ασφαλές να πραγματοποιηθούν.

### Σημειώσεις:

**NUMA:** Σε μία Non-Uniform Memory Access αρχιτεκτονική, ο χρόνος πρόσβασης στην μνήμη εξαρτάται από την τοποθεσία της μνήμης RAM σε σχέση με τον αντίστοιχο επεξεργαστή και δεν είναι ομοιόμορφη για όλους τους επεξεργαστές.

**First Touch Policy:** Τα συστήματα NUMA εφαρμόζουν την πολιτική αυτή για memory allocation. Η μνήμη παρέχεται από το node στο οποίο “ανήκει” ο επεξεργαστής στον οποίο εκτελείται το νήμα που έγραψε πρώτο στην μνήμη.

**Αρχιτεκτονική μηχανήματος sandman:** Αρχιτεκτονική NUMA, αποτελείται από 4 nodes, 8 επεξεργαστές στο καθένα με 64GB RAM. Κάθε επεξεργαστής έχει 2 τοπικά επίπεδα cache και ένα διαμοιραζόμενο με το node.



Προκειμένου να ελέγξουμε την επίδραση των ατομικών εντολών στην επίδοση του προγράμματος τις αφαιρούμε και ξαναπραγματοποιούμε τις μετρήσεις. Προφανώς τα αποτελέσματα θα είναι εσφαλμένα, επικεντρωνόμαστε στον χρόνο εκτέλεσης:

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_naive_kmeans	9.9265s	15.9681s	19.0513s	17.2718s	16.1752s	13.6271s	11.6955s
omp_naive_noSync	7.7488	12.8828	15.588	12.9483	11.5986	10.0246	7.0833

Παρατηρούμε ότι:

-Για ένα νήμα: Ο χρόνος εκτέλεσης μειώνεται σημαντικά και γίνεται περίπου ίσος με αυτόν του σειριακού προγράμματος. Δηλαδή ακόμη και για μη συνθήκες ανταγωνισμού μεταξύ πολλών threads ο συγχρονισμός προσθέτει μεγάλο overhead.

Οι υπόλοιποι παράγοντες αναφέρθηκαν δεν επιδρούν στον χρόνο.

-Για περισσότερα νήματα: Ο χρόνος μειώνεται σημαντικά (ο συγχρονισμός προσθέτει μεγάλο overhead) όμως δεν γίνεται ίσος με αυτόν του σειριακού προγράμματος. Αυτό σημαίνει ότι και άλλοι παράγοντες επιδρούν.

Αναζητούμε τους παράγοντες αυτούς:

Παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται μέχρι τα 4 threads και στην συνέχεια μειώνεται. Αναρωτιόμαστε για ποιον λόγο να συμβαίνει αυτό και θυμόμαστε ότι το sandman έχει 4 διαφορετικά nodes. Υποψιαζόμαστε ότι το κάθε thread γίνεται allocate σε ξεχωριστό node. Γνωρίζουμε ότι το λειτουργικό σύστημα και το runtime system του OpenMP έχει τον πρωταρχικό λόγο στο να κατανείμει τα threads στα αντίστοιχα cores. Συνεπώς δεν μπορούμε εκ των προτέρων να γνωρίζουμε σε ποια nodes θα εκτελεστούν τα ανωτέρω threads.

Πραγματοποιούμε δύο μετρήσεις:

- 1) Ρητό allocate των threads για το omp\_naive\_kmeans σε ξεχωριστά nodes ορίζοντας την environmental variable GOMP\_CPU\_AFFINITY="0 8 16 24". Έχουμε χρόνο εκτέλεσης για 4 threads 14.7906sec
- 2) Ρητό allocate των threads για το omp\_naive\_kmeans στο ίδιο node ορίζοντας την environmental variable GOMP\_CPU\_AFFINITY="0-3". Έχουμε χρόνο εκτέλεσης για 4 threads 4.9642sec

Παρατηρούμε ότι στην πρώτη περίπτωση έχουμε περίπου ίσο χρόνο με το omp\_naive\_kmeans και συμπεραίνουμε ότι τα threads γίνονται allocate σε ξεχωριστά nodes και για αυτό έχουμε αύξηση του χρόνου εκτέλεσης μέχρι τα 4 threads και πτώση από τα 8 threads και μετά.

[Η μνήμη έχει δεσμευτεί στην RAM του πρώτου node (λόγω first touch policy) και προσβάσεις στις διευθύνσεις αυτές από cores άλλων nodes κοστίζουν χρονικά (NUMA).

Επιπρόσθετα, τα νήματα σε διαφορετικά nodes δεν κάνουν χρήση κοινής cache]

Συγκρίνοντας την περίπτωση 1 με την omp\_naive\_kmeans έχουμε περίπου ίσο χρόνο αλλά όχι ίσο. Γιατί συμβαίνει αυτό; Με την χρήση GOMP\_CPU\_AFFINITY δεσμεύουμε συγκεκριμένα threads σε συγκεκριμένα cores και συνεπώς έχουμε λιγότερο "thread migration" μεταξύ των πυρήνων και συνεπώς λιγότερα cache misses.

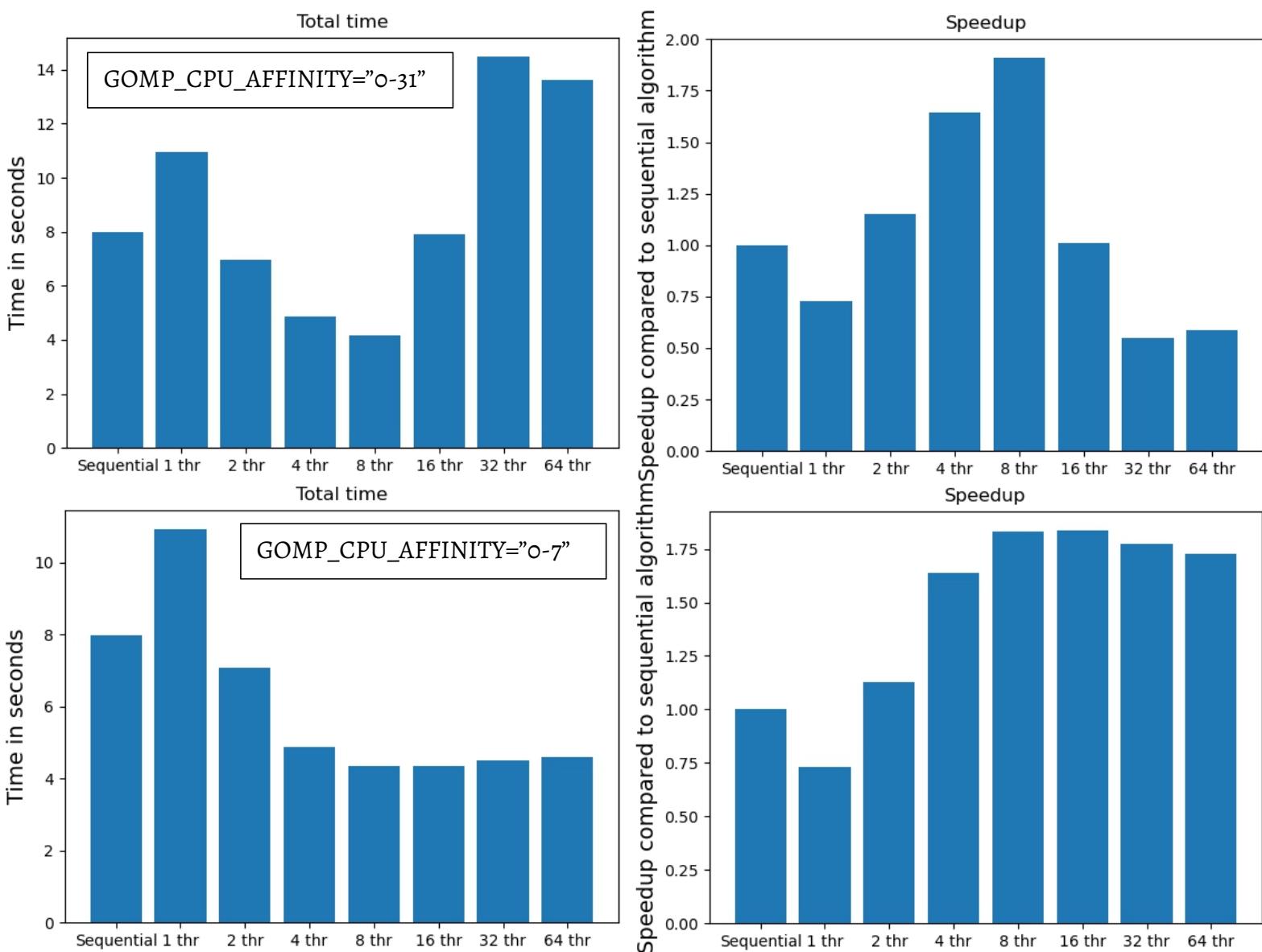
Συγκρίνοντας τις περιπτώσεις 1-2 συμπεραίνουμε ότι το κύριο overhead για την εκτέλεση με περισσότερα νήματα είναι λόγω μη βέλτιστης κατανομής των νημάτων στα nodes, που οδηγεί στην μη βέλτιστη χρήση των caches αλλά και σε καθυστερήσεις απομακρυσμένης πρόσβασης cores στην μνήμη (NUMA). Τέλος το δέσιμο συγκεκριμένων threads σε συγκεκριμένα cores επιφέρει αξιόλογη βελτίωση.

- 2.** Όπως προαναφέρθηκε, με τη δέσμευση συγκεκριμένων threads σε συγκεκριμένα cores έχουμε λιγότερο “thread migration” μεταξύ των πυρήνων και συνεπώς λιγότερα cache misses.

Χρησιμοποιούμε την μεταβλητή περιβάλλοντος GOMP\_CPU\_AFFINITY για να:

- 1) Δεσμεύσουμε τα threads σε συγκεκριμένα cores αξιοποιώντας όλα τα nodes του sandman (GOMP\_CPU\_AFFINITY="0-31")
- 2) Δεσμεύσουμε τα threads σε συγκεκριμένα cores αξιοποιώντας μόνο το πρώτο node του sandman (GOMP\_CPU\_AFFINITY="0-7")

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_naive_kmeans	9.9265s	15.968s	19.051s	17.2718s	16.1752s	13.6271s	11.6955s
omp_naive_noSync	7.7488s	12.882s	15.588s	12.9483s	11.5986s	10.0246s	7.0833s
omp_naive_31_GOMP	10.9562	6.9419	4.8476	4.1753	7.9089	14.4787	13.6059
omp_naive_7_GOMP	10.9087	7.0606	4.8683	4.3489	4.3413	4.4908	4.6077



Παρατηρούμε ότι παρόλο που στην πρώτη περίπτωση έχουμε πολύ περισσότερη επεξεργαστική ισχύ και επίσης δεν έχουμε συχνή εναλλαγή threads στο ίδιο core, η δεύτερη περίπτωση είναι ταχύτερη μόνο με τη χρήση 8 πυρήνων στο 1<sup>o</sup> node λόγω της locality της μνήμης (NUMA).

Το μηχάνημα sandman επιτρέπει hyperthreading, δηλαδή πάνω από ένα software thread να τρέχει σε ένα core. Η παρουσίαση της άσκησης αναφέρει ότι έχουμε 32 cores και 64 threads. Ουσιαστικά το κάθε core χωρίζεται σε δύο "λογικά" cores και το κάθε λογικό core μπορεί να χειρίζεται ένα ξεχωριστό thread επιτρέποντας έτσι πολύ μεγαλύτερη παραλληλοποίηση. Η αύξηση της επίδοσης προκύπτει από το hardware καθώς κάποια τμήματα του επεξεργαστή είναι αντιγραμμένα για να υποστηρίζει καλύτερο scheduling και thread manipulation. Το hyperthreading περιμένουμε να έχει ίδια συμπεριφορά με το προηγούμενο ερώτημα και έτσι θα δεσμεύσουμε το node ο για το οποίο πριν με GOMP\_CPU\_AFFINITY="0-7" πετύχαμε τον καλύτερο χρόνο. Σκοπός μας πλέον είναι να δεσμεύσουμε όλα τα λογικά threads του πρώτου node.

Στέλνοντας την εντολή: ls /sys/devices/system/cpu/cpu\* -d στο μηχάνημα sandman, λαμβάνουμε ως απάντηση 64 entries της μορφής :/sys/devices/system/cpu/cpuX, όπου X = {0,1,2,3,...,63}. Αυτά είναι τα λογικά cores. Για να δούμε σε ποιο core ανήκουν και πως είναι χωρισμένα στέλνουμε την εντολή :

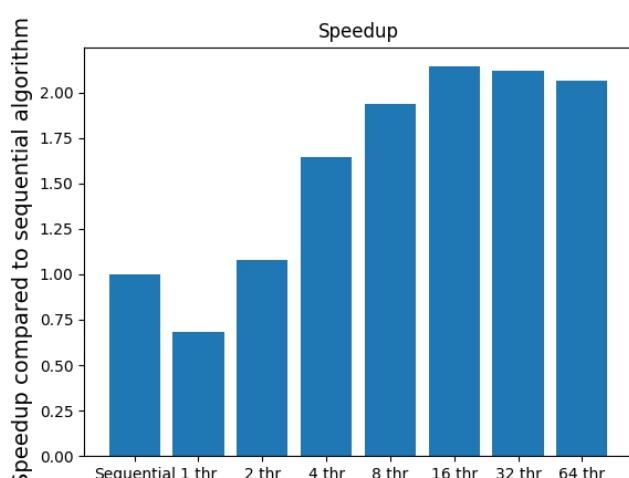
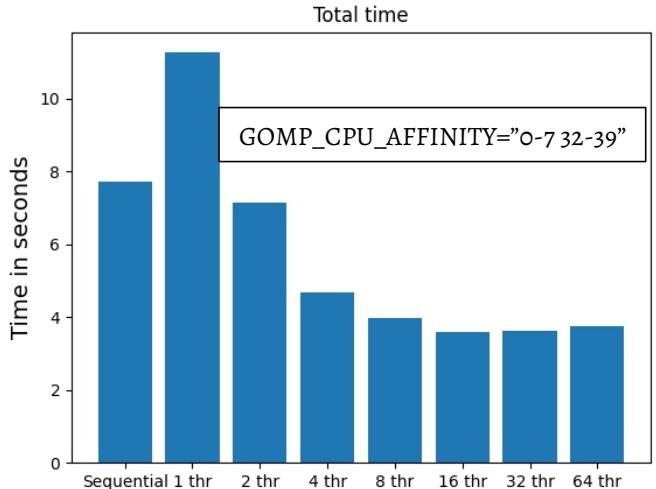
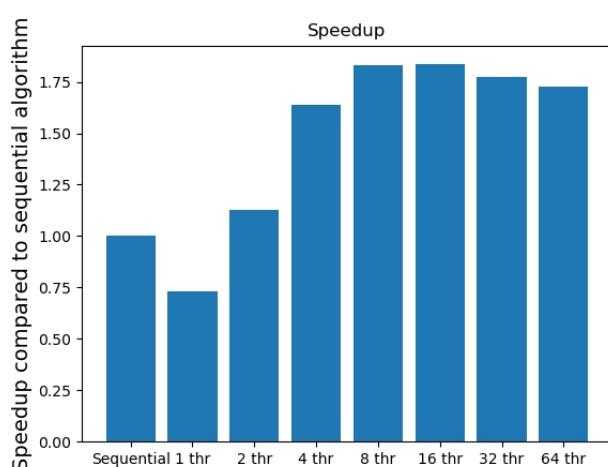
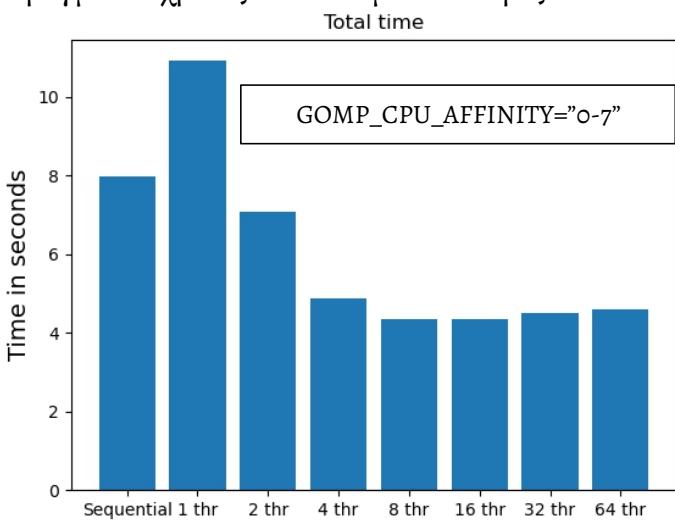
cat "/sys/devices/system/cpu/cpu\${i}/topology/thread\_siblings\_list", όπου i η τιμή του λογικού core για το οποίο θέλουμε να μάθουμε με ποιο είναι "αδελφοποιημένο". Η απάντηση που παίρνουμε είναι η εξής: 0,32 1,33 κ.λπ.

Δηλαδή τα λογικά threads ο και 32 ζουν στο core 0, τα λογικά threads 1 και 33 ζουν στο core 1 κ.λπ.

Συνεπώς για να δεσμεύσουμε ολόκληρο το node 0 και μόνο αυτό, αρκεί να ορίσουμε την μεταβλητή περιβάλλοντος GOMP\_CPU\_AFFINITY ως εξής GOMP\_CPU\_AFFINITY="0-7 32-39". Τα αποτελέσματα βρίσκονται στο παρακάτω πινακά:

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_naive_7_GOMP	10.9087s	7.0606s	4.8683s	4.3489s	4.3413s	4.4908s	4.6077s
omp_naive_7_32_GOMP	11.2602s	7.1361s	4.6811s	3.9750s	3.5981s	3.6326s	3.7335s

Πράγματι ο χρόνος είναι ακόμα καλύτερος όταν τα threads αυξάνονται.



## 2. copied clusters and reduce

- I. Με την ίδια λογική με τα shared clusters παραλληλοποιούμε τα for αυτή τη φορά όμως χρησιμοποιώντας αντιγραμμένα clusters για κάθε νήμα τα οποία στο τέλος συνδυάζονται (μόνο από ένα νήμα) ώστε να υπολογιστεί το τελικό αποτέλεσμα. Ο κώδικας για την περιοχή ενδιαφέροντος διαμορφώνεται ως εξής: (omp\_reduction)

```
#pragma omp single
{
    int tid;
    // the default behavior for all the variables in the surrounding scope
    // is to be shared. only loop iteration counters in pragma omp for are
    // made private by default
    #pragma omp parallel private(i, j, index, tid)
    {
        tid = omp_get_thread_num();
        /*
         * TODO: Initialize local cluster data to zero (separate for each thread)
         */
        for (i=0; i<numClusters; i++) {
            for (j=0; j<numCoords; j++)
                local_newClusters[tid][i*numCoords + j] = 0.0;
            local_newClusterSize[tid][i] = 0;
        }

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

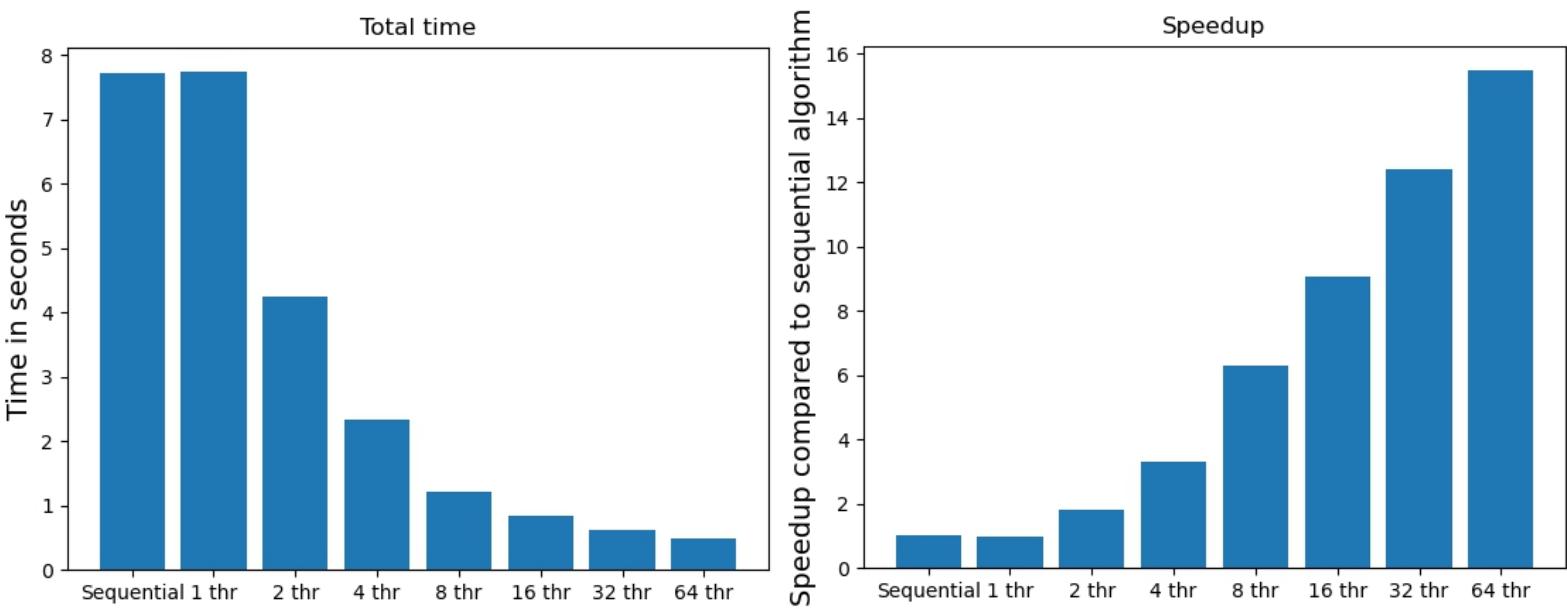
            // if membership changes, increase delta by 1
            if (membership[i] != index)
                delta += 1.0;

            // assign the membership to object i
            membership[i] = index;

            // update new cluster centers : sum of all objects located within
            // (average will be performed later)
            /*
             * TODO: Collect cluster data in local arrays (local to each thread)
             * Replace global arrays with local per-thread
             */
            local_newClusterSize[tid][index]++;
            for (j=0; j<numCoords; j++)
                local_newClusters[tid][index*numCoords+j]
                    += objects[i*numCoords+ j];
        }
    }

    /*
     * TODO: Reduction of cluster data from local arrays to shared.
     * This operation will be performed by one thread
     */
    for (i=0; i<numClusters; i++) {
        for (tid=0; tid<nthreads; tid++) {
            newClusterSize[i] += local_newClusterSize[tid][i];
            for (j=0; j<numCoords; j++)
                newClusters[i*numCoords + j] +=
                    local_newClusters[tid][i*numCoords + j];
        }
    }
}
```

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_naive_kmeans	9.9265s	15.968s	19.051s	17.2718s	16.1752s	13.6271s	11.6955s
omp_naive_noSync	7.7488s	12.882s	15.588s	12.9483s	11.5986s	10.0246s	7.0833s
omp_naive_31_GOMP	10.9562s	6.9419s	4.8476s	4.1753s	7.9089s	14.4787s	13.6059s
omp_naive_7_GOMP	10.9087s	7.0606s	4.8683s	4.3489s	4.3413s	4.4908s	4.6077s
omp_reduction	7.7378s	4.2554s	2.343s	1.222s	0.85s	0.6218s	0.4988s



Παρατηρούμε ότι η επίδοση αυτής της έκδοσης παραλληλίας είναι πολύ καλύτερη από εκείνης των shared clusters. Χρησιμοποιώντας αντιγραμμένα δεδομένα “πληρώνουμε” επιπλέον μνήμη και χρόνο allocation της μνήμης αυτής και χρόνο του reduction στο τέλος. Ωστόσο γλιτώνουμε από τον συγχρονισμό και το cache coherence protocol (+ μείωση των φαινομέων false sharing). Λόγω των local clusters μειώνονται οι προσβάσεις στην μνήμη και αυξάνεται η αξιοποίηση των caches.

Η επίδοση αυτής της έκδοσης είναι ικανοποιητική όμως με την τρέχουσα υλοποίηση είχαμε προβήματα με την μεταβλητή GOMP\_CPU\_AFFINITY. Συνεπώς υλοποίήσαμε και μια δεύτερη version όπου ουσιαστικά παραλληλοποιήσαμε τα for της sequential version. Ο κώδικας για την περιοχή ενδιαφέροντος διαμορφώνεται ως εξής (omp\_reduction\_2nd\_version):

```
// we could parallelize this loop but it is too small and the overhead is not worth it
// #pragma omp parallel for private(i,j)
shared(nthreads, numClusters, local_newClusterSize, numCoords, local_newClusters) default(none)
for(k = 0; k < nthreads; ++k) {
    for(i = 0; i < numClusters; ++i) {
        local_newClusterSize[k][i] = 0;
        for(j = 0; j < numCoords; ++j) {
            local_newClusters[k][i*numCoords + j] = 0;
        }
    }
}
```

```

#pragma omp parallel for
shared(numObjs, numClusters, numCoords, objects, clusters, membership, newClusterSize)
private(index, j, k) \
    firstprivate(local_newClusterSize, local_newClusters)
lastprivate(local_newClusterSize, local_newClusters) reduction(:delta) default(none)
    for (i=0; i<numObjs; i++)
    {
        // find the array index of nearest cluster center
        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords],
clusters);

        // if membership changes, increase delta by 1
        if (membership[i] != index)
            delta += 1.0;
        k = omp_get_thread_num();
        // assign the membership to object i
        membership[i] = index;

        // update new cluster centers : sum of all objects located within (average will be
performed later)
        /** TODO: Collect cluster data in local arrays (local to each thread)
Replace global arrays with local per-thread */
        local_newClusterSize[k][index]++;
        for (j=0; j<numCoords; j++)
            local_newClusters[k][index*numCoords + j] += objects[i*numCoords + j];
    }

/*
* TODO: Reduction of cluster data from local arrays to shared.
*       This operation will be performed by one thread
*/

```

```

for(i = 0; i < numClusters; ++i) {
    for(k = 0; k < nthreads; ++k) {
        for(j = 0; j < numCoords; ++j) {
            newClusters[i*numCoords + j] += local_newClusters[k][i*numCoords +
j];
        }
        newClusterSize[i] += local_newClusterSize[k][i];
    }
}

```

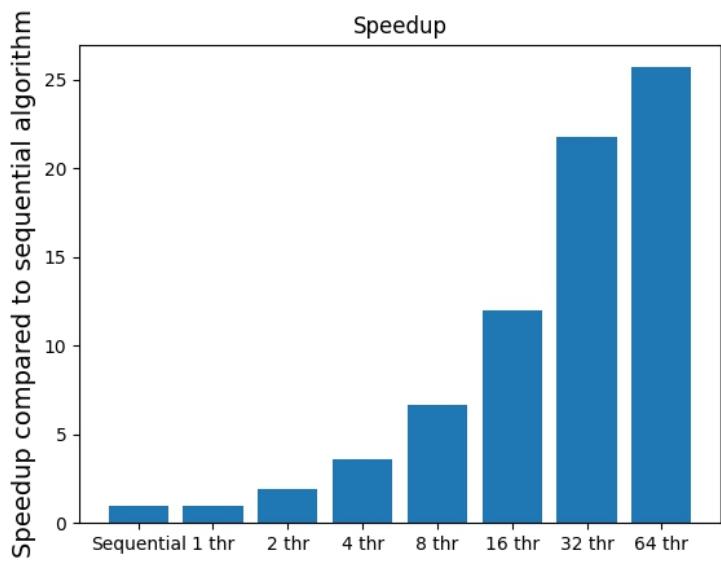
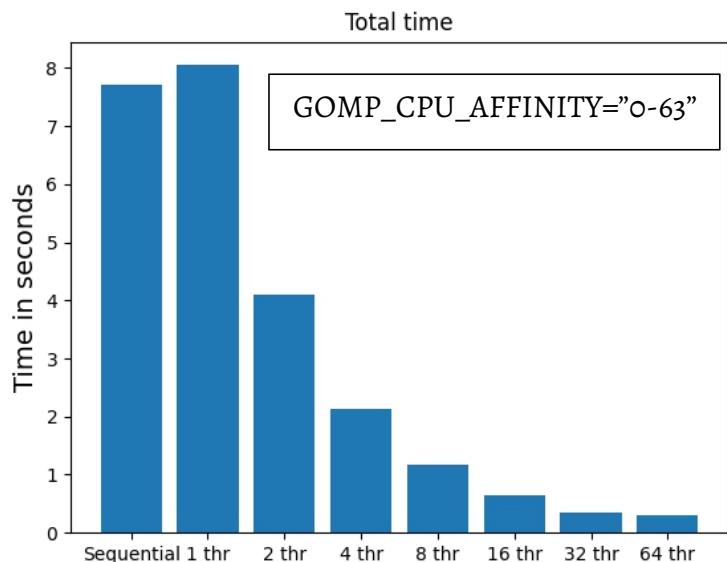
Οι χρόνοι που πετυχαίνουμε είναι:

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_reduction_v1	7.7378s	4.2554s	2.343s	1.222s	0.85s	0.6218s	0.4988s
omp_reduction_v2	7.8026s	4.2035s	2.3099s	1.2339s	0.8373s	0.5744s	0.5147s

Πλέον με την version 2 μπορούμε να χρησιμοποιήσουμε την GOMP\_CPU\_AFFINITY.

Όπως είναι φυσιολογικό, αφού πλέον κάθε threads έχει τον δικό του local πίνακα, η τοπικότητα των caches θα είναι πολύ χρήσιμη. Είναι λοιπόν λογικό που μετά από διάφορα πειράματα πετύχαμε τον καλύτερο χρόνο με GOMP\_CPU\_AFFINITY="0-63".

Program\Threads	1	2	4	8	16	32	64
seq_kmeans	7.8458s	7.7496s	7.7734s	7.9424s	7.7837s	7.7396s	7.7517s
omp_reduction_v1	7.7378s	4.2554s	2.343s	1.222s	0.85s	0.6218s	0.4988s
omp_reduction_v2	7.8026s	4.2035s	2.3099s	1.2339s	0.8373s	0.5744s	0.5147s
omp_reduction_v2_63_GOMP	8.0500s	4.0953s	2.1380s	1.1621s	0.6410s	0.3544s	0.2998s



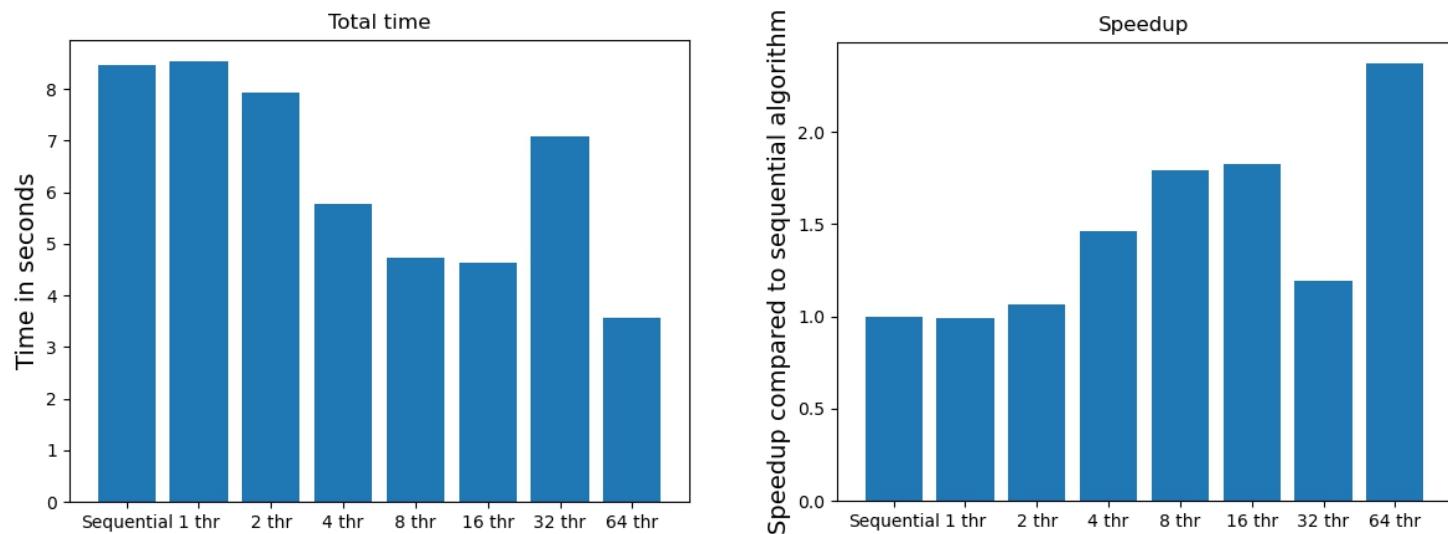
Είναι ξεκάθαρη η υπεροχή των local clusters έναντι των shared clusters αφού το μέγιστο speedup (για 64 threads) που πετυχαίναμε στο προηγούμενο ερώτημα ήταν 1.75 ενώ τώρα είναι 25.

2.

Δοκιμάζουμε το ακόλουθο configuration {Size, Coords, Clusters, Loops} = {256, 1, 8, 10}, και συγκρίνουμε το scalability με το προηγούμενο {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}.

Program\Threads	1	2	4	8	16	32	64
omp_reduction	7.7378s	4.2554s	2.343s	1.222s	0.85s	0.6218s	0.4988s
omp_reduction_different_config	8.529s	7.9245s	5.78s	4.7179s	4.6402	7.0824	3.57

Παρατηρούμε ότι σε σύγκριση με τις προηγούμενες εκδόσεις, το νέο configuration παρουσιάζει scalability, σημαντικά λιγότερο όμως συγκριτικά με το παλιό configuration.



Η τεράστια πτώση της κλιμάκωσης οφείλεται στο first touch policy αλλά και σε φαινόμενα false sharing που εμφανίζονται εξαιτίας των αλλαγών του configuration.

First touch policy: **Memory is mapped to the NUMA domain that first touches it.**

Συνεπώς αφού το master thread είναι αυτό που “ακουμπάει” πρώτο τον χώρο των local πινάκων (αφού η calloc όχι μόνο δεσμεύει χώρο αλλά γράφει και σε αυτόν μηδενικά), αυτά τοποθετούνται στη δικιά του μνήμη. Αυτό αυξάνει το access time των υπολοίπων threads στα local arrays που τους αναλογούν. Για να το διορθώσουμε αυτό θέλουμε κάθε thread να είναι το πρώτο που “ακουμπάει” το array του:

```
//first touch: allocate the data page in the memory closest to the thread accessing this page for the first time
//since we have nthreads and each one gets its own local arrays , we want each thread to initialize its own array
//so we make this loop parallel
#pragma omp parallel for
for (k=0; k<nthreads; k++)
{
    local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters,
sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords,
sizeof(**local_newClusters));
}
```

Αυτό λύνει το πρόβλημα που δημιουργούσε το first touch.

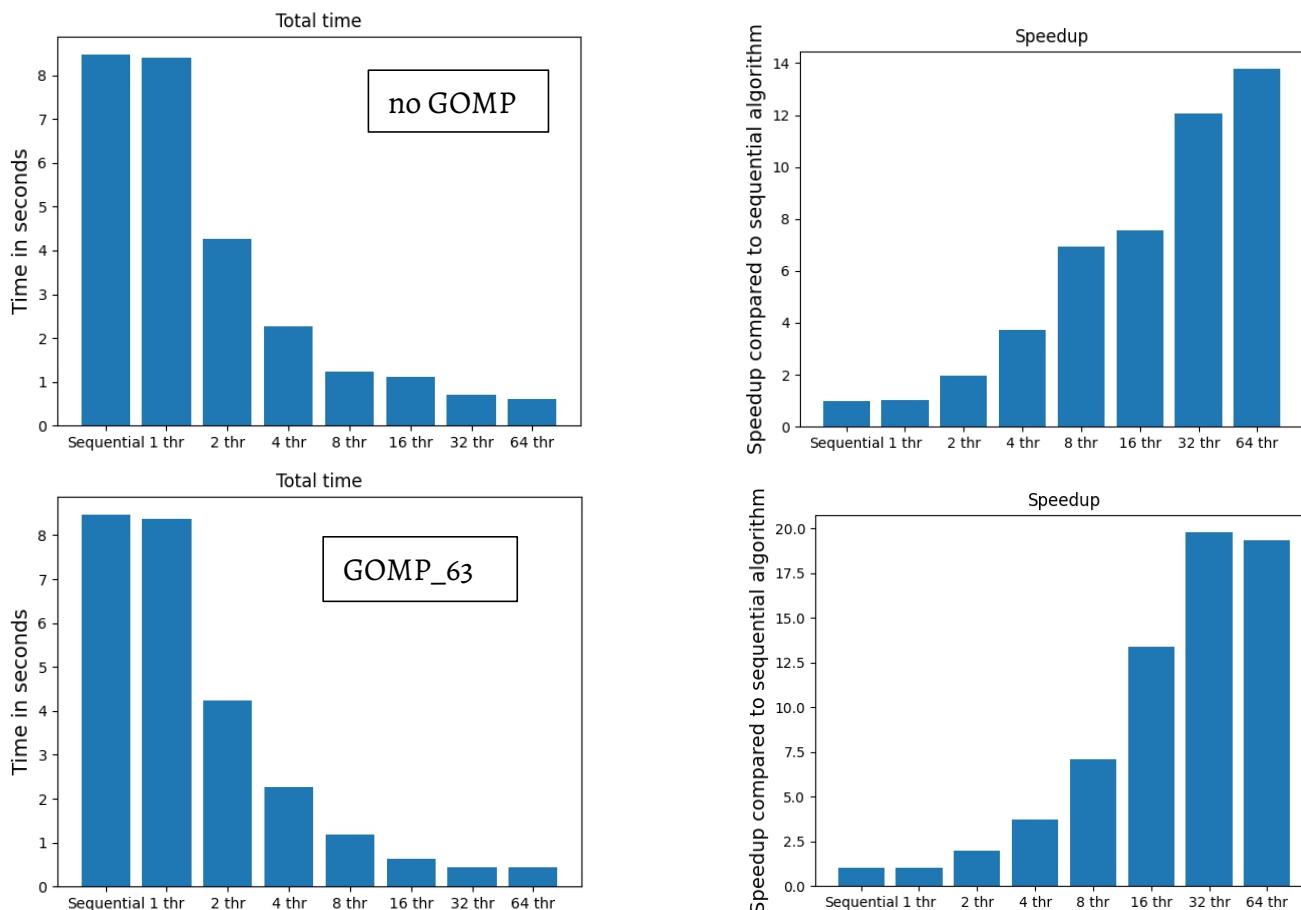
Τώρα ψάχνουμε false sharing που μπορεί να έχει προκύψει στο πρόγραμμα.

Διαβάζουμε στο διαδίκτυο ότι οι επεξεργαστές Intel Xeon E5-4620 που χρησιμοποιούνται στο sandman έχουν cache line ίση με 64B. Ο κάθε πίνακας local\_newClusterSize[nthreads] έχει μέγεθος numClusters και είναι τύπου int. Τα numClusters είναι 8 και το int έχει μέγεθος 4 bytes. Συνεπώς ένας τέτοιος πίνακας καταλαμβάνει χώρο ίσο με  $4^*8 = 32$ B. Αυτό είναι πρόβλημα καθώς δύο πίνακες local\_newClusterSize χωράνε στο ίδιο cache line π.χ ο local\_newClusterSize[0] και ο local\_newClusterSize[1]. Πρόκειται λοιπόν για μια ξεκάθαρη περίπτωση false sharing αφού το thread ο απαιτεί πρόσβαση μόνο στον πίνακα local\_newClusterSize[0] όμως όταν θέλει να τον φέρει στην cache του αναγκαστικά φέρνει το cache line που περιέχει και τον local\_newClusterSize[0] αλλά και τον local\_newClusterSize[1]. Αυτό δημιουργεί καθυστερήσεις λόγω του cache coherence protocol. Κάθε φορά που το thread ο αλλάζει την τιμή του local\_newClusterSize[0], το cache line του thread1 που περιέχει τον local\_newClusterSize[1] γίνεται invalidate και πρέπει να ανανεωθεί στην επόμενη πρόσβαση. Για να αποφύγουμε το πρόβλημα αυτό αλλάξαμε το type του local\_newClusterSize[nthreads] σε int64\_t. Έτσι κάθε πίνακας έχει μέγεθος numClusters\*8 = 64B δηλαδή ένα cache line.

Παρατηρούμε επίσης ότι δεν υπάρχει false sharing στους πίνακες local\_newClusters που είναι double (=8B) και διαστάσεων numClusters\*numCoords. Συνεπώς κάθε τέτοιος πίνακας απαιτεί χώρο numClusters\*numCoords\*8B =  $8^*1^*8B = 64B$  δηλαδή ακριβώς ένα cache line.

Οι νέοι χρόνοι είναι:

Program\Threads	1	2	4	8	16	32	64
omp_reduction_different_config	8.529s	7.9245s	5.78s	4.7179s	4.6402s	7.0824s	3.57s
omp_reduction_first_touch_false_sharing	8.3920s	4.2688s	2.2681s	1.2215s	1.1183s	0.7011s	0.6148s
omp_reduction_first_touch_false_sharing_63_GOMP	8.3640s	4.2468s	2.2594s	1.1917s	0.6320s	0.4279s	0.4377s



Μετά από τη διόρθωση που στάλθηκε θα τρέξουμε και το καινούργιο configuration {256,1,4,10}. Αρχικά θα σβήσουμε τις διορθώσεις που κάνουμε για το first touch policy και το false sharing. Περιμένουμε η κλιμάκωση του προγράμματός μας να είναι κάκιστη:

Program\Threads	1	2	4	8	16	32	64
omp_reduction_new_config	5.1887s	13.6155s	14.6321s	13.0046s	10.8415s	8.4366s	7.1399s

Ξανακάνουνε το loop της δέσμευσης χώρου παραλλήλο για να εκμεταλλευτούμε το first touch policy.

Πρέπει να σκεφτούμε και τα φαινόμενα false sharing. Το numClusters είναι πλέον 4. Συνεπώς έχουμε ξεκάθαρο false sharing στους πίνακες local\_newClusters οι οποίοι είναι τύπου double (8B) άρα ένας τέτοιος πίνακας  $\text{numClusters}^*\text{numCoords}^*8B = 4^*1^*8B = 32B$ . Δύο τέτοιοι πίνακες χωράνε σε ένα cache line. Για να το αποφύγουμε αυτό θα αλλάξουμε τον τύπο των πινάκων σε long double (=16B). Τότε το μέγεθος κάθε τέτοιου πίνακα είναι  $4^*16B = 64B$  δηλαδή ακριβώς ένα cache line. Πρέπει να μελετήσουμε και το false sharing στους πίνακες local\_newClusterSize που είναι τύπου int (=4B) και μεγέθους numClusters (=4) άρα κάθε τέτοιος πίνακας έχει μέγεθος 16B και 4 χωράνε στο ίδιο cache line. Θα θέλαμε να αλλάξουμε τον τύπο τους σε ένα int με 16 bytes όμως δεν βρήκαμε να υπάρχει τέτοιος τύπος στην C, για αυτό θα αλλάξουμε τον τρόπο με τον οποίο η calloc δεσμεύει τον χώρο για κάθε τέτοιο πίνακα:

```
local_newClusterSize[k] = calloc(numClusters, 16);
```

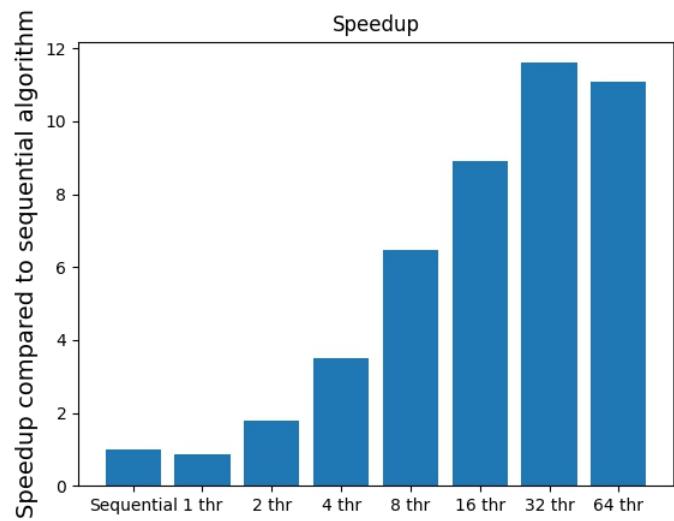
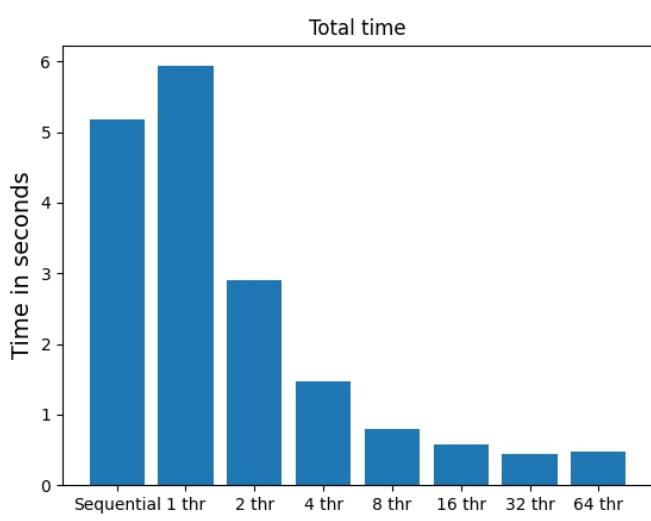
Έτσι για κάθε στοιχείο του πίνακα θα δεσμεύουμε 16 bytes και το συνολικό μέγεθος κάθε τέτοιου πίνακα θα είναι 64B αποφεύγοντας πλήρως το false sharing. Περιμένουμε και αυτό να μας δώσει μια μικρή βελτίωση παρότι δεν γίνονται πάρα πολλές προσβάσεις στους πίνακες local\_newClusterSize.

Τέλος, μιας και αξιοποιούμε το first touch policy, το loop της αρχικοποίησης των local πινάκων σε ο, θα παραλληλοποιηθεί ώστε κάθε thread να μηδενίσει τον δικό του πίνακα. Έτσι, δεν θα έχουμε άσκοπες μεταφορές από μακρινές μνήμες στις caches του master thread το οποίο ήταν εκείνο που αρχικοποιούσε πριν τους πίνακες στο ο. Επίσης, αυτό το loop θα λειτουργήσει σαν prefetching των πινάκων στις τοπικές caches των threads. Το loop στο οποίο αναφερόμαστε είναι το παρακάτω:

```
#pragma omp parallel for private(i,j)
shared(nthreads,numClusters,local_newClusterSize,numCoords,local_newClusters) default(none)
    for(k = 0; k < nthreads; ++k) {
        for(i = 0; i < numClusters; ++i) {
            local_newClusterSize[k][i] = 0;
            for(j = 0; j < numCoords; ++j) {
                local_newClusters[k][i*numCoords + j] = 0;
            }
        }
    }
}
```

Περιμένουμε λοιπόν μια μικρή βελτίωση και από εκεί. Στο προηγούμενο ερώτημα όπου δεν αξιοποιούσαμε το first touch policy αυτή η λούπα δεν είχε παραλληλοποιηθεί καθώς εξαιτίας του μικρού μεγέθους της ( $nthreads^*\text{numClusters}^*\text{numCoords}$ ) το overhead της δημιουργίας και επικοινωνίας των νημάτων δεν άξιζε την παραλληλοποίηση και είχαμε παρατηρήσει ελάχιστα μεγαλύτερους χρόνους όταν το είχαμε παραλληλοποιήσει. Οι τελικοί χρόνοι για το configuration {256,1,4,10} φαίνονται στην επόμενη σελίδα μαζί με τα διαγράμματα συνολικού χρόνου και speedup:

Program\Threads	1	2	4	8	16	32	64
omp_reduction_new_config	5.1887s	13.6155s	14.6321s	13.0046s	10.8415s	8.4366s	7.1399s
omp_reduction_new_config_first_touch_false_sharing	5.5238s	2.8783s	1.5291s	0.8218s	0.9792s	0.6716s	0.6192s
omp_reduction_new_config_first_touch_false_sharing_GOMP63	5.9338s	2.8994s	1.4769s	0.8010s	0.5815s	0.4459s	0.4675s



3.

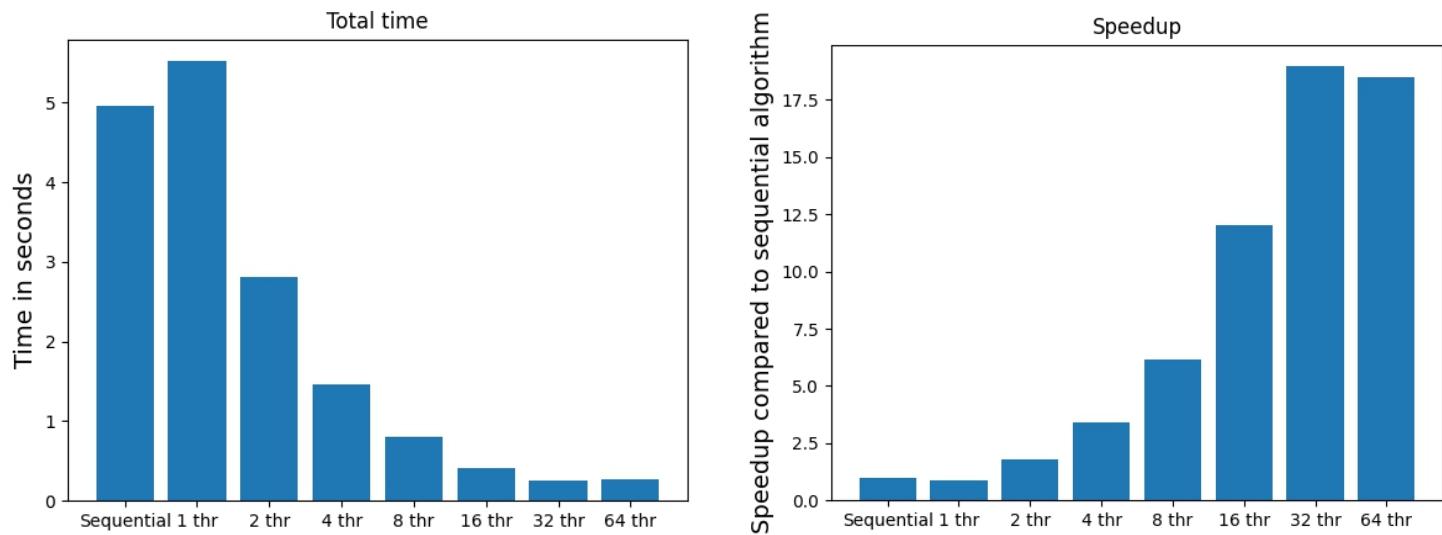
Παρατηρούμε ότι στον κώδικα που παραλληλοποιήσαμε, το threado είναι αυτό που θέλει πρόσβαση στα objects[i\*numCoords+j] για χαμηλά i. Στο αρχείο file\_io.c γίνεται αρχικοποίηση αυτών των objects. Λαμβάνοντας υπόψη τα NUMA χαρακτηριστικά του μηχανήματος, σκοπός μας είναι να τοποθετήσουμε τα κατάλληλα objects κοντά στα threads που θα τα χρησιμοποιήσουν. Με βάση την παραπάνω παρατήρηση και δεδομένου ότι τα υπόλοιπα threads θέλουν πρόσβαση στα objects με τη σειρά, παραλληλοποιούμε το for loop αρχικοποίησης που βλέπουμε στο αρχείο file\_io.c αφού προσθέσουμε και το header file του openMP:

```
#pragma omp parallel for private(j)
for (i=0; i<numObjs; i++)
{
    unsigned int seed = i;
    for (j=0; j<numCoords; j++)
    {
        objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) * val_range;
        if (_debug && i == 0)
            printf("object[i=%ld] [j=%ld]=%f\n", i, j, objects[i*numCoords + j]);
    }
}
```

Αρχικά δοκιμάζουμε το configuration {256,1,4,10}:

Program\Threads	1	2	4	8	16	32	64
omp_reduction_new_config_first_touch_false_sharing_GOMP63	5.9338s	2.8994s	1.4769s	0.8010s	0.5815s	0.4459s	0.4675s
omp_reduction_new_config_NUMA_aware	5.6848s	2.7780s	1.4707s	0.7610s	0.5640s	0.3780s	0.3646s
omp_reduction_new_config_NUMA_aware_GOMP	5.5210s	2.8055s	1.4669s	0.8068s	0.4129s	0.2617s	0.2683s

Πράγματι η κλιμάκωση είναι ακόμα καλύτερη.

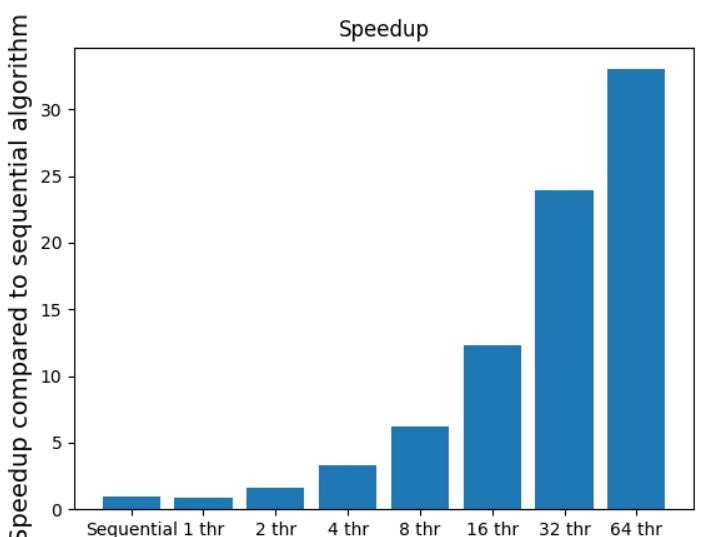
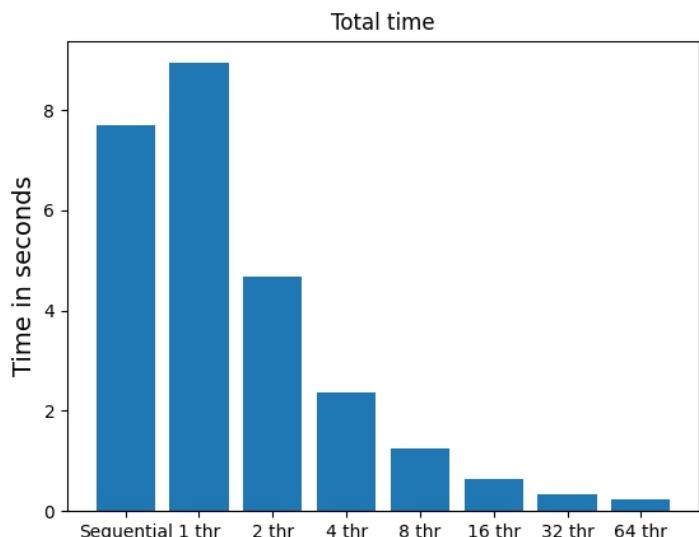


Παρατηρούμε ότι για αυτό το configuration δεν υπάρχει σοβαρή κλιμάκωση για nthreads > 32. Μάλιστα όταν χρησιμοποιούμε τη μεταβλητή περιβάλλοντος GOMP\_CPU\_AFFINITY τα 32 threads κάνουν λιγότερο χρόνο από τα 64. Ακόμα όμως και όταν δεν χρησιμοποιούσαμε την GOMP\_CPU\_AFFINITY, η βελτίωση μεταξύ 32 και 64 threads ήταν ελάχιστη.

Υποθέτουμε ότι το bottleneck της υλοποίησης αυτής είναι ότι δουλεύουμε μόνο σε μία διάσταση (numCoords = 1), συνεπώς, επειδή κάθε thread θέλει πρόσβαση στα objects[i\*numCoords] για να βρει το index αλλά και πιο κάτω στα objects[i\*numCoords + j] (το j “τρέχει” από 0 μέχρι numCoords) για να ανανεώσει τις τιμές στους local\_newClusters[k][index\*numCoords + j], υπάρχει μεγάλος διαμοιρασμός των πίνακα objects ο οποίος δημιουργεί καθυστερήσεις λόγω του cache coherence protocol. πχ αν το thread0 πάρει το i = 0 και το thread1 πάρει το i = 1, το thread0 θα απαιτεί πρόσβαση στα objects[0] και το thread1 στα objects[1], αυτά βρίσκονται στο ίδιο cache block (τα objects είναι type double) και έχουμε false sharing. Σε κάποια άλλη περίπτωση όπου το numCoords θα ήταν 8, ο διαμοιρασμός των objects στα cache block θα γινόταν με πολύ πιο αποδοτικό για την υλοποίησή μας τρόπο αφού τα objects[0,1,2,3,4,5,6,7] (τα οποία χρειάζεται το thread0) θα χωρούσαν σε ένα cache line ενώ τα objects[8,9,10,11,12,13,14,15] (τα οποία χρησιμοποιεί το thread1) θα βρισκόντουσαν σε διαφορετικό cache block και θα αποφεύγαμε το false sharing.

Για το αρχικό configuration {256, 16, 16, 10}:

Program\Threads	1	2	4	8	16	32	64
omp_reduction	7.8026s	4.2035s	2.3099s	1.2339s	0.8373s	0.5744s	0.5147s
omp_reduction_GOMP	8.0500s	4.0953s	2.1380s	1.1621s	0.6410s	0.3544s	0.2998s
omp_reduction_NUMA_aware	8.9238s	4.3571s	2.2666s	1.1615s	0.8880s	0.4742s	0.3217s
omp_reduction_NUMA_aware_GOMP	8.9390s	4.6766s	2.3686s	1.2378s	0.6290s	0.3228s	0.2336s



Σε αυτό το configuration δεν έχουμε false sharing στον πίνακα objects, όμως κάθε thread χρειάζεται πρόσβαση σε 2 cache lines από objects αφού numCoords = 16 και τα objects είναι τύπου double (=4B). Συνεπώς περιμένουμε να υπάρξουν capacity και conflict misses κατά τη διάρκεια του προγράμματος.

### 3. Αμοιβαίος Αποκλεισμός - Κλειδώματα

Σκοπός μας είναι να συγχρίνουμε και να αξιολογήσουμε διαφορετικούς τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό. Για να πετύχουμε αυτό θα χρησιμοποιήσουμε το κρίσιμο τμήμα της shared clusters υλοποίησης του αλγορίθμου K-means. Υπενθυμίζουμε ότι η δική μας υλοποίηση του κρίσιμου τμήματος ήταν:

```
/*
 * TODO: Detect parallelizable region and use appropriate OpenMP pragmas
 */
#pragma omp parallel for shared(numObjs, numClusters, numCoords, clusters, objects,
membership, newClusters, newClusterSize, delta) private(i, j, index)
for (i=0; i<numObjs; i++) {
    // find the array index of nearest cluster center
    index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);

    // if membership changes, increase delta by 1
    if (membership[i] != index)
        #pragma omp atomic
        delta += 1.0;

    // assign the membership to object i
    // no need for synchronization, as only one thread will modify membership[i]
    membership[i] = index;

    // update new cluster centers : sum of objects located within
    /*
     * TODO: protect update on shared "newClusterSize" array
     * DONE: We use the atomic pragma. Its more efficient than critical.
     * Critical is used for more complex (multiple lined) statements
     */
    #pragma omp atomic
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        /*
         * TODO: protect update on shared "newClusters" array
         * DONE: Usage of atomic pragma
         */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
}
}
```

Χρησιμοποιήσαμε αμοιβαίο αποκλεισμό και στην αλλαγή του delta καθώς όπως προαναφέραμε, με χρήση reduction δεν είδαμε ιδιαίτερη βελτίωση.

Στο αρχείο `omp_lock_kmeans.c` βλέπουμε την υλοποίηση όπου ο αμοιβαίος αποκλεισμός των threads επιτυγχάνεται με τις συνάρτησεις `lock_acquire()` και `lock_release()` αντί για pragmas:

```
#pragma omp parallel for \      private(i,j,index) \
firstprivate(numObjs,numClusters,numCoords) \
shared(objects,clusters,membership,newClusters,newClusterSize) \      schedule(static)
reduction(+:delta)

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

    // if membership changes, increase delta by 1
    if (membership[i] != index)
        delta += 1.0;
```

```

// assign the membership to object i
membership[i] = index;

// update new cluster centers : sum of objects located within
lock_acquire(lock);
newClusterSize[index]++;
for (j=0; j<numCoords; j++) {
    newClusters[index*numCoords + j] += objects[i*numCoords + j];
}
lock_release(lock);
}

```

Στο directory locks , μας δίνονται οι υλοποιήσεις των κλειδωμάτων. Το κάθε κλειδωμα ορίζει τις συναρτήσεις lock\_acquire και lock\_release ανάλογα με τη λειτουργία του. Έτσι μέσω του Makefile δημιουργούνται τα εκτελέσιμα αρχεία και το καθένα χρησιμοποιεί διαφορετικό lock.

Θα συγκρίνουμε 9 διαφορετικές υλοποιήσεις:

- τον δικό μας κώδικα , τον οποίο θα αποκαλούμε naive υλοποίηση (χρησιμοποιήσαμε pragma omp atomic)
- omp\_critical\_kmeans(χρησιμοποιείται pragma omp critical αντί των συναρτήσεων lock\_acquire και lock\_release)
- nosync\_lock : Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα. Ωστόσο, θα χρησιμοποιηθεί ως άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων.
- pthread\_mutex\_lock: To pthread\_mutex\_t κλειδωμα που παρέχει η βιβλιοθήκη Pthreads.
- pthread\_spin\_lock: To pthread\_spinlock\_t κλειδωμα που παρέχει η βιβλιοθήκη Pthreads.
- tas\_lock: To test-and-set κλειδωμα όπως περιγράφεται στις διαφάνειες του μαθήματος.
- ttas\_lock: To test-and-test-and-set κλειδωμα όπως περιγράφεται στις διαφάνειες του μαθήματος.
- array\_lock: To array-based κλειδωμα όπως περιγράφεται στις διαφάνειες του μαθήματος.
- clh\_lock: Ένα είδος κλειδώματος που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας.

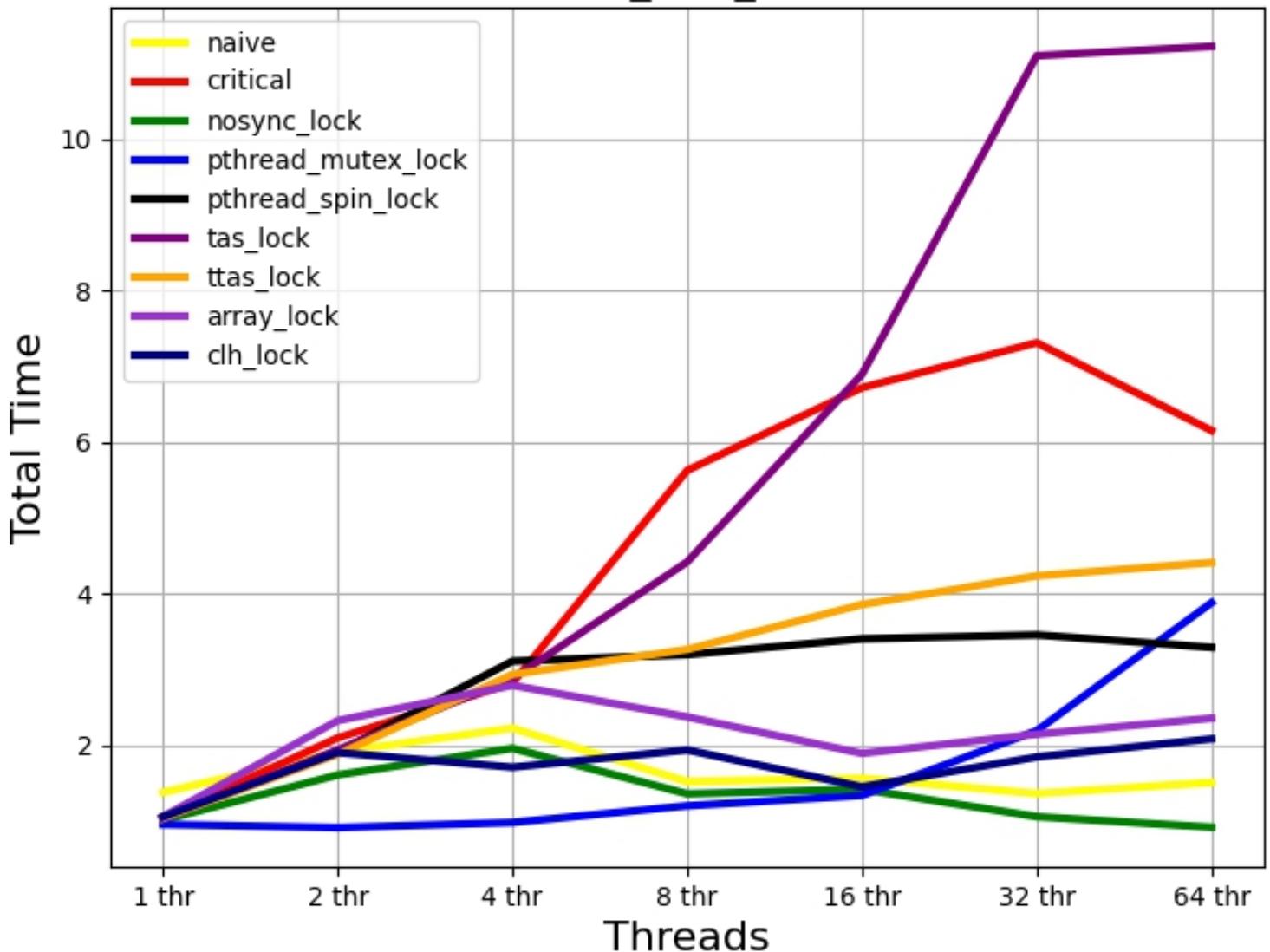
Οι μετρήσεις μας έγιναν με το configuration {Size, Coords, Clusters, Loops} = {32, 16, 16, 10} για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχάνημα sandman. Όπως έχουμε αναφέρει πιο πάνω, στην υλοποίηση shared clusters πετύχαμε τους καλύτερους χρόνους μας για GOMP\_CPU\_AFFINITY = “0-7 32-39” δηλαδή δεσμεύοντας μόνο το πρώτο node του sandman. Θα κάνουμε μετρήσεις χωρίς την χρήση της μεταβλητής περιβάλλοντος GOMP\_CPU\_AFFINITY , με GOMP\_CPU\_AFFINITY = “0-7 32-39” αλλά και με GOMP\_CPU\_AFFINITY = “0-63” (δεσμεύονται όλους τους πυρήνες και όλα τα λογικά threads). Επισημαίνεται ότι για οι υλοποιήσεις array\_lock και clh\_lock δεν τερμάτιζαν για GOMP\_CPU\_AFFINITY = “0-7 32-39” οπότε χρησιμοποιήσαμε μόνο GOMP\_CPU\_AFFINITY = “0-63” για αυτές τις δύο.

Στην επόμενη σελίδα φαίνονται τα αποτελέσματα των μετρήσεών μας.

### Χωρίς την χρήση της μεταβλητής περιβάλλοντος GOMP\_CPU\_AFFINITY:

Program\Threads	1	2	4	8	16	32	64
naive	1.3732s	1.9033s	2.2215s	1.5155s	1.5614s	1.3585s	1.5041s
omp_critical_kmeans	1.0454s	2.0926s	2.8148s	5.6227s	6.7139s	7.3075s	6.1486s
nosync_lock	1.0060s	1.6033s	1.9549s	1.3548s	1.4147s	1.0527s	0.9133s
pthread_mutex_lock	0.9506s	0.9087s	0.9757s	1.1960s	1.3325s	2.1875s	3.8771s
pthread_spin_lock	1.0354s	1.9031s	3.1052s	3.1913s	3.4001s	3.4518s	3.2874s
tas_lock	1.0275s	1.9356s	2.8752s	4.4153s	6.8905s	11.0949s	11.2178s
ttas_lock	1.0309s	1.8746s	2.9300s	3.2595s	3.8526s	4.2312s	4.4066s
array_lock	1.0515s	2.3203s	2.7862s	2.3695s	1.8888s	2.1420s	2.3533s
clh_lock	1.0498s	1.8993s	1.7054s	1.9333s	1.4471s	1.8411s	2.0802s

No GOMP\_CPU\_AFFINITY

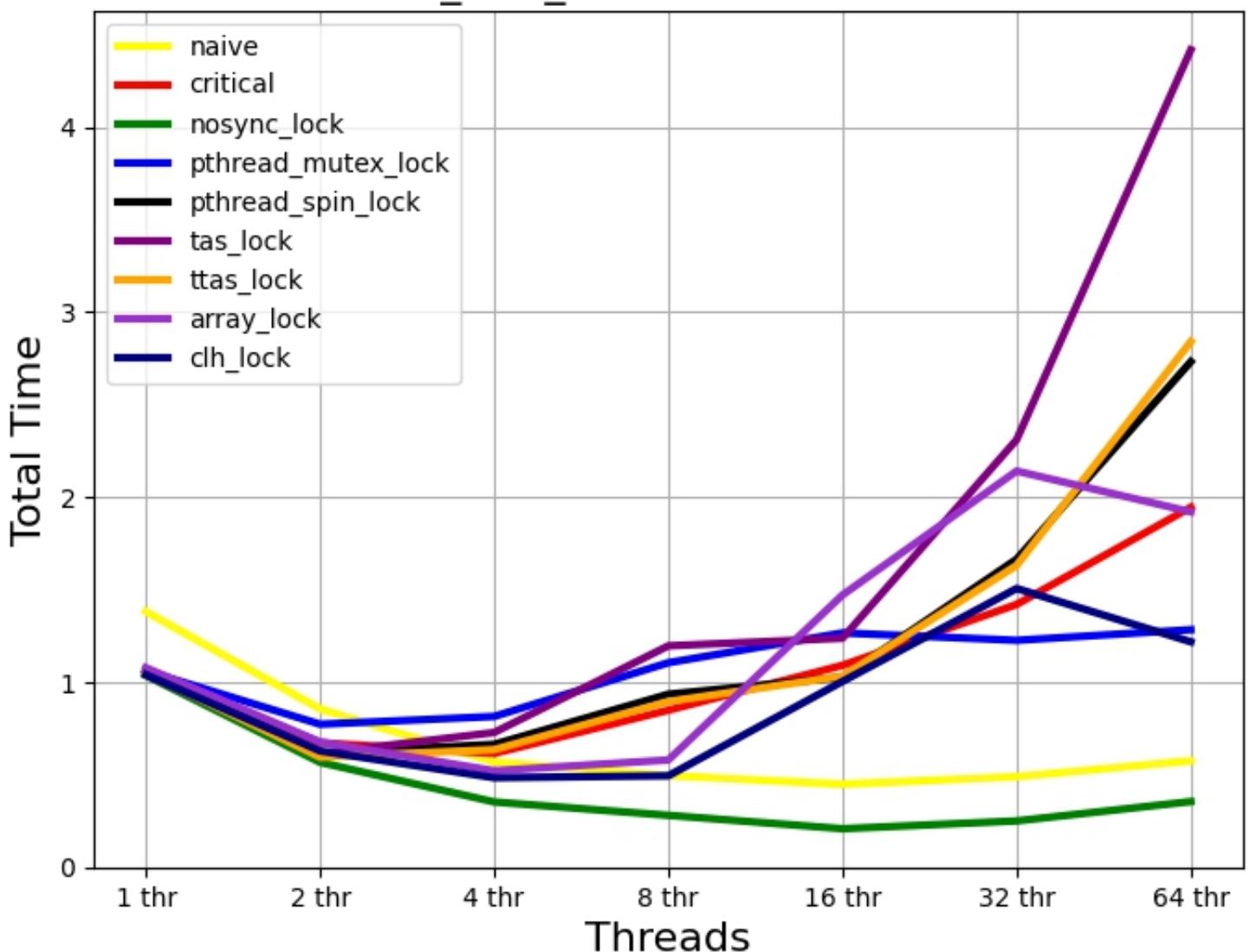


Νικητής (πέρα από το nosync) είναι το naive (η δικιά μας υλοποίηση), ακολουθούν το clh που δεν έχει αύξηση χρόνου όσο αυξάνονται τα threads, το array, το spinlock που ενώ αρχικά είχε μεγάλη αύξηση χρόνου εν τέλει σταθεροποιήθηκε για threads > 4 και το mutex που ενώ ξεκίνησε πολύ καλά για λίγα threads, είχε μεγάλη αύξηση χρόνου για 32 και 64.

Με GOMP\_CPU\_AFFINITY = "0-7 32-39" (0-63 για το array\_lock και clh\_lock):

Program\Threads	1	2	4	8	16	32	64
naive	1.3834s	0.8574s	0.5704s	0.4970s	0.4509s	0.4911s	0.5770s
omp_critical_kmeans	1.0640s	0.6746s	0.6170s	0.8503s	1.0927s	1.4216s	1.9463s
nosync_lock	1.0380s	0.5689s	0.3548s	0.2826s	0.2102s	0.2524s	0.3569s
pthread_mutex_lock	1.0632s	0.7745s	0.8166s	1.1068s	1.2678s	1.2278s	1.2851s
pthread_spin_lock	1.0650s	0.6185s	0.6658s	0.9366s	1.0286s	1.6658s	2.7326s
tas_lock	1.0439s	0.6169s	0.7300s	1.1980s	1.2393s	2.3115s	4.4167s
ttas_lock	1.0524s	0.5971s	0.6374s	0.8973s	1.0343s	1.6324s	2.8407s
array_lock	1.0798s	0.6791s	0.5233s	0.5816s	1.4724s	2.1421s	1.9218s
clh_lock	1.0426s	0.6305s	0.4856s	0.4963s	1.0057s	1.5078s	1.2198s

GOMP\_CPU\_AFFINITY="0-7 32-39"

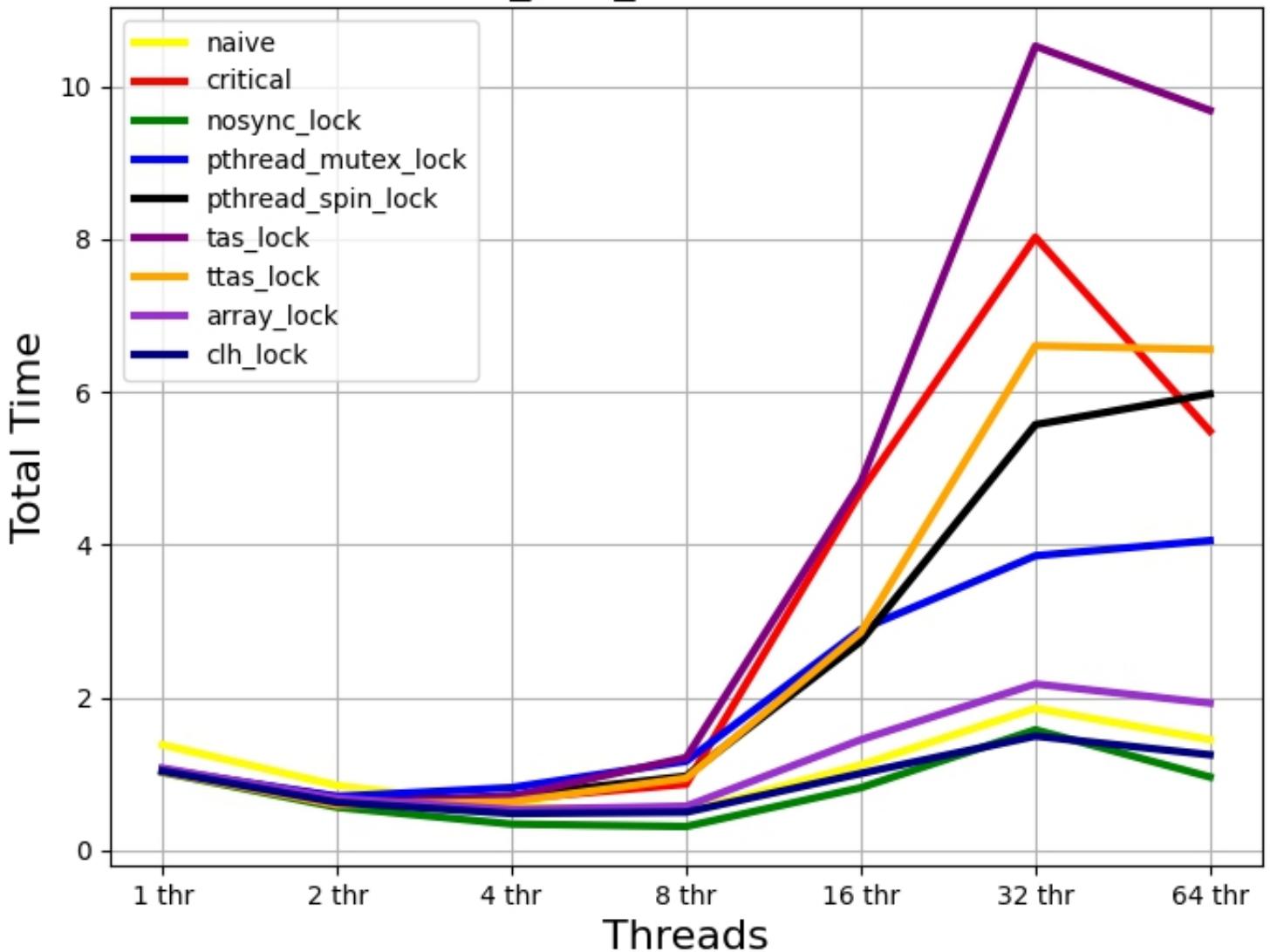


Νικητής είναι το naive, οι υπόλοιπες υλοποιήσεις είναι χειρότερες με το clh να ζεκινάει σαν το naive αλλά να τερματίζει σαν το mutex, το critical να είναι λίγο καλύτερο των ttas και spinlock για πολλά threads και το tas να είναι ξανά το χειρότερο.

Mε GOMP\_CPU\_AFFINITY = "0-63":

Program\Threads	1	2	4	8	16	32	64
naive	1.3825s	0.8483s	0.5782s	0.5064s	1.1173s	1.8597s	1.4490s
omp_critical_kmeans	1.0440s	0.7034s	0.6692s	0.8660s	4.7000s	8.0265s	5.4882s
nosync_lock	1.0216s	0.5642s	0.3443s	0.3128s	0.8200s	1.5780s	0.9607s
pthread_mutex_lock	1.0604s	0.7059s	0.8217s	1.1675s	2.8902s	3.8567s	4.0547s
pthread_spin_lock	1.0377s	0.6086s	0.7125s	0.9711s	2.7358s	5.5721s	5.9728s
tas_lock	1.0455s	0.6029s	0.7079s	1.2140s	4.8192s	10.5272s	9.6853s
ttas_lock	1.0309s	0.5953s	0.6371s	0.9448s	2.8528s	6.6050s	6.5566s
array_lock	1.0766s	0.6796s	0.5362s	0.5779s	1.4485s	2.1779s	1.9280s
clh_lock	1.0389s	0.6277s	0.4825s	0.5001s	1.0068s	1.4959s	1.2512s

GOMP\_CPU\_AFFINITY="0-63"



Νικητής το clh , ακολουθούν κοντά του το naive και το array. Το tas είναι το χειρότερο και πάλι.

Αφού είδαμε τα αποτελέσματα των πειραμάτων, μπορούμε να εξάγουμε συμπεράσματα για την επίδοση κάθε κλειδώματος σχολιάζοντας και την υλοποίησή του.

- Για το naive (δικιά μας υλοποίηση):

Παρατηρούμε ότι η δικιά μας υλοποίηση είναι αρκετά καλή και είναι σε όλα τα πειράματα πρώτη ή πάρα πολύ κοντά στην κορυφή. Αυτό συμβαίνει για 2 σημαντικούς λόγους:

- 1) Χρησιμοποιήσαμε `pragma omp atomic` που υλοποιείται με ατομικές εντολές σε επίπεδο hardware, γεγονός που το καθιστά πιο γρήγορο και αποδοτικό.
- 2) Με την χρήση του `pragma omp atomic` επιβάλλουμε ατομικότητα μόνο για συγκεκριμένες θέσεις μνήμης, δίδοντας τη δυνατότητα σε διαφορετικά threads να τροποποιούν διαφορετικές θέσεις της ίδιας δομής δεδομένων χωρίς επιπρόσθετο synchronization overhead.

(Για παράδειγμα `#pragma omp atomic` το ένα thread μπορεί να τροποποιήσει τη θέση `newClusterSize[0]` `newClusterSize[index]++;` και το άλλο την `newClusterSize[12]` χωρίς να παρεμβάλλονται μεταξύ τους.)

Αντιθέτως, στην υλοποίηση των locks όλο αυτό έχει γίνει ένα μεγάλο κρίσιμο τμήμα, μειώνοντας τον παραλληλισμό και αυξάνοντας το κόστος συγχρονισμού μεταξύ των νημάτων.

- Για το nosync\_lock:

Δεν προσφέρει αμοιβαίο αποκλεισμό συνεπώς δεν έχουμε κάτι να σχολιάσουμε. Η υλοποίησή του είναι κενή και οι συναρτήσεις του δεν τρέχουν καθόλου κώδικα. Είναι απολύτως φυσιολογικό να είναι αυτό που είχε πάντα τους καλύτερους χρόνους αφού ο αμοιβαίος αποκλεισμός συνεπάγεται καθυστέρηση.

- Για το critical:

Το critical προκαλεί χειροτέρευση του χρόνου καθώς αυξάνονται τα threads αφού δημιουργείται ένα bottleneck σε εκείνο το κομμάτι στο οποίο θα εισέλθει όποιο thread προλάβει. Η υλοποίηση critical είναι συνήθως προτελευταία στα πειράματά μας με μόνη χειρότερη την tas. Τυπικά το directive critical υλοποιείται με τη χρήση mutexes, γεγονός που το καθιστά πολύ πιο αργό σε σύγκριση με το # pragma `omp atomic`.

- Για το pthread\_mutex\_lock:

Πρόκειται για το κλασικό κλείδωμα `pthread_mutex_t` `mutex` της βιβλιοθήκης `pthreads` που είχαμε χρησιμοποιήσει και στο mandelbrot του μαθήματος του 6ου εξαμήνου. Η `lock_init` του καλεί την `pthread_mutex_init`, η `lock_free` την `pthread_mutex_destroy`, η `lock_acquire` την `pthread_mutex_lock` και η `lock_release` την `pthread_mutex_unlock`. Όταν ένα thread προσπαθήσει να κλειδώσει το mutex αλλά αποτύχει (γιατί είναι ήδη κλειδωμένο από άλλο thread), θα κάνει `sleep` αφήνοντας τον επεξεργαστή ελεύθερο για ένα άλλο thread να τρέξει. Συνεπώς τα mutexes συμπεριλαμβάνουν context switch. Όταν το mutex απελευθερωθεί, το thread θα ξυπνήσει και θα προσπαθήσει να το κλειδώσει. Όπως γνωρίζουμε, το να βάζουμε threads σε `sleep` και να τα ξυπνάμε είναι μια αργή και ακριβή διαδικασία και αν το κρίσιμο τμήμα είναι μικρό, ο χρόνος στον οποίο τα υπόλοιπα threads κοιμόντουσαν + τον χρόνο για να ξυπνήσουν μπορεί να είναι πολύ μεγαλύτερος από τον χρόνο εκτέλεσης του κρίσιμου τμήματος. Όταν χρησιμοποιούμε τη μεταβλητή `GOMP_CPU_AFFINITY`, το `mutex_lock` συμπεριφέρεται πολύ καλύτερα τον spinlock το οποίο σχολιάζουμε στη συνέχεια.

## ● Για το pthread\_spin\_lock:

Πρόκειται για το κλασικό κλειδωμα pthread\_spinlock\_t της βιβλιοθήκης pthreads και χρησιμοποιεί τις γνωστές σε μας συναρτήσεις pthread\_spin\_init, pthread\_spin\_destroy, pthread\_spin\_lock, pthread\_spin\_unlock . Η διαφορά spinlocks και mutexes έγκειται στο γεγονός ότι σε αντίθεση με το mutex, όταν ένα thread προσπαθήσει να κλειδώσει ένα ήδη κλειδωμένο spinlock , δεν θα κάνει sleep αλλά θα συνεχίσει να κάνει spin προσπαθώντας να αποκτήσει το lock μέχρι να το αποκτήσει (ή να αφαιρεθεί βίαια από τον επεξεργαστή από το λειτουργικό σύστημα όταν ξεπεράσει το κβάντο χρόνου που του αναλογεί). Αυτό έχει το θετικό ότι δεν κοιμίζουμε τα threads και έτσι δεν χάνουμε χρόνο στη διαδικασία ύπνου-ξυπνήματος αλλά έχει το αρνητικό ότι το thread κάνει spin για να αποκτήσει ένα lock το οποίο μπορεί να αργήσει να αφεθεί ελεύθερο και συνεπώς αφήνουμε κύκλους CPU να πάνε χαμένοι οι οποίοι θα μπορούσαν να χρησιμοποιηθούν από κάποιο άλλο thread που θα έτρεχε διαφορετικό κώδικα. Στην περίπτωση όπου CPU\_GOMP\_AFFINITY="0-63" δεν έχουμε τέτοιο πρόβλημα αφού έχουμε μέγιστο αριθμό threads = 64 και λογικές θέσεις ακριβώς τόσες οπότε ναι μεν τα threads κάνουν spin αλλά δεν θα μπορούσε κάποιο άλλο να πάρει τη θέση τους. Στην περίπτωση όμως όπου CPU\_GOMP\_AFFINITY="0-7 32-39" έχουμε σίγουρα τέτοια φαινόμενα αφού έχουμε μόνο 16 διαθέσιμες θέσεις συνεπώς για 32 και 64 threads, χάνουμε κύκλους CPU όταν ένα νήμα κάνει spin ενώ κάποιο άλλο που έχει αργήσει και δεν έχει φτάσει στο κρίσιμο τμήμα θα μπορούσε να τους χρησιμοποιήσει για πρόσδο. Λόγω των φαινομένων cache misses κ.λπ που προκύπτουν με τα διαφορετικά CPU\_GOMP\_AFFINITY αυτό που περιγράφουμε δεν φαίνεται στα διαγράμματα αλλά είναι σίγουρα ένας από τους λόγους όπου το spinlock πετυχαίνει χειρότερους χρόνους από το mutex στην περίπτωση CPU\_GOMP\_AFFINITY="0-7 32-39". Αυτό μάλλον μας οδηγεί στο συμπέρασμα ότι το κρίσιμο τμήμα μας είναι αρκετά μεγάλο ώστε να αξίζει τον χαμένο χρόνο από τον ύπνο των threads κατά τη χρήση του mutex κλειδώματος.

## ● Για το tas\_lock:

tas = Test And Set. Πρόκειται για έναν τύπο κλειδώματος όπου ενεργεί σε δύο μεταβλητές , την State και την Test. Όταν ένα thread θέλει να εισέλθει στο κρίσιμο τμήμα, θέτει ατομικά το State = 1 (κλειδωμένο) και μεταφέρει την προηγούμενη κατάσταση στη μεταβλητή Test. Διαβάζοντας την τιμή του Test καταλαβαίνει ότι εάν Test = 1 (προηγούμενη κατάσταση κλειδωμένη) τότε κάποιο άλλο thread είναι στην χρήσιμη περιοχή και δεν μπορεί να μπει και αυτό. Αντιθέτως εάν Test = 0 (προηγούμενη κατάσταση ξεκλείδωτη) τότε μπορεί να εισέλθει στο critical section. Πρόκειται για έναν απλό τύπου κλειδώματος βελτίωση του οποίου είναι το ttas που θα δούμε παρακάτω. Αυτή υλοποίηση έχει ως πρόβλημα την υπερβολική χρήση του διαδρόμου λόγω του cache coherence protocol αφού τα threads θέτουν συνεχώς τη μεταβλητή State ακόμα και όταν είναι κλειδωμένη με αποτέλεσμα να κάνουν invalidate τα blocks στις caches των υπολοίπων επεξεργαστών. Όσα περισσότερα threads και όσους περισσότερους ενεργούς πυρήνες έχουμε , η κατάσταση γίνεται όλο και πιο χαοτική λόγω της συνεχόμενης χρήσης του bus για cache invalidations. Αυτό είναι ξεκάθαρο και από τα διαγράμματα αφού είναι μακράν το χειρότερο lock όταν αυξάνεται ο αριθμός των threads.

## ● Για το ttas\_lock:

ttas = Test and Test And Set. Πρόκειται για βελτίωση του Test And Set κατά την οποία όσο το State είναι κλειδωμένο απλά το διαβάζουμε συνεχώς μέχρι να δούμε ότι ελευθερώθηκε. Όταν ελευθερωθεί θα γίνει διεκδίκηση θέτοντας το State = 1 και ακολουθώντας την ίδια ακριβώς διαδικασία με το TAS. Όσο λοιπόν το State είναι κλειδωμένο, δεν γράφουμε σε αυτό δημιουργώντας κίνηση στο bus αλλά μόνο διαβάζουμε για να μην προκαλέσουμε invalidations. Ουσιαστικά δεν διεκδικούμε το lock όσο είναι κλειδωμένο γιατί δεν πρόκειται να το πάρουμε αλλά το διεκδικούμε μόνο όταν αφεθεί ελεύθερο σε αντίθεση με το TAS όπου το lock διεκδικείται είτε είναι κλειδωμένο είτε όχι. Η υπεροχή του έναντι του tas είναι ξεκάθαρη στα διαγράμματα.

- Για το array\_lock:

Σε αυτόν τον τύπο κλειδώματος έχουμε έναν κυκλικό πίνακα μεγέθους ίσο με τον αριθμό των threads, έναν δείκτη στο τέλος της ουράς και μια τοπική μεταβλητή ανά thread. Όταν ένα thread θελήσει να κλειδώσει το lock, πιάνει θέση προτεραιότητας στον πίνακα, την επόμενη από το tail και περιμένει αυτή να γίνει true. Όταν ένα thread αφήσει το lock του, αλλάζει την τιμή του slot του στον πίνακα από true σε false και κάνει την επόμενη τιμή του πίνακα ίση με true, δηλαδή ενημερώνει το επόμενο thread ότι μπορεί να εισέλθει στο κρίσιμο τμήμα. Αυτό το κλειδωμα είναι δίκαιο και αποδοτικό αφού δεν εμπλέκεται σε μεγάλο βαθμό το cache coherence protocol (πέρα από την εγγραφή στον πίνακα προτεραιότητας) αφού το spin γίνεται σε διαφορετικές διευθύνσεις (διαφορετικές θέσεις του πίνακα) σε αντίθεση με τα δύο προηγούμενα locks. Δεν είναι λοιπόν τυχαίο που έχει πολύ καλή επίδοση όταν CPU\_GOMP\_AFFINITY="0-63".

- Για το clh\_lock:

Πρόκειται για κλειδωμα που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας. Η έμπνευση για να χρησιμοποιήσουμε συνδεδεμένη λίστα είναι ότι ένα thread μπορεί να μάθει εάν είναι η σειρά του να εισέλθει στο κρίσιμο τμήμα ελέχγοντας εάν το προγούμενο έχει τελείωσει. Το spin γίνεται σε διαφορετικό location. Επίσης όπως και πριν έχουμε FIFO δικαιότητα. Παρατηρούμε ότι στο προηγούμενο lock χρειαζόμαστε έναν πίνακα μεγέθους ίσο με τον αριθμό των threads για κάθε lock, αυτό δεν είναι αποδοτικά χωρικά. Εδώ, καταγράφουμε το status κάθε thread σε ένα object, αν η boolean τιμή είναι true τότε το thread είτε έχει το lock είτε το περιμένει ενώ αν είναι false το έχει απελευθερώσει. Το lock είναι ουσιαστικά μια linked list αντών των objects όπου το κάθε thread ξέρει τον πρόγονό του με μια τοπική μεταβλητή ενώ κρατάμε και μια εγγραφή για το ποιο είναι το node που προστέθηκε τελευταίο στη λίστα. Εάν ένα thread θέλει να αποκτήσει το lock θέτει την boolean τιμή του object του σε true και γίνεται αυτό το tail της συνδεδεμένης λίστας ενώ κρατάει το παλιό tail ως πρόγονό του. Υστερα κάνει spin στη boolean τιμή του προγόνου του μέχρι εκείνος να ελευθερώσει το lock (να θέσει τη μεταβλητή του ίση με false). Η διαφορά με το προγούμενο lock είναι ότι μπορεί να χρησιμοποιήσει το node του προγόνου του για μελλοντικές accesses σε locks αφού ο πρόγονός του έχει πλέον ελευθερώσει το προηγούμενο node του. Έχουμε έτσι τα θετικά του array\_lock (ότι κάνουμε spin σε διαφορετικά locations και ότι τα invalidation είναι ελάχιστα) και έχουμε και καλύτερη αξιοποίηση του χώρου (λόγω της επαναχρησιμοποίησης του node του προγόνου). Αυτό το lock έχει πράγματι λίγο καλύτερα αποτελέσματα από το array\_lock σε όλα τα πειράματά μας.

## 2.2 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου Floyd-Warshall

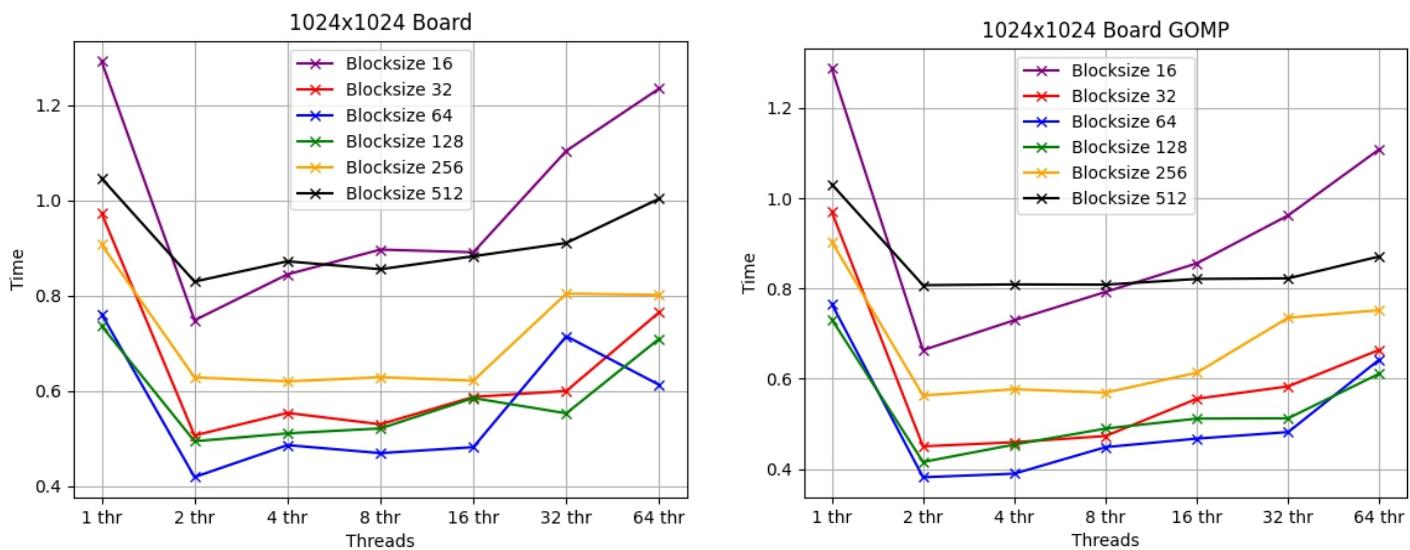
Σκοπός μας είναι να παραλληλοποιήσουμε την αναδρομική υλοποίηση του αλγορίθμου Floyd-Warshall. Για να το πετύχουμε αυτό απαιτείται να χρησιμοποιήσουμε tasks.

Παρακάτω φαίνεται ο κώδικας του οποίου παραλληλοποιήσαμε:

```
if (myN<=bsize)
    for (k=0; k<myN; k++)
        for (i=0; i<myN; i++)
            for (j=0; j<myN; j++)
                A[arow+i][acol+j]=min(A[arow+i][acol+j],
B[brow+i][bcol+k]+C[crow+k][ccol+j]);
    else {
        FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
        #pragma omp parallel
        {
            #pragma omp single
            {
                #pragma omp task shared(A, B, C)
                FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2,
myN/2, bsize);
                FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2,
bsize);
                #pragma omp taskwait
            }
        }
        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2,
bsize);
        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
ccol+myN/2, myN/2, bsize);
        #pragma omp parallel
        {
            #pragma omp single
            {
                #pragma omp task shared(A, B, C)
                FW_SR(A,arow+myN/2, acol,B,brow+myN/2,
bcol+myN/2,C,crow+myN/2, myN/2, bsize);
                FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2,
ccol+myN/2, myN/2, bsize);
                #pragma omp taskwait
            }
        }
        FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
    }
```

Για να μετρήσουμε την κλιμάκωση του προγράμματός μας, θα πραγματοποιήσουμε μετρήσεις για πίνακες 1024x1024, 2048x2048, 4096x4096, threads = {1, 2, 4, 8, 16, 32, 64} και διάφορα block sizes.

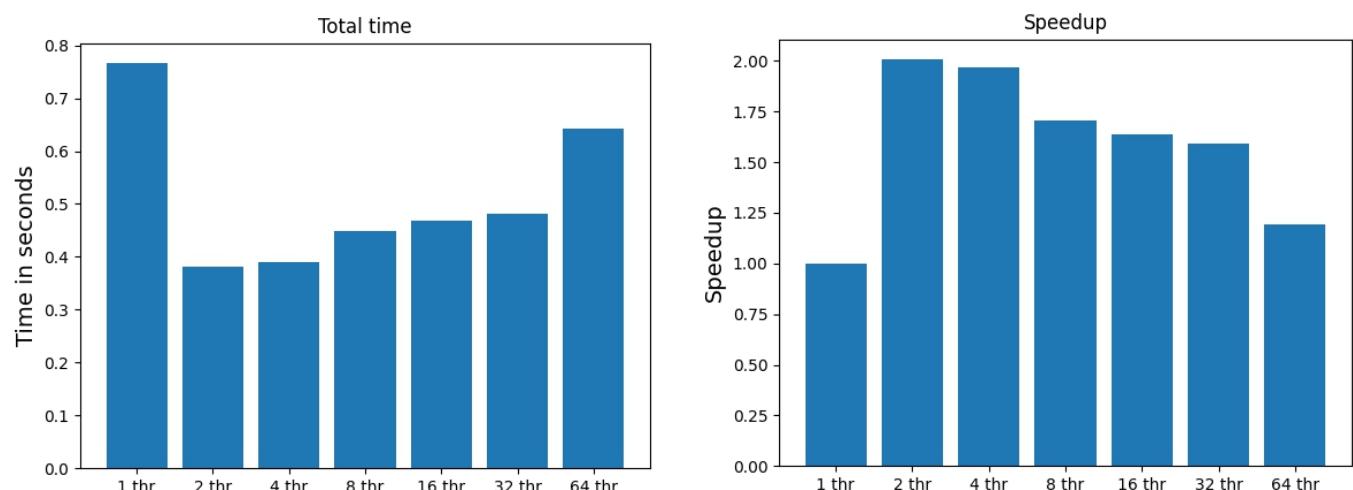
## 1024x1024:



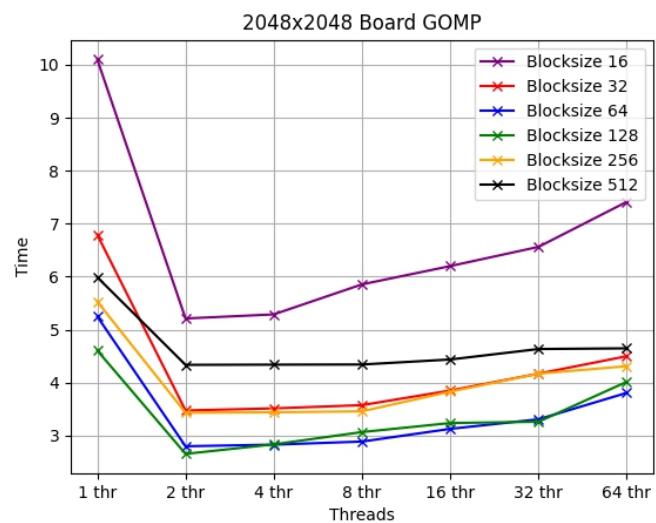
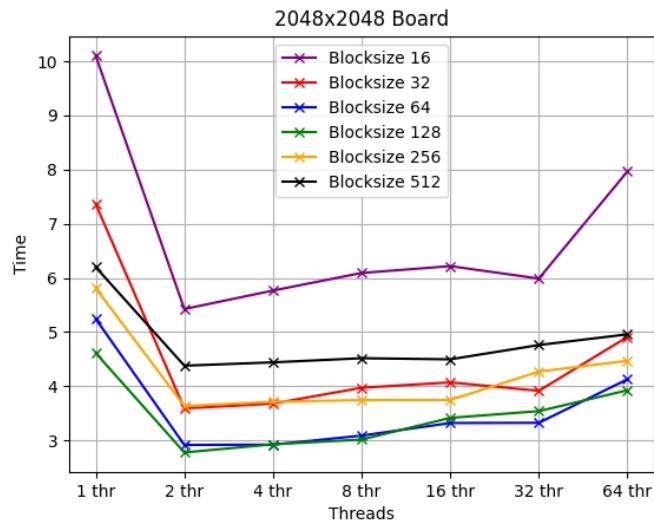
Εδώ φαίνεται ότι το καλύτερο blocksize είναι το 64.

Θα δημιουργήσουμε τα απαιτούμενα barplots με βάση αυτό.

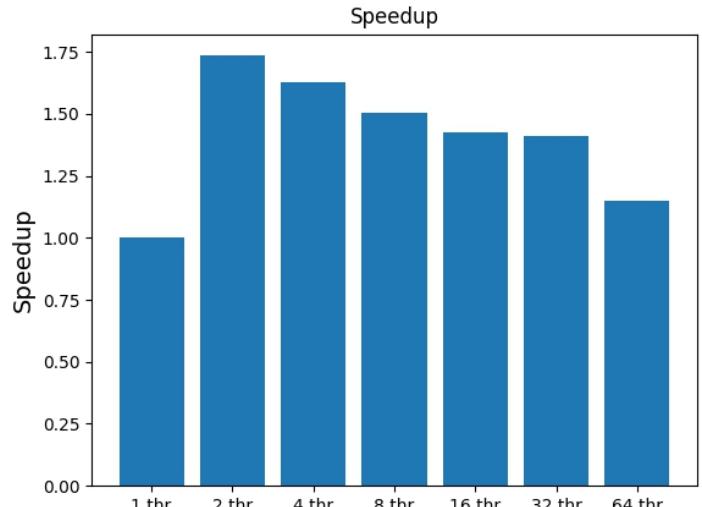
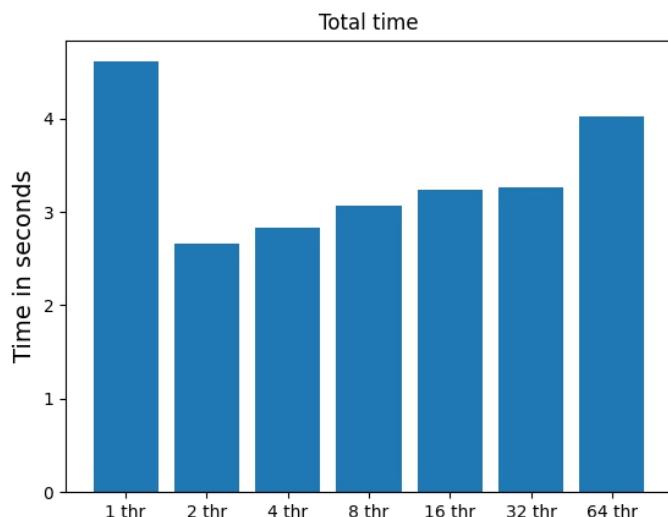
Τα απαιτούμενα barplots (για το speedup, χρησιμοποιήσαμε ως βάση τον αλγόριθμο με 1 thread):



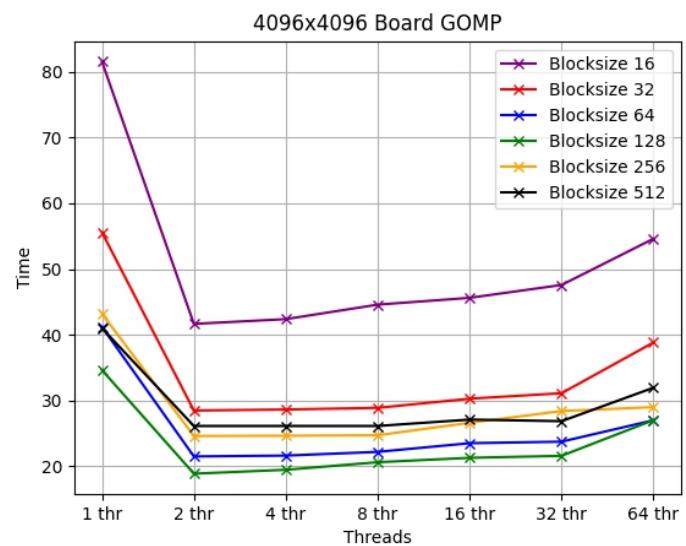
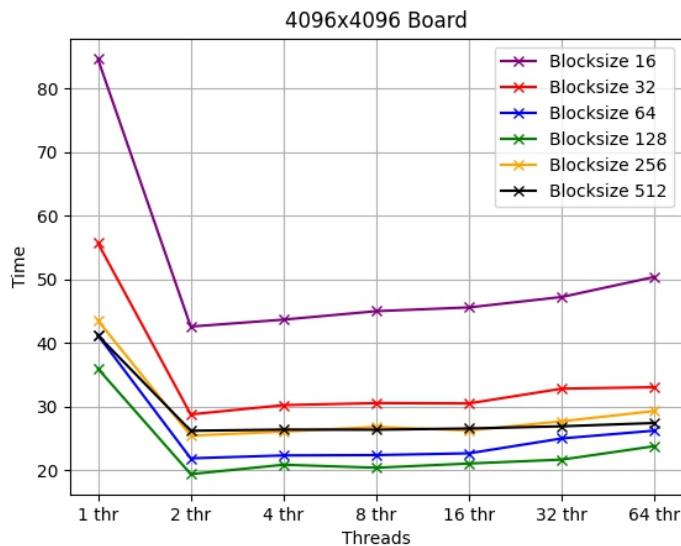
## 2048x2048:



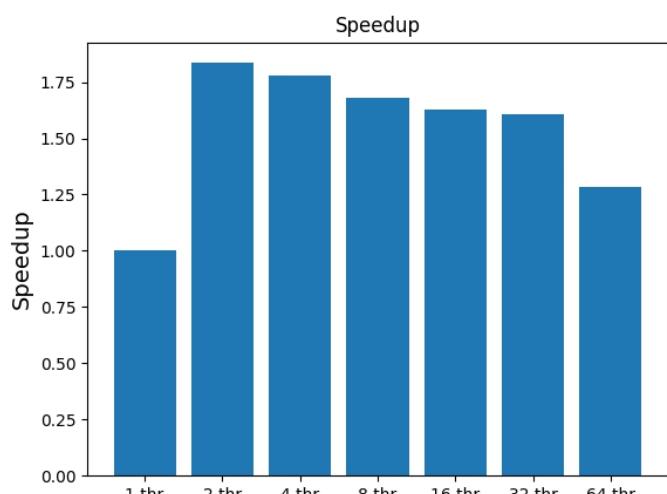
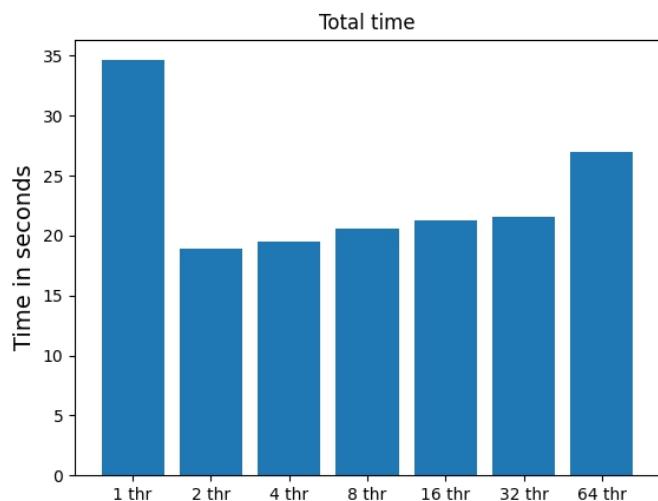
Tα barplots μας θα δημιουργηθούν με βάση το blocksize 128:



## 4096x4096:



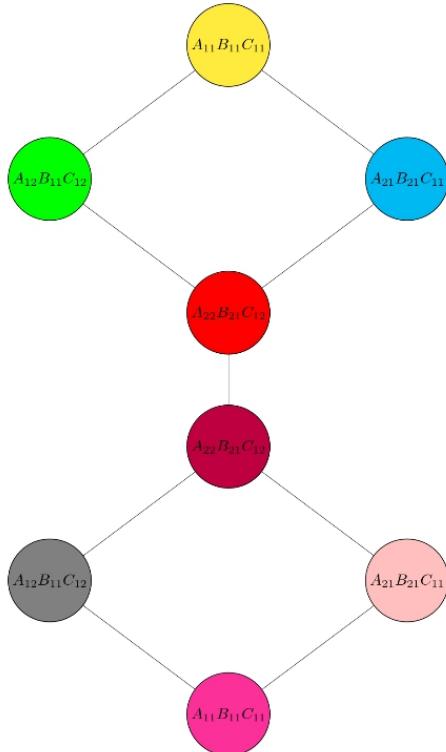
Τα barplots μας θα δημιουργηθούν με βάση το blocksize 128:



Παρατηρούμε ότι η κλιμάκωση του προγράμματός μας είναι απογοητευτική. Ωστόσο αυτή η συμπεριφορά είναι φυσιολογική. Για να βρούμε τον λόγο για τον οποίο βλέπουμε εξαιρετική βελτίωση μεταξύ ενός και δύο threads, αρκεί να σκεφτούμε το task graph του προγράμματος. Ουσιαστικά ο αλγόριθμός μας λειτουργεί κάπως έτσι:

```
FWR(A, B, C)
if(base case)
    FWiter(A, B, C)
else
    FWR(A11 , B11, C11)
    FWR(A12 , B11, C12)
    FWR(A21 , B21, C11)
    FWR(A22 , B21, C12)
    FWR(A22 , B21, C12)
    FWR(A21 , B21, C11)
    FWR(A12 , B11, C12)
    FWR(A11 , B11, C11)
```

To task graph μας δηλαδή είναι το παρακάτω:



Ο παραλληλισμός του αλγορίθμου μας είναι ξεκάθαρα περιορισμένος. Συνεπώς δεν αποτελεί ερώτημα γιατί το πρόγραμμά μας δεν κλιμακώνει για πάνω από 2 threads, αφού το overhead της δημιουργίας και επικοινωνίας των έξτρα νημάτων δεν αξίζει σε καμία περίπτωση αφού δεν υπάρχει διαθέσιμος παραλληλισμός για να εκμεταλλευτούμε.

Τέλος, θα κάνουμε σύγκριση της standard version του αλγορίθμου Floyd Warshall με την recursive υλοποίησή μας. Η σύγκριση θα γίνει board 4096x4096 ενώ για την recursive υλοποίηση θα κρατήσουμε τα δύο καλύτερα board size που πετύχαμε (δηλαδή 64 και 128):

Program\Threads	1	2	4	8	16	32	64
FW_standard	93.8440	94.0524	93.7867	93.8128	93.8160	93.9929	93.9513
FW_recursive_64	41.2168	21.8546	22.4588	22.8117	23.0917	23.2561	25.8662
FW_recursive_128	34.5148	19.4707	20.3097	20.7092	21.7661	22.8736	23.9522

Παρατηρούμε ότι υπάρχουν σημαντικές διαφορές στους χρόνους μεταξύ standard υλοποίησης και αναδρομικής ακόμα και για ένα thread. Αυτές οι διαφορές εξηγούνται με βάση τις προσβάσεις κάθε υλοποίησης στους πίνακες. Η standard υλοποίηση υπολογίζει σε κάθε γύρο το στοιχείο  $A[i][j]$  στο οποίο θα επανέλθει στον επόμενο γύρο. Οι γύροι είναι όσοι το  $N$  δηλαδή στην προκειμένη περίπτωση 4096. Το πρόβλημα αυτής της υλοποίησης λοιπόν είναι ότι το κάθε  $A[i][j]$  θα κληθεί 4096 φορές από την μνήμη (δεν θα το βρούμε στις caches μας) διότι μεταξύ δύο γύρων, το cache line στο οποίο βρίσκεται το  $A[i][j]$  θα έχει φύγει από την cache, καθώς θα πρέπει να κληθούν όλα τα υπόλοιπα στοιχεία του πίνακα  $A$  μέχρι να φτάσουμε στον επόμενο γύρο και να ξαναϋπολογίσουμε το  $A[i][j]$ . Η L1 cache μας έχει μέγεθος 32KB, η L2 256KB και η L3 16MB. Ο πίνακας  $A$  έχει μέγεθος  $N^2N^24B$  (είναι τύπου int) δηλαδή  $67,108,864 B = 65,536 KB = 64 MB$ . Συνεπώς ο πίνακας  $A$  είναι πολύ μεγαλύτερος ακόμα και από την L3 cache με αποτέλεσμα στον επόμενο γύρο να αγκαστούμε να καλέσουμε το cache line του  $A[i][j]$  από τη μνήμη και να χάνουμε πολύ χρόνο. Ακόμη, επειδή για τον υπολογισμό του  $A[i][j]$  χρειαζόμαστε και τα στοιχεία  $A[i][k]$  και  $A[k][j]$ , λόγω του μεγέθους του πίνακα  $A$ , θα έχουμε και εκεί cache misses που θα δυσχεραίνουν την κατάσταση.

Αντιθέτως, η recursive υλοποίηση περιορίζεται σε τετράγωνα μεγέθους board size κάνοντας πολλούς υπολογισμούς σε αυτά τα δεδομένα. Εκμεταλλεύεται δηλαδή πολύ καλύτερα το locality των δεδομένων βρίσκοντας κάθε φορά τα απαιτούμενα στοιχεία ακόμα και στην L1 cache.

Παρατηρούμε λοιπόν πόσο σημαντική μπορεί να είναι η αξιοποίηση των caches σε έναν memory bound αλγόριθμο όπως ο Floyd-Warshall.

## 2.3 Ταυτόχρονες Δομές Δεδομένων

Σκοπός αυτού του ερωτήματος είναι η εμβάθυνση σε μερικούς τρόπους συγχρονισμού νημάτων όταν απαιτούν πρόσβαση σε κοινές δομές δεδομένων. Για δομή δεδομένων θα χρησιμοποιήσουμε μία απλά συνδεδεμένη λίστα και θα μελετήσουμε τις παρακάτω ταυτόχρονες υλοποιήσεις που είδαμε και στις διαλέξεις:

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

Θα χρησιμοποιήσουμε το μηχάνημα sandman (32 φυσικοί πυρήνες και 64 λογικοί πυρήνες). Θα τοποθετούμε τα threads μας σε γειτονικούς φυσικούς πυρήνες, ενώ όταν τα threads είναι περισσότερα από 32 θα χρησιμοποιήσουμε και hyperthreading δηλαδή σε κάθε φυσικό πυρήνα θα τρέχουν πάνω από 2 threads τα οποία θα προσκολλώνται σε αδελφικούς λογικούς πυρήνες. Τα πειράματά μας περιλαμβάνουν διαφορετικά μεγέθη λίστας (1024, 8192), διαφορετικά ποσοστά λειτουργιών αναζητήσεων, εισαγωγών και διαγραφών (100-0-0, 80-10-10, 20-40-40, 0-50-50) και διαφορετικό αριθμό threads (1, 2, 4, 8, 16, 32, 64, 128).

Θα προβάλλουμε διαγράμματα για τα διαφορετικά ποσοστά λειτουργιών ενώ σε αυτά θα αποτυπώνουμε το Throughput(Kops/sec).

Πρωτού παρουσιάσουμε τα αποτελέσματά μας, θα εξηγήσουμε πως κάθε υλοποίηση επιτυγχάνει συγχρονισμό για κάθε λειτουργία (contains, add, remove).

### ● Coarse-grain locking

Για όλη τη λίστα έχουμε ένα και μόνο lock (στην υλοποίηση που μας δίνεται χρησιμοποιείται spinlock). Για να εκτελέσει κάποιο νήμα οποιαδήποτε από τις λειτουργίες θα πρέπει να αποκτήσει το lock. Αυτό δεν είναι καθόλου optimal αφού κλειδώνει ολόκληρη την τεράστια δομή με αποτέλεσμα κάθε φορά να τρέχει ένα νήμα. Μια πρώτη βελτίωση είναι το readers-writers lock αφού συνειδητοποιούμε ότι οι μέθοδοι που μπορούν να δημιουργήσουν πρόβλημα στο συγχρονισμό και στη σωστή λειτουργία της δομής είναι αυτές που κάνουν write (δηλαδή η add και remove) και αρκούν αυτές να γίνουν κρατώντας το lock ενώ η μέθοδος contains μπορεί να εκτελεστεί παράλληλα από πολλά threads. Ωστόσο το κύριο bottleneck παραμένει και αυτό είναι ότι κλειδώνουμε ολόκληρη την τεράστια δομή αντί να κλειδώσουμε τη “γειτονιά” πάνω στην οποία θέλουμε να επενεργήσουμε. Αυτή η υλοποίηση δεν είναι αποδοτική, είναι όμως πολύ εύκολο να προγραμματιστεί και δεν χρειάζεται να αγχωθούμε για το εάν θα λειτουργήσει ή όχι αφού είναι προφανές ότι δεν θα έχουμε προβλήματα.

### ● Fine-grain locking

Σε αυτήν την υλοποίηση έχουμε ένα lock για κάθε κόμβο της λίστας αντί για ένα lock για ολόκληρη την λίστα. Παρατηρούμε ότι για ορθό add και remove πρέπει να κλειδώσουμε μόνο δύο nodes, στο add τους δύο που θέλουμε να προσθέσουμε κάτι ανάμεσά τους και στο remove τον προηγούμενο του προς διαγραφή κόμβου και τον ίδιο τον προς διαγραφή κόμβο. Για να φτάσουμε όμως μέχρι το σημείο στο οποίο επενεργούμε θα πρέπει να διασχίσουμε τη λίστα με την τεχνική “hand-over-hand” locking δηλαδή κλειδώνουμε τους πρώτους δύο κόμβους, μετά κλειδώνουμε τον τρίτο και ξεκλειδώνουμε τον πρώτο, μετά κλειδώνουμε τον τέταρτο και ξεκλειδώνουμε τον δεύτερο κ.λπ. Αυτό συμβαίνει ώστε να βεβαιωθούμε ότι ο κόμβος στον οποίο βρισκόμαστε δεν έχει διαγραφεί από κάποιο άλλο thread ενώ κάνουμε traverse τη λίστα αλλά και για να πετύχουμε όλοι οι κόμβοι να αποκτούν τα locks που τους χρειαστούν με την κατάλληλη σειρά (ξεκινώντας από την αρχή της λίστας και προχωρώντας) ώστε να αποφύγουμε deadlocks σε περιπτώσεις που ένα thread έχει το lock του κόμβου 1 και ένα άλλο το lock του κόμβου 2 ενώ θέλουν να επενεργήσουν ανάμεσα στους δύο κόμβους. Το “hand-over-hand” locking χρησιμοποιείται και στις τρεις μεθόδους. Η υλοποίηση αυτή είναι πιο αποδοτική από το coarse-grain locking αφού κλειδώνει γειτονιές αντί για ολόκληρη τη λίστα, είναι όμως λιγότερο programmable.

## ● Optimistic synchronization

Συνεχίζοντας στο μοτίβο του κλειδώματος γειτονιών, θέλουμε να αποφύγουμε τη βαριά και αργή λειτουργία του “hand-over-hand” locking ώστε το κάθε thread που έχει κλειδώσει μία περιοχή να μην μπλοκάρει κάποιο άλλο που θέλει να προσπεράσει τη συγκεκριμένη γειτονιά. Για να το πετύχουμε αυτό, ξεκινάμε κάνοντας traverse τη λίστα χωρίς να πάρουμε κανένα lock. Όταν φτάσουμε στη γειτονιά που θέλουμε να επεξεργαστούμε, κρατάμε ένα στιγμιότυπο των δύο κόμβων που μας ενδιαφέρουν. Υστερα κλειδώνουμε τους δύο κόμβους και συγκρίνουμε εάν πράγματι το τωρινό στιγμιότυπο ταιριάζει με το προηγούμενο, δηλαδή ότι οι δύο κόμβοι παραμένουν γειτονικοί και δεν έχει προστεθεί ανάμεσά τους κάποιος καινούργιος στον χρόνο που μας πήρε να αποκτήσουμε τα locks. Αυτός όμως ο έλεγχος δεν αρκεί, θα πρέπει με κάποιον τρόπο να βεβαιωθούμε πως δεν έχει διαγραφεί ο πρώτος κλειδωμένος κόμβος από τη λίστα, εάν έχει διαγραφεί μπορεί πράγματι οι κόμβοι να παραμένουν γειτονικοί όμως να μην υπάρχει ο αρχικός κόμβος στη λίστα. Ο τρόπος για να το ελέγξουμε αυτό είναι : κρατώντας τα κλειδώματα των κόμβων που μας ενδιαφέρουν, κάνουμε traverse τη λίστα για να δούμε εάν ο πρώτος κλειδωμένος κόμβος είναι προσπελάσιμος. Εάν είναι , τότε πράγματι βρίσκεται ακόμα στη λίστα και αν ταιριάζουν και τα δύο στιγμιότυπα, η κατάσταση στη γειτονιά ενδιαφέροντος είναι πράγματι η ίδια που επικρατούσε πριν αποκτήσουμε τα locks και μπορούμε να συνεχίσουμε τη λειτουργία μας, ενώ εάν δεν ισχύει μία από τις δύο συνθήκες, ξεκλειδώνουμε και ξαναρχίζουμε από την αρχή. Το προφανές πρόβλημα αυτής της υλοποίησης είναι ότι διασχίζουμε όλη τη λίστα μέχρι το σημείο που μας ενδιαφέρει δύο φορές , μία χωρίς τα locks και μία κρατώντας τα. Είναι λοιπόν ξεκάθαρο bottleneck ειδικά εάν σκεφτούμε ότι εάν η λίστα μας είναι τεράστια, κατά το validation θα πρέπει να τη διασχίσουμε ολόκληρη ενώ κρατάμε τα locks και καθυστερούμε όσα threads θέλουν να περάσουν από την περιοχή μας. Ειδικά αν σκεφτούμε ότι και στη δικιά μας διαδρομή μέχρι τους κλειδωμένους κόμβους θα αναγκαστούμε να περάσουμε από προηγούμενες γειτονιές κλειδωμένων κόμβων, καταλαβαίνουμε ότι δημιουργείται μεγάλο bottleneck καθώς τα threads αυξάνονται.

## ● Lazy synchronization

Θέλουμε να λύσουμε το πρόβλημα της διπλής διάσχισης που δημιουργήθηκε κατά το validation της optimistic υλοποίησης. Ενώ ένα ακόμα μειονέκτημα της προηγούμενης υλοποίησης που θέλουμε να βελτιώσουμε είναι το γεγονός ότι ακόμα και η μέθοδος contains απαιτεί να κλειδώσει τη γειτονιά ενδιαφέροντος. Δεδομένου ότι στα πρώτα πειράματά μας έχουμε πολλά contains , εκεί θα δημιουργείται σίγουρα bottleneck. Ψάχνοντας μια υλοποίηση όπου οι μέθοδοι add και remove θα γίνονται με κλειδώματα αλλά θα διασχίζουν τη λίστα μόνο μία φορά, ενώ η contains δεν θα χρειάζεται κανένα κλειδωμα φτάνουμε στη lazy υλοποίηση όπου κάθε node ,πέρα από το κλειδωμά του, έχει και μια boolean μεταβλητή (marked) που μας δείχνει εάν ο κόμβος βρίσκεται ή όχι στη λίστα. Πλέον δεν χρειάζεται να διασχίσουμε ολόκληρη τη λίστα για να ελέγξουμε εάν είναι ακόμη προσπελάσιμος ο πρώτος κλειδωμένος κόμβος αλλά αρκεί αφού κλειδώσουμε, να ελέγξουμε την τιμή marked για να βεβαιωθούμε ότι δεν διεγράφη στο χρονικό διάστημα που πέρασε μέχρι να αποκτήσουμε το lock. Επίσης, η contains αρκεί να κάνει έναν εξίσου εύκολο έλεγχο στο πεδίο marked χωρίς να χρειαστεί καθόλου τα κλειδώματα αφού το marked μας εξηγεί ότι κάθε unmarked κόμβος είναι προσπελάσιμος άρα η contains θα επιστρέψει false εάν δεν βρούμε τον κόμβο ή εάν τον βρούμε και είναι marked(έχει γίνει η “λογική” διαγραφή του από τη λίστα). Η μέθοδος remove είναι αυτή που “βρωμίζει” το marked πεδίο του προς διαγραφή κόμβου. Απαιτείται ένα έξτρα επίπεδο προσοχής όταν διασχίζουμε marked κόμβους όπως περιγράφεται στο παράδειγμα της σελίδας 211 του βιβλίου μας. Καταλήγουμε λοιπόν στο ότι η validate είναι πολύ πιο γρήγορη από εκείνη της optimistic υλοποίησης αφού απλά ελέγχει ότι οι κόμβοι παραμένουν γειτονικοί και ότι δεν έγινε κάποιος marked. Περιμένουμε λοιπόν πολύ καλή απόδοση ενώ στα αρνητικά αυτού του σχήματος συγχρονισμού είναι η δυσκολία της υλοποίησής του.

## ● Non-blocking synchronization

Και οι 4 όμως προηγούμενες υλοποίήσεις χρησιμοποιούν locks σε κάποιο βαθμό μικρότερο ή μεγαλύτερο η μία από την άλλη. Θέλουμε μια υλοποίηση που δεν θα χρησιμοποιεί καθόλου κλειδώματα για καμία μέθοδο , τότε θα πετύχουμε robustness δηλαδή θα αποφύγουμε πλήρως καταστάσεις κατά τις οποίες η αποτυχία ενός νήματος που κρατάει ένα κλειδωμα θα οδηγήσει σε αποτυχία όλης της εφαρμογής. Βασική ιδέα για να το κάνουμε αυτό είναι να ενσωματώσουμε το πεδίο marked του κόμβου στο πεδίο next του ίδιου (που δείχνει ποιος είναι ο επόμενος του κόμβου). Σκοπός μας είναι να μην επιτρέπουμε την αλλαγή των πεδίων ενός node εφόσον αυτός έχει διαγραφεί (λογικά ή φυσικά) από τη λίστα δηλαδή κάθε προσπάθεια αλλαγής του next όταν ο κόμβος είναι marked να αποτύχει. Η συνένωση των δύο πεδίων θα

επιτρέψει τη δημιουργία μιας combo ατομικής εντολής σε χαμηλό επίπεδο. Η combo εντολή αυτή θα ελέγχει ότι ο κόμβος δεν είναι marked αλλά και ότι δείχνει στο αναμενόμενο σημείο και αν επιτύχει τότε θα επιτρέψουμε την αλλαγή του πεδίου next. Η μέθοδος remove θα κάνει τον κόμβο marked (λογική διαγραφή) χωρίς να επιμείνει στη φυσική διαγραφή του η οποία θα γίνει αργότερα από επόμενους εργάτες που θα διατρέχουν τη λίστα. Οι αλλαγές από την νέα εντολή γίνονται ατομικά και σε περίπτωση αποτυχίας, ξαναξεκινάμε από την αρχή. Αυτή η υλοποίηση περιμένουμε να έχει εξαιρετική επίδοση στην πλειονότητα των περιπτώσεων αφού έχουμε συνεχόμενη πρόοδο.

Στις επόμενες σελίδες ακολουθούν τα αποτελέσματα των πειραμάτων μας.

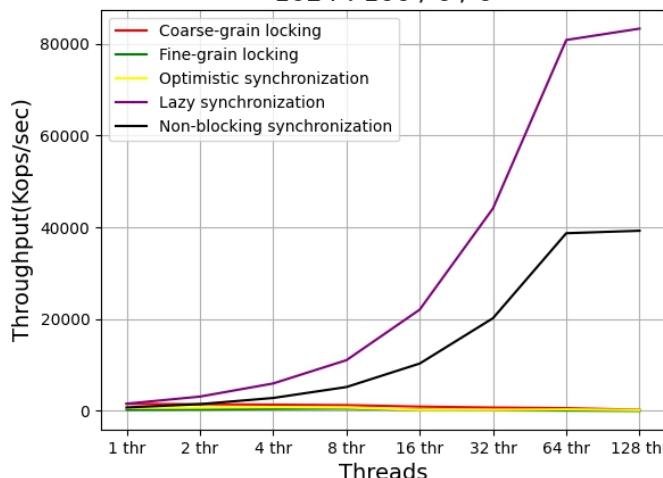
Για μέγεθος λίστας = 1024:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	1524.87	1437.84	1347.98	1244.42	923.44	720.76	583.22	224.0
Fine-grain	234.06	308.1	405.8	491.13	229.31	117.56	71.21	1.5
Optimistic	644.61	696.76	797.97	707.03	198.78	186.52	318.72	159.86
Lazy	1562.27	3102.55	5968.56	11043.94	22070.68	44137.96	80846.3	83321.97
Non-blocking	734.81	1458.68	2805.34	5191.87	10311.84	20200.9	38724.2	39261.63

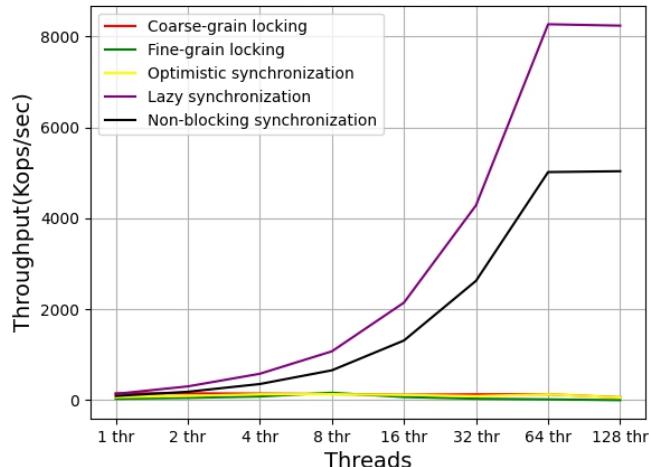
Για μέγεθος λίστας = 8192:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	151.55	150.42	143.9	132.91	117.81	128.26	127.92	61.63
Fine-grain	29.34	50.53	80.49	161.69	68.47	31.42	17.35	0.35
Optimistic	64.41	95.31	129.59	131.25	114.39	90.0	128.07	59.21
Lazy	137.74	302.58	580.44	1073.89	2145.2	4286.08	8267.09	8236.13
Non-blocking	92.81	184.4	354.67	656.44	1311.94	2624.89	5015.7	5034.57

1024 : 100 / 0 / 0



8192 : 100 / 0 / 0



Όταν έχουμε μόνο reads και κανένα write η lazy υλοποίηση είναι αυτή με την καλύτερη κλιμάκωση (περισσότερα Kops/sec) καθώς τα threads αυξάνονται ενώ ακολουθεί η non blocking. Και οι δύο δεν χρησιμοποιούν κλειδώματα στη μέθοδο contains και για αυτό ξεχωρίζουν. Η διαφορά τους υποθέτουμε ότι οφείλεται στο επιπλέον κόστος που πληρώνει η non-blocking τρέχοντας τη συνάρτηση list\_search() ενώ το traverse που κάνει η lazy υλοποίηση δεν έχει ελέγχους πέρα από τη συνθήκη του while. Η coarse grain δεν έχει μεγάλες διακυμάνσεις στην επίδοσή της όμως έχει ξεκάθαρη πτωτική τάση καθώς αυξάνονται τα threads. Αφού ουσιαστικά μόνο ένα thread έχει το lock, όλες οι άλλες περιπτώσεις είναι ίδιες με την περίπτωση του ενός thread αλλά έχουμε έξτρα κόστος για δημιουργία και επικοινωνία threads αλλά και cache invalidations λόγω του spin σε κοινό lock. Στην περίπτωση με 128 threads βλέπουμε ότι η πτώση είναι πολύ μεγάλη καθώς ενδέχεται το thread που έχει το lock να γίνει schedule out ενώ το κρατάει αφήνοντας έτσι όλους του πόρους ανεκμετάλλευτους αφού τα υπόλοιπα threads προσπαθούν μάταια να αποκτήσουν πρόσβαση στο lock που κατέχει ένα thread που δεν προοδεύει. Η fine grain καθυστερεί λόγω του hand-over-hand locking που δημιουργεί καθυστερήσεις ενώ η optimistic για λίγα threads δεν τα πηγαίνει άσχημα, όμως καθώς αυξάνονται επικρατεί χάος με τα κλειδώματα και τις διασχίσεις του validate όπως περιγράψαμε στις αρχικές σελίδες. Σημειώνεται ότι η πτώση των Kops/sec μεταξύ 1024 και 8192 είναι φυσιολογική αφού η λίστα είναι πολύ μεγαλύτερη στη δεύτερη περίπτωση συνεπώς πολλοί περισσότεροι κόμβοι πρέπει να προσπεραστούν για να φτάσουμε στο επιθυμητό σημείο της λίστας.

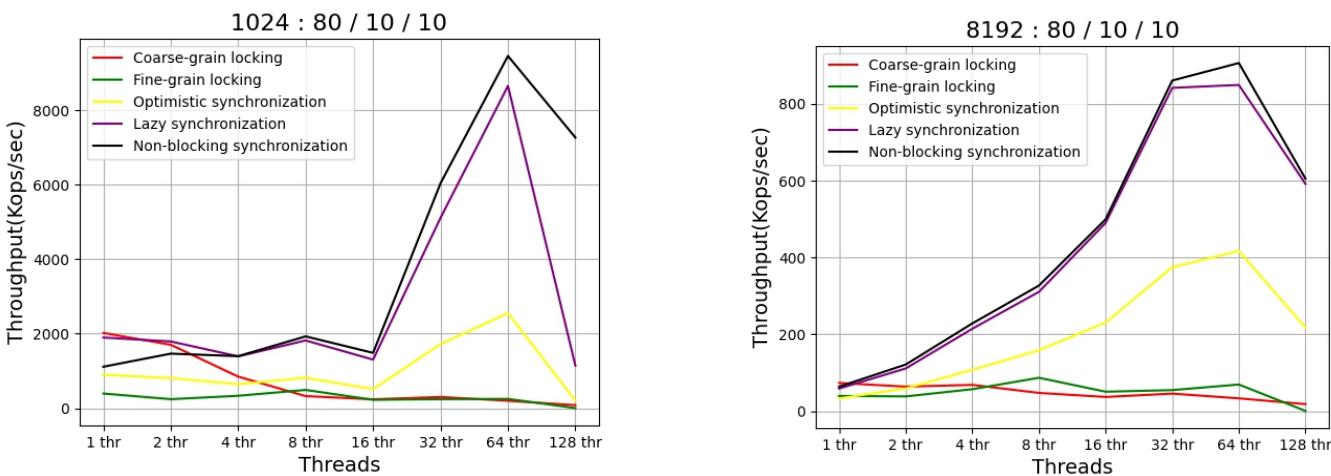
## 80-10-10:

Για μέγεθος λίστας = 1024:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	2020.8	1702.55	847.56	324.51	238.3	300.64	198.14	80.95
Fine-grain	392.93	241.68	332.45	489.71	224.23	242.54	248.76	2.84
Optimistic	899.99	807.2	644.05	815.08	510.68	1719.22	2552.39	204.76
Lazy	1897.49	1792.15	1391.62	1819.66	1304.43	5113.49	8657.59	1143.05
Non-blocking	1112.72	1465.36	1397.46	1928.51	1485.52	6042.01	9462.51	7270.49

Για μέγεθος λίστας = 8192:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	73.19	63.67	68.19	47.27	36.73	45.46	33.11	18.02
Fine-grain	39.34	38.16	56.88	86.72	50.36	54.58	69.14	0.28
Optimistic	32.87	58.95	107.4	158.38	230.53	374.55	417.01	217.62
Lazy	58.63	110.86	214.42	310.71	490.09	842.17	849.71	591.91
Non-blocking	63.54	121.09	228.32	326.93	499.52	861.2	906.82	605.49



Σε αυτήν την περίπτωση η non blocking υλοποίηση είναι αυτή που ξεχωρίζει ενώ η lazy την ακολουθεί. Λόγω της ανυπαρξίας των locks στη non blocking έχουμε αυτή τη βελτίωση σε σχέση με τη lazy που χρησιμοποιεί locks για add και remove τα οποία ωστόσο δεν είναι τόσο πολλά και για αυτό η διαφορά των δύο υλοποιήσεων δεν είναι τόσο μεγάλη στην πλειονότητα των πειραμάτων. Η πτώση που παρατηρείται για 128 threads έχει σίγουρα να κάνει και με τα schedule out των threads και τα context switch που συμβαίνουν τα οποία σπαταλάνε πολύτιμο χρόνο. Παρατηρούμε ότι η optimistic υλοποίηση έχει σχετικά καλή κλιμάκωση ειδικά στην περίπτωση 8192 όπου η λίστα είναι πιο μεγάλη και δεν θα έχουμε πολύ συχνά conflicts στις γειτονιές που θα θέλουν να κλειδώσουν δύο threads. Η coarse grain για 1024 ξεκινάει πολύ καλά όμως έχει σημαντική πτώση λόγω της διαμάχης των threads για απόκτηση του lock. Η fine grain όπως και πριν δεν είναι γρήγορη έχοντας μικρές αυξομειώσεις κατά τη διάρκεια των πειραμάτων, φαίνεται ότι το hand-over-hand locking είναι αργό ενώ δεν περιμένουμε η υλοποίηση αυτή να γίνει καλύτερη στα επόμενα πειράματα όπου θα αυξηθούν οι χρονοβόρες πράξεις add και remove.

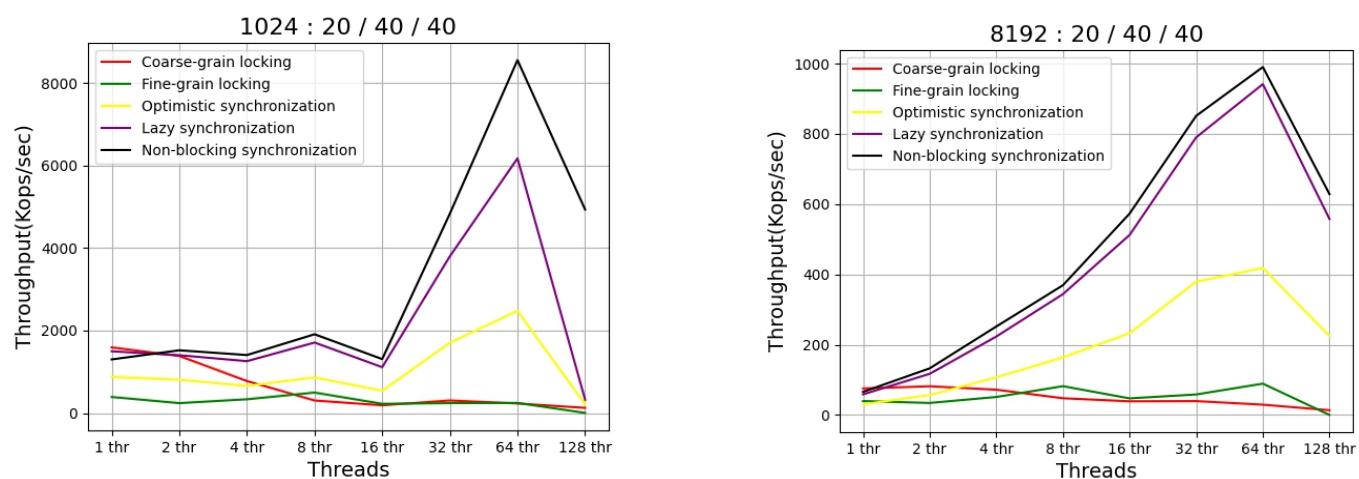
## 20- 40 - 40:

Για μέγεθος λίστας = 1024:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	1592.0	1383.19	777.26	305.43	189.48	306.34	233.35	126.76
Fine-grain	390.65	241.94	334.73	497.29	225.57	242.3	247.19	4.37
Optimistic	878.89	809.69	658.14	864.86	540.14	1701.91	2477.79	211.9
Lazy	1498.01	1402.1	1257.65	1709.63	1112.45	3799.42	6171.99	325.11
Non-blocking	1300.15	1521.76	1406.34	1910.23	1308.33	4831.63	8552.92	4931.79

Για μέγεθος λίστας = 8192:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	75.66	81.87	72.09	47.68	39.04	39.61	29.45	13.75
Fine-grain	39.76	34.35	51.32	82.15	47.34	58.55	89.34	0.29
Optimistic	30.43	56.89	107.23	163.95	232.94	379.51	418.38	225.06
Lazy	58.87	117.27	223.44	344.12	512.4	790.4	941.67	557.91
Non-blocking	65.9	132.88	251.98	369.38	573.19	851.49	990.35	628.68



Ξανά οι καλύτερες υλοποιήσεις είναι η lazy και η non blocking. Η διαφορά των δύο αυτών έχει μεγαλώσει σε σχέση με το προηγούμενο πείραμα καθώς η lazy χρησιμοποιεί κλειδώματα για τις πλέον περισσότερες add και remove. Η coarse grain πάσχει από τα ίδια προβλήματα που έχουμε αναφέρει στα προηγούμενα πειράματα (είναι πάντα το ίδιο αφού μόνο ένα thread έχει το lock). Η optimistic βελτιώνεται, όπως και πριν, μέχρι να γεμίσουμε όλους τους επεξεργαστές ενώ υπάρχει και μικρή βελτίωση όταν αξιοποιούμε το multithreading του sandman. Όταν όμως έχουμε 128 threads, λόγω των schedule outs έχουμε καθυστερήσεις. Επίσης φαίνεται το γεγονός ότι από το πρώτο στιγμιότυπο μέχρι να κλειδώσουμε, μπορεί να έχει γίνει schedule out το thread, να έχει αλλάξει η κατάσταση της γειτονιάς του και να πρέπει να ξαναρχίσει από την αρχή αφού αποτυγχάνει το validation. Η fine grain κινείται στο ίδιο μοτίβο με τα προηγούμενα πειράματα.

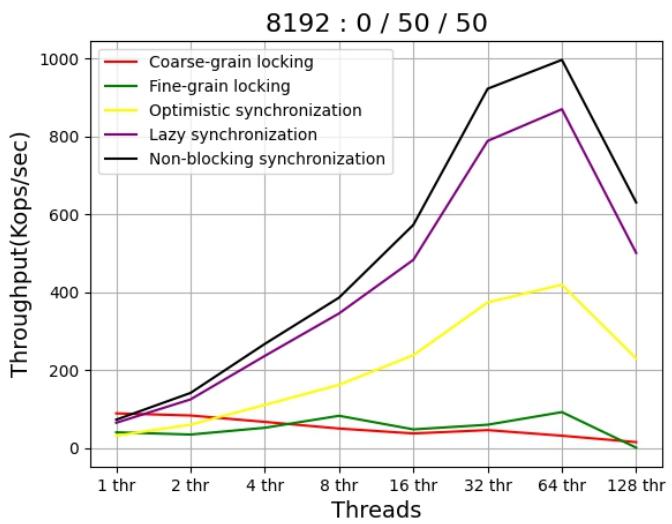
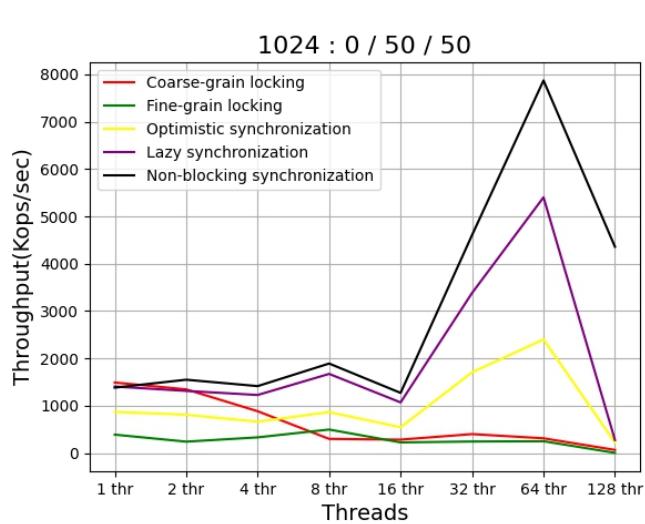
## 0-50-50:

Για μέγεθος λίστας = 1024:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	1491.46	1346.53	884.94	300.43	285.09	401.75	313.33	67.89
Fine-grain	390.33	242.66	332.97	498.4	226.7	245.03	251.38	5.71
Optimistic	871.97	811.31	663.97	866.47	546.35	1707.21	2400.71	225.1
Lazy	1406.32	1314.39	1227.92	1673.43	1071.27	3382.58	5403.59	276.2
Non-blocking	1383.03	1550.41	1415.49	1892.27	1272.27	4605.96	7871.23	4358.02

Για μέγεθος λίστας = 8192:

Sync Type \ Threads	1	2	4	8	16	32	64	128
Coarse-grain	88.55	83.11	66.65	49.6	36.96	45.53	31.12	14.64
Fine-grain	39.9	34.15	51.55	82.39	47.39	59.35	91.84	0.4
Optimistic	31.02	59.58	110.36	161.94	238.1	373.72	419.42	229.55
Lazy	64.38	124.47	236.29	345.56	483.26	788.95	870.47	500.85
Non-blocking	72.32	141.08	267.13	386.16	573.15	923.05	997.28	630.7



Για άλλη μια φορά οι καλύτερες υλοποιήσεις είναι η lazy και η non blocking. Έχει μεγαλώσει η διαφορά μεταξύ τους καθώς πλέον δεν έχουμε contains αλλά μόνο add και remove συνεπώς η lazy χρησιμοποιεί locks για κάθε πράξη που ζητείται. Επίσης επειδή δεν έχουμε contains, έχει μειωθεί η απόσταση μεταξύ της lazy και της optimistic που ούτως ή άλλως χρησιμοποιεί locks παντού. Σε αυτό το πείραμα είναι ολοφάνερη η καθυστέρηση που δημιουργεί το validation της optimistic και η υπεροχή της lazy ως προς το validation (έλεγχος του πεδίου marked). Οι άλλες δύο έχουν τα ίδια προβλήματα με τα προηγούμενα πειράματα. Μεγάλες πτώσεις παρατηρούνται για 128 threads λόγω των schedule out

## Τελικά σχόλια-παρατηρήσεις:

Η coarse-grain υλοποίηση έχει την ίδια λειτουργία για οποιονδήποτε αριθμό threads αφού κάθε φορά μόνο ένα thread θα έχει το lock. Πληρώνουμε όμως μεγάλος κόστος για δημιουργία και επικοινωνία των threads ενώ γίνονται και cache invalidations επειδή όλα τα threads κάνουν spin στο ίδιο lock.

Η fine-grain εκμεταλλεύεται καλύτερα τα threads καθώς αυτά αυξάνονται αφού τα κλειδώματα είναι σε μικρότερες γειτονιές. Παρατηρούμε μικρές αυξήσεις στην επίδοση όσο γεμίζουμε πυρήνες του sandman ενώ υπάρχει και μικρή βελτίωση (στα περισσότερα πειράματα τουλάχιστον) για 64 threads, όταν δηλαδή αξιοποιούμε και το hyperthreading. Όταν όμως ο αριθμός των threads υπερβαίνει αυτόν των λογικών πυρήνων (περίπτωση 128 threads) έχουμε τεράστια μείωση της απόδοσης σε βαθμό που θα μπορούσε να παρομοιαστεί με deadlock (π.χ για λίστα μεγέθους 8192 στο τελευταίο πείραμα έχουμε 0.4 Kops/sec). Αυτό συμβαίνει καθώς όλα τα threads, για όλες τις λειτουργίες διασχίζουν τη λίστα με “hand-over-hand” lock. Έτσι λοιπόν κάθε thread που θα γίνει schedule out θα έχει στην κατοχή του locks που δεν θα επιτρέψουν στα υπόλοιπα να προχωρήσουν πέρα από αυτό το σημείο π.χ άμα γίνει schedule out το thread που κάνει add μεταξύ των δύο πρώτων κόμβων και κρατάει τα locks αυτών, κανένα thread δεν μπορεί καν να ξεκινήσει τη διάσχιση της λίστας. Επίσης η απόδοση αυτής της υλοποίησης είναι περιορισμένη λόγω των cache invalidations που συμβαίνουν όταν κάποια threads κάνουν spin σε κάποιο κλειδωμένο lock ο κάτοχος του οποίου είτε έχει γίνει schedule out είτε κάνει μια χρονοβόρα διαδικασία (π.χ δεσμεύει χώρο με malloc για να προσθέσει έναν νέο κόμβο).

Η optimistic υλοποίηση είναι μεγάλη βελτίωση σε σχέση με τις δύο προηγούμενες γιατί δεν χρησιμοποιούμε hand-over-hand αλλά κινούμαστε πιο ελεύθερα. Το κόστος και η καθυστέρησή της προκύπτουν από το validation της που πρέπει να διασχίσει ξανά όλη τη λίστα. Ουσιαστικά κάθε λειτουργία κάνει τον διπλάσιο χρόνο απ'όσο παίρνει στη lazy. Επίσης, στα πειράματα που περιέχουν contains, η optimistic καθυστερεί γιατί χρησιμοποιεί locks και σε αυτήν την μέθοδο. Και εδώ βλέπουμε τα αποτελέσματα του να γίνεται schedule out ένα thread που κρατάει lock αφού και θα μπλοκάρει όλα τα υπόλοιπα threads αλλά και υπάρχει πιθανότητα εάν άλλαξε το στιγμιότυπο της γειτονιάς μέχρι να πάρει το lock να αναγκαστεί να ξαναξεκινήσει απ'την αρχή η λειτουργία.

Η lazy δεν χρησιμοποιεί locks στην contains αλλά μόνο στις add και remove. Για αυτό είναι πολύ κοντά στην non blocking υλοποίηση στα πρώτα πειράματα και απομακρύνεται στα επόμενα. Το validation της είναι στιγμιαίο σε αντίθεση με την optimistic. Και εδώ στα 128 threads έχουμε πολύ μειωμένη επίδοση αφού αν αποτύχει το validate (άλλαξε το στιγμιότυπο της περιοχής) θα πρέπει η λειτουργία να ξεκινήσει και πάλι απ'την αρχή.

Τέλος, η non-blocking υλοποίηση που δεν χρησιμοποιεί κανένα lock έχει την καλύτερη επίδοση σχεδόν σε όλα τα πειράματα. Όταν έχουμε 128 threads βλέπουμε για άλλη μια φορά το κόστος που πληρώνουμε όταν αποτυγχάνουν τα validations (επανεκκίνηση της διαδικασίας).