

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ ΠΡΟΠΑΡΑΣΚΕΥΑΣΤΙΚΗΣ ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 21.10.2025

■ Ενότητα 1.4.3 – Το Πρόγραμμα

Διοθέντος του αρχικού προγράμματος του Game of Life με τη σειριακή υλοποίηση και έχοντας μελετήσει τα παραδείγματα των εισαγωγικών διαφανειών του εργαστηρίου, συντάξαμε το ακόλουθο παράλληλο πρόγραμμα, με χρήση του OpenMP:

```

a1/life_par.c

1  ****
2  ***** Conway's game of life ****
3  ****
4
5  Usage: ./exec ArraySize TimeSteps
6
7  Compile with -DOUTPUT to print output in output.gif
8  (You will need ImageMagick for that - Install with
9  sudo apt-get install imagemagick)
10 WARNING: Do not print output for large array sizes!
11 or multiple time steps!
12 ****
13
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <sys/time.h>
17 #include <omp.h>
18
19 #define FINALIZE \
20 convert -delay 20 `ls -1 out*.pgm | sort -V` output.gif\n\
21 rm *pgm\n\
22 "
23
24 int **allocate_array(int N);
25 void free_array(int **array, int N);
26 void init_random(int **array1, int **array2, int N);
27 void print_to_pgm(int **array, int N, int t);
28
29 int main(int argc, char *argv[])
30 {
31     int N;                      // array dimensions
32     int T;                      // time steps
33     int **current, **previous; // arrays - one for current timestep, one for previous
timestep
34     int **swap;                 // array pointer
35     int i, j, t, nbrs;          // helper variables
36
37     double time; // variables for timing
38     struct timeval ts, tf;
39
40     /*Read input arguments*/
41     if (argc != 3)
42     {
43         fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
44         exit(-1);
45     }
46     else
47     {
48         N = atoi(argv[1]);
49         T = atoi(argv[2]);
50     }
51

```

```

52  /*Allocate and initialize matrices*/
53  current = allocate_array(N); // allocate array for current time step
54  previous = allocate_array(N); // allocate array for previous time step
55
56  init_random(previous, current, N); // initialize previous array with pattern
57
58 #ifdef OUTPUT
59  print_to_pgm(previous, N, 0);
60#endif
61
62  /*Game of Life*/
63
64  gettimeofday(&ts, NULL);
65
66  gettimeofday(&ts, NULL);
67
68  for (t = 0; t < T; ++t)
69  {
70
71  /* Parallelize rows; implicit barrier at loop end */
72  /* schedule is static
73   (i, j) as loop indices are private
74   (N, previous, current) are shared, as they are enclosed by the loop
75   nbrs must be private
76   by default */
77  /* Kept here for clarity */
78 #pragma omp parallel for schedule(static) private(i, j, nbrs) shared(N, previous, current)
79      for (i = 1; i < N - 1; ++i)
80      {
81          for (j = 1; j < N - 1; ++j)
82          {
83              nbrs =
84                  previous[i + 1][j + 1] + previous[i + 1][j] + previous[i + 1][j - 1] +
85                  previous[i][j - 1] + previous[i][j + 1] +
86                  previous[i - 1][j - 1] + previous[i - 1][j] + previous[i - 1][j + 1];
87
88              current[i][j] = (nbrs == 3 || (previous[i][j] + nbrs == 3)) ? 1 : 0;
89          }
90      } /* implicit barrier here: all threads finished step t */
91
92 #ifdef OUTPUT
93     print_to_pgm(current, N, t + 1); /* single thread here: we're back in serial */
94#endif
95
96     /* Safe to swap: we're outside the parallel region created by 'parallel for' */
97     swap = current;
98     current = previous;
99     previous = swap;
100 }
101
102 gettimeofday(&tf, NULL);
103 time = (tf.tv_sec - ts.tv_sec) + (tf.tv_usec - ts.tv_usec) * 0.000001;
104
105 free_array(current, N);

```

```
106     free_array(previous, N);
107     printf("GameOfLife: Size %d Steps %d Time %lf\n", N, T, time);
108 #ifdef OUTPUT
109     system(FINALIZE);
110 #endif
111 }
112
113 int **allocate_array(int N)
114 {
115     int **array;
116     int i, j;
117     array = malloc(N * sizeof(int *));
118     for (i = 0; i < N; i++)
119         array[i] = malloc(N * sizeof(int));
120     for (i = 0; i < N; i++)
121         for (j = 0; j < N; j++)
122             array[i][j] = 0;
123     return array;
124 }
125
126 void free_array(int **array, int N)
127 {
128     int i;
129     for (i = 0; i < N; i++)
130         free(array[i]);
131     free(array);
132 }
133
134 void init_random(int **array1, int **array2, int N)
135 {
136     int i, pos, x, y;
137
138     for (i = 0; i < (N * N) / 10; i++)
139     {
140         pos = rand() % ((N - 2) * (N - 2));
141         array1[pos % (N - 2) + 1][pos / (N - 2) + 1] = 1;
142         array2[pos % (N - 2) + 1][pos / (N - 2) + 1] = 1;
143     }
144 }
145
146 void print_to_pgm(int **array, int N, int t)
147 {
148     int i, j;
149     char *s = malloc(30 * sizeof(char));
150     sprintf(s, "out%d.pgm", t);
151     FILE *f = fopen(s, "wb");
152     fprintf(f, "P5\n%d %d 1\n", N, N);
153     for (i = 0; i < N; i++)
154         for (j = 0; j < N; j++)
155             if (array[i][j] == 1)
156                 fputc(1, f);
157             else
158                 fputc(0, f);
159     fclose(f);
```

```
160     free(s);  
161 }  
162  
163
```

Όσον αφορά τις λεπτομέρειες της υλοποίησής μας (όσον αφορά το παράλληλο τμήμα του προγράμματος, δηλ. τις γραμμές 69-100), επισημαίνουμε τα εξής:

- Η μεταβλητή *t*, δηλαδή ο αριθμός των γενεών, δεν είναι μια διαδικασία που επιδέχεται παραλληλοποίησης, καθώς αφενός μεν η μία διαδέχεται την άλλη, χωρίς να μπορούμε να πάμε στην επόμενη χωρίς να έχει τελειώσει η προηγούμενη, αφετέρου δε το μεγαλύτερο «κέρδος» προκύπτει από την παραλληλοποίηση των υπολογισμών εντός μιας γενιάς. Αυτό διότι, στο τέλος κάθε γενιάς πρέπει όλα τα threads να «συναντηθούν», ώστε να ενημερώσουν τα κοινά μας δεδομένα (δηλαδή τους πίνακες). Επομένως, στο τέλος κάθε γενιάς, υπάρχει ένα νοητό «φράγμα», στο οποίο συναντιούνται και συγχρονίζονται όλα τα threads, μέχρι όλα να τελειώσουν. Δηλαδή, οι γενιές εκτελούνται σειριακά, ενώ οι υπολογισμοί καθεμίας παράλληλα.
- Η παραλληλοποίηση του προγράμματος γίνεται στην γραμμή:

```
#pragma omp parallel for schedule(static) private(i, j, nbrs) shared(N, previous, current)
```

όπου παραλληλοποιούμε το for loop για το *i*. Σημειώνουμε (υπάρχουν και σχόλια στον κώδικα) ότι το scheduling είναι by default static, οι μεταβλητές του loop {*i*, *j*} είναι by default private και τα {*N*, previous, shared} by default shared, καθώς βρίσκονται εντός του παράλληλου τμήματος. Επομένως, η δήλωση των ανωτέρω δεν χρειάζεται, αλλά γίνεται για λόγους διαφάνειας. Η μεταβλητή *nbrs* πρέπει να δηλωθεί ως private, για να αποφευχθούν race conditions, εφόσον δεν δηλώνεται μέσα στο παράλληλο τμήμα.

- Στο τέλος του nested for loop, τα threads περιμένουν μέχρι να τελειώσουν όλα (βλ. σχόλιο) και να γίνει η ασφαλής-ατομική πρόσβαση στους πίνακες που ενημερώνουμε {previous, current}, καθώς αυτοί βρίσκονται εκτός του παράλληλου τμήματος.

Τέλος, τα αρχεία Makefile, make_on_queue.sh και run_on_queue.sh, έχουν συνταχθεί σε πλήρη αντιστοιχία με τα παραδείγματα των διαφανειών, φέρουν μικρές διαφορές που διευκολύνουν την υλοποίηση και την οργάνωση και παρουσιάζονται, για λόγους πληρότητας (χωρίς να χρειάζεται κάποια επεξήγηση), ακολούθως:

a1/Makefile

```
1 all: life_par
2
3 life_par: life_par.c
4     gcc -O3 -fopenmp -o life_par life_par.c
5
6 clean:
7     rm life_par
8
```

a1/make_on_queue.sh

```
1 #!/bin/bash
2
3 ## Give the Job a descriptive name
4 #PBS -N makejob
5
6 ## Output and error files
7 #PBS -o makejob.out
8 #PBS -e makejob.err
9
10 ## How many machines should we get?
11 #PBS -l nodes=1
12
13 ## Start
14 ## Load appropriate module
15 module load openmp
16
17 ## Run make in the src folder (modify properly)
18 cd /home/parallel/parlab05/a1/
19 make
20
```

a1/run_on_queue.sh

```
1 #!/bin/bash
2
3 ## Give the Job a descriptive name
4 #PBS -N life_par
5
6 ## Output and error files
7 #PBS -o life_par.out
8 #PBS -e life_par.err
9
10 ## How many machines should we get?
11 #PBS -l nodes=1:ppn=8
12
13 ## How long should the job run for?
14 #PBS -l walltime=01:00:00
15
16 ## Module Load
17 module load openmp
18
19 ## Defaults if not passed via -v
20 : "${THREADS:=8}"
21 : "${N:=1024}"
22 : "${STEPS:=1000}"
23
24 ## Start
25 cd /home/parallel/parlab05/a1/ || exit 1
26
27 # --- OpenMP runtime settings ---
28 export OMP_NUM_THREADS="${THREADS}"      # 1,2,4,6,8 per the assignment
29
30 # Run and capture outputs by config
31 RESULT_DIR="benchmarks/N${N}_T${THREADS}"
32 mkdir -p "${RESULT_DIR}"
33
34 ./life_par "${N}" "${STEPS}" \
35   > "${RESULT_DIR}/life_${THREADS}_${N}.out" \
36   2> "${RESULT_DIR}/life_${THREADS}_${N}.err"
37
38
```

■ Ενότητα 1.4.4 – Οι Μετρήσεις

Έχοντας συντάξει το παράλληλο πρόγραμμά μας, το υποβάλλαμε στα μηχανήματα του εργαστηρίου για κάθε συνδυασμό των παραμέτρων {μέγεθος ταμπλό, πυρήνες}. Οι παράμετροι αυτές, πήραν τις ακόλουθες τιμές, όπως ζητούνταν από την εκφώνηση της άσκησης:

- Μέγεθος ταμπλό: {64, 1024, 4096}
- Πυρήνες: {1, 2, 4, 6, 8}

Σύνολο $3 \times 5 = 15$ υποβολές.

Έτσι, ανοίγοντας τα αντίστοιχα αρχεία “filename.out”, λάβαμε τα ακόλουθα αποτελέσματα, τα οποία και παρουσιάζουμε στον κάτωθι πίνακα.

SIZE	THREADS	TIME	SPEEDUP
64	1	0.022613	1.000000
64	2	0.013416	1.685525
64	4	0.009780	2.312168
64	6	0.008942	2.528853
64	8	0.009163	2.467860
1024	1	11.793298	1.000000
1024	2	5.868753	2.009507
1024	4	2.929962	4.025069
1024	6	1.965196	6.001080
1024	8	1.476472	7.987485
4096	1	189.347966	1.000000
4096	2	94.832356	1.996660
4096	4	47.729763	3.967084
4096	6	32.393775	5.845196
4096	8	28.594441	6.621845

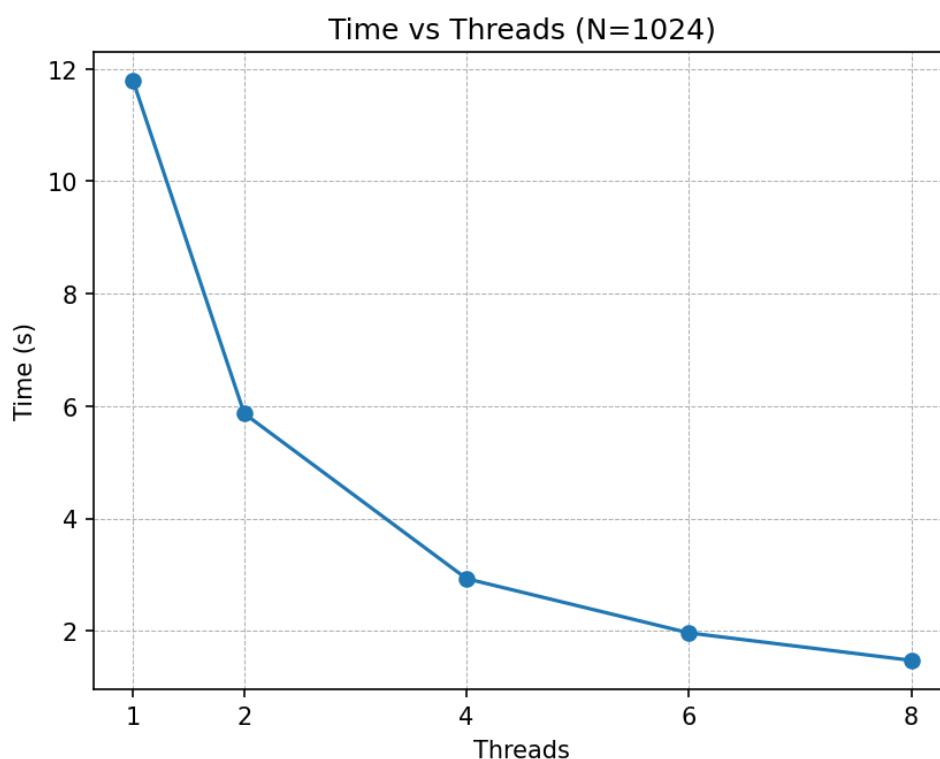
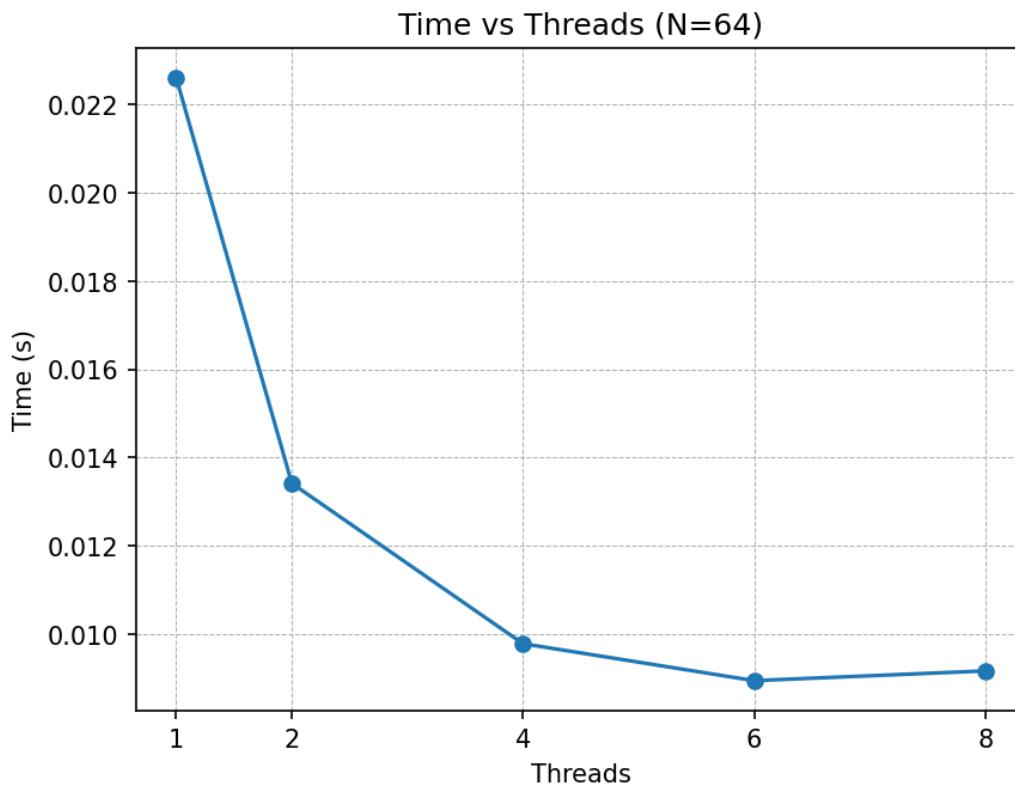
Τα αποτελέσματα αυτά αποτελούν τα δείγματα αναφοράς, με βάση τα οποία θα καταστρώσουμε τα ζητούμενα διαγράμματα (χρόνου και speedup) του επόμενου ερωτήματος. Επισημαίνουμε ότι:

$$Speedup = \frac{Time_of_N_cores}{Time_of_1_core}$$

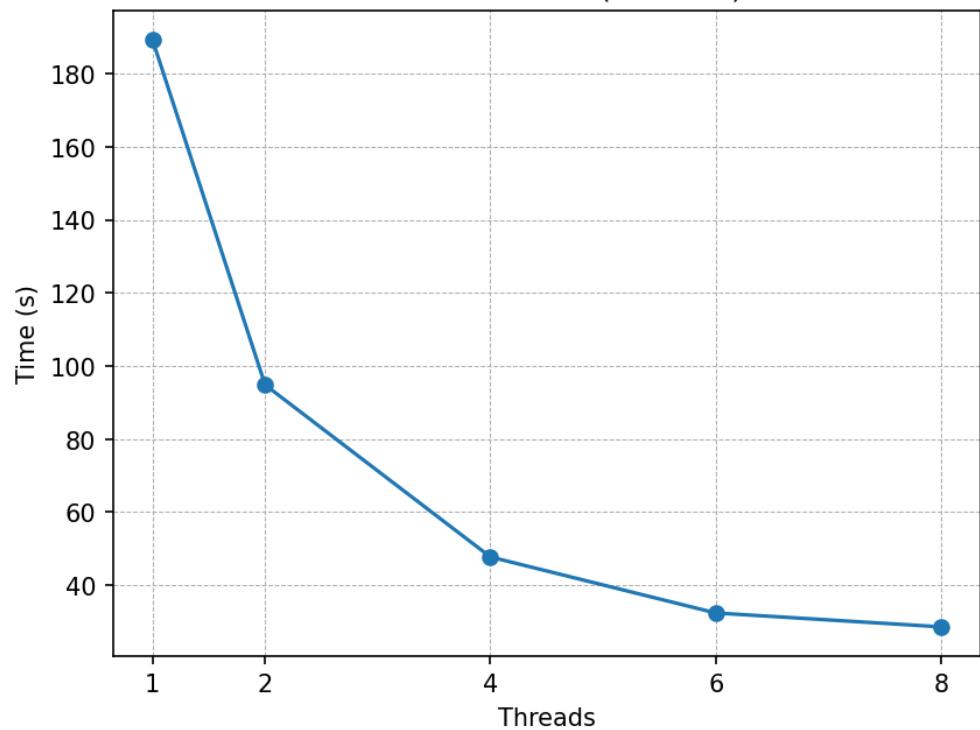
■ Ενότητα 1.4.5 – Τα Διαγράμματα

Σύμφωνα με τις μετρήσεις του παραπάνω πίνακα, καταστρώνουμε τα ακόλουθα διαγράμματα για κάθε μέγεθος ταμπλό (με χρήση ενός προγράμματος Python):

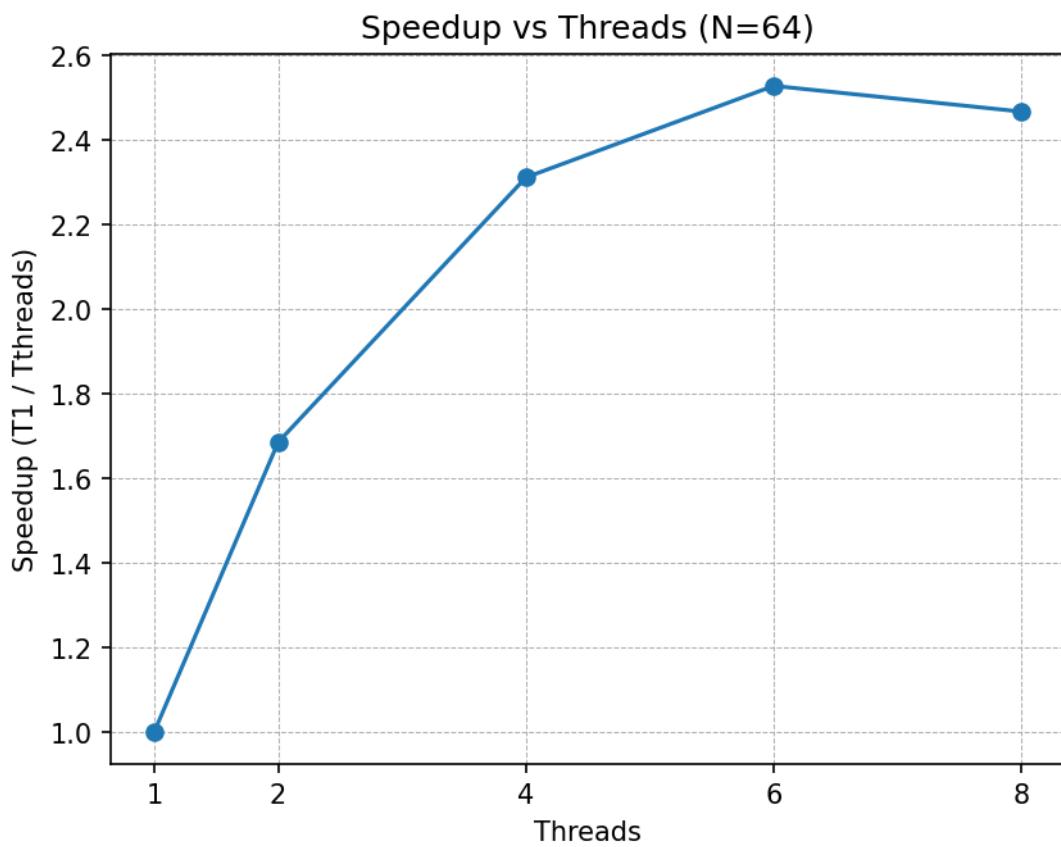
Διαγράμματα Χρόνου



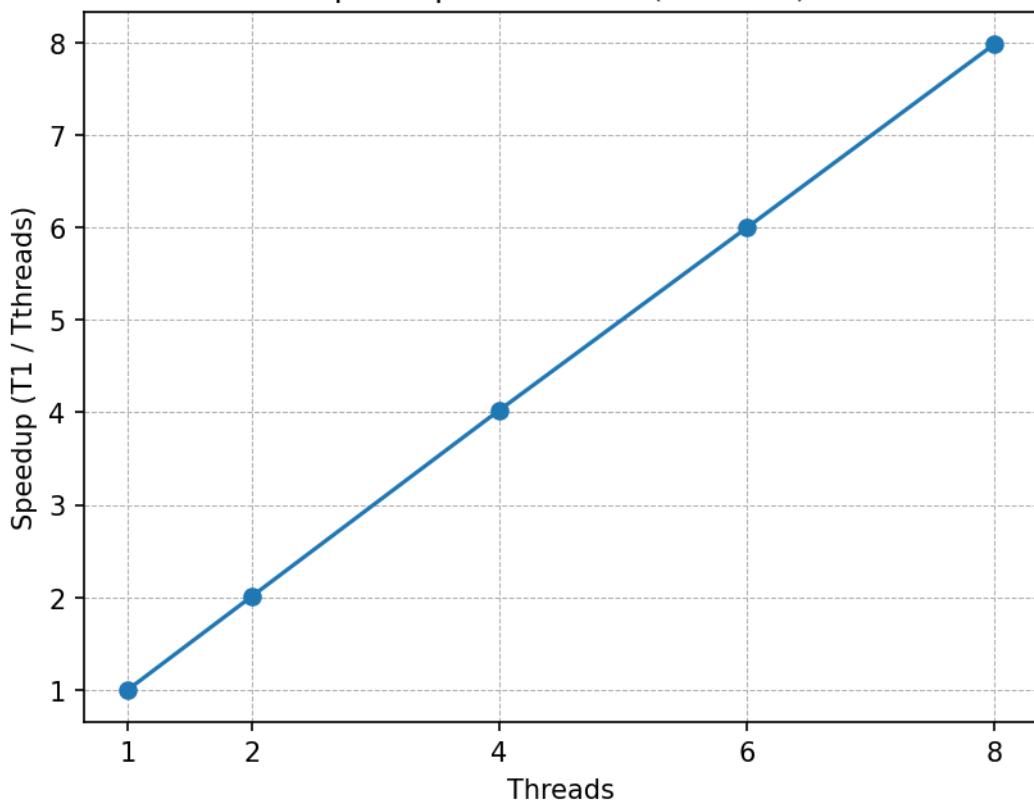
Time vs Threads (N=4096)



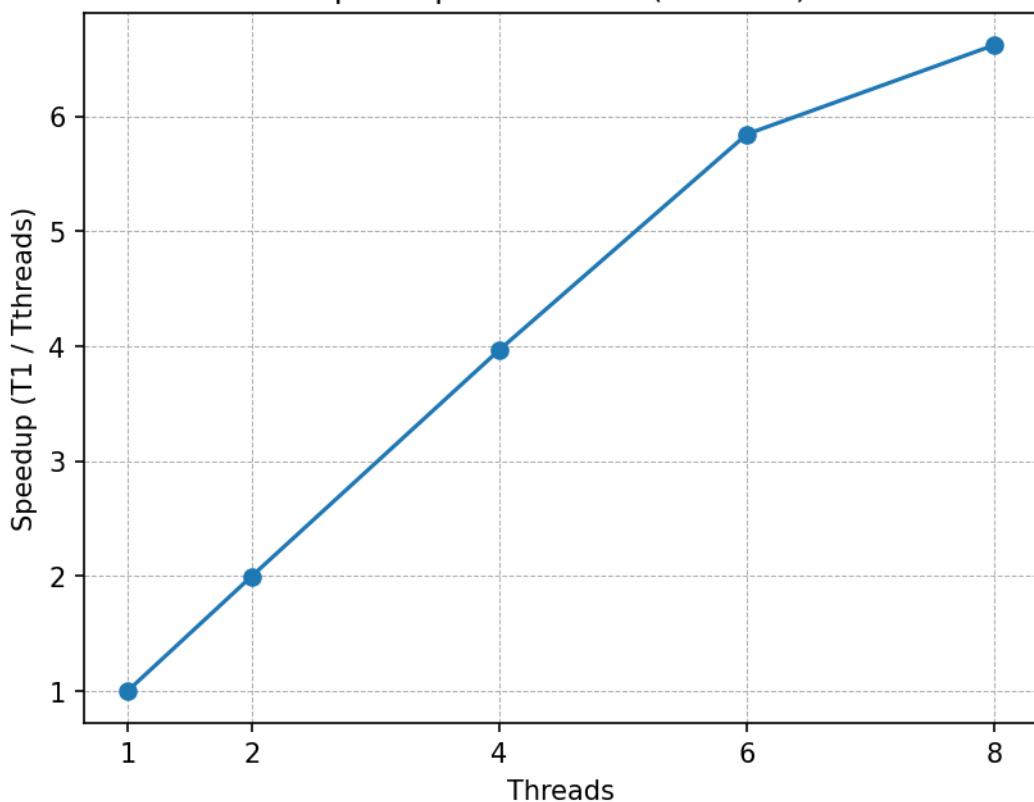
Διαγράμματα Speedup



Speedup vs Threads (N=1024)



Speedup vs Threads (N=4096)



Με βάση τα παραπάνω αποτελέσματα και διαγράμματα παρατηρούμε ότι:

- **Για N=64**, ο χρόνος εκτέλεσης δεν κλιμακώνει πέραν των 2 threads (δηλ. δεν είναι ανάλογος του 1/threads) και από τα 4 threads και έπειτα μειώνεται ελάχιστα με την αύξηση των νημάτων. Το speedup είναι μικρότερο από γραμμικό (κοίλη συνάρτηση) και στα 8 νήματα εμφανίζει μικρή υποχώρηση. Αυτό οφείλεται στο μικρό φορτίο ανά νήμα: το κόστος δημιουργίας και συγχρονισμού των νημάτων αποτελεί σημαντικό ποσοστό του συνολικού χρόνου και άρα δεν έχουμε μεγάλο CPU intensity.
- **Για N=1024**, ο χρόνος μειώνεται σχεδόν γραμμικά με τον αριθμό νημάτων, επιτυγχάνοντας ιδανική κλιμάκωση ($\sim 1/\text{threads}$). Το πρόβλημα είναι αρκετά μεγάλο ώστε να επικρατεί ο υπολογιστικός φόρτος έναντι του overhead δημιουργίας νημάτων, ενώ τα δεδομένα χωρούν πλήρως στην cache, αποφεύγοντας καθυστερήσεις από τη μνήμη. Έτσι, και η επιτάχυνση είναι γραμμική.
- **Για N=4096**, η επιτάχυνση αρχικά είναι σχεδόν γραμμική έως τα 4 νήματα, αλλά στη συνέχεια μειώνεται. Από τα 6 και ειδικά στα 8 νήματα, η βελτίωση της απόδοσης περιορίζεται, καθώς το πρόβλημα γίνεται memory-bound (έχουμε αρχιτεκτονική κοινής μνήμης, αφού τρέχουμε το πρόγραμμα σε 1 node, με πολλά threads): η αυξημένη κίνηση στη μνήμη και το περιορισμένο εύρος ζώνης (bandwidth) επιβραδύνουν την περαιτέρω κλιμάκωση, παρότι ο συνολικός χρόνος συνεχίζει να μειώνεται. Έτσι, η επιτάχυνση αρχικά είναι γραμμική, αλλά στο τέλος γίνεται κούλη.

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 1^{ης} ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 19.11.2025

▪ **Ενότητα 2.1 – Παραλληλοποίηση και Βελτιστοποίηση του Αλγορίθμου K-means**

Στόχος της άσκησης είναι η ανάπτυξη δύο παράλληλων εκδόσεων του αλγορίθμου K-means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με τη χρήση του προγραμματιστικού εργαλείου OpenMP.

Αρχικά, μελετήσαμε το υλικό και τα αρχεία του εργαστηρίου (στον φάκελο του kmeans που μας δίνεται), όπως και το αντίστοιχο υλικό των διαλέξεων του μαθήματος. Έτσι, τροποποιήσαμε τα περιφερειακά αρχεία (Makefile, make_on_queue.sh, run_on_queue.sh, file_io.sh) που εξυπηρετούν την ορθή μεταγλώττιση και λειτουργία των κυρίων αρχείων με τις παράλληλες εκδόσεις του αλγορίθμου μας (omp_naive_kmeans.c, omp_reduction_kmeans.c), ως εξής:

1. **Makefile:** Μετονομάσαμε τα αρχεία, ώστε να ανταποκρίνονται στο εργαστηριακό υλικό που μας δόθηκε, κάναμε uncomment τα σχόλια που μεταγλωττίζουν τα αρχεία με τον παράλληλο κώδικα και συμπεριλάβαμε το -fopenmp, ώστε να δηλώσουμε ότι τα αρχεία μας χρησιμοποιούν τη βιβλιοθήκη OpenMP.
2. **make_on_queue.sh:** Αλλάξαμε τη διεύθυνση του φακέλου src σε «/home/parallel/parlab05/a2/kmeans», ώστε να εξυπηρετεί τις ανάγκες της άσκησης, όπως ζητήθηκε. Καθώς η αλλαγή αυτή είναι μικρή, το συγκεκριμένο αρχείο δεν θα συμπεριληφθεί στην αναφορά.
3. **run_on_queue.sh:** Το αρχείο προσαρμόστηκε ώστε να επιτρέπει την εκτέλεση των πειραμάτων με διαφορετικές πολιτικές δέσμευσης νημάτων (affinity), μέσω παραμέτρων στην εντολή qsub. Υποστηρίζονται οι δύο κύριες φάσεις που ζητούνται στην άσκηση:
(α) εκτέλεση χωρίς καμία πολιτική δέσμευσης (noaff) και
(β) εκτέλεση με προκαθορισμένη πολιτική affinity (aff), όπου τα N νήματα του OpenMP δένονται ρητά στα N πρώτα λογικά CPU slots του κόμβου (0, 1, ..., N-1).

Η επιλογή αυτή πάρθηκε κατόπιν συζήτησης με τον διδάσκοντα και καθώς συνιστά την standard επιλογή της βιβλιοθήκης (κατόπιν αναζήτησης στο διαδίκτυο). Μια άλλη επιλογή με την οποία πειραματίστηκαμε ήταν ο διαμοιρασμός των threads στα 4 nodes του μηχανήματος ισάριθμα, ώστε να

αξιοποιήσουμε στο μέγιστο το διαθέσιμο memory bandwidth (η εφαρμογή μας είναι memory bound), παρά να είναι τοποθετημένα κοντά, στο ίδιο node και αυξάνοντας τη συμφόρηση στον δίαυλο μνήμης. Ωστόσο, για λόγους απλότητας και έκτασης της άσκησης, αφήσαμε την πολιτική στο default.

Σημειώνουμε ότι η επιλογή της πολιτικής γίνεται αυτόματα κατά την υποβολή του πειράματος, ενώ τα αποτελέσματα αποθηκεύονται σε ξεχωριστούς φακέλους, οργανωμένους ανά εκτελέσιμο και ανά επιλεγμένη πολιτική affinity, ώστε να διευκολύνεται η σύγκριση των μετρήσεων.

4. **file io.c:** Συμπληρώσαμε το header που ζητούνταν, ως: #include <omp.h>. Καθώς η αλλαγή αυτή είναι μικρή, το συγκεκριμένο αρχείο δεν θα συμπεριληφθεί στην αναφορά.

Τα αρχεία αυτά βρίσκονται στον orion και στον scirouter της ομάδας μας και παρουσιάζονται (αυτά που άλλαξαν σημαντικά, για λόγους πληρότητας) ακολούθως:

a2/kmeans/Makefile

```
1 .KEEP_STATE:
2
3 CC = gcc
4
5 CFLAGS = -Wall -Wextra -Wno-unused -O3 -std=gnu11
6 # Compile OpenMP sources with -fopenmp (+CFLAGS)
7 OMPFLAGS = $(CFLAGS) -fopenmp
8
9 # Link step includes -fopenmp so binaries link libgomp if any object used OpenMP
10 LDFLAGS = -fopenmp
11
12 H_FILES = kmeans.h
13 COMM_SRC = file_io.c util.c
14
15 # Build all variants
16 all: seq_kmeans omp_naive_kmeans omp_reduction_kmeans
17 seq_kmeans: main.o file_io.o util.o seq_kmeans.o
18     $(CC) $(CFLAGS) $^ -o $@ $(LDFLAGS)
19
20 omp_naive_kmeans: main.o file_io.o util.o omp_naive_kmeans.o
21     $(CC) $(CFLAGS) $^ -o $@ $(LDFLAGS)
22
23 omp_reduction_kmeans: main.o file_io.o util.o omp_reduction_kmeans.o
24     $(CC) $(CFLAGS) $^ -o $@ $(LDFLAGS)
25
26 main.o: main.c $(H_FILES)
27     $(CC) $(CFLAGS) -c $< -o $@
28
29 seq_kmeans.o: seq_kmeans.c $(COMM_SRC) $(H_FILES)
30     $(CC) $(CFLAGS) -c $< -o $@
31
32 # OpenMP objects use OMPFLAGS so pragmas are honored
33 omp_naive_kmeans.o: omp_naive_kmeans.c $(COMM_SRC) $(H_FILES)
34     $(CC) $(OMPFLAGS) -c $< -o $@
35
36 omp_reduction_kmeans.o: omp_reduction_kmeans.c $(COMM_SRC) $(H_FILES)
37     $(CC) $(OMPFLAGS) -c $< -o $@
38
39 file_io.o: file_io.c
40     $(CC) $(CFLAGS) -c $< -o $@
41
42 util.o: util.c
43     $(CC) $(CFLAGS) -c $< -o $@
44
45 clean:
46     rm -rf *.o seq_kmeans omp_naive_kmeans omp_reduction_kmeans
47
48
```

a2/kmeans/make_on_queue.sh

```
1 #!/bin/bash
2
3 ## How to run (example)
4 ## qsub -q parlab make_on_queue.sh
5
6 ## Give the Job a descriptive name
7 #PBS -N make_kmeans
8
9 ## Output and error files
10 #PBS -o make_kmeans.out
11 #PBS -e make_kmeans.err
12
13 ## How many machines should we get?
14 #PBS -l nodes=1:ppn=1
15
16 ## How long should the job run for?
17 #PBS -l walltime=00:10:00
18
19 ## Start
20 ## Run make in the src folder (modify properly)
21
22 cd /home/parallel/parlab05/a2/kmeans
23 make
24
25
```

```

1 #!/bin/bash
2
3 #PBS -N run_kmeans
4 #PBS -o run_kmeans.out
5 #PBS -e run_kmeans.err
6 #PBS -l nodes=1:ppn=64
7 #PBS -l walltime=01:00:00
8
9 # Submission details
10 # usage--no affinity (default): qsub -q serial -l nodes=sandman:ppn=64 -v
# THREADS=32,BIN=omp_naive_kmeans run_on_queue.sh
11 # with default affinity (bind 0..T-1): qsub -q serial -l nodes=sandman:ppn=64 -v
# THREADS=32,AFFINITY=default,BIN=omp_naive_kmeans run_on_queue.sh
12 # BIN=seq_kmeans|omp_naive_kmeans|omp_reduction_kmeans
13 # optional VARS: SIZE=256,COORDS=16,CLUSTERS=32,LOOPS=10
14
15 set -euo pipefail
16 cd /home/parallel/parlab05/a2/kmeans || exit 1
17
18 : "${BIN:=seq_kmeans}"
19 : "${SIZE:=256}"
20 : "${COORDS:=16}"
21 : "${CLUSTERS:=32}"
22 : "${LOOPS:=10}"
23 : "${THREADS:?Set THREADS via qsub -v THREADS=...}"
24 : "${AFFINITY:=none}"
25
26 export OMP_NUM_THREADS="${THREADS}"
27 AFF_LABEL="noaff"
28 if [[ "${AFFINITY,,}" == "default" ]]; then
29   CPUSERT=$(seq 0 $((THREADS-1)) | paste -sd' ' -)
30   export GOMP_CPU_AFFINITY="${CPUSERT}"
31   AFF_LABEL="aff"
32 else
33   unset GOMP_CPU_AFFINITY || true
34 fi
35
36 BENCH_ROOT="/home/parallel/parlab05/a2/kmeans/benchmarks"
37 case "${BIN}" in
38   *seq*) BENCH_SUBDIR_BASE="serial" ;;
39   *naive*) BENCH_SUBDIR_BASE="naive" ;;
40   *reduction*|*copied*) BENCH_SUBDIR_BASE="reduction" ;;
41   *) BENCH_SUBDIR_BASE="other" ;;
42 esac
43 BENCH_SUBDIR="${BENCH_SUBDIR_BASE}/${AFF_LABEL}"
44
45 RUN_TAG="S${SIZE}_N${COORDS}_C${CLUSTERS}_L${LOOPS}_T${THREADS}"
46 RESULT_DIR="${BENCH_ROOT}/${BENCH_SUBDIR}/${RUN_TAG}"
47 mkdir -p "${RESULT_DIR}"
48 {
49   echo "[run_on_queue] BIN=${BIN}"
50   echo "[run_on_queue] OMP_NUM_THREADS=${OMP_NUM_THREADS}"
51   echo "[run_on_queue] GOMP_CPU_AFFINITY=${GOMP_CPU_AFFINITY:-<unset>}"

```

```
53 echo "[run_on_queue] AFF_LABEL=${AFF_LABEL}"
54 echo "[run_on_queue] Params: -s ${SIZE} -n ${COORDS} -c ${CLUSTERS} -l ${LOOPS}"
55 echo "[run_on_queue] Result dir: ${RESULT_DIR}"
56 } | tee "${RESULT_DIR}/meta.txt"
57
58 "./${BIN}" -s "${SIZE}" -n "${COORDS}" -c "${CLUSTERS}" -l "${LOOPS}" \
59 | tee "${RESULT_DIR}/output.txt"
```

a2/kmeans/file_io.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>      /* strtok() */
4 #include <sys/types.h>   /* open() */
5 #include <sys/stat.h>
6 #include <fcntl.h>
7 #include <unistd.h>      /* read(), close() */
8 // TODO: remove comment from following line
9 #include <omp.h>
10
11 #include "kmeans.h"
12
13 double * dataset_generation(int numObjs, int numCoords)
14 {
15     double * objects = NULL;
16     long i, j;
17     // Random values that will be generated will be between 0 and 10.
18     double val_range = 10;
19
20     /* allocate space for objects[][] and read all objects */
21     objects = (typeof(objects)) malloc(numObjs * numCoords * sizeof(*objects));
22
23     /*
24      * Hint : Could dataset generation be performed in a more "NUMA-Aware" way?
25      *         Need to place data "close" to the threads that will perform operations on
26      * them.
27      *         reminder : First-touch data placement policy
28      */
29
30     for (i=0; i<numObjs; i++)
31     {
32         unsigned int seed = i;
33         for (j=0; j<numCoords; j++)
34         {
35             objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) * val_range;
36             if (_debug && i == 0)
37                 printf("object[i=%ld][j=%ld]=%f\n", i, j, objects[i*numCoords + j]);
38         }
39     }
40
41     return objects;
42 }
```

2.1.1 – Shared Clusters

Στην πρώτη παράλληλη υλοποίηση του αλγορίθμου K-means υιοθετήσαμε το μοντέλο shared clusters, χωρίς καμία βελτιστοποίηση στη συλλογή των μερικών αποτελεσμάτων. Πρόκειται για μια «αφελή» (naive) προσέγγιση, όπου όλα τα νήματα ενημερώνουν απευθείας τους κοινόχρηστους πίνακες newClusters[] και newClusterSize[]. Η πρόσβαση σε αυτά τα κοινόχρηστα δεδομένα απαιτεί συγχρονισμό, προκειμένου να αποφευχθούν πιθανά race conditions, ο οποίος στη συγκεκριμένη εκδοχή υλοποιείται αποκλειστικά με #pragma omp atomic για κάθε ενημέρωση-πρόσβαση. Σημειώνουμε ότι θα μπορούσε να έχει χρησιμοποιηθεί και #pragma omp critical, ωστόσο η εντολή αυτή είναι πιο αργή και υποβέλτιστη σε απλές προσβάσεις-πράξεις, όπως αυτή.

Η προσέγγιση αυτή επιτρέπει την εύκολη και άμεση παραλληλοποίηση του βρόχο, αλλά εισάγει σημαντικό κόστος εξαιτίας των συχνών πράξεων που γίνονται με atomic και της υψηλής πιθανότητας contention, ειδικά για μικρό αριθμό συντεταγμένων (numCoords) ή για μεγάλο αριθμό νημάτων. Δηλαδή, η ευκολία υλοποίησης έρχεται με υψηλό κόστος συγχρονισμού. Έτσι, η 2.1.1 συμβάλλει κυρίως ως σημείο αναφοράς για τη σύγκριση με πιο αποδοτικές τεχνικές συγχώνευσης που αναπτύσσονται στην επόμενη ενότητα (2.1.2 – copied clusters and reduce).

Όσον αφορά τις λεπτομέρειες της υλοποίησής μας, επισημαίνουμε τα εξής:

- Ο παράλληλος βρόχος (#pragma omp parallel for) αναθέτει σε κάθε νήμα ένα υποσύνολο αντικειμένων για επεξεργασία.
- Η αύξηση της μεταβλητής delta προστατεύεται με atomic operation, καθώς εδώ δεν χρησιμοποιείται reduction. Ωστόσο, ακόμα και χωρίς atomic (δεν ζητούταν με σχόλιο) η ορθότητα του προγράμματος δεν θα άλλαζε, αφού ναι μεν η delta θα είχε λάθος τιμή, αλλά σίγουρα $\text{delta} \geq 1 > \text{threshold}$, άρα ο έλεγχος θα ήταν πάντα σωστός και αληθής και θα μπορούσαμε να αποφύγουμε αυτό το atomic (η σχετική συζήτηση έγινε και με τον διδάσκοντα και επιλέξαμε να το βάλουμε).
- Η ενημέρωση των δομών newClusterSize[] και newClusters[] γίνεται επίσης με atomic σε κάθε πρόσβαση, γεγονός που καθιστά την υλοποίηση μεν εύκολη, αλλά δε καθόλου αποδοτική, αφού έχουμε μεγάλο κόστος συγχρονισμού, σε πολλαπλά σημεία.

Ο ολοκληρωμένος κώδικας της υλοποίησης (`omp_naive_kmeans.c`) βρίσκεται στον `scirouter` και στον `orion` της ομάδας μας, αλλά παρατίθεται και στην παρούσα αναφορά για λόγους πληρότητας.

```

a2/kmeans/omp_naive_kmeans.c

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "kmeans.h"
4 /*
5  * TODO: include openmp header file
6  */
7 #include <omp.h>
8
9 // square of Euclid distance between two multi-dimensional points
10 inline static double euclid_dist_2(int numdims, /* no. dimensions */
11                                     double *coord1, /* [numdims] */
12                                     double *coord2) /* [numdims] */
13 {
14     int i;
15     double ans = 0.0;
16
17     for (i = 0; i < numdims; i++)
18         ans += (coord1[i] - coord2[i]) * (coord1[i] - coord2[i]);
19
20     return ans;
21 }
22
23 inline static int find_nearest_cluster(int numClusters, /* no. clusters */
24                                         int numCoords, /* no. coordinates */
25                                         double *object, /* [numCoords] */
26                                         double *clusters) /* [numClusters][numCoords] */
27 {
28     int index, i;
29     double dist, min_dist;
30
31     // find the cluster id that has min distance to object
32     index = 0;
33     min_dist = euclid_dist_2(numCoords, object, clusters);
34
35     for (i = 1; i < numClusters; i++)
36     {
37         dist = euclid_dist_2(numCoords, object, &clusters[i * numCoords]);
38         // no need square root
39         if (dist < min_dist)
40             { // find the min and its array index
41                 min_dist = dist;
42                 index = i;
43             }
44     }
45     return index;
46 }
47
48 void kmeans(double *objects, /* in: [numObjs][numCoords] */
49             int numCoords, /* no. coordinates */
50             int numObjs, /* no. objects */
51             int numClusters, /* no. clusters */

```

```

52     double threshold,      /* minimum fraction of objects that change membership */
53     long loop_threshold, /* maximum number of iterations */
54     int *membership,      /* out: [numObjs] */
55     double *clusters)     /* out: [numClusters][numCoords] */
56 {
57     int i, j;
58     int index, loop = 0;
59     double timing = 0;
60
61     double delta;          // fraction of objects whose clusters change in each loop
62     int *newClusterSize; // [numClusters]: no. objects assigned in each new cluster
63     double *newClusters; // [numClusters][numCoords]
64     int nthreads;         // no. threads
65
66     nthreads = omp_get_max_threads();
67     printf("OpenMP Kmeans - Naive\t(number of threads: %d)\n", nthreads);
68
69     // initialize membership
70     for (i = 0; i < numObjs; i++)
71         membership[i] = -1;
72
73     // initialize newClusterSize and newClusters to all 0
74     newClusterSize = (typeof(newClusterSize))calloc(numClusters, sizeof(*newClusterSize));
75     newClusters = (typeof(newClusters))calloc(numClusters * numCoords,
76         sizeof(*newClusters));
76
77     timing = wtime();
78
79     do
80     {
81         // before each loop, set cluster data to 0
82         for (i = 0; i < numClusters; i++)
83         {
84             for (j = 0; j < numCoords; j++)
85                 newClusters[i * numCoords + j] = 0.0;
86             newClusterSize[i] = 0;
87         }
88
89         delta = 0.0;
90
91     /*
92     * TODO: Detect parallelizable region and use appropriate OpenMP pragmas
93     */
94     #pragma omp parallel for private(index, j)
95         for (i = 0; i < numObjs; i++)
96         {
97             // find the array index of nearest cluster center
98             index = find_nearest_cluster(numClusters, numCoords, &objects[i * numCoords],
99             clusters);
100
101             // if membership changes, increase delta by 1
102             if (membership[i] != index)
103             {

```

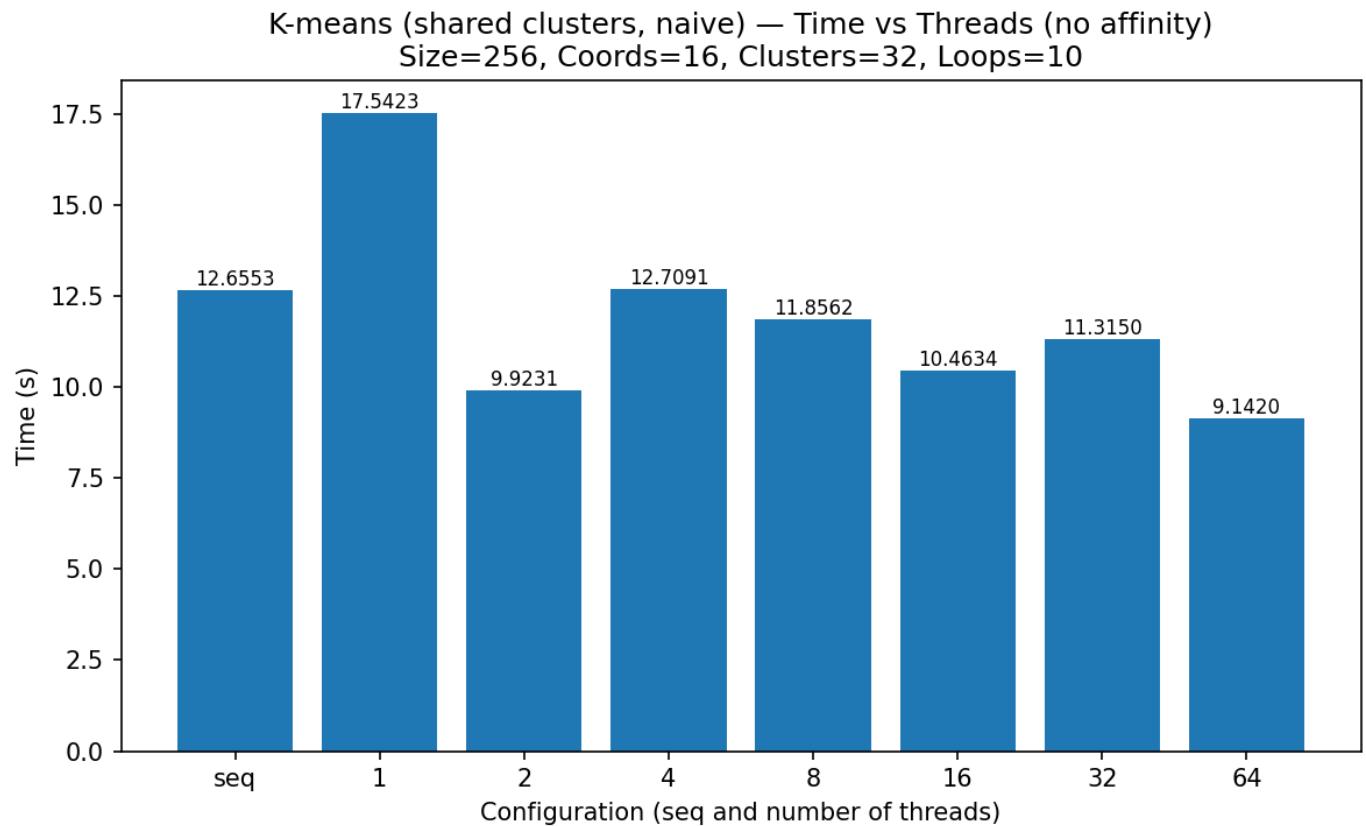
```
104 #pragma omp atomic // protect update on shared "delta" variable
105         delta += 1.0;
106     }
107
108     // assign the membership to object i
109     membership[i] = index;
110
111 // update new cluster centers : sum of objects located within
112 /*
113 * TODO: protect update on shared "newClusterSize" array
114 */
115 #pragma omp atomic
116         newClusterSize[index]++;
117         for (j = 0; j < numCoords; j++)
118     /*
119     * TODO: protect update on shared "newClusters" array
120     */
121 #pragma omp atomic
122         newClusters[index * numCoords + j] += objects[i * numCoords + j];
123     }
124
125     // average the sum and replace old cluster centers with newClusters
126     for (i = 0; i < numClusters; i++)
127     {
128         if (newClusterSize[i] > 0)
129         {
130             for (j = 0; j < numCoords; j++)
131             {
132                 clusters[i * numCoords + j] = newClusters[i * numCoords + j] /
133             newClusterSize[i];
134             }
135         }
136     }
137
138     // Get fraction of objects whose membership changed during this loop. This is used
139     // as a convergence criterion.
140     delta /= numObjs;
141
142     loop++;
143     printf("\r\tcompleted loop %d", loop);
144     fflush(stdout);
145 } while (delta > threshold && loop < loop_threshold);
146     timing = wtime() - timing;
147     printf("\n          nloops = %3d    (total = %7.4fs)  (per loop = %7.4fs)\n", loop,
148 timing, timing / loop);
149
150     free(newClusters);
151     free(newClusterSize);
152 }
```

1. No Affinity

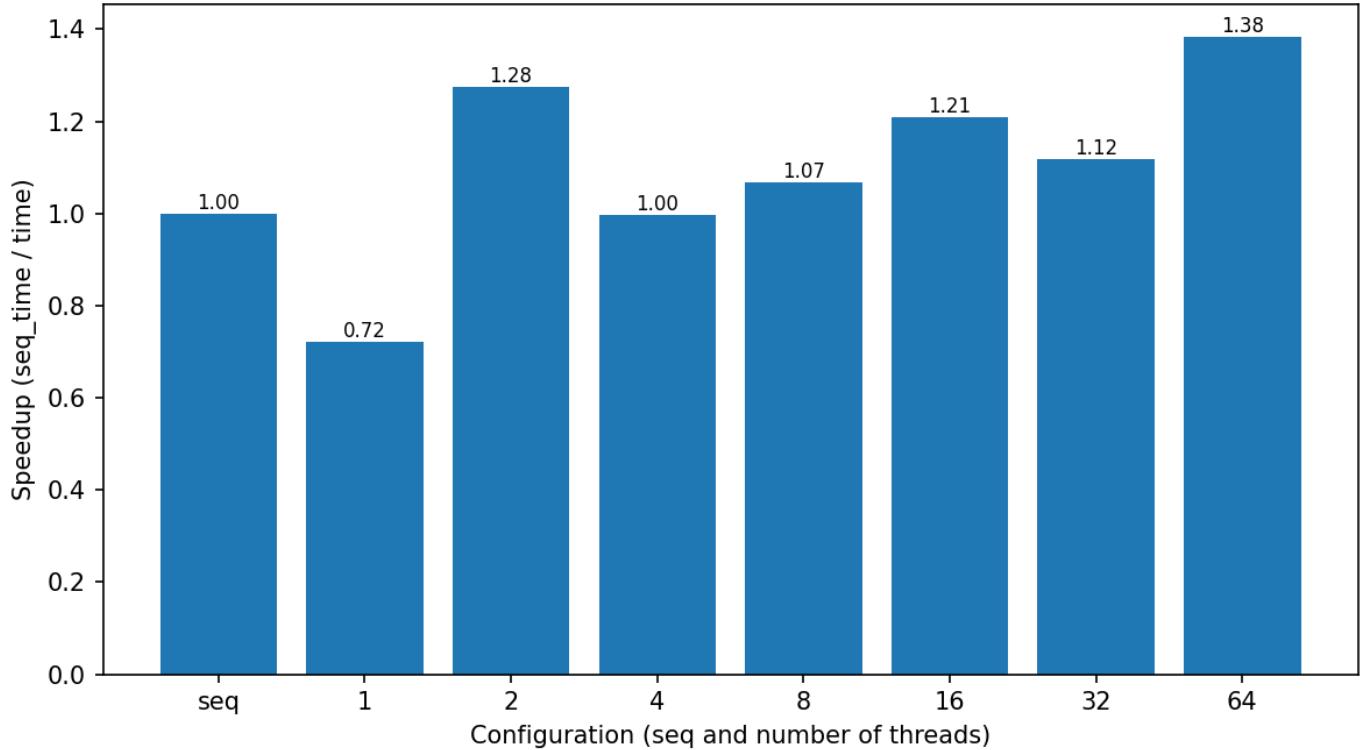
Το παραπάνω παράλληλο πρόγραμμα (omp_naive_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και χωρίς affinity. Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	12.65
1	17.54
2	9.92
4	12.70
8	11.85
16	10.46
32	11.31
64	9.14

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



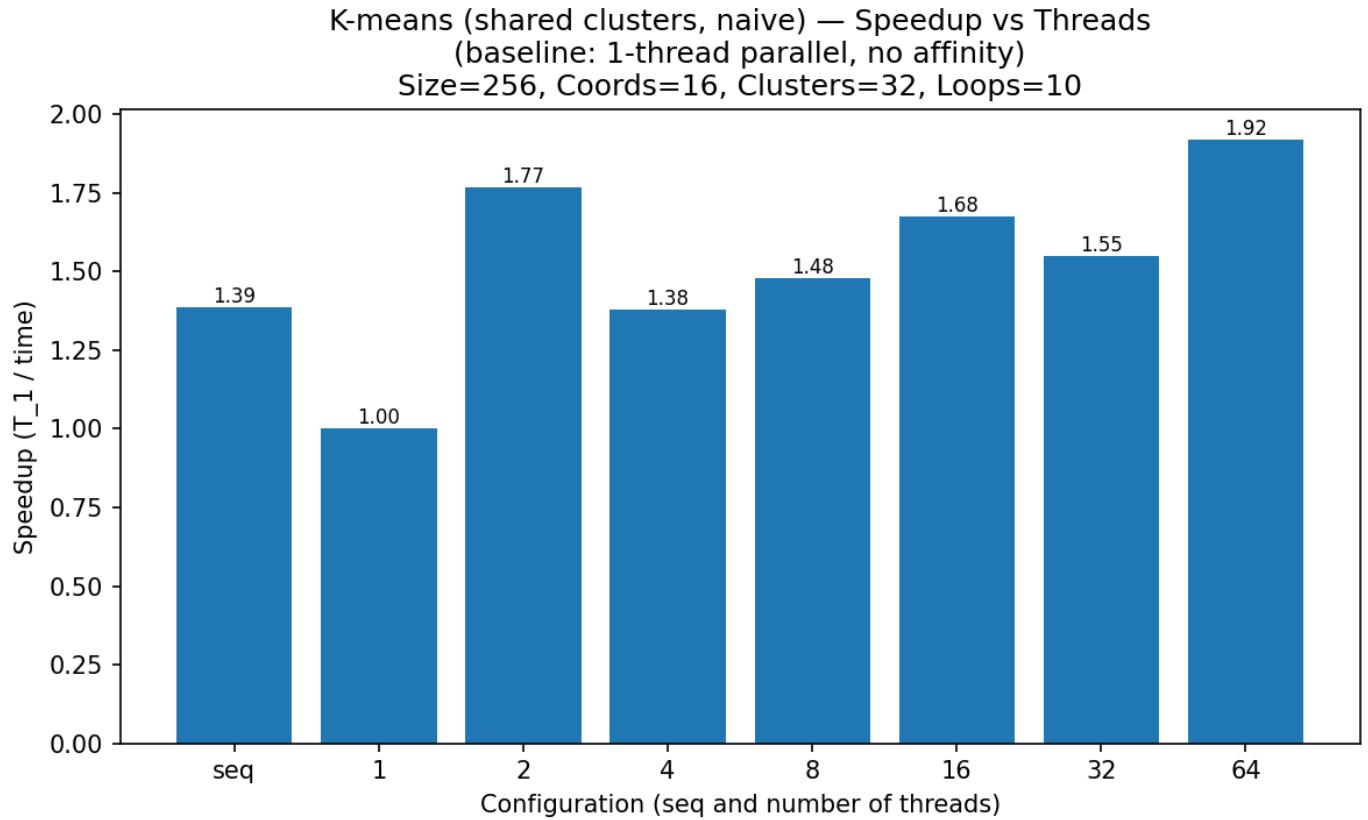
K-means (shared clusters, naive) — Speedup vs Threads (no affinity)
 Size=256, Coords=16, Clusters=32, Loops=10



Σύμφωνα με τα παραπάνω, συμπεραίνουμε ότι υλοποίηση χωρίς affinity δεν κλιμακώνει ικανοποιητικά. Ο σειριακός χρόνος είναι περίπου 12.7s, ενώ η παράλληλη έκδοση με 1 νήμα είναι σαφώς χειρότερη ($\approx 17.5s$, speedup ≈ 0.72), κάπι που αποδίδεται στο κόστος δημιουργίας του παράλληλου προγράμματος-κώδικα (δημιουργία/συγχρονισμός νημάτων, επιπλέον κώδικας OpenMP), όπως αναφέρεται και στις διαφάνειες. Για περισσότερα νήματα, τα διαγράμματα χρόνου και speedup δείχνουν μικρές μόνο βελτιώσεις: γύρω στο 1.28 \times στα 2 νήματα (αν και παρατηρείται κλιμάκωση σχεδόν 2x από το 1 νήμα του παράλληλου προγράμματος), τιμές κοντά στο 1.0–1.2 \times για 4–32 νήματα και μέγιστο περίπου 1.38 \times στα 64 νήματα, πολύ μακριά από την ιδανική γραμμική κλιμάκωση.

Η συμπεριφορά αυτή ταιριάζει ακριβώς με τη θεωρία για synchronization bottlenecks: στη naive shared έκδοση όλες οι ενημερώσεις των πινάκων newClusterSize[] και newClusters[] γίνονται με #pragma omp atomic, άρα πολλές προσπελάσεις σε λίγες κοινόχρηστες μεταβλητές σειριοποιούνται και δημιουργούν έντονο contention, όπως στα παραδείγματα των διαφανειών για fine-grained synchronization. Έτσι, μεγάλο μέρος του χρόνου δαπανάται σε συγχρονισμό αντί για πραγματικό υπολογισμό, ενώ και το σειριακό τμήμα του αλγορίθμου (σύμφωνα με τον νόμο του Amdahl) βάζει χαμηλό άνω φράγμα στο speedup. Τα παραπάνω αποτελέσματα και τα διαγράμματα, λοιπόν, επιβεβαιώνουν ότι η λύση αυτή είναι μεν ορθή και εύκολη στην υλοποίηση, αλλά καθόλου αποδοτική.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



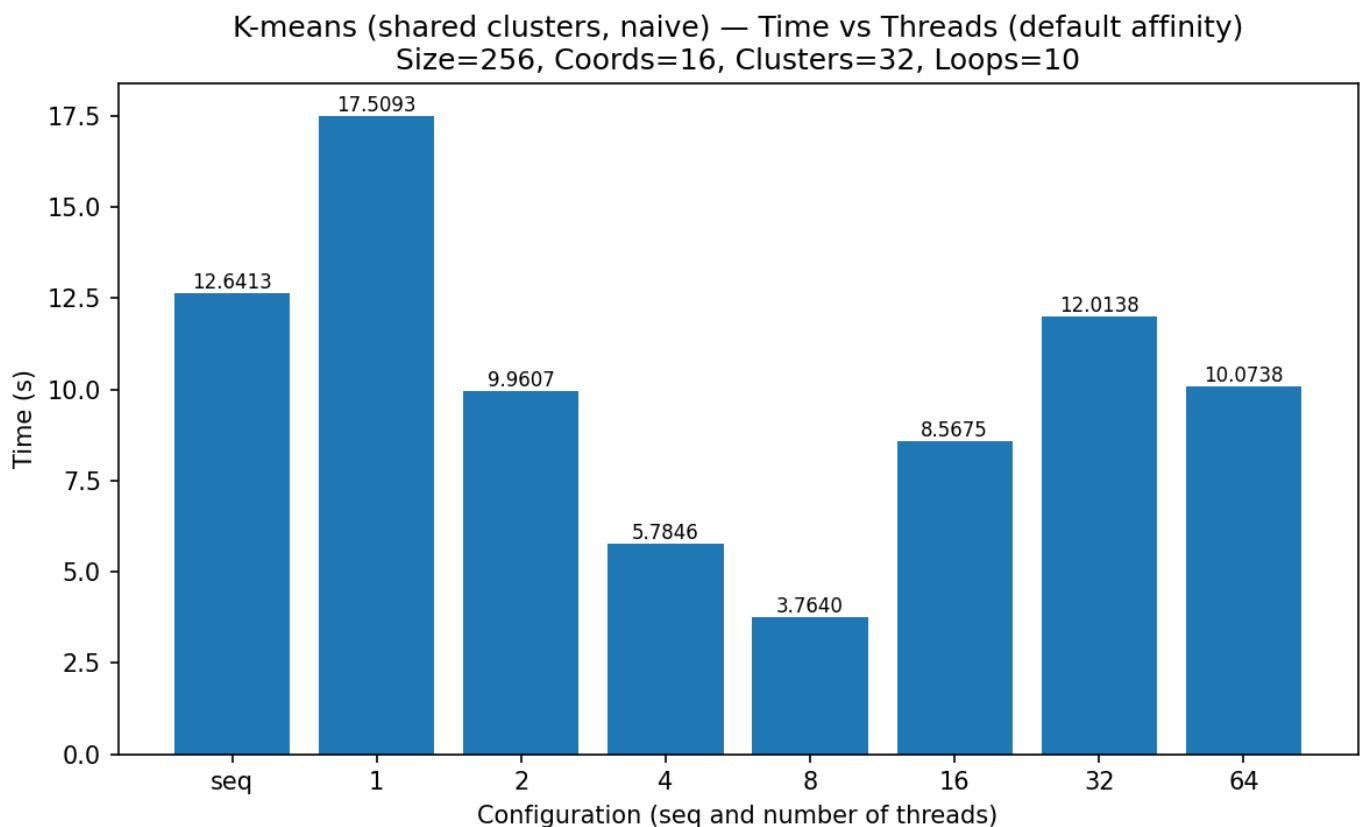
Από το παραπάνω διάγραμμα παρατηρούμε ότι, αν θεωρήσουμε ως βάση το παράλληλο πρόγραμμα με 1 νήμα, όλες οι εκτελέσεις με περισσότερα νήματα εμφανίζουν πλέον speedup μεγαλύτερο της μονάδας. Ενδεικτικά, για 2 νήματα το κέρδος είναι περίπου $1.8\times$ σε σχέση με το 1-thread (σχεδόν γραμμικό και το μόνο ικανοποιητικό), ενώ για 64 νήματα φτάνει σχεδόν το $2\times$ (πολύ κακή κλιμάκωση γενικότερα). Αυτό επιβεβαιώνει ότι ένα σημαντικό τμήμα του κόστους στην περίπτωση του 1 νήματος οφείλεται αποκλειστικά στο parallel overhead του OpenMP (δημιουργία ομάδας νημάτων, συγχρονισμοί κ.λπ.), το οποίο «απλώνεται» σε περισσότερα νήματα και αντισταθμίζεται μερικώς όταν αυξάνουμε τον βαθμό παραλληλίας και ότι η εφαρμογή είναι memory bound. Παρ' όλα αυτά, η κλιμάκωση παραμένει εξαιρετικά κακή και σε απόλυτους όρους ως προς το σειριακό πρόγραμμα και τα κέρδη παραμένουν μικρά, γεγονός που δείχνει ότι το synchronization bottleneck της naive shared υλοποίησης δεν επιτρέπει ουσιαστική κλιμάκωση.

2. Default Affinity

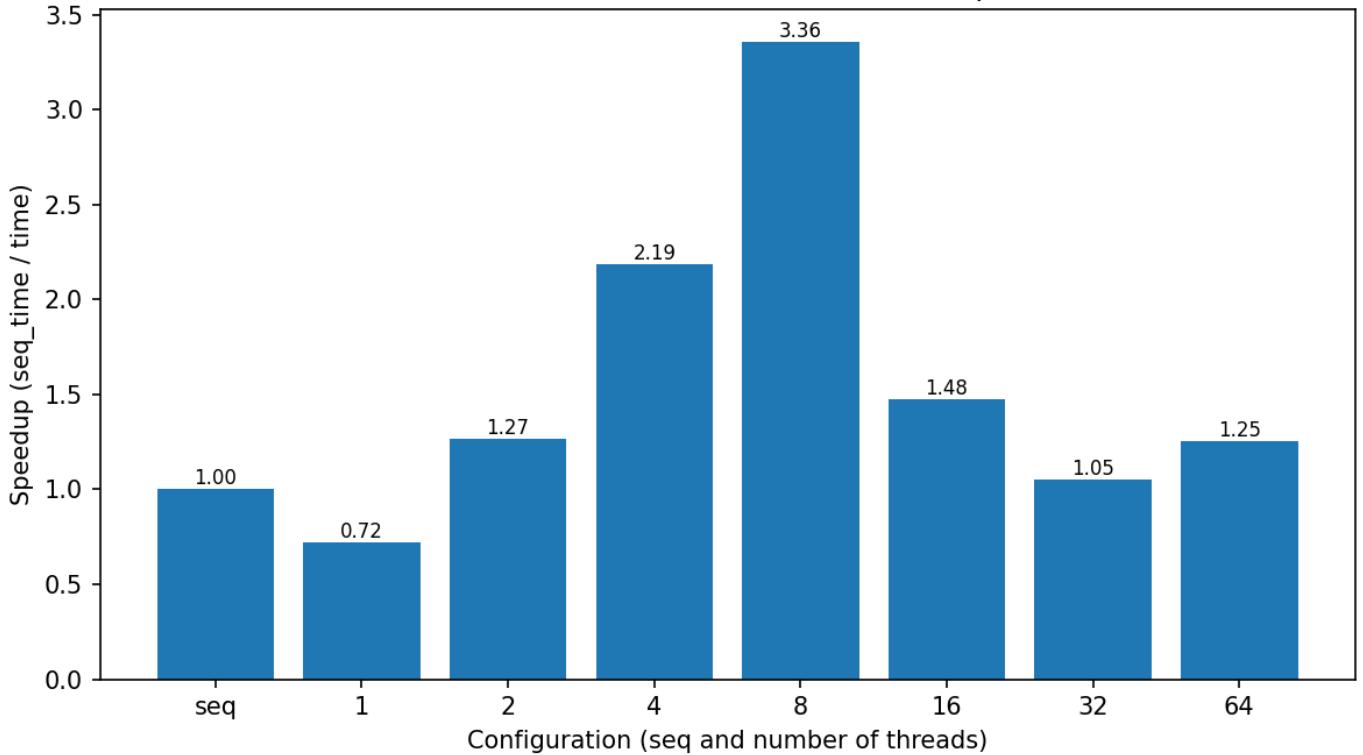
Το παραπάνω παράλληλο πρόγραμμα (omp_naive_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64, αλλά αυτή τη φορά με affinity. Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	12.64
1	17.50
2	9.96
4	5.78
8	3.76
16	8.56
32	12.01
64	10.07

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



K-means (shared clusters, naive) — Speedup vs Threads (default affinity)
Size=256, Coords=16, Clusters=32, Loops=10

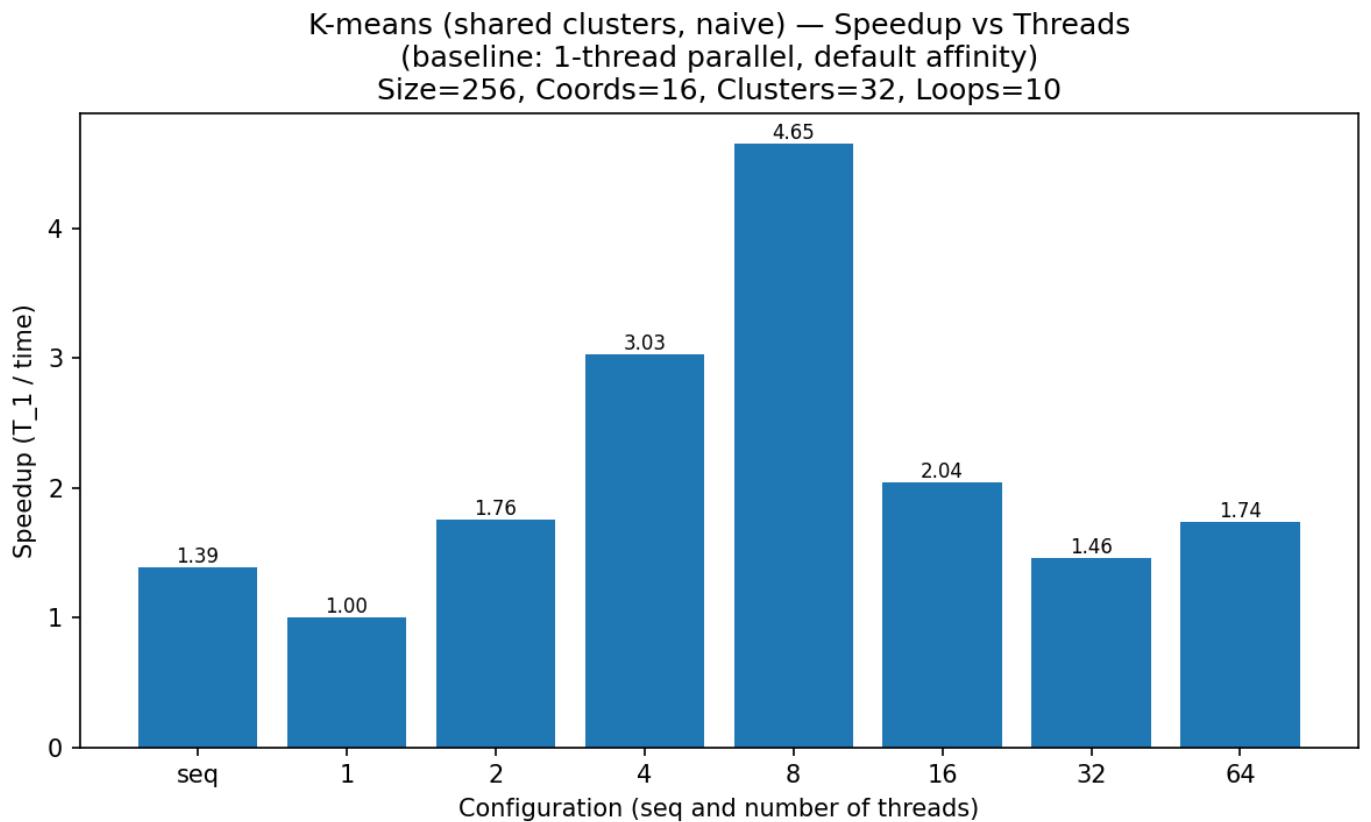


Παρατηρούμε ότι με ενεργοποιημένο το default affinity η συμπεριφορά της naive shared υλοποίησης βελτιώνεται σημαντικά σε σχέση με την περίπτωση χωρίς affinity. Ο χρόνος εκτέλεσης μειώνεται πολύ καλά οριακά γραμμικά ως προς το παράλληλο πρόγραμμα με 1 νήμα) μέχρι τα 8 νήματα (από ~17.5s στο 1 νήμα σε ~3.8s στα 8 νήματα), με αντίστοιχο speedup ~3.4x σε σχέση με το σειριακό πρόγραμμα, γεγονός που δείχνει ότι η δέσμευση των νημάτων σε σταθερούς πυρήνες αξιοποιεί καλύτερα την τοπικότητα cache και μνήμης μέσα στο ίδιο NUMA node, όπως επισημαίνεται και στις διαφάνειες για affinity και locality. Ωστόσο, για 16, 32 και 64 νήματα η επίδοση υποβαθμίζεται (ο χρόνος αυξάνεται ξανά και το speedup πέφτει κοντά στη μονάδα), κάτι που είναι αναμενόμενο για έναν κατά βάση memory-bound αλγόριθμο με έντονο synchronization μέσω atomic πράξεων.

Πιο συγκεκριμένα, όταν ξεπερνάμε τα νήματα που «χωράει άνετα» ένα socket/NUMA node, η εκτέλεση αρχίζει να μοιράζεται σε πολλαπλούς κόμβους μνήμης και αυξάνονται οι απομακρυσμένες προσπελάσεις (remote NUMA accesses) και η συμφόρηση στον δίαυλο μνήμης. Ταυτόχρονα, οι atomic ενημερώσεις στους κοινόχρηστους πίνακες newClusters[] και newClusterSize[] δημιουργούν έντονο contention στις ίδιες cache lines, με αποτέλεσμα η θεωρητική παραλληλία να χάνεται από τον συγχρονισμό, όπως ακριβώς περιγράφεται στις διαφάνειες για synchronization bottlenecks και NUMA αρχιτεκτονικές. Συνολικά, το affinity εκμεταλλεύεται καλά τη δομή του κόμβου μέχρι τα 8 νήματα, αλλά τα αρχιτεκτονικά

και αλγορίθμικά όρια της naive shared λύσης δεν επιτρέπουν ουσιαστική κλιμάκωση πέρα από αυτό το σημείο.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



Από το τελευταίο διάγραμμα, όπου ως βάση λαμβάνουμε πλέον την παράλληλη εκτέλεση με 1 νήμα, βλέπουμε ότι τα speedups για 2, 4 και 8 νήματα είναι ιδιαίτερα υψηλά (της τάξης του 1.7–1.8x, ~3x και ~4.5x αντίστοιχα). Αυτό δείχνει ότι το σημαντικό parallel overhead της OpenMP (δημιουργία και οργάνωση της ομάδας νημάτων, συγχρονισμοί, barriers) κατανέμεται αποτελεσματικά σε λίγα νήματα όταν αυτά «μένουν» σε κοντινούς πυρήνες του ίδιου NUMA node, με αποτέλεσμα η αύξηση του βαθμού παραλληλίας από 1 σε 2–8 threads να αποδίδει καθαρό κέρδος εντός της ίδιας αρχιτεκτονικής (κάπως κοντά σε γραμμικά, ειδικά αρχικά).

Παρ' όλα αυτά, όταν συνεχίζουμε πέραν από τα 8 νήματα, τα speedups ως προς το 1-thread παράλληλο πρόγραμμα μειώνονται αισθητά, γεγονός που υποδηλώνει ότι έχουμε φτάσει πρακτικά το όριο των πόρων (πυρήνων, cache, memory bandwidth) ενός sockets και αρχίζουμε να «πατάμε» σε δεύτερο NUMA

node ή/και να ενεργοποιούμε hardware multithreading στους ίδιους φυσικούς πυρήνες. Σε έναν αλγόριθμο όπως o K-means, που είναι memory-bound και επιβαρυμένος με atomic operations και συγχρονισμό σε κοινόχρηστες δομές, η περαιτέρω αύξηση των νημάτων δεν μπορεί να εκμεταλλευτεί αποτελεσματικά τις διαθέσιμες memory lanes και οδηγεί σε κορεσμό και σε περισσότερη εμπλοκή μεταξύ των νημάτων. Έτσι, το affinity βελτιώνει σημαντικά την απόδοση μέχρι τα 8 threads, αλλά τα εγγενή NUMA και synchronization bottlenecks της naive shared υλοποίησης εξακολουθούν να περιορίζουν την κλιμάκωση σε μεγαλύτερο αριθμό νημάτων.

2.1.1 – Copied Clusters and Reduce

Στη δεύτερη παράλληλη υλοποίηση του αλγορίθμου K-means υιοθετούμε και πάλι το μοντέλο shared clusters, αλλά αυτή τη φορά με τεχνική copied clusters και reduction για τη συλλογή των μερικών αποτελεσμάτων. Αντί όλα τα νήματα να ενημερώνουν απευθείας τους κοινόχρηστους πίνακες newClusters[] και newClusterSize[], κάθε νήμα διατηρεί δικά του, τοπικά αντίγραφα (local_newClusters[tid], local_newClusterSize[tid]) τα οποία ενημερώνει ελεύθερα, χωρίς atomic operations ή άλλον συγχρονισμό. Στο τέλος του παράλληλου βρόχου, τα τοπικά αυτά αντίγραφα συγχωνεύονται σε έναν κοινό πίνακα μέσω μιας φάσης reduction, η οποία εκτελείται από ένα νήμα (ή σε ένα μικρό, καλά οριοθετημένο σειριακό τμήμα κώδικα).

Η προσέγγιση αυτή αυξάνει λίγο τη χρήση μνήμης και προσθέτει ένα επιπλέον βήμα συγχώνευσης, αλλά μειώνει δραστικά το κόστος συγχρονισμού σε σχέση με τη naïve εκδοχή, καθώς αποφεύγονται οι χιλιάδες ατομικές ενημερώσεις πάνω στις ίδιες cache lines. Έτσι, η 2.1.2 στοχεύει σε πολύ καλύτερη κλιμάκωση με τον αριθμό νημάτων, ειδικά σε NUMA αρχιτεκτονικές, όπου η μείωση του contention στη μνήμη παίζει καθοριστικό ρόλο στην επίδοση.

Όσον αφορά τις λεπτομέρειες της υλοποίησής μας, επισημαίνουμε τα εξής:

- Ο παράλληλος βρόχος (#pragma omp parallel) αναθέτει σε κάθε νήμα ένα υποσύνολο αντικειμένων, όπως και πριν, αλλά οι ενημερώσεις των clusters γίνονται αποκλειστικά στα τοπικά arrays local_newClusterSize[tid] και local_newClusters[tid], χωρίς χρήση atomic.
- Η μεταβλητή delta, που μετράει τις αλλαγές στα memberships, υπολογίζεται πλέον με κατάλληλο reduction μέσα στο parallel, ώστε να αποφεύγονται επιπλέον ατομικές προσπελάσεις και να διατηρείται η ορθότητα του κριτηρίου σύγκλισης.
- Στο τέλος του βρόχου, μια #pragma omp single περιοχή εκτελεί τη φάση reduction, αθροίζοντας τα τοπικά αντίγραφα όλων των νημάτων στους κοινόχρηστους πίνακες newClusterSize[] και newClusters[]. Με αυτόν τον τρόπο συγκεντρώνονται τα μερικά αποτελέσματα με ελάχιστο συγχρονισμό, σε ένα καλά ελεγχόμενο σημείο του προγράμματος.

Ο ολοκληρωμένος κώδικας της υλοποίησης (omp_reduction_kmeans.c) βρίσκεται στον scirouter και στον orion της ομάδας μας, αλλά παρατίθεται και στην παρούσα αναφορά για λόγους πληρότητας.

a2/kmeans/omp_reduction_kmeans.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "kmeans.h"
4 /*
5  * TODO: include openmp header file
6  */
7 #include <omp.h>
8
9 // square of Euclid distance between two multi-dimensional points
10 inline static double euclid_dist_2(int numdims, /* no. dimensions */
11                                     double *coord1, /* [numdims] */
12                                     double *coord2) /* [numdims] */
13 {
14     int i;
15     double ans = 0.0;
16
17     for (i = 0; i < numdims; i++)
18         ans += (coord1[i] - coord2[i]) * (coord1[i] - coord2[i]);
19
20     return ans;
21 }
22
23 inline static int find_nearest_cluster(int numClusters, /* no. clusters */
24                                         int numCoords, /* no. coordinates */
25                                         double *object, /* [numCoords] */
26                                         double *clusters) /* [numClusters][numCoords] */
27 {
28     int index, i;
29     double dist, min_dist;
30
31     // find the cluster id that has min distance to object
32     index = 0;
33     min_dist = euclid_dist_2(numCoords, object, clusters);
34
35     for (i = 1; i < numClusters; i++)
36     {
37         dist = euclid_dist_2(numCoords, object, &clusters[i * numCoords]);
38         // no need square root
39         if (dist < min_dist)
40             { // find the min and its array index
41                 min_dist = dist;
42                 index = i;
43             }
44     }
45     return index;
46 }
47
48 void kmeans(double *objects, /* in: [numObjs][numCoords] */
49             int numCoords, /* no. coordinates */
50             int numObjs, /* no. objects */
51             int numClusters, /* no. clusters */

```

```

52     double threshold,      /* minimum fraction of objects that change membership */
53     long loop_threshold, /* maximum number of iterations */
54     int *membership,      /* out: [numObjs] */
55     double *clusters)    /* out: [numClusters][numCoords] */
56 {
57     int i, j, k;
58     int index, loop = 0;
59     double timing = 0;
60
61     double delta;          // fraction of objects whose clusters change in each loop
62     int *newClusterSize; // [numClusters]: no. objects assigned in each new cluster
63     double *newClusters; // [numClusters][numCoords]
64     int nthreads;         // no. threads
65
66     nthreads = omp_get_max_threads();
67     printf("OpenMP Kmeans - Reduction\t(number of threads: %d)\n", nthreads);
68
69     // initialize membership
70     for (i = 0; i < numObjs; i++)
71         membership[i] = -1;
72
73     // initialize newClusterSize and newClusters to all 0
74     newClusterSize = (typeof(newClusterSize))calloc(numClusters, sizeof(*newClusterSize));
75     newClusters = (typeof(newClusters))calloc(numClusters * numCoords,
76         sizeof(*newClusters));
76
77     // Each thread calculates new centers using a private space. After that, thread 0 does
78     // an array reduction on them.
78     int *local_newClusterSize[nthreads]; // [nthreads][numClusters]
79     double *local_newClusters[nthreads]; // [nthreads][numClusters][numCoords]
80
81     /*
82      * Hint for false-sharing
83      * This is noticed when numCoords is low (and neighboring local_newClusters exist
84      * close to each other).
85      * Allocate local cluster data with a "first-touch" policy.
86      */
86     // Initialize local (per-thread) arrays (and later collect result on global arrays)
87     for (k = 0; k < nthreads; k++)
88     {
89         local_newClusterSize[k] = (typeof(*local_newClusterSize))calloc(numClusters,
90             sizeof(**local_newClusterSize));
90         local_newClusters[k] = (typeof(*local_newClusters))calloc(numClusters * numCoords,
91             sizeof(**local_newClusters));
91     }
92
93     timing = wtime();
94     do
95     {
96         // before each loop, set cluster data to 0
97         for (i = 0; i < numClusters; i++)
98         {
99             for (j = 0; j < numCoords; j++)
100                 newClusters[i * numCoords + j] = 0.0;

```

```

101         newClusterSize[i] = 0;
102     }
103
104     // reset delta before each iteration; it will be updated via reduction in the
105     // parallel region
106     delta = 0.0;
107
108     /*
109      * TODO: Initialize local cluster data to zero (separate for each thread)
110      *
111      * We now use an OpenMP parallel region where:
112      * - Each thread zeroes its own local_newClusterSize/local_newClusters.
113      * - The object loop is distributed with 'omp for' and 'reduction(+ : delta)'.
114      * - A single thread reduces the per-thread local arrays into the shared arrays.
115      */
116 #pragma omp parallel private(i, j, k, index)
117 {
118     int tid = omp_get_thread_num();
119     int T   = omp_get_num_threads(); // actual number of threads in this team
120
121     /* per-thread zeroing (first-touch initialization of local cluster data) */
122     for (i = 0; i < numClusters; i++)
123         local_newClusterSize[tid][i] = 0;
124     for (i = 0; i < numClusters * numCoords; i++)
125         local_newClusters[tid][i] = 0.0;
126
127     // Distribute objects across threads and compute per-thread contributions.
128     // delta is accumulated using a reduction to avoid atomics on a shared
129     // variable.
130 #pragma omp for reduction(+ : delta)
131     for (i = 0; i < numObjs; i++)
132     {
133         // find the array index of nearest cluster center
134         index = find_nearest_cluster(numClusters, numCoords,
135                                     &objects[i * numCoords], clusters);
136
137         // if membership changes, increase delta by 1
138         if (membership[i] != index)
139             delta += 1.0;
140
141         // assign the membership to object i
142         membership[i] = index;
143
144         // update new cluster centers : sum of all objects located within (average
145         // will be performed later)
146         /*
147          * TODO: Collect cluster data in local arrays (local to each thread)
148          * Replace global arrays with local per-thread
149          */
150         local_newClusterSize[tid][index]++;
151         for (j = 0; j < numCoords; j++)
152             local_newClusters[tid][index * numCoords + j] += objects[i * numCoords
153 + j];
154     }

```

```

151
152     /*
153      * TODO: Reduction of cluster data from local arrays to shared.
154      * This operation will be performed by one thread
155      *
156      * Here we use 'omp single' so that exactly one thread accumulates
157      * all per-thread local arrays into the shared newClusterSize/newClusters.
158      */
159 #pragma omp single
160 {
161     for (k = 0; k < T; k++) // only sum over the threads actually in this
team
162     {
163         int *srcS = local_newClusterSize[k];
164         double *srcC = local_newClusters[k];
165         if (!srcS || !srcC)
166             continue;
167         for (i = 0; i < numClusters; i++)
168         {
169             newClusterSize[i] += srcS[i];
170             for (j = 0; j < numCoords; j++)
171                 newClusters[i * numCoords + j] += srcC[i * numCoords + j];
172         }
173     }
174     /* implicit barrier after single */
175 } /* end parallel region */

176
177 // average the sum and replace old cluster centers with newClusters
178 for (i = 0; i < numClusters; i++)
179 {
180     if (newClusterSize[i] > 0)
181     {
182         for (j = 0; j < numCoords; j++)
183         {
184             clusters[i * numCoords + j] = newClusters[i * numCoords + j] /
newClusterSize[i];
185         }
186     }
187 }

188
189 // Get fraction of objects whose membership changed during this loop. This is used
as a convergence criterion.
190     delta /= numObjs;

191
192     loop++;
193     printf("\r\tcompleted loop %d", loop);
194     fflush(stdout);
195 } while (delta > threshold && loop < loop_threshold);
196     timing = wtime() - timing;
197     printf("\n nloops = %3d (total = %7.4fs) (per loop = %7.4fs)\n", loop, timing, timing
/ loop);

198
199     for (k = 0; k < nthreads; k++)
200     {

```

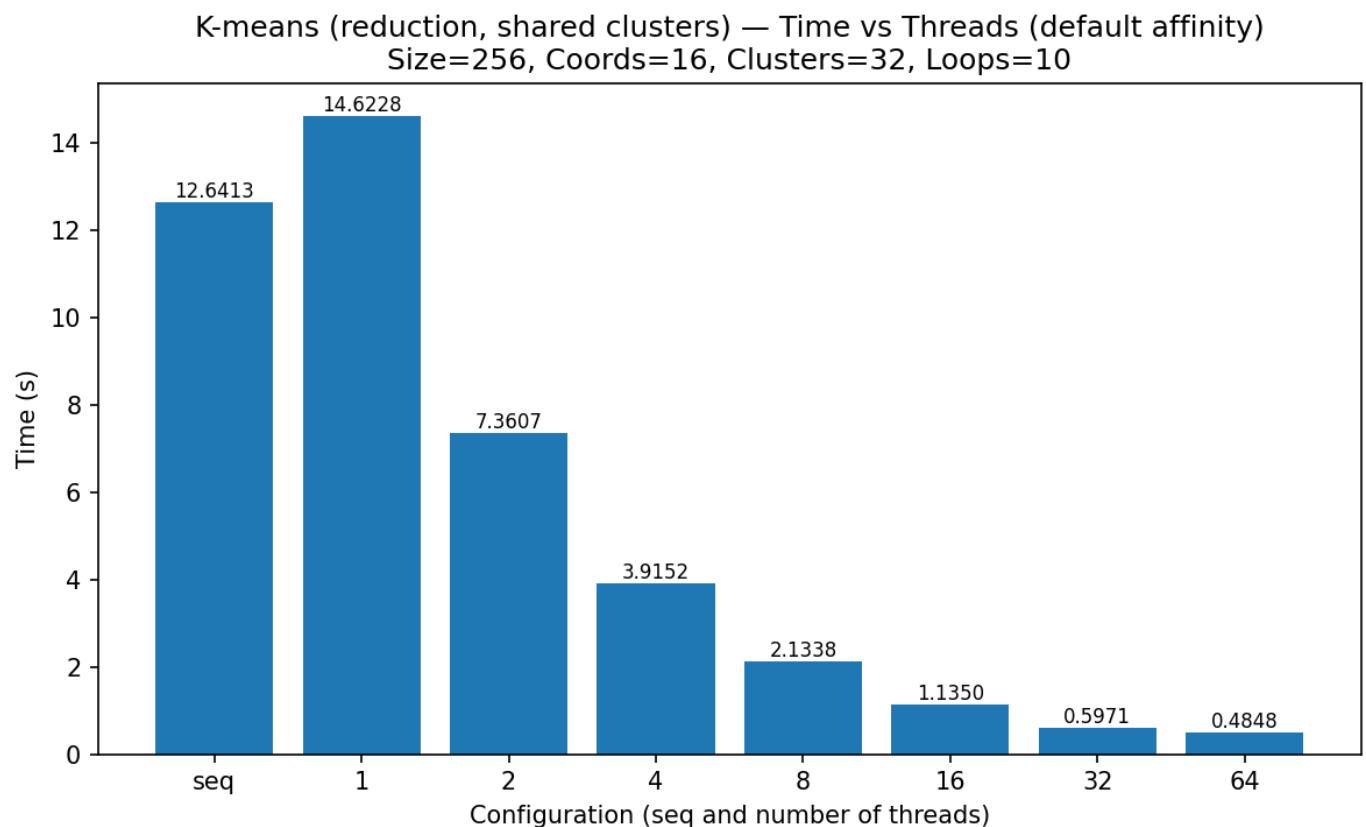
```
201     free(local_newClusterSize[k]);
202     free(local_newClusters[k]);
203 }
204 free(newClusters);
205 free(newClusterSize);
206 }
207
208 }
```

1. Παραλληλοποίηση για το αρχικό grid size και διαγράμματα

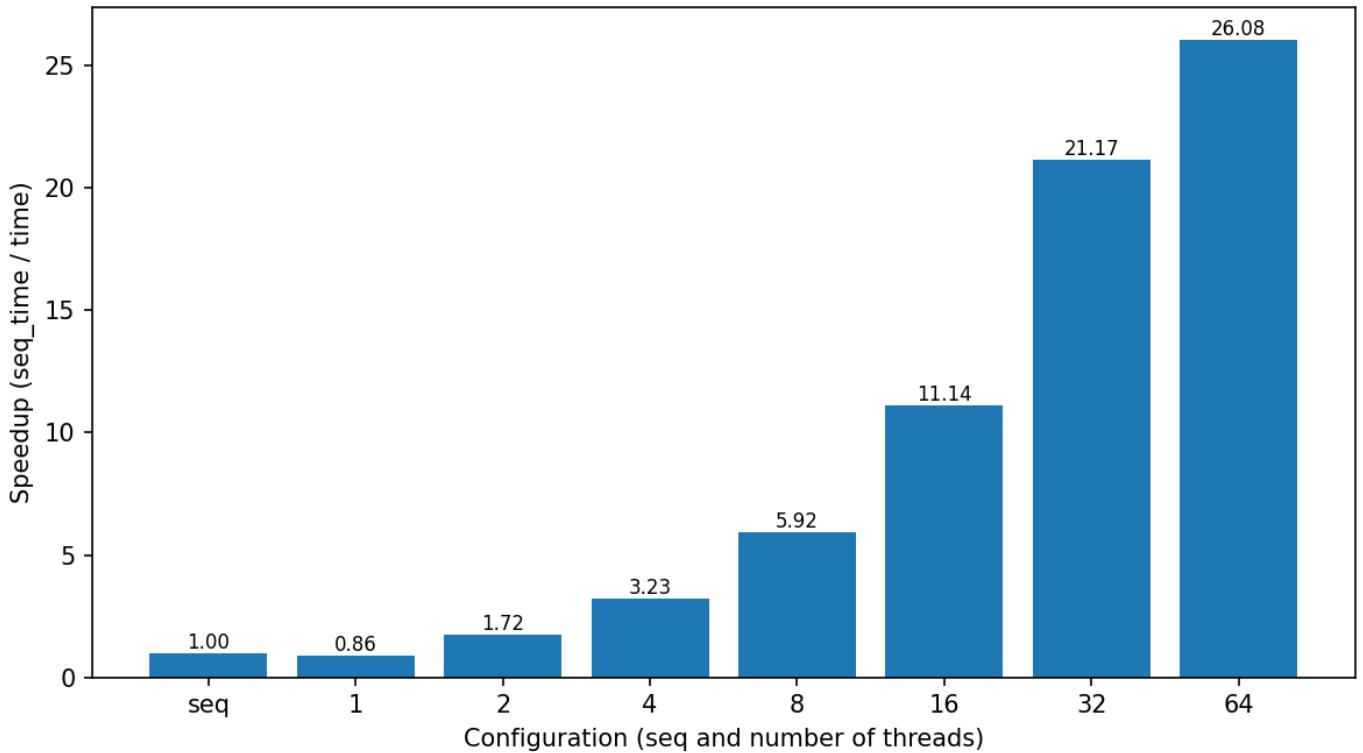
Το παραπάνω παράλληλο πρόγραμμα (omp_reduction_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και με affinity (όπως και πριν). Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	12.64
1	14.62
2	7.36
4	3.92
8	2.13
16	1.14
32	0.60
64	0.48

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



K-means (reduction, shared clusters) — Speedup vs Threads (default affinity)
Size=256, Coords=16, Clusters=32, Loops=10

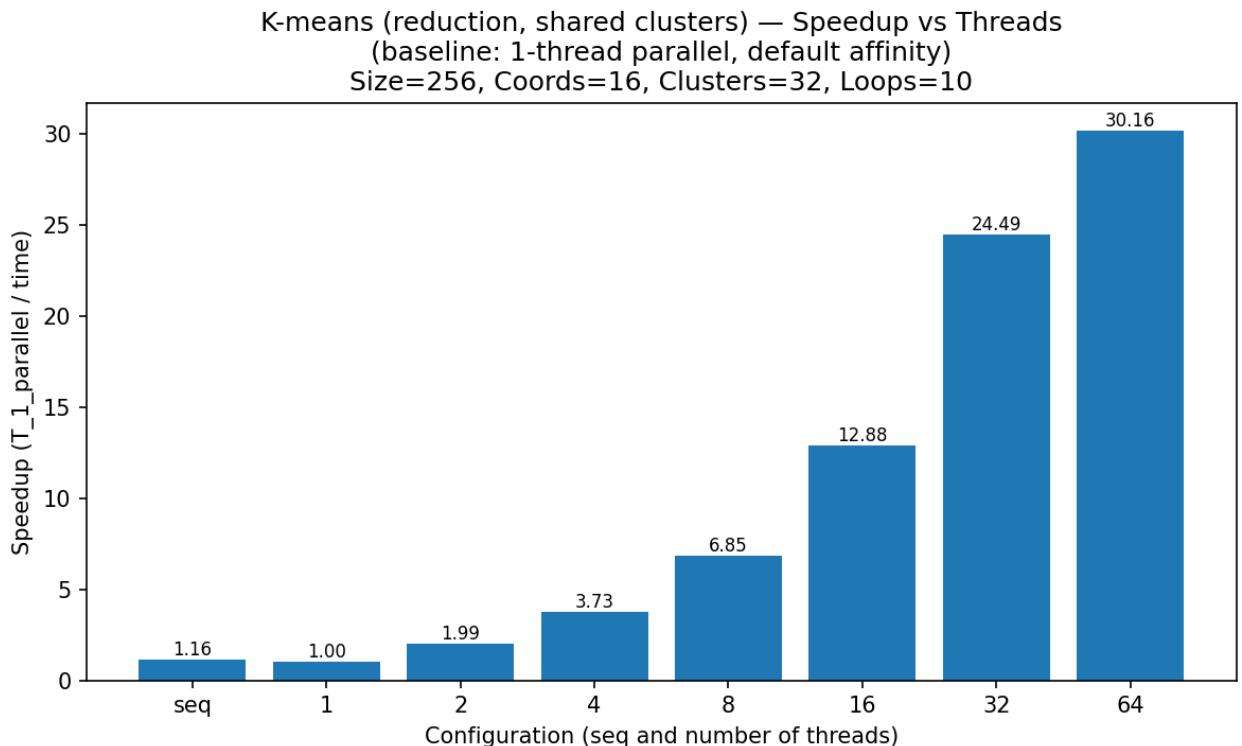


Από τα δύο πρώτα διαγράμματα παρατηρούμε ότι η έκδοση με copied clusters και reduction κλιμακώνει πλέον πολύ ικανοποιητικά σε σχέση με τη naive shared προσέγγιση. Ο σειριακός χρόνος είναι περίπου 12.6s, ενώ η παράλληλη εκτέλεση με 1 νήμα παραμένει λίγο χειρότερη (~14.6s), λόγω του parallel overhead του OpenMP, όπως και πριν. Ωστόσο, από τα 2 νήματα και πάνω ο χρόνος μειώνεται μονοτονικά και σχεδόν γραμμικά: ~7.4s στα 2 threads, ~3.9s στα 4, ~2.1s στα 8, ~1.1s στα 16, ~0.6s στα 32 και ~0.5s στα 64 threads. Αντίστοιχα, το speedup ως προς το σειριακό πρόγραμμα φτάνει περίπου τις 1.7x, 3.2x, 5.9x, 11x, 21x και 26x για 2, 4, 8, 16, 32 και 64 νήματα αντίστοιχα, πολύ κοντά στην ιδανική κλιμάκωση που παρουσιάζεται και στις διαφάνειες.

Η διαφορά σε σχέση με τη naive υλοποίηση εξηγείται από την αρχιτεκτονική της reduction λύσης: κάθε νήμα ενημερώνει αποκλειστικά τα δικά του local αντίγραφα (local_newClusters, local_newClusterSize), αποφεύγοντας atomic ενημερώσεις σε κοινές cache lines και μειώνοντας δραστικά το synchronization bottleneck. Έτσι, το μεγαλύτερο μέρος του χρόνου ξοδεύεται σε πραγματικό υπολογισμό και όχι σε συγχρονισμό, κάτι που επιτρέπει στον αλγόριθμο να κλιμακώνεται πολύ καλύτερα σε ένα NUMA σύστημα όπως ο sandman. Μέχρι τα 32 threads (ένας hardware thread ανά φυσικό πυρήνα) αξιοποιούνται αποτελεσματικά οι πόροι όλων των sockets, ενώ η μικρή “κάμψη” της κλιμάκωσης από τα 32 στα 64 νήματα αποδίδεται κυρίως στο hardware multithreading και στον κορεσμό του

memory bandwidth: δύο λογικά νήματα ανά πυρήνα μοιράζονται την ίδια εκτέλεση και τις ίδιες memory lanes σε έναν ήδη memory-bound αλγόριθμο.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



Το τελευταίο διάγραμμα, όπου ως βάση λαμβάνουμε την παράλληλη εκτέλεση με 1 νήμα, αναδεικνύει ακόμη πιο καθαρά το όφελος της reduction υλοποίησης. Σε αυτή τη σύγκριση, το σειριακό πρόγραμμα εμφανίζεται ήδη ταχύτερο από το 1-thread parallel (speedup $\approx 1.2 \times$), επιβεβαιώνοντας ότι το κόστος δημιουργίας και οργάνωσης της ομάδας νημάτων είναι σημαντικό όταν χρησιμοποιείται μόνο ένα νήμα. Από τα 2 threads και πάνω, όμως, η κλιμάκωση γίνεται εντυπωσιακή: το speedup φτάνει περίπου τις $2 \times$ στα 2 νήματα, $\sim 3.7 \times$ στα 4, $\sim 6.8 \times$ στα 8, $\sim 12.9 \times$ στα 16, $\sim 24.5 \times$ στα 32 και πάνω από $30 \times$ στα 64 νήματα σε σχέση με το 1-thread parallel.

Τα αποτελέσματα αυτά δείχνουν ότι, μόλις “ξεπληρωθεί” το parallel overhead, η copied clusters and reduction προσέγγιση εκμεταλλεύεται πλήρως την παραλληλία που προσφέρει ο κόμβος: μέχρι τα 32 threads αξιοποιείται ουσιαστικά κάθε φυσικός πυρήνας όλων των NUMA nodes, με πολύ μικρό synchronization cost χάρη στα local arrays, ενώ η περαιτέρω αύξηση στα 64 threads δίνει μικρότερο, αλλά υπαρκτό, πρόσθετο κέρδος λόγω του hardware multithreading. Σε αντίθεση με τη naive shared υλοποίηση, εδώ η κλιμάκωση περιορίζεται κυρίως από το διαθέσιμο memory

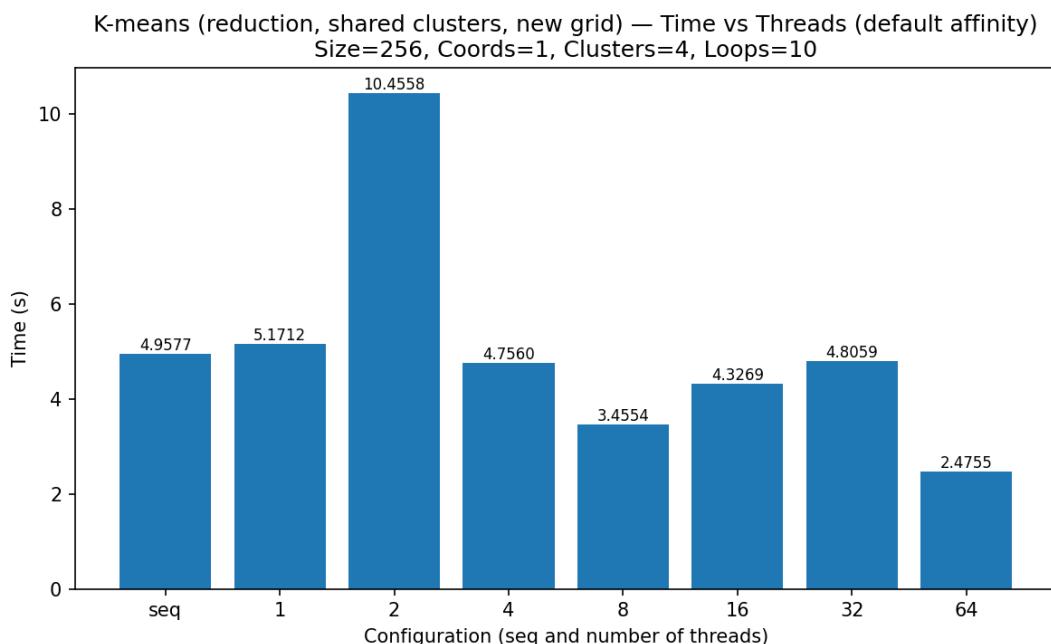
bandwidth και το μικρό σειριακό τμήμα του κώδικα (νόμος του Amdahl), ενώ το synchronization bottleneck έχει ουσιαστικά εξαλειφθεί.

2. Παραλληλοποίηση για το μειωμένο grid size και διαγράμματα

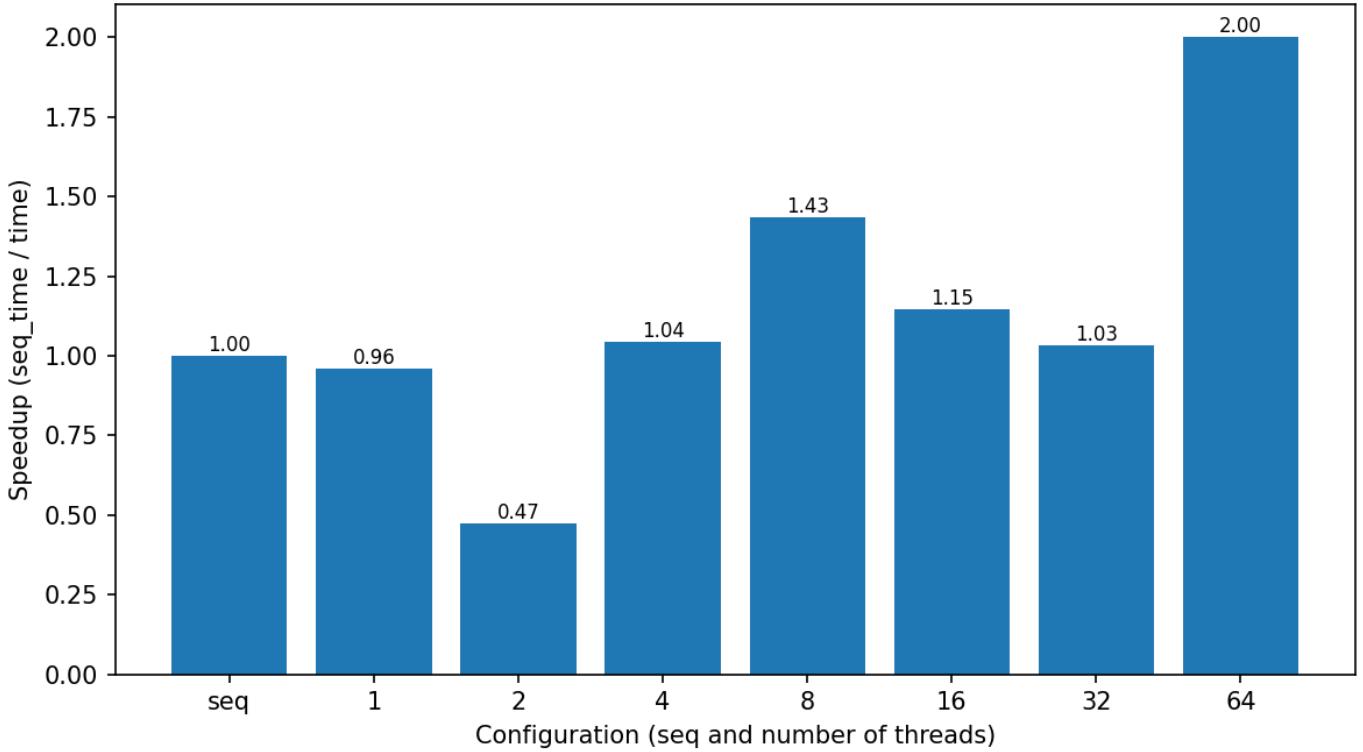
Το παραπάνω παράλληλο πρόγραμμα (omp_reduction_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και με affinity (όπως και πριν). Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

THREADS	TIME
seq	4.96
1	5.17
2	10.46
4	4.76
8	3.46
16	4.33
32	4.81
64	2.48

Τα ζητούμενα διαγράμματα (πάντα με βάση τον χρόνο του σειριακού προγράμματος, όπως αναφέρεται) φαίνονται ακολούθως:



K-means (reduction, shared clusters, new grid) — Speedup vs Threads (default affinity)
Size=256, Coords=1, Clusters=4, Loops=10

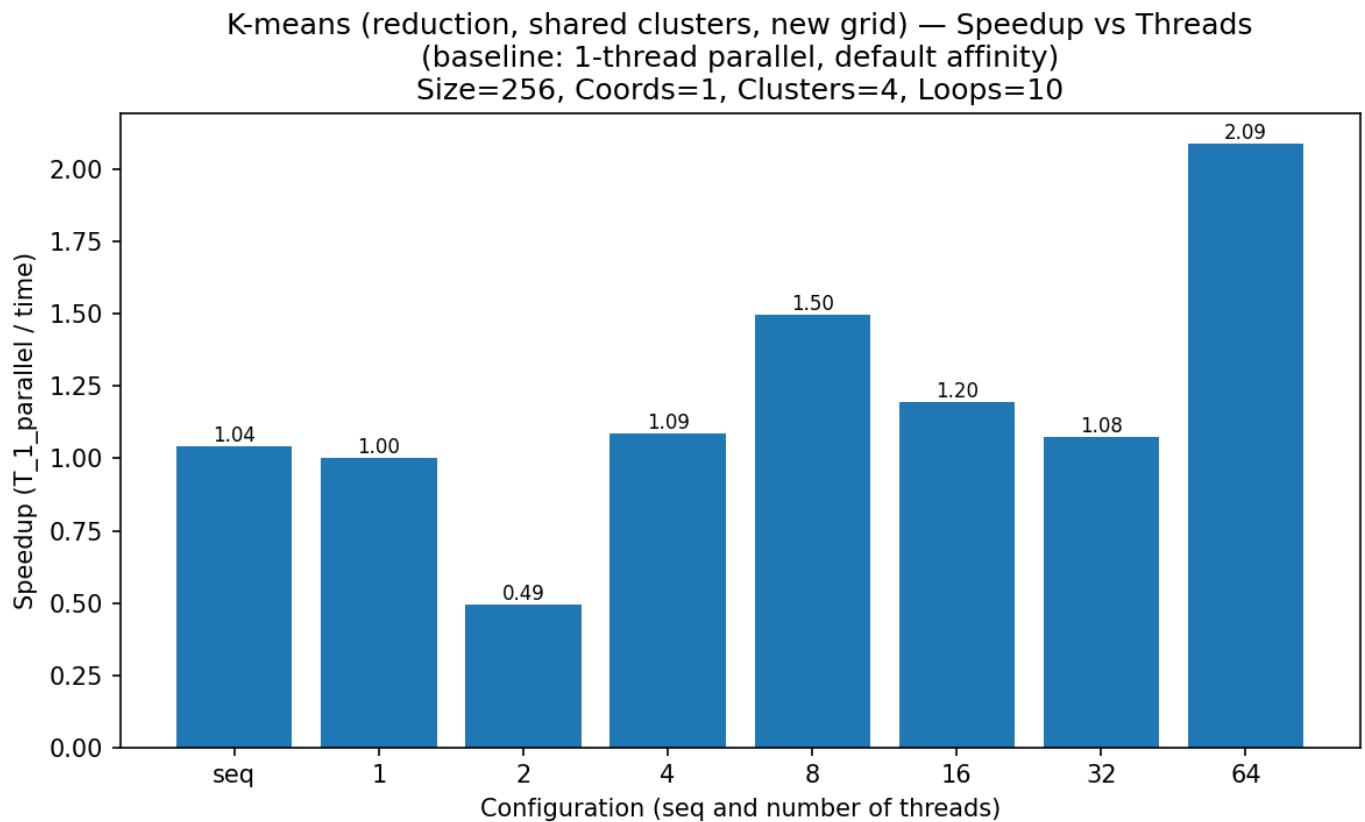


Από τα δύο πρώτα διαγράμματα για το μειωμένο grid παρατηρούμε ότι η συμπεριφορά της έκδοσης με reduction είναι σαφώς χειρότερη από αυτή στο αρχικό, πιο «υπολογιστικά βαρύ» (compute intensive) grid (<{256,16,32,10}). Εδώ ο σειριακός χρόνος είναι περίπου 4.96s και η παράλληλη εκτέλεση με 1 νήμα λίγο χειρότερη (~5.17s), αλλά η κλιμάκωση με περισσότερα νήματα δεν είναι πλέον καλή: στα 2 threads ο χρόνος μάλιστα χειροτερεύει σημαντικά (~10.46s), στα 4 και 8 νήματα έχουμε μια μικρή βελτίωση (4.76s και 3.46s αντίστοιχα), ενώ στα 16 και 32 νήματα ο χρόνος ξανανεβαίνει κοντά στον σειριακό (4.33s και 4.81s) και μόνο στα 64 threads πέφτει στα ~2.48s. Αντίστοιχα, το speedup ως προς το σειριακό πρόγραμμα μόλις που ξεπερνά το 1.4x στα 8 νήματα και φτάνει περίπου το 2x στα 64 νήματα.

Η βασική διαφορά στα scalability plots, σε σχέση με το προηγούμενο grid, είναι ότι εδώ η κλιμάκωση «επιπεδώνει» πολύ νωρίς και είναι έντονα ακανόνιστη. Στο αρχικό grid με 16 συντεταγμένες και 32 clusters, ο αλγόριθμος ήταν πολύ πιο compute-intensive και η έκδοση με reduction κατάφερνε να φτάσει speedup ~26x στα 64 threads. Αντίθετα, στο μειωμένο grid με μόνο 1 συντεταγμένη και 4 clusters, το workload ανά αντικείμενο είναι πολύ μικρότερο και η εφαρμογή είναι ακόμη πιο έντονα memory-bound: κάθε νήμα κάνει ελάχιστες πράξεις ανά προσπέλαση μνήμης, με αποτέλεσμα το κόστος πρόσβασης σε μνήμη και το overhead του OpenMP να κυριαρχούν. Έτσι, ο διαθέσιμος υπολογισμός δεν αρκεί για να «κρύψει»

τη latency της μνήμης και το scalability περιορίζεται σημαντικά, όπως φαίνεται στα διαγράμματα.

Από απλή περιέργεια και χωρίς να ζητείται, καταστρώσαμε και ένα διάγραμμα speedup με βάση όχι τώρα το σειριακό πρόγραμμα, αλλά το παράλληλο με 1 thread. Τα αποτελέσματα φαίνονται ακολούθως:



Στο τελευταίο διάγραμμα, όπου ως βάση λαμβάνουμε την παράλληλη εκτέλεση με 1 νήμα, γίνεται ακόμη πιο εμφανές ότι στο μειωμένο grid το scaling είναι περιορισμένο και ασταθές. Ο σειριακός κώδικας παραμένει ελαφρώς ταχύτερος από το 1-thread parallel, ενώ τα speedups ως προς το 1-thread πρόγραμμα κινούνται γύρω στη μονάδα για τα περισσότερα T: το 2-thread run είναι σαφώς χειρότερο (speedup < 1), στα 4 και 8 νήματα έχουμε κάποια βελτίωση, ενώ στα 16 και 32 νήματα ουσιαστικά δεν κερδίζουμε τίποτα. Μόνο στα 64 threads παρατηρείται πιο αξιοσημείωτο κέρδος, της τάξης περίπου του 2x, αλλά και πάλι πολύ μακριά από τα speedups που είδαμε στο αρχικό grid και την επιθυμητή κλιμάκωση.

Η διαφορά στα scalability plots σε σχέση με την περίπτωση {256,16,32,10} εξηγείται από το ότι εδώ το πρόβλημα είναι πλέον «πολύ μικρό» υπολογιστικά ανά στοιχείο και κυριαρχείται από τις προσπελάσεις στη μνήμη (memory-bound). Στο αρχικό grid, ο μεγάλος αριθμός συντεταγμένων και clusters παρείχε αρκετό

υπολογισμό ώστε η έκδοση με reduction να εκμεταλλεύεται ουσιαστικά όλους τους πυρήνες μέχρι τα 32 threads και να έχει πολύ καλή κλιμάκωση. Στο νέο grid, το per-object work είναι μικρό και η ταυτόχρονη πρόσβαση πολλών νημάτων στα ίδια δεδομένα καταναλώνει γρήγορα το memory bandwidth, με αποτέλεσμα ο επιπλέον παραλληλισμός να μην μπορεί να μεταφραστεί σε αντίστοιχο speedup. Ουσιαστικά, έχουμε ένα χαρακτηριστικό παράδειγμα από τις διαφάνειες: όσο πιο memory-bound είναι μια εφαρμογή και όσο μικρότερο το διαθέσιμο υπολογιστικό workload, τόσο πιο γρήγορα «σκάει» η κλιμάκωση και τα scalability plots γίνονται ρηχά και ασταθή.

Στο Linux, η πολιτική first-touch σε NUMA συστήματα ορίζει ότι οι φυσικές σελίδες μνήμης δεσμεύονται στο NUMA node του πυρήνα που τις «ακουμπά» πρώτος (πρώτη εγγραφή). Στο πρόγραμμά μας, οι πίνακες local_newClusters και local_newClusterSize δεσμεύονται αρχικά σειριακά (μέσα σε loops στο main thread), οπότε οι αντίστοιχες σελίδες μνήμης τοποθετούνται κατά κανόνα στο NUMA node όπου εκτελείται το main thread. Στη συνέχεια, όταν άλλα νήματα OpenMP που τρέχουν σε διαφορετικούς πυρήνες και nodes προσπελαύνουν αυτές τις δομές, μεγάλο μέρος των προσβάσεων είναι «απομακρυσμένο» (remote NUMA), με αυξημένη latency και μειωμένο effective bandwidth. Αυτό περιορίζει την επίδοση, ειδικά στη reduction υλοποίηση, όπου η πρόσβαση στα τοπικά clusters δεδομένα είναι πολύ συχνή.

Επιπλέον, εμφανίζεται και το φαινόμενο false-sharing: παρόλο που κάθε νήμα ενημερώνει διαφορετικά elements μέσα στο local_newClusters[tid], οι επιμέρους πίνακες για διαφορετικά tid μπορεί να τοποθετούνται σε συνεχόμενες διευθύνσεις και να μοιράζονται τις ίδιες cache lines. Έτσι, όταν δύο threads γράφουν σε διαφορετικά στοιχεία που τυχαίνει να κατοικούν στην ίδια γραμμή cache, οι γραμμές αυτές κάνουν συνεχώς invalidate μεταξύ των πυρήνων, δημιουργώντας σημαντικό overhead, χωρίς να υπάρχει πραγματικό data sharing σε επίπεδο προγράμματος.

Για να αντιμετωπίσουμε προβλήματα NUMA τοποθέτησης (first-touch), μπορούμε να αφήσουμε το κάθε νήμα να κάνει τη δική του δέσμευση μνήμης, π.χ. με malloc μέσα στην παράλληλη περιοχή: κάθε thread εκτελεί το malloc και την αρχικοποίηση των δικών του local_newClusters[tid] και local_newClusterSize[tid], οπότε οι σελίδες του κάθε πίνακα first-touched από το αντίστοιχο νήμα καταλήγουν στον «σωστό» NUMA node. Με αυτόν τον τρόπο, οι επαναλαμβανόμενες προσπελάσεις σε τοπικά δεδομένα γίνονται κυρίως σε τοπική μνήμη, μειώνοντας σημαντικά τα remote NUMA accesses.

Για να περιορίσουμε το false-sharing, κάναμε χρήση padding στα τοπικά arrays: φροντίζουμε κάθε «γραμμή» local_newClusters[tid] να ευθυγραμμίζεται σε μέγεθος cache line (π.χ. 64 bytes) και να μεσολαβεί αρκετό κενό (padding) ανάμεσα στα δεδομένα διαφορετικών νημάτων, ώστε καμία cache line να μην περιέχει ταυτόχρονα δεδομένα από δύο διαφορετικά tid. Με αυτόν τον τρόπο, κάθε γραμμή cache ανήκει ουσιαστικά σε ένα μόνο thread και δεν υπάρχει αλληλοεπικάλυψη που θα προκαλούσε invalidations. Συνδυάζοντας την τεχνική του per-thread malloc (για σωστό first-touch ανά thread και NUMA node) με το κατάλληλο padding, μειώνουμε τόσο τα προβλήματα NUMA τοποθέτησης όσο και τα φαινόμενα false-sharing, όπως προτείνεται και στο hint της εκφώνησης.

Γενικές Παρατηρήσεις

- Η υλοποίηση με reduction αποδείχθηκε σαφώς πιο αποδοτική από αυτή με atomic operations, καθώς μεταφέρει το κόστος συγχρονισμού σε ένα μικρό, καλά οριοθετημένο στάδιο συγχώνευσης (reduction) και αφήνει τον κυρίως βρόχο να εκτελείται χωρίς locks. Αντίθετα, στην παίνε υλοποίηση μεγάλο μέρος του χρόνου χάνεται σε atomic ενημερώσεις πάνω στις ίδιες cache lines. Παρ' όλα αυτά, το reduction δεν είναι πάντα προτιμότερο: σε σενάρια με μικρό πρόβλημα ή λίγες συγκρούσεις στα shared δεδομένα, το επιπλέον κόστος αντιγραφής και συγχώνευσης μπορεί να εξανεμίσει τα κέρδη.
- Πέραν των 32 threads δεν παρατηρείται ποτέ ουσιαστική (σχεδόν γραμμική) βελτίωση, καθώς στο sandman έχουμε 32 φυσικούς πυρήνες και 64 λογικά νήματα μέσω hardware multithreading (hyperthreading). Σε έναν κατά βάση memory-bound αλγόριθμο, όπως ο K-means με shared clusters, δύο λογικά νήματα στον ίδιο πυρήνα μοιράζονται τους ίδιους execution πόρους και τις ίδιες memory lanes, οπότε η περαιτέρω αύξηση των νημάτων δεν μεταφράζεται σε αντίστοιχο speedup και μπορεί να οδηγήσει ακόμη και σε υποβάθμιση της επίδοσης.
- Η επιλογή μιας κατάλληλης πολιτικής affinity βελτιώνει αισθητά την επίδοση. Η δέσμευση των νημάτων σε συγκεκριμένους πυρήνες (ώστε να «μένουν κοντά» σε δεδομένα και cache) μειώνει το κόστος επικοινωνίας με τη μνήμη και εκμεταλλεύεται καλύτερα την τοπικότητα. Παράλληλα, η προσεκτική διασπορά των threads στα NUMA nodes μπορεί να αυξήσει το διαθέσιμο memory bandwidth για memory-bound εφαρμογές. Τα πειράματά μας δείχνουν ξεκάθαρα ότι με ενεργό affinity η επίδοση μέχρι τα 8–16 threads βελτιώνεται σημαντικά σε σχέση με την πλήρως “noaff” περίπτωση.

▪ Ενότητα 2.2 – Παραλληλοποίηση του Αλγορίθμου Floyd-Warshall

- **Υλοποίηση και ανάλυση εξαρτήσεων**

Για την παραλληλοποίηση του recursive Floyd-Warshall άλγορίθμου χρησιμοποιήθηκαν OpenMP tasks. Η βασική πρόκληση προέκυψε από το γεγονός ότι ο αλγόριθμος παίρνει ως ορίσματα τον ίδιο πίνακα (A) τρεις φορές (A,B,C), αν και με διαφορετικά offsets. Αυτό έχει ως αποτέλεσμα, παρόλο που τα A,B,C έχουν διαφορετικό όνομα, να αναφέρονται συχνά στην ίδια θέση μνήμης, δημιουργώντας εξαρτήσεις τύπου RAW (Read After Write) και WAW (Write After Write).

- **Μελέτη των Αναδρομικών Κλήσεων**

Η μελέτη των εξαρτήσεων ανά αναδρομική κλήση βασίστηκε στις ακόλουθες παρατηρήσεις:

1. Κάθε αναδρομική κλήση γίνεται για blocks μισού μεγέθους σε σχέση με αυτά της εισόδου (3 blocks/arrays ίδιων διαστάσεων)
2. Κάθε πίνακας/block χωρίζεται σε 4 ίσα μέρη, επιτρέποντας σε διαφορετικά blocks ίδιων διαστάσεων να μην έχουν μερικές επικαλύψεις.
3. Σε πιο "βαθύ επίπεδο" της αναδρομής, ο αλγόριθμος περιορίζεται σε εγγραφή στον A (και υπό-blocks του) και ανάγνωση από τους B,C (και υπό-blocks τους) στην δεδομένη κλήση. Αυτό εξασφαλίζει ότι σε βαθύτερο επίπεδο δεν υπάρχει επικάλυψη πρόσβασης μνήμης που δεν φαίνεται στο ψηλότερο.

Μελετώντας τις κλήσεις, το πρόβλημα χωρίστηκε σε 4 περιπτώσεις ανάλογα με τις σχέσεις των blocks A,B,C.

- **Σενάρια Παραλληλισμού (Task Chains)**

Θεωρώντας ως call_i την i-οστή σε σειρά αναδρομική κλήση του παρακάτω αλγορίθμου βγάζουμε τα συμπεράσματα του πίνακα:

FWR (A00, B00, C00);

FWR (A01, B00, C01);

FWR (A10, B10, C00);

FWR (A11, B10, C01);

FWR (A11, B10, C01);

FWR (A10, B10, C00);

FWR (A01, B00, C01);

FWR (A00, B00, C00);

Case ID	Σχέση Blocks	Αλυσίδα Εξαρτήσεων	Παραλληλισμός
1	A=B=C (όλα ταυτίζονται)	1 -> {2,3} -> 4 -> 5 -> {6,7}->8	Τα ζευγάρια {2, 3} και {6, 7} εκτελούνται παράλληλα 10.
2/3	A=B ή A=C (αλλά B ≠ C)	Αν A=B: 1->2->7->8. και 3->4->5->6. Αν A=C: 1->3->6->8 και 2->4->5->6->8	Προκύπτουν 2 παράλληλες και εντελώς ανεξάρτητες αλυσίδες εκτέλεσης
4	A≠ B≠ C (όλα διαφορετικά)	1-> 8, 2->7, 3->6, 4->5	Προκύπτουν 4 παράλληλες και εντελώς ανεξάρτητες αλυσίδες

Στην αρχική κλήση ($A, 0, 0, A, 0, 0, A, 0, 0$), εφαρμόζεται η Case 1. Η εξάρτηση RAW των κλήσεων 2, 3 από το 1, και η εξάρτηση του 4 από 2, 3, καθορίζουν τη σειρά εκτέλεσης.

- **Επιλογή block size (B)**

Για την επιλογή του μεγέθους του ελάχιστου block (bsize) όπου γίνεται ο υπολογισμός του min (base case), δοκιμάστηκαν $B=\{16, 32, 64, 128, 256\}$ για $N=1024$:

- ✓ Παρατηρήθηκε ότι η επίδοση βελτιωνόταν σημαντικά με την αύξηση του B.
- ✓ Στο $B=128$ δεν σημειώθηκε βελτίωση, ενώ στο $B=256$ ο χρόνος εκτέλεσης μάλιστα αυξήθηκε.
- ✓ Επιλέχθηκε $B=64$ για την υπόλοιπη άσκηση. Αν και το $B=128$ ήταν αποδοτικό, το $B=64$ προτιμήθηκε για να αξιοποιηθεί περισσότερο η παραλληλία, καθώς μεγαλύτερο B αντιστοιχεί σε λιγότερα layers παράλληλου προγράμματος. Η επιλογή του B επηρεάζει ελάχιστα το speedup, καθώς επηρεάζει με τον ίδιο τρόπο το σειριακό και το παράλληλο πρόγραμμα.

- **Παράμετροι Εκτέλεσης και Περιβάλλοντος**

Η εκτέλεση και οι μετρήσεις του παράλληλου αλγορίθμου Recursive Floyd-Warshall (FW_SR) πραγματοποιήθηκαν μέσω του script run_on_queue.sh στο περιβάλλον PBS (Portable Batch System) του συστήματος sandman. Η εργασία υποβλήθηκε στην ουρά serial ζητώντας 64 πυρήνες (ppn=64) στον κόμβο sandman. Το πρόγραμμα εκτελέστηκε επαναληπτικά για τρία μεγέθη πίνακα (N): 1024, 2048, και 4096, χρησιμοποιώντας την βελτιστοποιημένη τιμή $B=64$ για το μέγεθος του ελάχιστου block. Ο παραλληλισμός OpenMP ελέγχθηκε μέσω της μεταβλητής περιβάλλοντος OMP_NUM_THREADS, η οποία διατρέχθηκε στις τιμές $T=\{1, 2, 4, 8, 16, 32, 64\}$. Κάθε συνδυασμός N και T καταγράφηκε σε ξεχωριστά αρχεία εξόδου (.out και .err), εξασφαλίζοντας ακριβή δεδομένα για την ανάλυση της κλιμάκωσης του αλγορίθμου.

Τα run_on_queue.sh και το κυρίως πρόγραμμα fw_sr_p.c φαίνονται ακολούθως:

a2/FW/run_on_queue.sh

```
1 #!/bin/bash
2
3 #PBS -N run_fw_sr_p
4
5 ## Output error
6 #PBS -o run_fw_sr_p.pbs_out
7 #PBS -e run_fw_sr_p.pbs_err
8
9 ## Sandman, serial queue, 64 threads
10 #PBS -q serial
11 #PBS -l nodes=sandman:ppn=64
12
13 #PBS -l walltime=01:00:00
14
15 cd /home/parallel/parlab05/a2/FW
16
17 module load openmp
18
19 N_VALUES="1024 2048 4096"
20
21 # Block size
22 B=64
23
24 OUTDIR="benchmarks"
25 mkdir -p "$OUTDIR"
26
27 for N in $N_VALUES; do
28     for T in 1 2 4 8 16 32 64; do
29
30         export OMP_NUM_THREADS=$T
31         echo "Running N=$N, B=$B, threads=$T"
32
33         #outputs
34         OUT="${OUTDIR}/fw_sr_p_N${N}_T${T}.out"
35         ERR="${OUTDIR}/fw_sr_p_N${N}_T${T}.err"
36
37         # - stdout → OUT
38         # - stderr → ERR
39         ./fw_sr_p "$N" "$B" >"$OUT" 2>"$ERR"
40     done
41 done
42
43
```

```

a2/FW/fw_sr_p.c

1  /*
2   * Recursive implementation of the Floyd-Warshall algorithm.
3   * command line arguments: N, B
4   * N = size of graph
5   * B = size of submatrix when recursion stops
6   * works only for N, B = 2^k
7   */
8
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <sys/time.h>
12 #include <omp.h>
13 #include "util.h"
14
15 inline int min(int a, int b);
16 void FW_SR (int **A, int arow, int acol,
17             int **B, int brow, int bcol,
18             int **C, int crow, int ccol,
19             int myN, int bsize);
20
21 int main(int argc, char **argv)
22 {
23     int **A;
24     int i,j,k;
25     struct timeval t1, t2;
26     double time;
27     int B=16;
28     int N=1024;
29
30     if (argc !=3){
31         fprintf(stdout, "Usage %s N B \n", argv[0]);
32         exit(0);
33     }
34
35     N=atoi(argv[1]);
36     B=atoi(argv[2]);
37
38     if ((N%B)!=0){
39         fprintf(stdout, "N must be multiple of B\n");
40         exit(0);
41     }
42
43     A = (int **) malloc(N*sizeof(int *));
44     for(i=0; i<N; i++) A[i] = (int *) malloc(N*sizeof(int));
45
46     graph_init_random(A, -1, N, 128*N);
47
48 //-----
49     gettimeofday(&t1,0);
50
51     #pragma omp parallel

```

```

52 #pragma omp single
53 {
54     FW_SR(A,0,0, A,0,0,A,0,0,N,B);
55 }
56
57     gettimeofday(&t2,0);
58
59     time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)/1000000;
60     printf("FW_SR,%d,%d,%4f\n", N, B, time);
61
62
63 // for(i=0; i<N; i++)
64 //     for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
65
66
67     return 0;
68 }
69
70 inline int min(int a, int b)
71 {
72     if(a<=b) return a;
73     else return b;
74 }
75
76 void FW_SR (int **A, int arow, int acol,
77             int **B, int brow, int bcol,
78             int **C, int crow, int ccol,
79             int myN, int bsize)
80 {
81     int k,i,j;
82     /*we use different task paral depending on the blocks A,B,C use.
83      If they use same blocks therre may be future depedencies , else not*/
84
85     //Arrays check
86     if (A!=B || A!=C){
87         printf("Different arrays not supported yet\n");
88         exit (1);
89     }
90     //row check
91     int RAB= arow==brow;
92     int RAC= arow==crow;
93     int RBC= brow==crow;
94     //col check
95     int CAB= acol==bcol;
96     int CAC= acol==ccol;
97     int CBC= bcol==ccol;
98     //case check
99     int case_id;
100    if (RAB&&RAC&&CAB&&CAC) case_id=0; //A,B,C same block
101    else if (RAB&&CAB) case_id=1; //A,B same block
102    else if (RAC&&CAC) case_id=2; //A,C same block
103    else case_id=3; //A separate from B and C
104
105    /*

```

```

106     * The base case (when recursion stops) is not allowed to be edited!
107     * What you can do is try different block sizes.
108     */
109     if(myN<=bsize)
110         for(k=0; k<myN; k++)
111             for(i=0; i<myN; i++)
112                 for(j=0; j<myN; j++)
113                     A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k]
114 [ccol+j]);
115     else {
116
117         switch(case_id){
118             case 0: //A,B,C same block
119             {
120                 //call1
121                 FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
122
123                 #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
124                 shared(A,B,C)
125                 {
126                     //call2
127                     FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2,
128 bsize);
129                 }
130                 #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
131                 shared(A,B,C)
132                 {
133                     //call3
134                     FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2,
135 bsize);
136
137                     //call4
138                     FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
139 myN/2, bsize);
140
141                     //call5
142                     FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
143 ccol+myN/2, myN/2, bsize);
144
145                     #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
146                     shared(A,B,C)
147                     {
148                         //call6
149                         FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol,

```

```

150         #pragma omp taskwait
151
152             //call8
153             FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
154         }
155         break;
156         case 1: //A,B same block
157         {
158             #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
159             shared(A,B,C)
160             {
161                 //call1
162                 FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
163                 //call2
164                 FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2,
165                 bsize);
166                 //call7
167                 FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
168                 myN/2, bsize);
169                 //call8
170                 FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2,
171                 bsize);
172                 //call9
173                 FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2,
174                 bsize);
175                 //call14
176                 FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
177                 myN/2, bsize);
178                 //call15
179                 FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
180                 ccol+myN/2, myN/2, bsize);
181                 //call16
182                 FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol,
183                 myN/2, bsize);
184             }
185             #pragma omp taskwait
186         }
187         break;
188         case 2: //A,C same block
189         {
190             #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
191             shared(A,B,C)
192             {
193                 //call1
194                 FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
195                 //call3
196                 FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2,
197                 bsize);
198                 //call16

```

```

193         FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol,
194         myN/2, bsize);
195             //call18
196             FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2,
197             bsize);
198         }
199             //call12
200             FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2,
201             bsize);
202             //call14
203             FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
204             myN/2, bsize);
205             //call15
206             FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
207             ccol+myN/2, myN/2, bsize);
208             //call17
209             FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
210             myN/2, bsize);
211         }
212         #pragma omp taskwait
213     }
214     break;
215     case 3: //A separate from B and C
216     #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
217     shared(A,B,C)
218     {
219         //call11
220         FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
221         //call18
222         FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
223     }
224     #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
225     shared(A,B,C)
226     {
227         //call12
228         FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
229         //call17
230         FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
231         myN/2, bsize);
232     }
233     #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
234     shared(A,B,C)
235     {
236         //call13
237         FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
238         //call16
239         FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol,
240         myN/2, bsize);
241     }

```

```
235     #pragma omp task firstprivate(arow,acol,brow,bcol,crow,ccol,myN,bsize)
236     shared(A,B,C)
237     {
238         //call4
239         FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2,
240         myN/2, bsize);
241         //call5
242         FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2,
243         ccol+myN/2, myN/2, bsize);
244         }
245         #pragma omp taskwait
246
247     }
248
249 /*
250 call1
251     FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
252 call2
253     FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
254 call3
255     FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
256 call4
257     FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2,
258     bsize);
259 call5
260     FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
261     myN/2, bsize);
262 call6
263     FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2,
264     bsize);
265 call7
266     FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2,
267     bsize);
268 call8
269     FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
*/

```

- **Αποτελέσματα Μετρήσεων Επίδοσης**

Αρχικά δοκιμάσαμε να τρέξουμε τον σειριακό (επαναληπτικό) αλγόριθμο. Ο χρόνος εκτέλεσης για N=1024 ήταν 1.3954.

Στη συνέχεια, τρέχοντας το πρόγραμμα με τις παραμέτρους της προηγούμενης παραγράφου στο sandman πήραμε τα εξής αποτελέσματα:

Χρόνοι εκτέλεσης

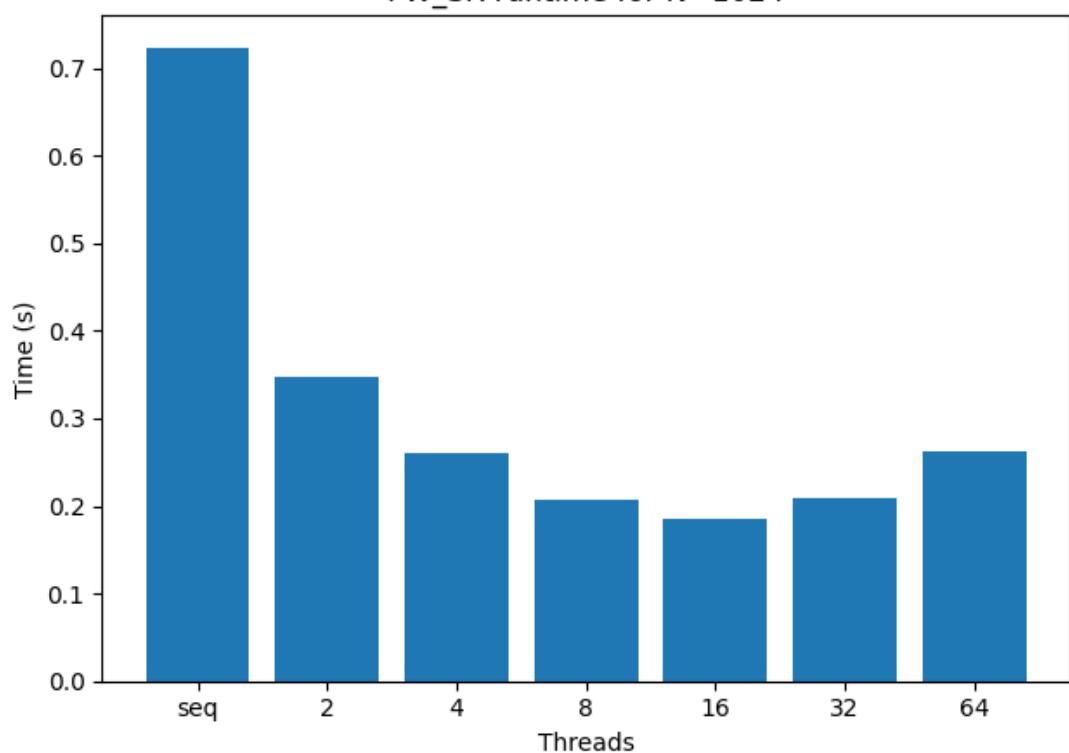
N	Tseq (T=1)	T=2	T=4	T=8	T=16	T=32	T=64
1024	0.7237	0.3467	0.2604	0.2078	0.1850	0.2097	0.2627
2048	5.1359	2.6523	1.9513	1.1305	0.7720	0.7496	0.9577
4096	43.4211	21.7275	14.3879	7.9806	4.4654	3.0423	3.1596

Συντελεστής επιτάχυνσης (speedup)

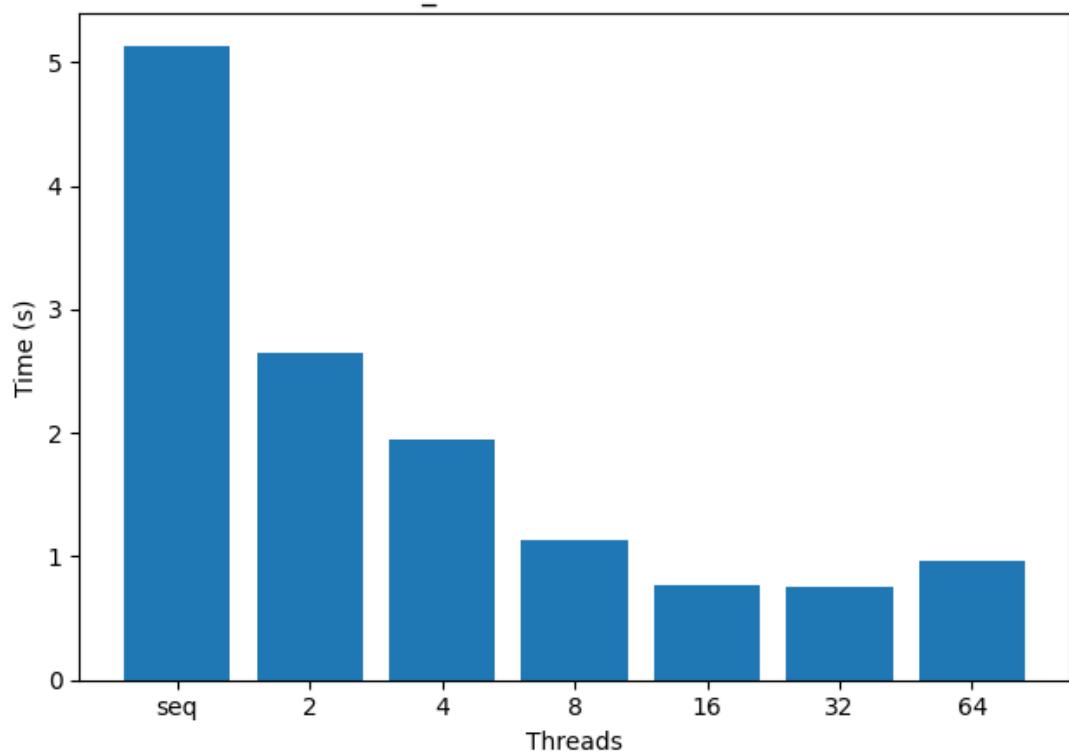
N	T=2	T=4	T=8	T=16	T=32	T=64	Max S
1024	2.09	2.78	3.48	3.91	3.45	2.75	3.91
2048	1.94	2.63	4.54	6.65	6.85	5.36	6.85
4096	2.00	3.02	5.44	9.72	14.27	13.74	14.27

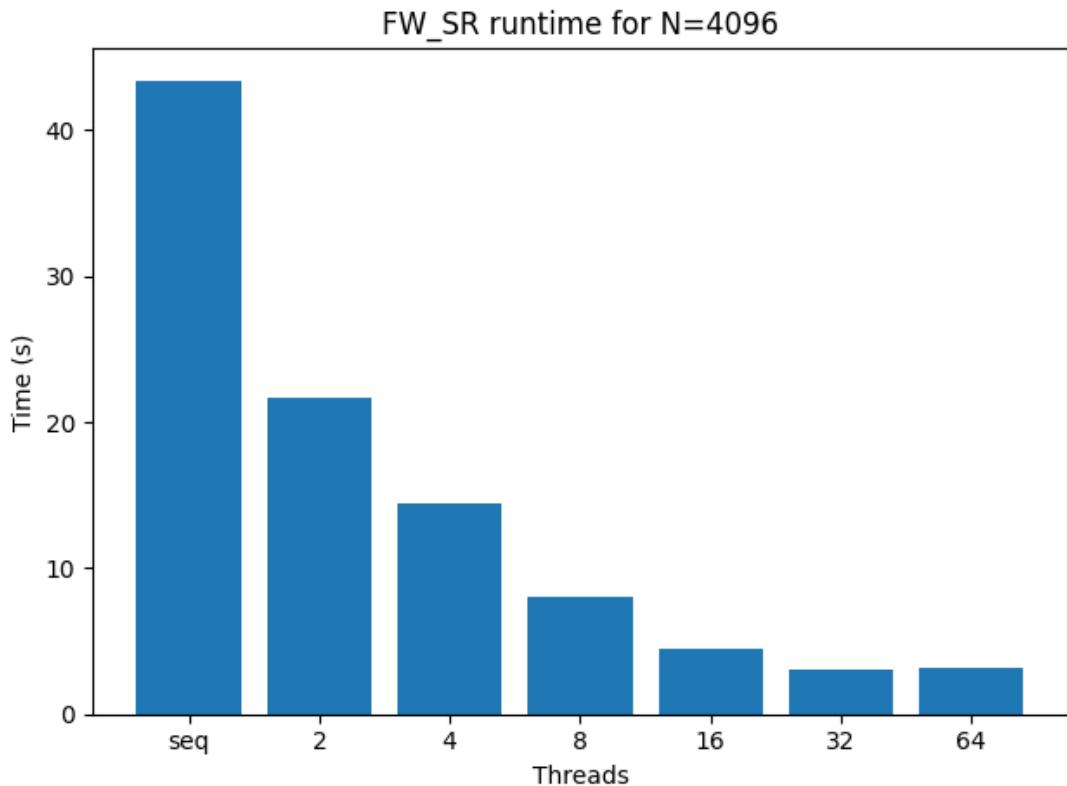
Ακολουθούν και τα barplots για τα runs των διαφορετικών N:

FW_SR runtime for N=1024



FW_SR runtime for N=2048





- **Συμπεράσματα και παρατηρήσεις**

- ✓ Αρχικά παρατηρούμε ότι η χρήση του σειριακού recursive αλγορίθμου έφερε από μόνη της σημαντική βελτίωση (από 1.3954 σε 0.7237). Αυτή οφείλεται πιθανώς στην βελτίωση της cache locality που επιτυγχάνεται με τη recursive δομή του αλγορίθμου. Αυτή η αναδρομική διάσπαση του πίνακα σε μικρότερα blocks επιτρέπει την πιο αποδοτική χρήση της ιεραρχίας της κρυφής μνήμης (cache hierarchy), μειώνοντας δραστικά τα misses.
- ✓ Η κλιμάκωση (speedup) είναι καλύτερη για το μεγαλύτερο dataset ($N=4096$), φτάνοντας το $14.27\times$ στους 32 threads. Αυτό συμβαίνει επειδή, για μεγάλο N , ο τεράστιος όγκος των υπολογισμών ($O(N^3)$) κυριαρχεί έναντι του overhead του OpenMP tasking και των καθυστερήσεων της μνήμης, επιτρέποντας την καλύτερη αξιοποίηση της παραλληλίας.

- ✓ Το ταβάνι στο scalability (Max Speedup) οφείλεται στους thread overheads και στη δομή του taskgraph, η οποία εισάγει περιορισμένη παραλληλία λόγω των εξαρτήσεων ιδιαίτερα στο case 1, όπου οι σειριακές εκτελέσεις μειώνονται απλά σε 6 από τις 8 αρχικές.
- ✓ Η αύξηση των threads στους 64 οδηγεί σε αύξηση του χρόνου εκτέλεσης σε όλες τις περιπτώσεις, υποδηλώνοντας ότι το overhead του tasking υπερβαίνει το όφελος της παραλληλοποίησης.

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Νοέμβριος 2025

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 2^{ης} ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 25.11.2025

▪ Αμοιβαίος Αποκλεισμός – Κλειδώματα

1. Εισαγωγή και Αρχεία

Στην παρούσα άσκηση μελετάμε διάφορους μηχανισμούς locks σε πολυνηματικές εφαρμογές, μέσω του αλγορίθμου ταξινόμησης K-means (στη shared έκδοσή του). Στόχος μας, είναι να κατανοήσουμε πώς διαφορετικές στρατηγικές συγχρονισμού (απουσία κλειδώματος, κλασικά mutex και spinlocks, locks τύπου TAS/TTAS, ιεραρχικά κλειδώματα τύπου array και CLH, καθώς και η εντολή critical του OpenMP) επηρεάζουν τον χρόνο εκτέλεσης και τη δυνατότητα κλιμάκωσης της εφαρμογής, όταν αυτή εκτελείται σε ένα πολυπύρηνο σύστημα με μέχρι και 64 λογικά νήματα (το sandman έχει $4 \times 8 = 32$ φυσικούς πυρήνες και $32 \times 2 = 64$ λογικούς). Το κυρίως σώμα του προγράμματος παραμένει το ίδιο και μεταβάλλεται μόνο ο μηχανισμός του lock, ώστε να μπορούμε να απομονώσουμε και να συγκρίνουμε το καθαρό κόστος συγχρονισμού κάθε κλειδώματος.

Ο δοσμένος κώδικας βασίζεται στον K-means από την προηγούμενη άσκηση και αποτελείται από τα κοινά αρχεία file_io.c και util.c, το header file kmeans.h και το πρόγραμμα main.c, το οποίο αναλαμβάνει να διαβάσει τις παραμέτρους και να καλέσει την υλοποίηση του αλγορίθμου. Το αρχείο seq_kmeans.c περιέχει τη σειριακή έκδοση του K-means, ενώ το omp_naive_kmeans.c παρέχει μια απλή παράλληλη προσέγγιση με OpenMP. Για τη μελέτη των κλειδωμάτων χρησιμοποιούνται δύο επιπλέον αρχεία: το omp_critical_kmeans.c, όπου ο συγχρονισμός υλοποιείται με #pragma omp critical, και το omp_lock_kmeans.c, το οποίο ορίζει την παράλληλη εκδοχή που βασίζεται σε μια διεπαφή με κλειδώματα, ως παράμετρο. Η υλοποίηση των επιμέρους κλειδωμάτων γίνεται στον φάκελο locks/, μέσω των αρχείων nosync_lock.c, pthread_mutex_lock.c, pthread_spin_lock.c, tas_lock.c, ttas_lock.c, array_lock.c και clh_lock.c, τα οποία μοιράζονται το κοινό header lock.h (ορισμός του τύπου lock_t και των συναρτήσεων lock_init, lock_acquire, lock_release, lock_free) και το αρχείο alloc.h για τις βοηθητικές δεσμεύσεις μνήμης. Έτσι, με κατάλληλο linking στο Makefile προκύπτουν τα διαφορετικά εκτελέσιμα (kmeans_omp_nosync_lock, kmeans_omp_pthread_mutex_lock, kmeans_omp_pthread_spin_lock, kmeans_omp_tas_lock, kmeans_omp_ttas_lock, kmeans_omp_array_lock, kmeans_omp_clh_lock), τα οποία μοιράζονται τον ίδιο κώδικα του K-means αλλά χρησιμοποιούν διαφορετικό μηχανισμό lock.

Το Makefile του φακέλου έχει προσαρμοστεί ώστε να συνθέτει όλα τα παραπάνω εκτελέσιμα, αξιοποιώντας τις κοινές πηγές file_io.c και util.c και προσθέτοντας κάθε φορά την αντίστοιχη υλοποίηση κλειδώματος από τον υποφάκελο locks/. Για τα κλειδώματα που βασίζονται σε POSIX mutex ή spinlocks ενεργοποιείται επιπλέον η παράμετρος -pthread, ενώ όλα τα παράλληλα εκτελέσιμα μεταγλωττίζονται με τις σημαίες του OpenMP (-fopenmp). Παράλληλα, το πρόγραμμα make_on_queue.sh έχει τροποποιηθεί, ώστε να μετακινείται στον σωστό φάκελο της άσκησης (cd /home/parallel/parlab05/a3) και να εκτελεί εκεί την εντολή make. Με αυτόν τον τρόπο εξασφαλίζουμε ότι όλες οι εκδόσεις του K-means (σειριακή, naive, critical και με κλειδώματα) μεταγλωττίζονται απευθείας στο περιβάλλον του sandman, πριν από τη λήψη των μετρήσεων.

Τέλος, το πρόγραμμα run_on_queue.sh είναι υπεύθυνο για την αυτοματοποιημένη εκτέλεση των πειραμάτων στο μηχάνημα του sandman. Στην τελική του μορφή το script τρέχει όλα τα εκτελέσιμα κλειδωμάτων και για όλους τους αριθμούς νημάτων που ζητούνται (1, 2, 4, 8, 16, 32, 64). Για κάθε συνδυασμό κλειδώματος και αριθμό νημάτων, θέτει κατάλληλα τη μεταβλητή OMP_NUM_THREADS και ορίζει την πολιτική δέσμευσης πυρήνων μέσω της GOMP_CPU_AFFINITY, ώστε τα νήματα να κατανέμονται στους πρώτους φυσικούς πυρήνες του συστήματος (εκτέλεση πάντα «με affinity», όπως απαιτείται). Τα αποτελέσματα κάθε εκτέλεσης αποθηκεύονται σε ξεχωριστό κατάλογο της μορφής benchmarks/<lock_name>/S32_N16_C32_L10_T<threads>/, όπου δημιουργούνται αρχεία meta.txt με τα μεταδεδομένα της εκτέλεσης (εκτελέσιμο, τύπος κλειδώματος, παράμετροι K-means, αριθμός νημάτων, τιμές affinity) και output.txt με την πλήρη έξοδο του προγράμματος, συμπεριλαμβανομένης της γραμμής με τον χρόνο εκτέλεσης και τον αριθμό επαναλήψεων. Έτσι, οι επόμενες ενότητες μπορούν να επεξεργαστούν τα δεδομένα των benchmarks με τρόπο αντίστοιχο της προηγούμενης άσκησης και να παράγουν συγκρίσιμα διαγράμματα για όλες τις υλοποίησεις κλειδωμάτων.

Για λόγους πληρότητας, το τροποποιημένο run_on_queue.sh (μόνο αυτό διαφοροποιήθηκε σημαντικά) παρουσιάζεται ακολούθως:

a3/run_on_queue.sh

```
1 #!/bin/bash
2
3 #PBS -N run_kmeans_locks
4 #PBS -o run_kmeans_locks.out
5 #PBS -e run_kmeans_locks.err
6 #PBS -l nodes=1:ppn=64
7 #PBS -l walltime=01:00:00
8
9 ## How to submit (runs all locks x all thread configs on sandman):
10 ## qsub -q serial -l nodes=sandman:ppn=64 run_on_queue.sh
11 ##
12 ## Defaults (can be overridden via -v):
13 ## SIZE=32
14 ## COORDS=16
15 ## CLUSTERS=32
16 ## LOOPS=10
17
18 set -euo pipefail
19
20 # Work in the directory where qsub was executed (your a3 folder)
21 cd "${PBS_O_WORKDIR:-.}" || exit 1
22
23 # Fixed configuration required by the exercise (override with env if needed)
24 SIZE="${SIZE:-32}"
25 COORDS="${COORDS:-16}"
26 CLUSTERS="${CLUSTERS:-32}"
27 LOOPS="${LOOPS:-10}"
28
29 # Thread configurations to test
30 THREADS_LIST=(1 2 4 8 16 32 64)
31
32 # Lock variants (names as they appear in the binary targets)
33 LOCKS=(
34     "nosync_lock"
35     "pthread_mutex_lock"
36     "pthread_spin_lock"
37     "tas_lock"
38     "ttas_lock"
39     "array_lock"
40     "clh_lock"
41 )
42
43 run_one() {
44     local lock_name="$1"
45     local threads="$2"
46     local bin=""
47
48     if [[ "$lock_name" == "critical" ]]; then
49         # OpenMP critical version
50         bin="kmeans_omp_critical"
51     else
```

```

52     # Lock-based versions (built from omp_lock_kmeans.c + one lock object)
53     bin="kmeans_omp_${lock_name}"
54   fi
55
56   if [[ ! -x "./${bin}" ]]; then
57     echo "[WARN] Skipping lock='${lock_name}', threads=${threads}: binary '${bin}' not
58   found"
59   return
60   fi
61
62   # OpenMP settings
63   export OMP_NUM_THREADS="${threads}"
64
65   # Always use thread binding (affinity) as required
66   local affinity=""
67   for ((i=0; i<threads; i++)); do
68     affinity+="${i} "
69   done
70   affinity="${affinity%% }"
71   export GOMP_CPU_AFFINITY="${affinity}"
72
73   # Result directory:
74   # benchmarks/<lock_name>/S32_N16_C32_L10_T8/
75   local
76   result_dir="benchmarks/${lock_name}/S${SIZE}_N${COORDS}_C${CLUSTERS}_L${LOOPS}_T${threads}"
77   mkdir -p "${result_dir}"
78
79   {
80     echo "[run_on_queue] BIN=${bin}"
81     echo "[run_on_queue] LOCK=${lock_name}"
82     echo "[run_on_queue] OMP_NUM_THREADS=${OMP_NUM_THREADS}"
83     echo "[run_on_queue] GOMP_CPU_AFFINITY=${GOMP_CPU_AFFINITY}"
84     echo "[run_on_queue] Params: -s ${SIZE} -n ${COORDS} -c ${CLUSTERS} -l ${LOOPS}"
85     echo "[run_on_queue] Result dir: ${result_dir}"
86   } > "${result_dir}/meta.txt"
87
88   echo "[INFO] Running lock='${lock_name}', threads=${threads}, bin='${bin}'"
89   ./"${bin}" -s "${SIZE}" -n "${COORDS}" -c "${CLUSTERS}" -l "${LOOPS}" \
90   | tee "${result_dir}/output.txt"
91
92   # 1) Run all lock implementations (omp_lock_kmeans.c + locks/)
93   for lock in "${LOCKS[@]}"; do
94     for t in "${THREADS_LIST[@]}"; do
95       run_one "${lock}" "${t}"
96     done
97   done
98
99   # 2) Run the critical version (omp_critical_kmeans.c → kmeans_omp_critical)
100  for t in "${THREADS_LIST[@]}"; do
101    run_one "critical" "${t}"
102  done
103

```

2. Λήψη Μετρήσεων και Διαγράμματα

Στη συνέχεια παρουσιάζουμε τους πίνακες με τα αποτελέσματα όλων των μετρήσεων που προέκυψαν από την εκτέλεση του K-means για τις παραμέτρους εισόδου SIZE=32, COORDS=16, CLUSTERS=32 και LOOPS=10 και για αριθμό νημάτων $T \in \{1, 2, 4, 8, 16, 32, 64\}$. Παραθέτουμε έναν πίνακα για κάθε μηχανισμό (συμπεριλαμβανομένου του critical). Με αυτόν τον τρόπο μπορούμε να συγκρίνουμε άμεσα το κόστος συγχρονισμού κάθε μηχανισμού και πώς αυτό κλιμακώνεται καθώς αυξάνεται ο βαθμός παραλληλισμού:

nosync_lock:

Threads	Total Time
1	1.3238
2	0.7005
4	0.4084
8	0.2984
16	0.7346
32	1.3143
64	0.9590

pthread_mutex_lock:

Threads	Total Time
1	1.3241
2	0.8372
4	0.7302
8	0.8456
16	2.6544
32	3.7170
64	3.8797

pthread_spin_lock:

Threads	Total Time
1	1.3174
2	0.7447
4	0.5589
8	0.8860
16	3.3755
32	7.7281
64	3.7769

tas_lock:

Threads	Total Time
1	1.3291
2	0.7396
4	0.5305
8	1.1465
16	4.5320
32	10.6944
64	9.3531

ttas_lock:

Threads	Total Time
1	1.3151
2	0.7513
4	0.5621
8	0.8613
16	3.2426
32	7.3978
64	4.3841

array_lock:

Threads	Total Time
1	1.3585
2	0.8212
4	0.5121
8	0.4942
16	1.5424
32	2.2050
64	2.1843

clh_lock:

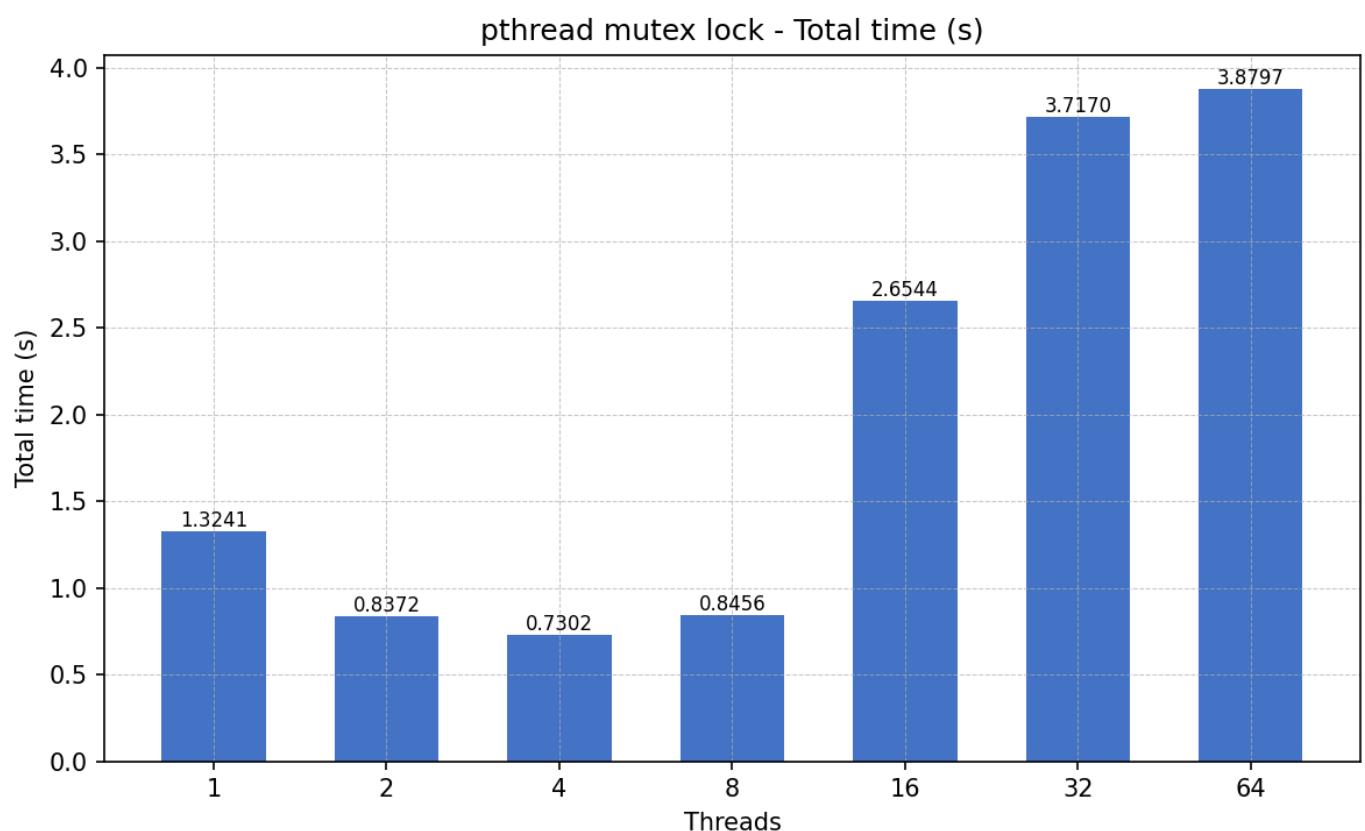
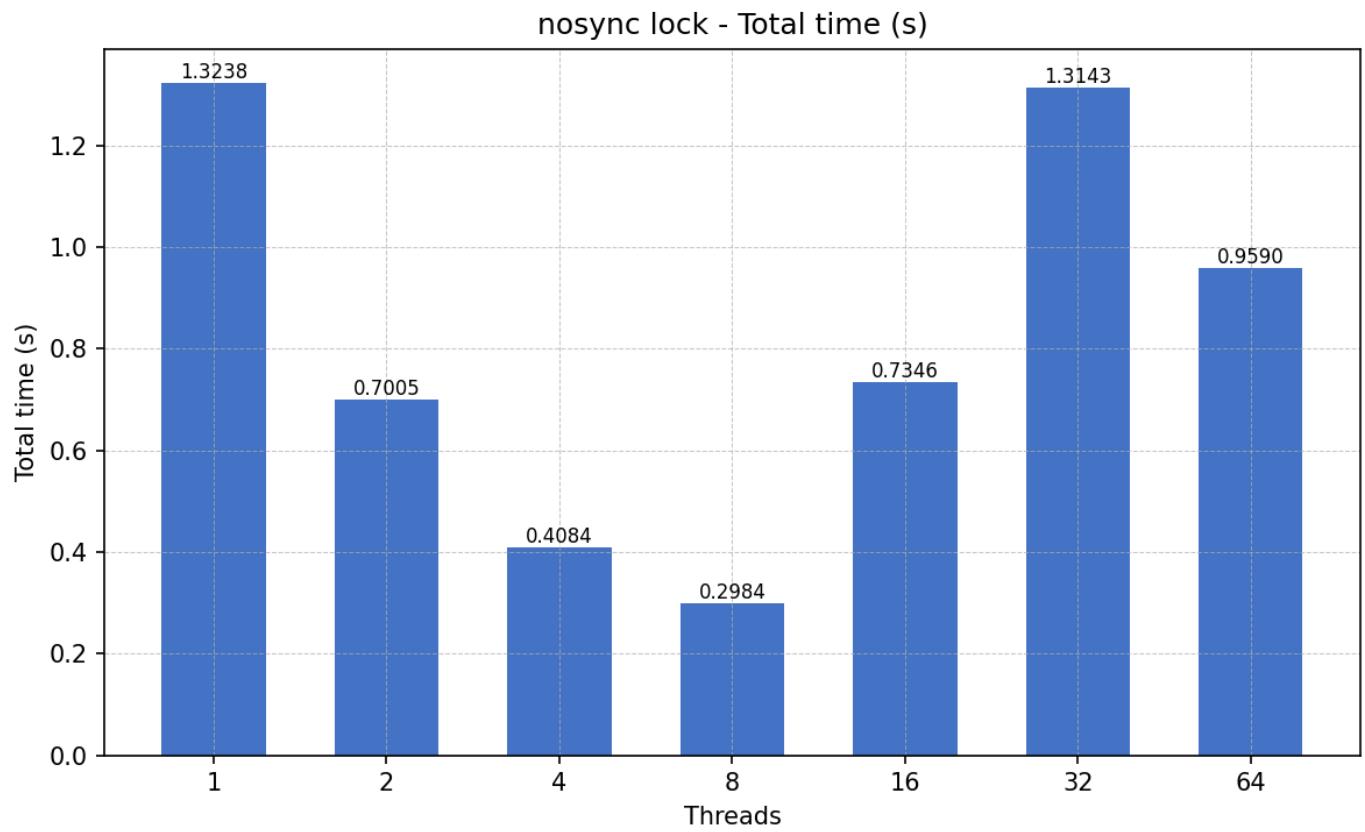
Threads	Total Time
1	1.3182
2	0.7995
4	0.4884
8	0.4319
16	1.2522
32	1.6519
64	1.4726

critical:

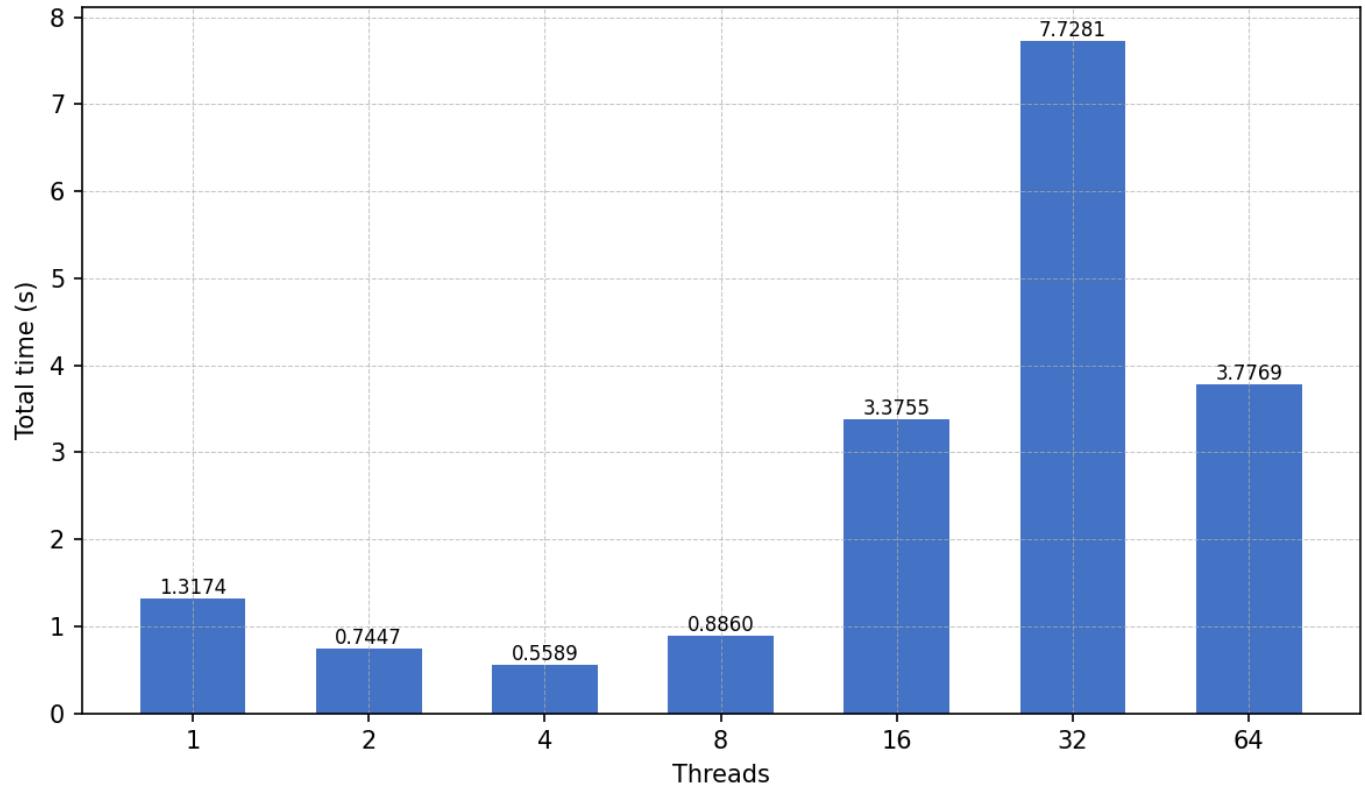
Threads	Total Time
1	1.3187
2	0.7827
4	0.4916
8	0.7655
16	3.4095
32	7.9911
64	6.1087

Για να είναι πιο ευδιάκριτες οι διαφορές μεταξύ των locks, παρουσιάζουμε τα αποτελέσματα και σε διαγράμματα. Στα διαγράμματα που ακολουθούν απεικονίζουμε τον χρόνο εκτέλεσης σε συνάρτηση με τον αριθμό νημάτων, για κάθε υλοποίηση κλειδώματος, καθώς και ένα συγκεντρωτικό διάγραμμα για όλους τους τύπους μαζί. Τα διαγράμματα αυτά μας επιτρέπουν να αξιολογήσουμε ποια

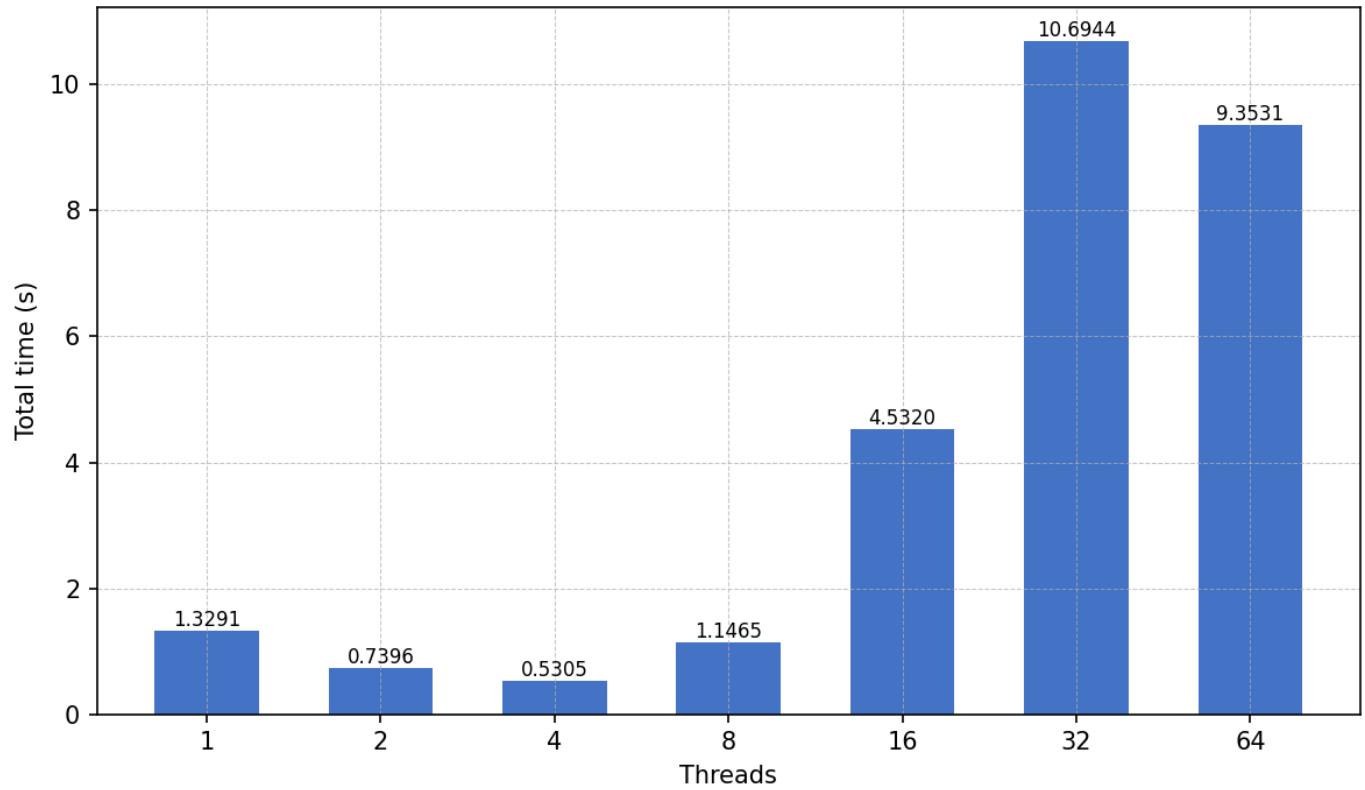
κλειδώματα παραμένουν αποδοτικά υπό αυξημένη συμφόρηση, ποια εμφανίζουν κορεσμό λόγω κόστους συγχρονισμού, καθώς και πώς συγκρίνεται η χρήση critical sections του OpenMP με τις υπόλοιπες υλοποιήσεις κλειδώματος:



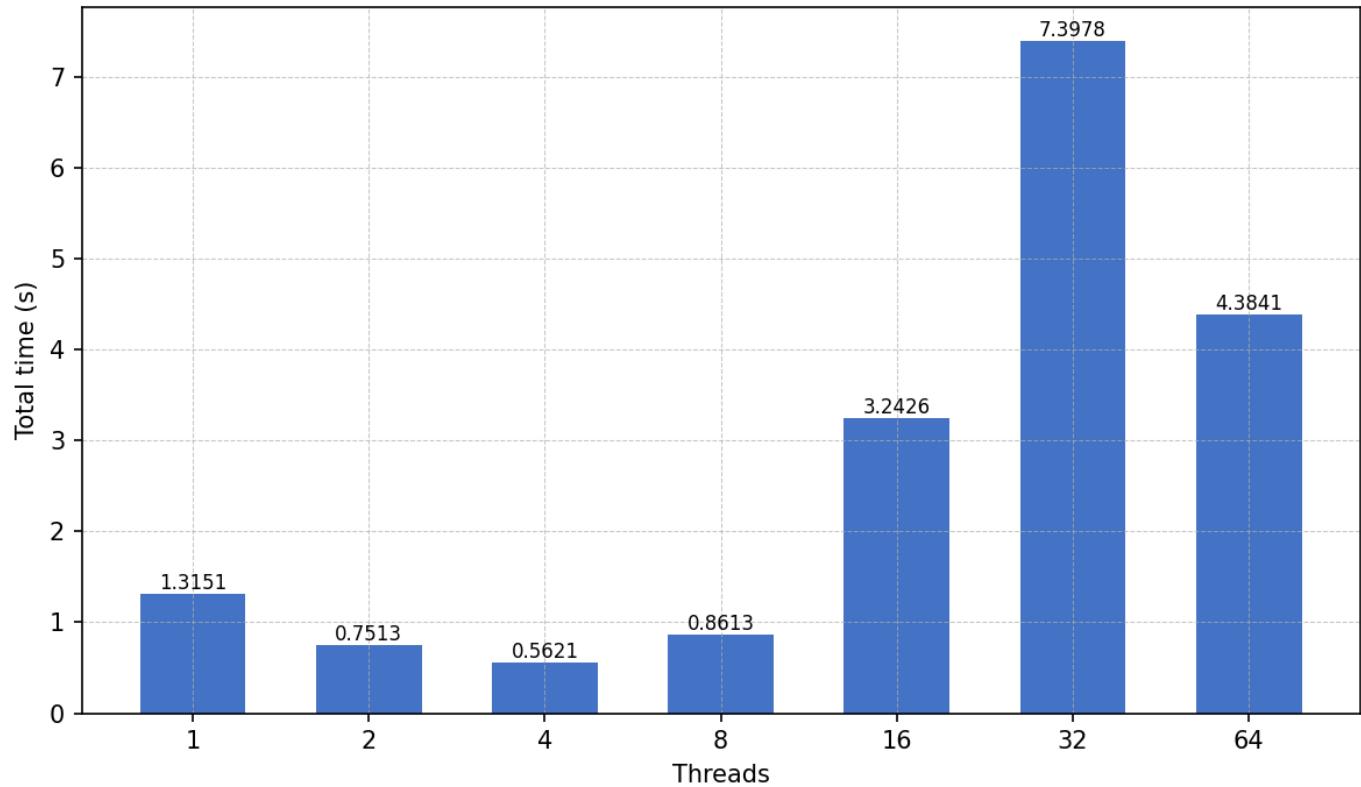
pthread spin lock - Total time (s)



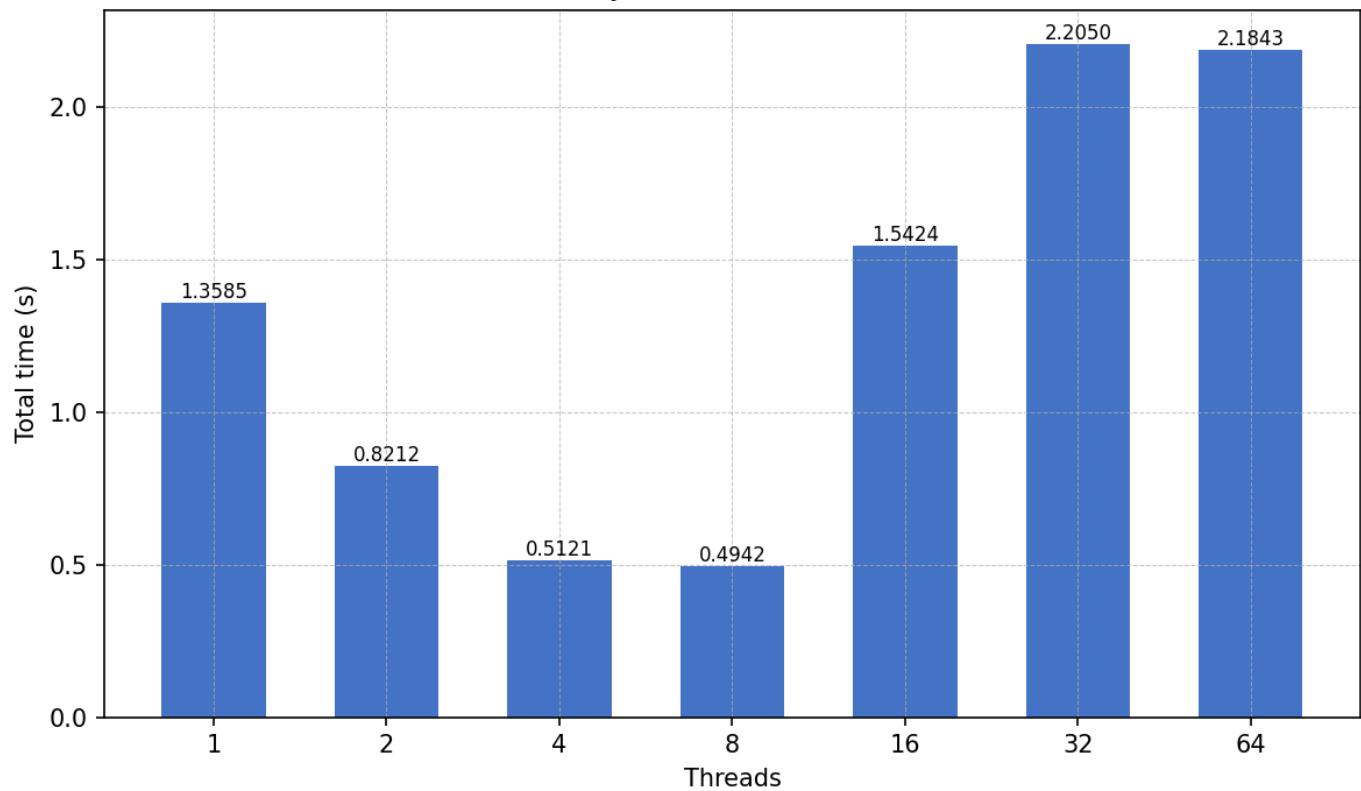
tas lock - Total time (s)

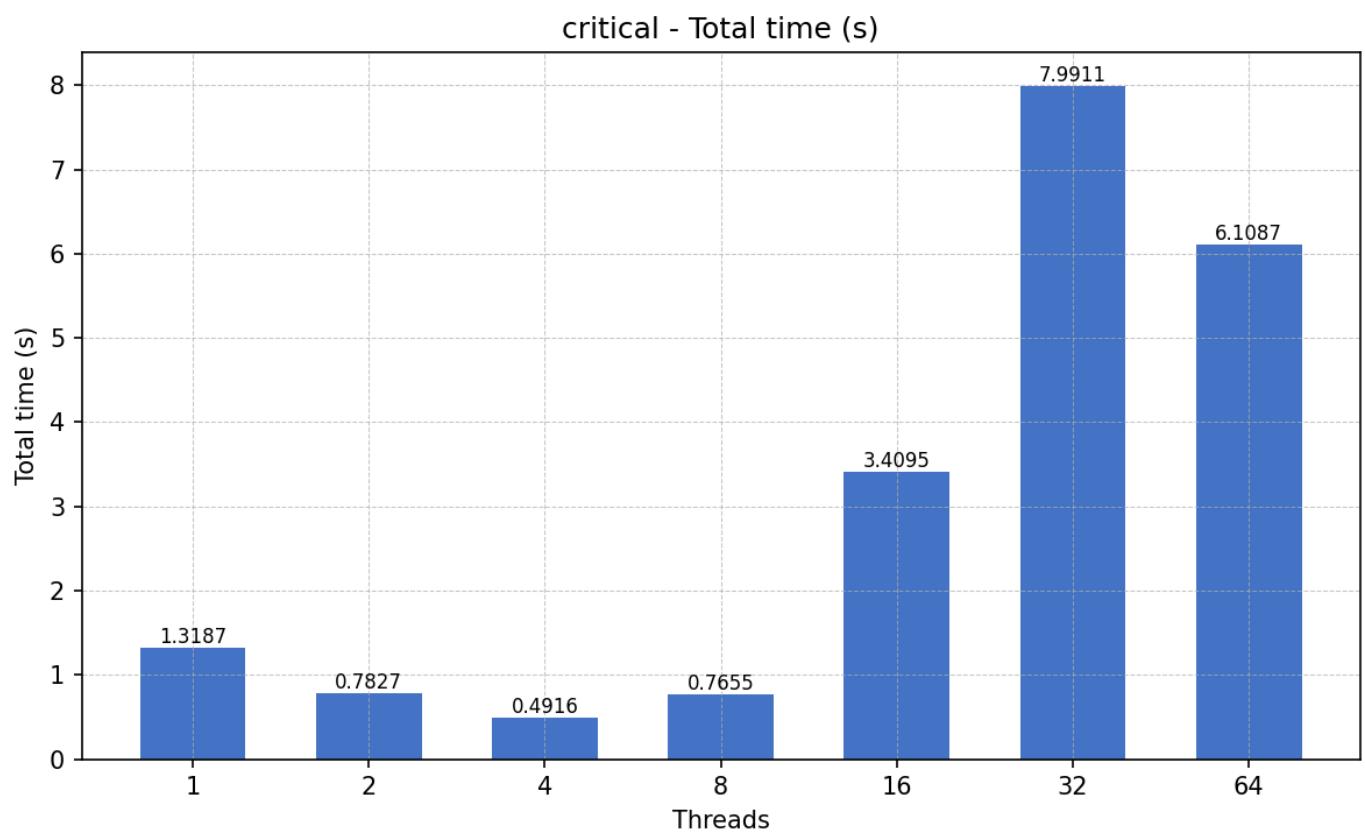
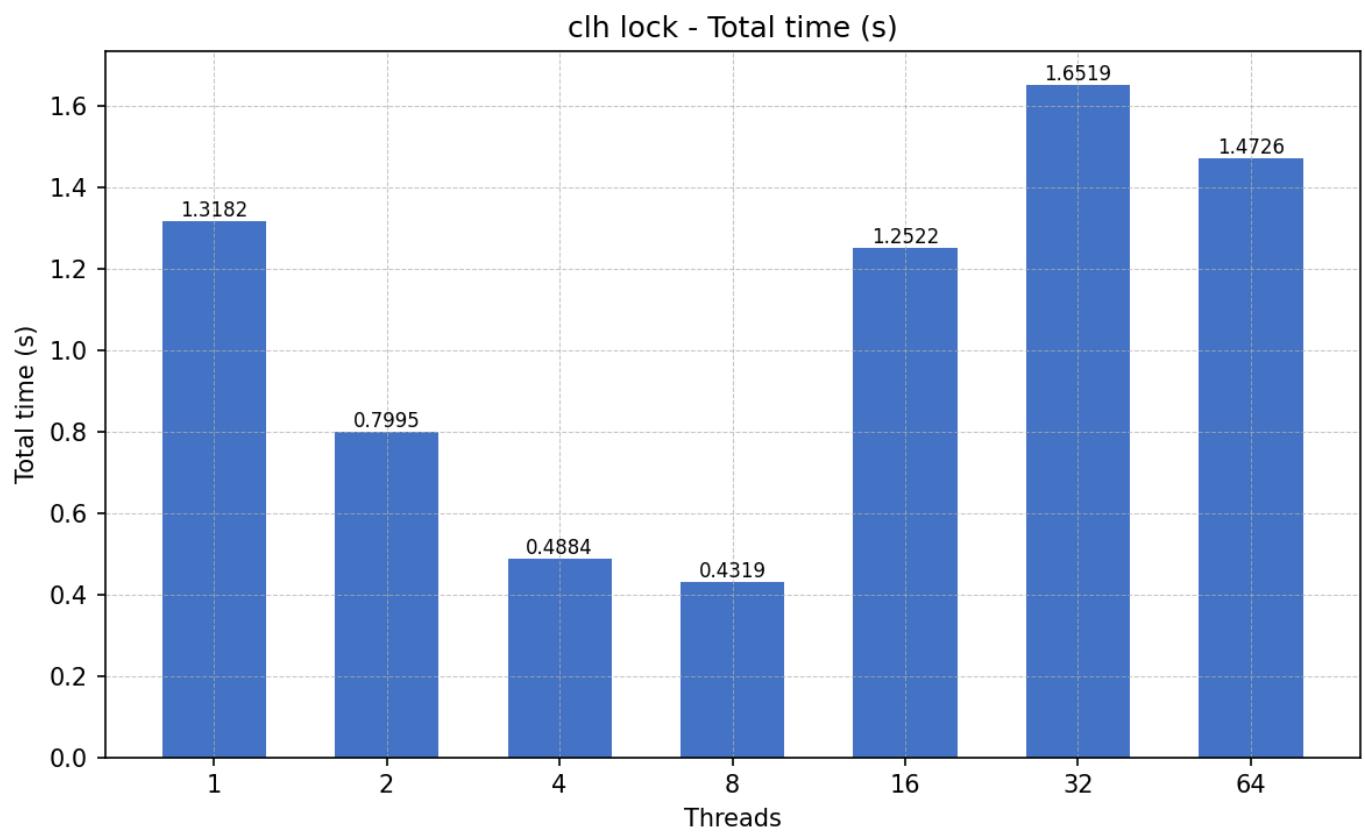


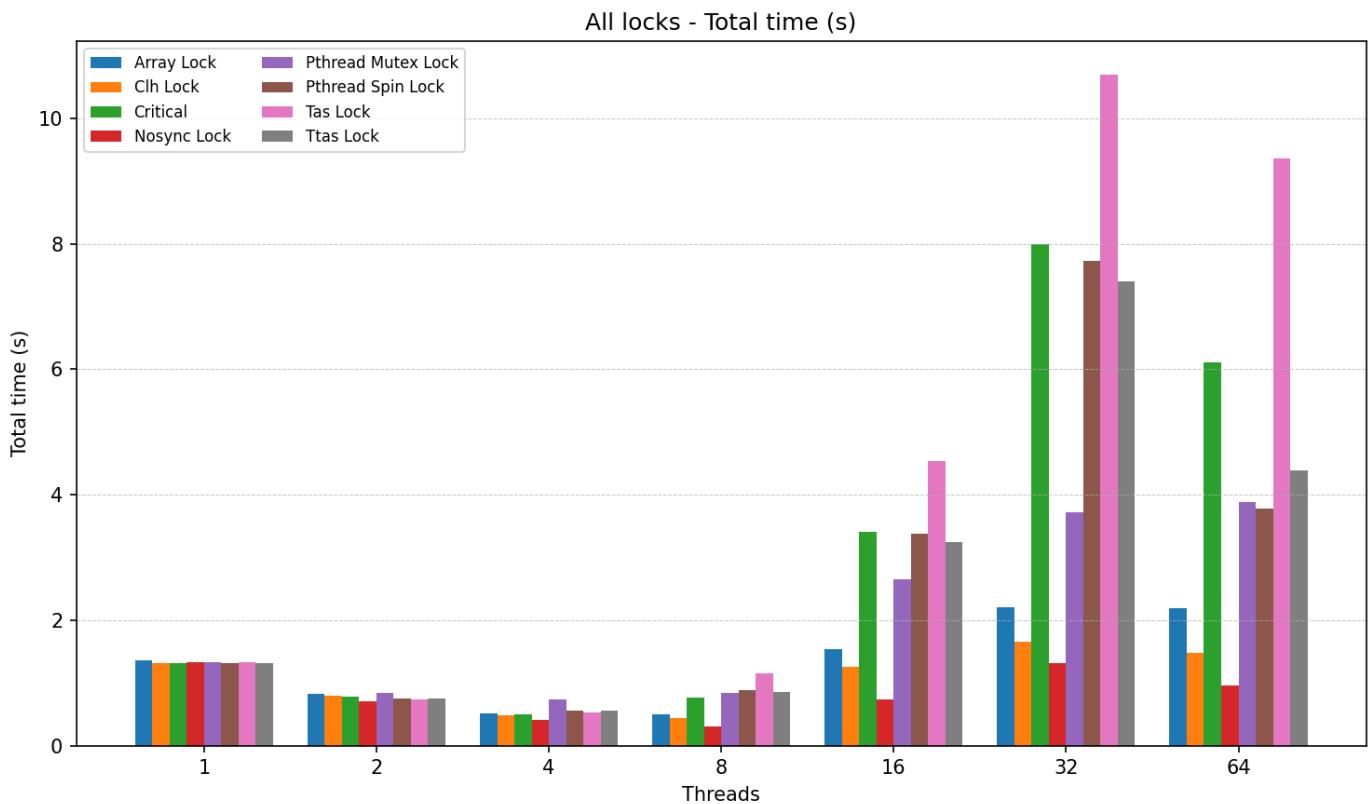
ttas lock - Total time (s)



array lock - Total time (s)







3. Σύγκριση Αποτελεσμάτων

Ξεκινώντας από τη σύγκριση μεταξύ των κλειδωμάτων, για $T=1$ όλοι οι μηχανισμοί (nosync, mutex, spin, TAS/TTAS, array, CLH, omp_critical) έχουν πολύ κοντινούς χρόνους, αφού δεν υπάρχει ουσιαστικό contention και το κόστος του lock είναι αμελητέο. Καθώς αυξάνουμε τα νήματα, ο «ιδανικός» δείκτης nosync_lock και τα queue-based locks (array_lock, clh_lock) βελτιώνουν τον χρόνο μέχρι περίπου τα 8 νήματα και από εκεί και πέρα χειροτερεύουν ήπια. Αντίθετα, τα pthread_mutex_lock, pthread_spin_lock, TAS/TTAS και το omp_critical_kmeans αρχίζουν να πέφτουν γρήγορα: για 8–16 νήματα ο χρόνος σταθεροποιείται ή αυξάνεται και για περισσότερα αυξάνεται ραγδαία, καθώς το κρίσιμο τμήμα σειριοποιείται και το lock γίνεται το βασικό bottleneck.

Η συμπεριφορά αυτή συνδέεται και με τη φύση του ίδιου του προβλήματος: στο configuration {32,16,32,10} ο K-means είναι έντονα memory-bound, με περιορισμένο υπολογιστικό φορτίο ανά στοιχείο (compute intensity). Όταν προσθέτουμε ένα «βαρύ» lock πάνω σε ένα μικρό, κυρίως memory-bound workload, το κόστος συγχρονισμού κυριαρχεί, ειδικά όταν πολλά νήματα προσπαθούν να μπουν στο ίδιο critical section. Τα queue locks περιορίζουν καλύτερα το contention

(λιγότερη τυχαία κίνηση σε κοινές cache lines) και γι' αυτό αντέχουν περισσότερο σε υψηλό παραλληλισμό από ό,τι τα απλά spinlocks ή το critical section.

Σε σχέση με την προηγούμενη άσκηση, τα ευρήματα είναι απολύτως συνεπή. Η shared-clusters υλοποίηση με `#pragma omp atomic` είχε λίγο κέρδος μέχρι περίπου τα 8 νήματα και μετά υπόκειται σε κορεσμό, ακριβώς επειδή πολλές `atomic` ενημερώσεις σε λίγες κοινόχρηστες δομές σειριαποιούσαν μεγάλο μέρος της δουλειάς. Στην τωρινή άσκηση, το `omp_critical_kmeans` και τα απλά spin/TAS locks εμφανίζουν την ίδια συμπεριφορά: λειτουργικά σωστά, αλλά με περιορισμένη κλιμάκωση λόγω έντονου synchronization και memory contention.

Αντίθετα, η λύση της προηγούμενης άσκησης με copied clusters και explicit reduction πέτυχε πολύ καλύτερη κλιμάκωση γιατί μείωσε δραστικά το fine-grained synchronization: κάθε νήμα δούλευε σε ιδιωτικές δομές και συγχρονιζόταν μόνο στη φάση του reduction. Τα queue locks της τωρινής άσκησης κινούνται προς αυτή την κατεύθυνση (περιορίζουν το contention και βελτιώνουν την shared υλοποίηση), αλλά δεν μπορούν να φτάσουν το επίπεδο της αλγορίθμικής βελτίωσης του copied clusters + reduction, ιδίως σε ένα μικρό και τόσο memory-bound πρόβλημα.

'Όπως τονίζεται και στις διαφάνειες του μαθήματος, η επίδοση κάθε μηχανισμού κλειδώματος εξαρτάται κρίσιμα από τον αριθμό νημάτων, το μέγεθος του κρίσιμου/μη κρίσιμου τμήματος και το επίπεδο συμφόρησης. Δεν υπάρχει ένα «καθολικά καλύτερο» κλείδωμα, αλλά διαφορετικοί μηχανισμοί που αποδίδουν καλύτερα σε διαφορετικά σενάρια.

4. Τα Κλειδώματα

Κάθε κλείδωμα υλοποιεί τον αμοιβαίο αποκλεισμό με διαφορετικό τρόπο και αυτό έχει άμεσο αντίκτυπο στην επίδοση, ειδικά σε έντονα memory-bound πρόβλημα, όπως το K-means της άσκησης.

Το `nosync_lock` δεν είναι πραγματικό κλείδωμα. Ουσιαστικά δεν κάνει καμία προστασία του κρίσιμου τμήματος και χρησιμοποιείται μόνο ως θεωρητικό baseline. Γι' αυτό εμφανίζει τους καλύτερους χρόνους: δεν υπάρχει overhead συγχρονισμού, άρα όλα τα νήματα γράφουν χωρίς συντονισμό πάνω στις κοινόχρηστες δομές. Στην πράξη όμως η υλοποίηση αυτή δεν είναι ορθά συγχρονισμένη και μπορεί να οδηγήσει σε λανθασμένα αποτελέσματα. Γι' αυτό χρησιμοποιείται μόνο για να απομονώσουμε το «καθαρό» κόστος του locking.

To `pthread_mutex_lock` υλοποιεί ένα blocking lock σε επίπεδο POSIX. Όταν υπάρχει μικρό contention, η απόδοσή του είναι σχετικά καλή, αλλά μόλις πολλαπλά νήματα αρχίσουν να μπλοκάρουν πάνω στο ίδιο mutex, ο μηχανισμός αναγκάζεται να κάνει system calls και πιθανές μεταβάσεις σε kernel space, με context switches κ.λπ. Σε ένα σενάριο με πολλές, μικρής διάρκειας κρίσιμες περιοχές (όπως οι ενημερώσεις στους μετρητές του K-means) αυτό το overhead γίνεται δυσανάλογα μεγάλο και εξηγεί γιατί το mutex δεν κλιμακώνεται καλά για $T \geq 8$. Το `pthread_spin_lock`, από την άλλη, είναι spin lock: τα νήματα δεν αποκοιμούνται αλλά κάνουν busy-wait σε μια μεταβλητή. Αυτό αποφεύγει τις ακριβές μεταβάσεις στο kernel όταν το lock κρατιέται για πολύ λίγο χρόνο, αλλά υπό έντονο contention το busy-wait καταναλώνει CPU cycles και δημιουργεί μεγάλη πίεση στο memory system, αφού όλα τα νήματα διαβάζουν/γράφουν την ίδια cache line – γι' αυτό βλέπουμε χειροτέρευση χρόνου για πολλά νήματα.

Τα `tas_lock` και `ttas_lock` είναι απλές υλοποιήσεις spin lock σε επίπεδο user-space. To TAS (test-and-set) κάνει συνεχές atomic γράψιμο σε μια κοινή μεταβλητή μέχρι να πάρει το lock, με αποτέλεσμα να προκαλεί μεγάλο traffic στο bus και να «πετάει» την cache line από πυρήνα σε πυρήνα. To TTAS (test-and-test-and-set) βελτιώνει την κατάσταση: τα νήματα πρώτα κάνουν απλό read (χωρίς write) και μόνο όταν φαίνεται ότι το lock είναι ελεύθερο επιχειρούν atomic test-and-set. Αυτό μειώνει κάπως τον αριθμό των writes, αλλά όλα τα νήματα εξακολουθούν να γυρίζουν γύρω από την ίδια cache line. Συνεπώς, σε χαμηλό contention τα TAS/TTAS είναι σχετικά ελαφριά, ενώ σε υψηλό contention έχουν πολύ κακή κλιμάκωση λόγω τρικυμίας στο πρωτόκολλο συνοχής της μνήμης.

Τα `array_lock` και `clh_lock` είναι queue-based locks και εδώ φαίνεται η διαφορά τους στην επίδοση. Στο array lock κάθε νήμα παίρνει μια θέση σε έναν «κυκλικό» πίνακα και περιμένει σε δικό του κελί, έτσι ώστε μόνο ένας μικρός αριθμός cache lines να αλλάζει χέρια κάθε φορά. Στο CLH lock κάθε νήμα δημιουργεί έναν κόμβο σε μια λογική ουρά και σπινάρει πάνω σε μια μεταβλητή που ανήκει στον «προκάτοχό» του. Και στις δύο περιπτώσεις, το spinning γίνεται πάνω σε τοπικά ή σχεδόν τοπικά δεδομένα, με λιγότερη ανταλλαγή cache lines ανά απόκτηση lock και με εγγενή δικαιοσύνη (FIFO σειρά). Αυτό εξηγεί γιατί τα queue locks είχαν πολύ καλύτερη συμπεριφορά για $T=8-32$: μειώνουν δραστικά το lock contention που βλέπαμε στα TAS/TTAS και spin locks, και δεν «πνίγουν» τον αλγόριθμο με άσκοπο coherence traffic.

Τέλος, το `omp_critical_kmeans` χρησιμοποιεί την εντολή `#pragma omp critical`, η οποία αντιστοιχεί σε ένα implicit global lock γύρω από το κρίσιμο τμήμα. Η

λειτουργία του είναι εννοιολογικά παρόμοια με έναν mutex: μόνο ένα νήμα κάθε φορά περνάει μέσα στο critical section, όλα τα υπόλοιπα περιμένουν. Ο απλός αυτός μηχανισμός είναι εύχρηστος προγραμματιστικά, αλλά, όπως και το pthread_mutex_lock, γίνεται γρήγορα bottleneck όταν πολλά νήματα προσπαθούν να ενημερώσουν την ίδια κοινόχρηστη δομή. Έτσι, οι χρόνοι του omp_critical_kmeans αντανακλούν την ίδια εικόνα με το naive shared-clusters της προηγούμενης άσκησης: σωστή αλλά βαριά μορφή συγχρονισμού, που περιορίζει την παραλληλία και δεν κλιμακώνεται καλά σε υψηλό contention, ειδικά σε ένα μικρό και έντονα memory-bound πρόβλημα.

Τα κλειδώματα τύπου TAS/TTAS και spin lock ανήκουν στην κατηγορία των απλών spinlocks, ενώ τα array και CLH συγκαταλέγονται στα scalable queue locks, τα οποία – όπως φαίνεται και από τις μετρήσεις μας – μειώνουν το coherence traffic και κλιμακώνονται καλύτερα υπό έντονο contention.

**Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Νοέμβριος 2025**

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 3^{ης} ΑΣΚΗΣΗΣ



Στοιχεία Ομάδας

- Αναγνωριστικό: parlab05
- Μέλος 1^ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
- Μέλος 2^ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
- Ημερομηνία Παράδοσης Αναφοράς: 20.10.2025

▪ Ταυτόχρονες Δομές Δεδομένων

1. Υλοποιήσεις

Στην άσκηση μελετάμε ταυτόχρονες υλοποιήσεις μίας ταξινομημένης απλά συνδεδεμένης λίστας με βασικές λειτουργίες `contains()`, `add()` και `remove()`. Οι υλοποιήσεις διαφέρουν μόνο στον μηχανισμό συγχρονισμού:

Coarse-grain locking (cgl)

Χρησιμοποιείται ένα κοινό lock (mutex) για ολόκληρη τη λίστα. Κάθε λειτουργία κλειδώνει τη δομή στην αρχή και την ξεκλειδώνει στο τέλος, άρα σε κάθε χρονική στιγμή μόνο ένα νήμα μπορεί να εκτελεί οποιαδήποτε λειτουργία στη λίστα.

Fine-grain locking (fgl)

Υπάρχει lock ανά κόμβο και η διάσχιση γίνεται με hand-over-hand (lock coupling): διατηρούνται κλειδωμένοι διαδοχικά ο πρόγονος (pred) και ο τρέχων κόμβος (curr), και καθώς προχωράμε ξεκλειδώνεται ο προηγούμενος κόμβος και κλειδώνεται ο επόμενος. Αυτό επιτρέπει παραλληλία σε διαφορετικά τμήματα της λίστας, αλλά αυξάνει το overhead από πολλά lock/unlock.

Optimistic synchronization (opt)

Οι λειτουργίες πρώτα διατρέχουν τη λίστα χωρίς locks για να εντοπίσουν το σημείο ενημέρωσης. Στη συνέχεια κλειδώνουν τοπικά τους κόμβους pred και curr και εκτελούν validation (επαλήθευση) ότι η δομή δεν άλλαξε ενδιάμεσα. Το validation περιλαμβάνει επαναδιάσχιση από την αρχή για να επιβεβαιωθεί ότι το ζεύγος (pred,curr) παραμένει έγκυρο, αλλιώς η λειτουργία επαναλαμβάνεται (retry).

Lazy synchronization (lazy)

Η `contains()` εκτελείται χωρίς locks και αγνοεί κόμβους που έχουν μαρκαριστεί ως διαγραμμένοι. Η `remove()` υλοποιείται σε δύο φάσεις: πρώτα λογική διαγραφή (θέτοντας `marked=true`) και έπειτα φυσική αφαίρεση (ενημέρωση `pred.next=curr.next`) με τοπικό locking και validation. Έτσι μειώνεται ο χρόνος που κρατιούνται locks και αποφεύγονται συγκρούσεις με αναζητήσεις.

Non-blocking (lock-free) synchronization (nb)

Δεν χρησιμοποιούνται locks. Οι ενημερώσεις γίνονται με ατομικές εντολές týπου CAS πάνω σε δείκτες (συχνά «πακετάροντας» και το mark μαζί με το next). Οι λειτουργίες επαναπροσπαθούν (retry) όταν μια CAS αποτύχει λόγω ταυτόχρονης ενημέρωσης από άλλο νήμα. Η `contains()` είναι wait-free (δεν μπλοκάρει), ενώ `add/remove` είναι lock-free (πάντα κάποιο νήμα προοδεύει).

2. Περιβάλλον Εκτέλεσης και Ρυθμίσεις Παραμέτρων

Τα πειράματα εκτελέστηκαν στο μηχάνημα sandman (ουρά serial) με χρήση `qsub`, σύμφωνα με τις οδηγίες της áskησης. Για κάθε εκτελέσιμο χρησιμοποιήθηκε η ίδια `main.c` διεπαφή και μετρήθηκε το throughput σε Kops/sec (χιλιάδες λειτουργίες ανά δευτερόλεπτο) για σταθερό χρόνο εκτέλεσης.

Ο αριθμός νημάτων και η αντιστοίχισή τους σε λογικούς πυρήνες καθορίστηκε αποκλειστικά από τη μεταβλητή περιβάλλοντος `MT_CONF`, ώστε να επιτυγχάνεται pinning και αναπαραγωγιμότητα. Σε εκτελέσεις έως 64 νήματα τα threads «δέθηκαν» σε διαδοχικούς πυρήνες (π.χ. `MT_CONF=0,1,2,3` για 4 νήματα). Για 64 και 128 νήματα αξιοποιήθηκε hyperthreading και (στην περίπτωση των 128) oversubscription, σύμφωνα με τις οδηγίες της εκφώνησης.

Οι παράμετροι που εξετάστηκαν ήταν:

- Threads: 1, 2, 4, 8, 16, 32, 64, 128
- Μέγεθος λίστας: 1024, 8192
- Workloads: 100-0-0, 80-10-10, 20-40-40, 0-50-50 (contains-add-remove)

To `run_on_queue.sh` φαίνεται, για λόγους πληρότητας, ακολούθως:

a4/conc_ll/run_on_queue.sh

```

1 #!/bin/bash
2
3 ## Job Name
4 #PBS -N run_conc_ll
5
6 ## Output and error of PBS (not the runs)
7 #PBS -o run_conc_ll.pbs_out
8 #PBS -e run_conc_ll.pbs_err
9
10 ## Sandman, serial queue, 64 threads available
11 #PBS -q serial
12 #PBS -l nodes=sandman:ppn=64
13
14 ## Maximum walltime (adjust if necessary)
15 #PBS -l walltime=01:00:00
16
17 ## Go to the directory where qsub was executed
18 # CHANGE THIS TO YOUR ACTUAL DIRECTORY
19 cd $HOME/a2/conc_ll
20
21 # --- Define Core Parameters ---
22 IMPLEMENTATIONS="serial cgl fgl opt lazy nb"
23 NTHREADS="1 2 4 8 16 32 64 128"
24 LIST_SIZES="1024 8192"
25
26 # Workloads: (Contains, Add, Remove)
27 # Format: "C_A_R"
28 WORKLOADS="100_0_0 80_10_10 20_40_40 0_50_50"
29
30 # Directory for results
31 OUTDIR="results_conc_ll"
32 mkdir -p "$OUTDIR"
33
34 # --- Helper Function to generate MT_CONF for thread pinning ---
35 # Generates a comma-separated list of logical core IDs.
36 # Assumes sandman has 64 logical cores (0-63).
37 # For N > 64, it cycles through the 64 available logical cores (oversubscription).
38 get_mt_conf() {
39     local N=$1
40     local CONFIG=""
41     local MAX_LOGICAL_CORES=64
42
43     for i in $(seq 0 $((N - 1))); do
44         # Core ID cycles through 0, 1, ..., 63, 0, 1, ...
45         local CORE_ID=$((i % MAX_LOGICAL_CORES))
46
47         CONFIG="${CONFIG}${CORE_ID}"
48         if [ $i -lt $((N - 1)) ]; then
49             CONFIG="${CONFIG},"
50         fi
51     done

```

```

52     echo "$CONFIG"
53 }
54
55 # --- Main Execution Loop ---
56 for IMPL in $IMPLEMENTATIONS; do
57     EXECUTABLE="./$x.$IMPL"
58
59     for S in $LIST_SIZES; do
60
61         for T in $NTHREADS; do
62
63             # For the serial implementation, only run T=1 (to establish baseline)
64             if [ "$IMPL" == "serial" ] && [ $T -gt 1 ]; then
65                 continue
66             fi
67
68             # --- MT_CONF Setting for Thread Pinning (pthreads) ---
69             if [ $T -gt 1 ]; then
70                 MT_CONF=$(get_mt_conf $T)
71                 export MT_CONF
72             else
73                 # Unset MT_CONF for single-threaded execution (T=1)
74                 unset MT_CONF
75             fi
76
77             echo "Running $IMPL: ListSize=$S, Nthreads=$T, MT_CONF=$MT_CONF"
78
79             for W in $WORKLOADS; do
80                 # Split the workload string (e.g., 100_0_0) into C, A, R variables
81                 IFS='_' read -r C A R <<< "$W"
82
83                 # Input arguments for the executable: <list_size> <contains_pct> <add_pct>
<remove_pct>
84                 ARGS="$S $C $A $R"
85
86                 # Output files for this run
87                 OUT="${OUTDIR}/conc_ll_${IMPL}_${S}${S}_T${T}_W${W}.out"
88                 ERR="${OUTDIR}/conc_ll_${IMPL}_${S}${S}_T${T}_W${W}.err"
89
90                 # Run the program:
91                 # - stdout → OUT
92                 # - stderr → ERR
93                 $EXECUTABLE $ARGS >"$OUT" 2>"$ERR"
94             done
95         done
96     done
97 done
98
99 echo "Execution finished. Results are in the $OUTDIR directory."
100

```

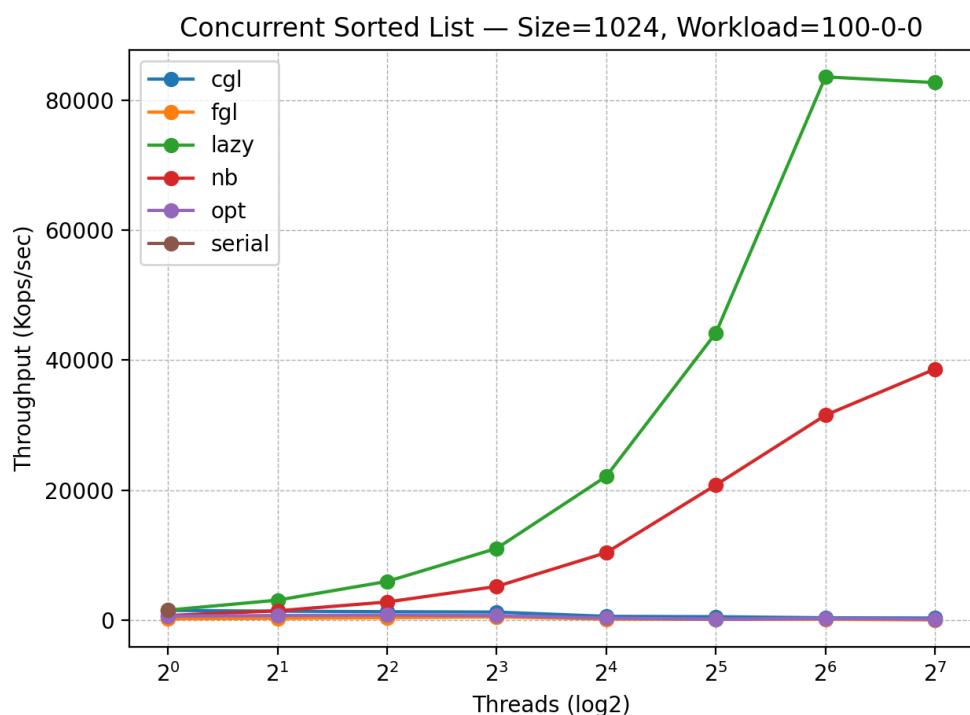
3. Αποτελέσματα και Σχολιασμός

Στα διαγράμματα που ακολουθούν παρουσιάζεται το throughput (Kops/sec), αλλά και το speedup ως συνάρτηση του αριθμού νημάτων, για διαφορετικά μεγέθη λίστας και διαφορετικά workloads. Κάθε γράφημα αντιστοιχεί σε σταθερό συνδυασμό μεγέθους λίστας και workload, ενώ οι καμπύλες αναπαριστούν τις διαφορετικές υλοποιήσεις συγχρονισμού. Για κάθε περίπτωση workload έχουμε :

A. Workload 100-0-0 (μόνο αναζητήσεις)

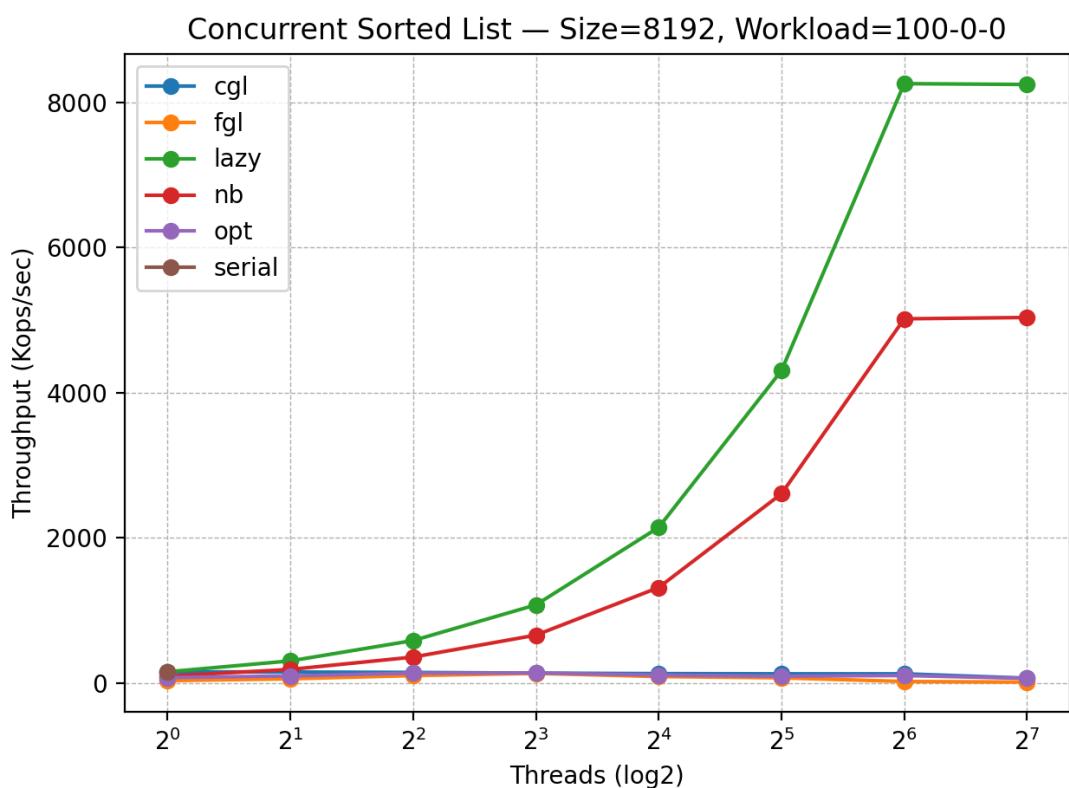
Στο συγκεκριμένο workload όλες οι λειτουργίες είναι contains(), χωρίς καμία εισαγωγή ή διαγραφή. Πρόκειται επομένως για ένα αμιγώς read-only σενάριο, στο οποίο δεν εμφανίζονται συγκρούσεις εγγραφών και η απόδοση εξαρτάται κυρίως από το αν και πόσο blocking απαιτεί η υλοποίηση για τις αναζητήσεις και το κόστος διάσχισης της λίστας (εξαρτώμενο από το μέγεθός της).

Throughput ως προς τον αριθμό νημάτων



Για μέγεθος λίστας $S = 1024$. Οι υλοποιήσεις lazy synchronization και non-blocking (lock-free) παρουσιάζουν τη σαφώς καλύτερη απόδοση και κλιμακώνονται έντονα με την αύξηση του αριθμού νημάτων. Το lazy synchronization επιτυγχάνει το υψηλότερο throughput, με σχεδόν γραμμική αύξηση έως περίπου 64 νήματα και ελαφρά σταθεροποίηση στη συνέχεια. Η non-blocking υλοποίηση ακολουθεί παρόμοια τάση, αν και με χαμηλότερες απόλυτες τιμές throughput.

Αντίθετα, η coarse-grain locking υλοποίηση εμφανίζει σχεδόν σταθερό και χαμηλό throughput ανεξαρτήτως αριθμού νημάτων, καθώς όλες οι αναζητήσεις σειριοποιούνται μέσω ενός καθολικού lock. Η fine-grain locking παρουσιάζει μικρή βελτίωση σε χαμηλό αριθμό νημάτων, ωστόσο η απόδοσή της υποβαθμίζεται καθώς αυξάνεται το concurrency, λόγω του αυξημένου κόστους από τα πολλαπλά lock/unlock σε κάθε βήμα της διάσχισης. Η optimistic synchronization παραμένει σε ενδιάμεσες τιμές, χωρίς να μπορεί να ανταγωνιστεί τις lazy και non-blocking υλοποιήσεις στο συγκεκριμένο σενάριο.



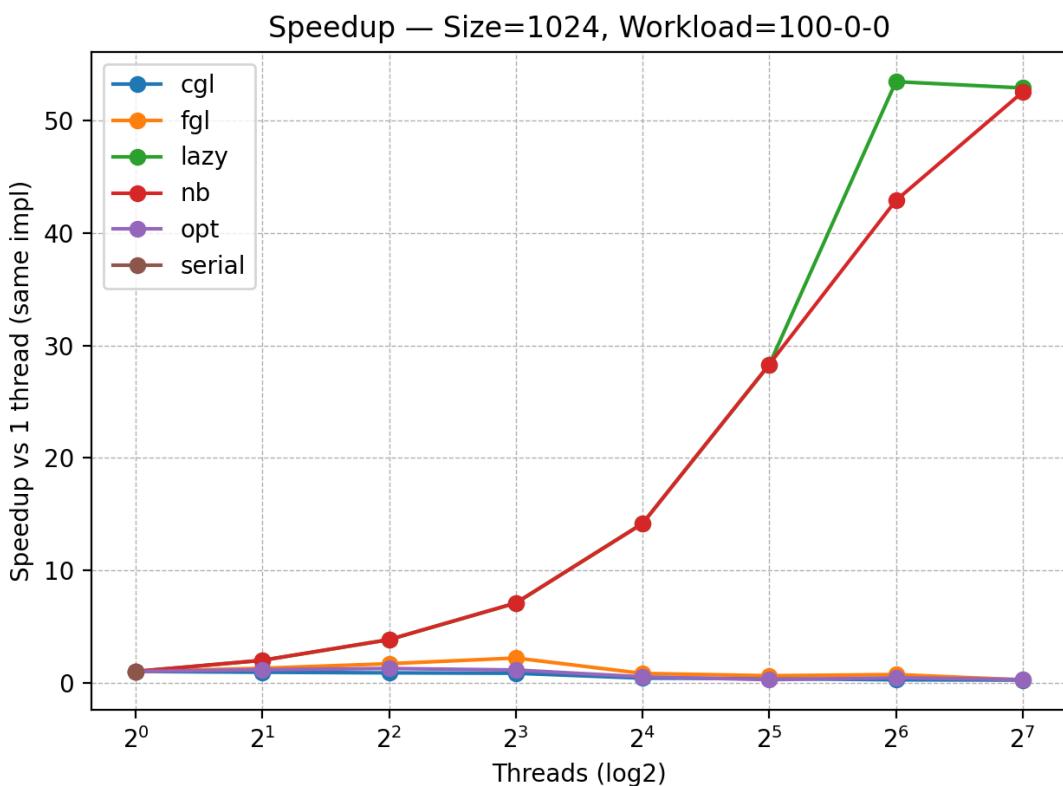
Για μεγαλύτερο μέγεθος λίστας, η ποιοτική εικόνα παραμένει ίδια, αλλά όλες οι υλοποιήσεις παρουσιάζουν σημαντικά χαμηλότερο throughput. Η αύξηση του

μήκους της λίστας συνεπάγεται μεγαλύτερο κόστος διάσχισης και αυξημένα cache misses, γεγονός που περιορίζει τον ρυθμό εκτέλεσης των αναζητήσεων.

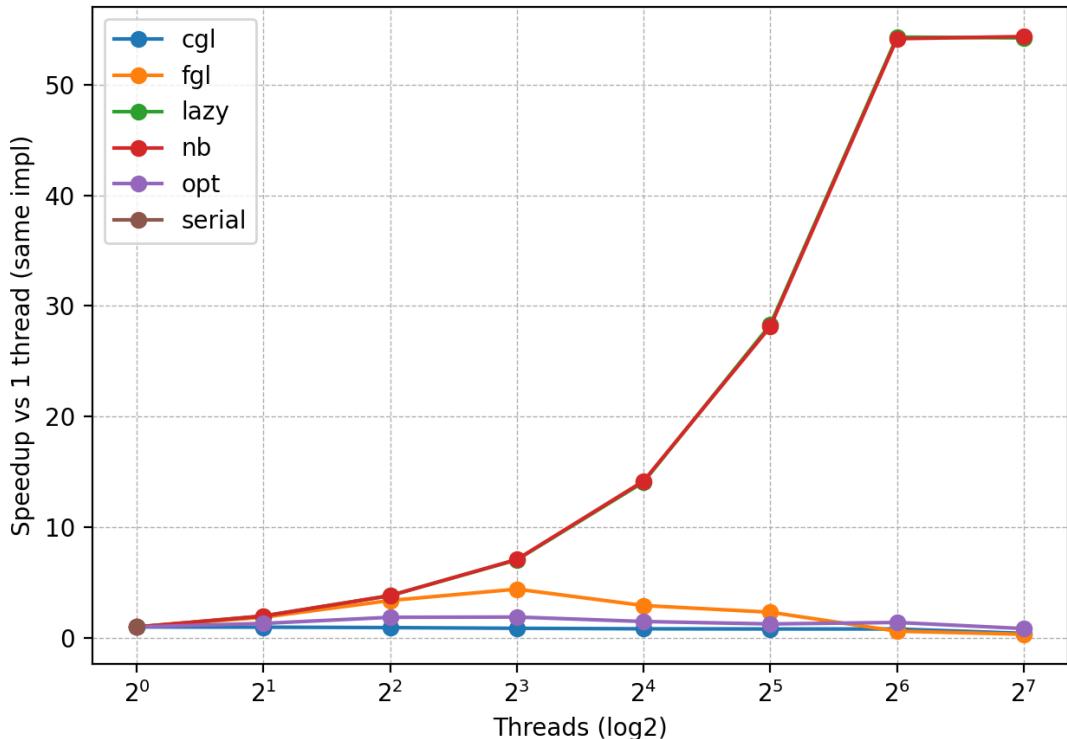
Παρόλα αυτά, οι lazy και non-blocking υλοποιήσεις εξακολουθούν να υπερέχουν σημαντικά έναντι των locking-based προσεγγίσεων, επιβεβαιώνοντας ότι η αποφυγή blocking είναι καθοριστικός παράγοντας απόδοσης σε read-only workloads.

Speedup ως προς τον αριθμό νημάτων

Το speedup υπολογίζεται σε σχέση με την απόδοση ενός νήματος της ίδιας υλοποίησης και αποτυπώνει την ικανότητα κλιμάκωσης ανεξάρτητα από τις απόλυτες τιμές throughput.



Speedup — Size=8192, Workload=100-0-0



Τόσο για $S=1024$ όσο και για $S=8192$, οι υλοποιήσεις *lazy* και *non-blocking* παρουσιάζουν εντυπωσιακό speedup, φτάνοντας περίπου έως $50\times$ σε 64 νήματα. Πέρα από αυτό το σημείο, και ειδικότερα στα 128 νήματα, η επιτάχυνση σταθεροποιείται, γεγονός που αποδίδεται στη χρήση hyperthreading και oversubscription, όπου τα threads μοιράζονται τους ίδιους φυσικούς πόρους.

Οι υλοποιήσεις *coarse-grain*, *fine-grain* και *optimistic* εμφανίζουν περιορισμένο speedup, με τις καμπύλες τους να παραμένουν κοντά στη μονάδα ή να παρουσιάζουν μικρή μόνο βελτίωση. Αυτό υποδηλώνει ότι το κόστος συγχρονισμού και το contention υπερισχύουν των ωφελειών από την παράλληλη εκτέλεση.

Επίδραση του μεγέθους της λίστας

Η σύγκριση μεταξύ $S=1024$ και $S=8192$ δείχνει ότι το μεγαλύτερο μέγεθος λίστας επηρεάζει αρνητικά το throughput σε όλες τις υλοποιήσεις, λόγω αυξημένου κόστους διάσχισης και μνήμης. Ωστόσο, το σχήμα των καμπυλών speedup παραμένει παρόμοιο, γεγονός που υποδηλώνει ότι οι βασικοί περιορισμοί κλιμάκωσης κάθε μηχανισμού συγχρονισμού δεν αλλάζουν με το μέγεθος της λίστας, αλλά σχετίζονται κυρίως με τον τρόπο συγχρονισμού.

Συμπεράσματα για το workload 100-0-0

Στο αμιγώς read-only workload προκύπτουν τα εξής:

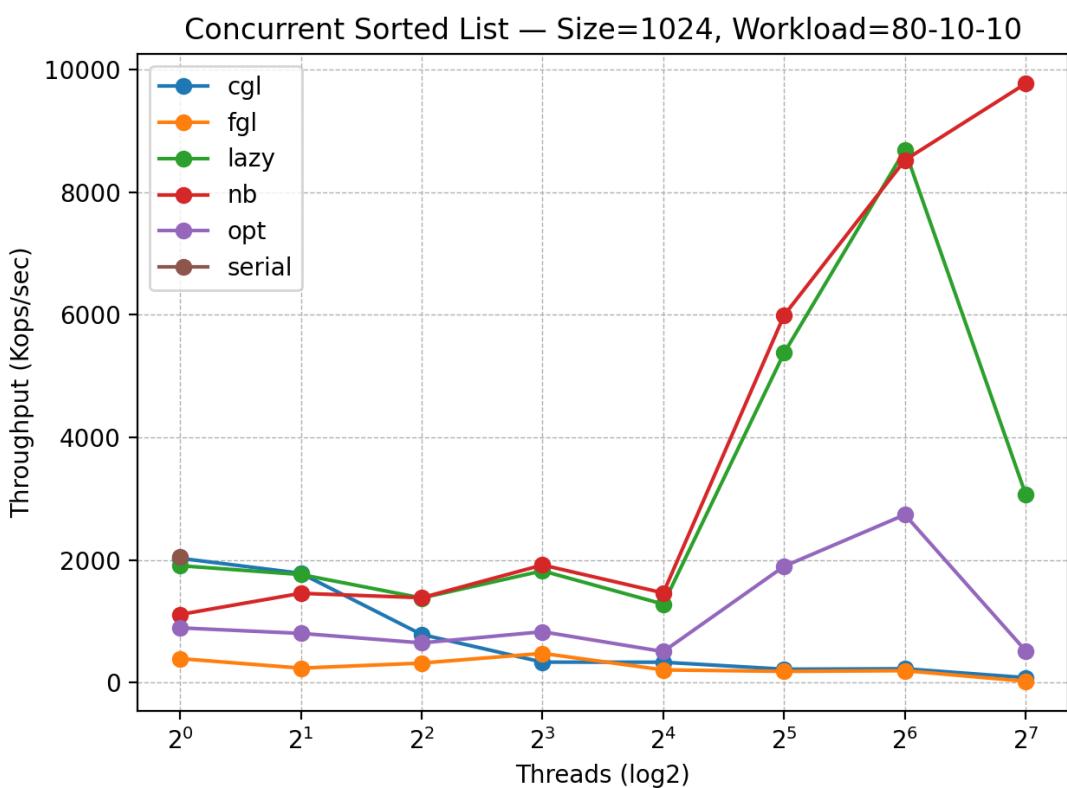
- Οι lazy synchronization και non-blocking υλοποιήσεις είναι οι πλέον κατάλληλες, καθώς οι αναζητήσεις εκτελούνται με ελάχιστο ή μηδενικό blocking.
- Η coarse-grain locking υλοποίηση δεν κλιμακώνεται, λόγω της πλήρους σειριοποίησης των λειτουργιών.
- Οι fine-grain και optimistic υλοποιήσεις παρουσιάζουν καλύτερη συμπεριφορά από την coarse-grain, αλλά παραμένουν κατώτερες σε σχέση με lazy και non-blocking λόγω αυξημένου κόστους συγχρονισμού.
- Η κλιμάκωση περιορίζεται μετά τα 64 νήματα, κυρίως λόγω αρχιτεκτονικών περιορισμών (hyperthreading και oversubscription) και όχι λόγω αλγορίθμικής αστοχίας.

B. Workload 80-10-10 (κυρίως αναζητήσεις)

Στο workload 80-10-10, το 80% των λειτουργιών είναι `contains()`, ενώ το υπόλοιπο 20% κατανέμεται ισομερώς σε `add()` και `remove()`. Το σενάριο αυτό προσομοιώνει ένα ρεαλιστικό `read-mostly` workload, όπου συνυπάρχουν αναζητήσεις και περιορισμένος αριθμός ενημερώσεων.

Σε αντίθεση με το 100-0-0, εδώ αρχίζουν να εμφανίζονται συγκρούσεις μεταξύ νημάτων λόγω των updates, γεγονός που επιτρέπει να αξιολογηθεί η συμπεριφορά των διαφορετικών μηχανισμών συγχρονισμού υπό μέτριο contention.

Throughput ως προς τον αριθμό νημάτων

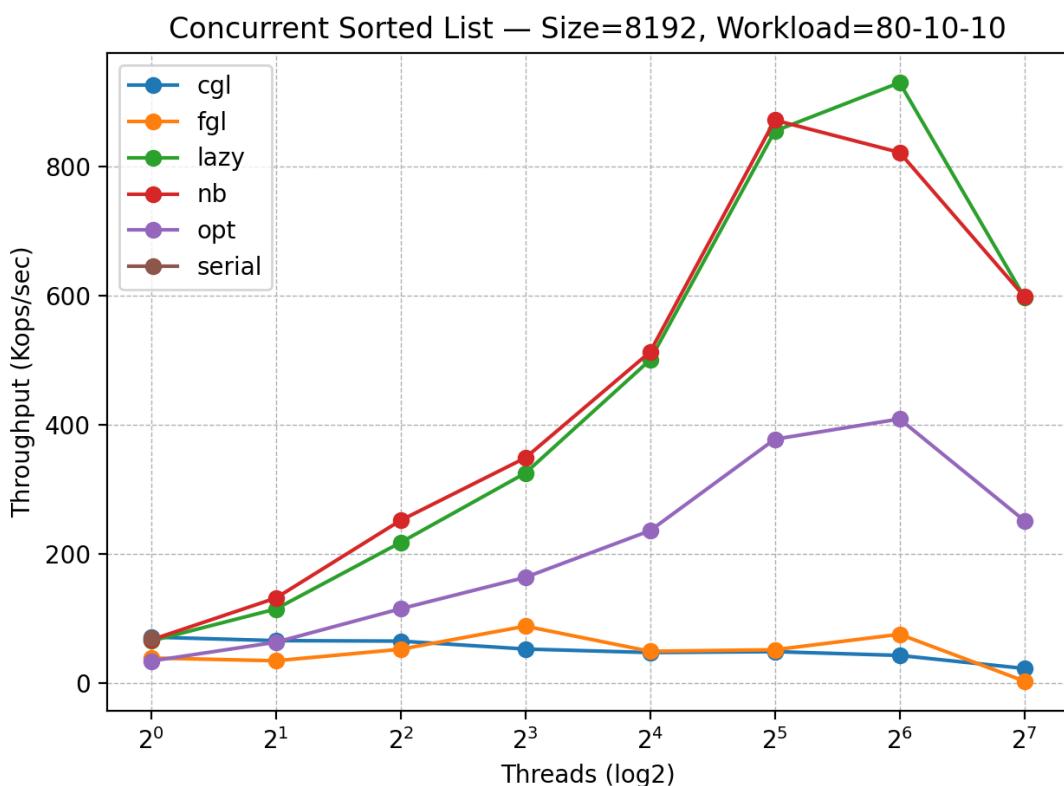


Για μικρό μέγεθος λίστας, η `lazy synchronization` εμφανίζει την καλύτερη συνολική απόδοση. Το throughput αυξάνεται σταθερά μέχρι τα 32–64 νήματα και στη συνέχεια παρουσιάζει ελαφρά σταθεροποίηση. Η καλή αυτή συμπεριφορά

οφείλεται στο γεγονός ότι οι `contains()` εκτελούνται χωρίς locks, ενώ οι ενημερώσεις υλοποιούνται με σύντομο τοπικό locking και λογική διαγραφή.

Η optimistic synchronization ακολουθεί σε απόδοση, όμως η αύξηση του αριθμού των νημάτων οδηγεί σε συχνότερα αποτυχημένα validations, με αποτέλεσμα η κλιμάκωση να περιορίζεται νωρίτερα σε σχέση με το lazy. Η non-blocking υλοποίηση παρουσιάζει καλή απόδοση σε χαμηλό και μεσαίο αριθμό νημάτων, αλλά σε υψηλότερο concurrency αρχίζει να επηρεάζεται από αποτυχημένες CAS και αυξημένο contention σε κοινά σημεία της λίστας.

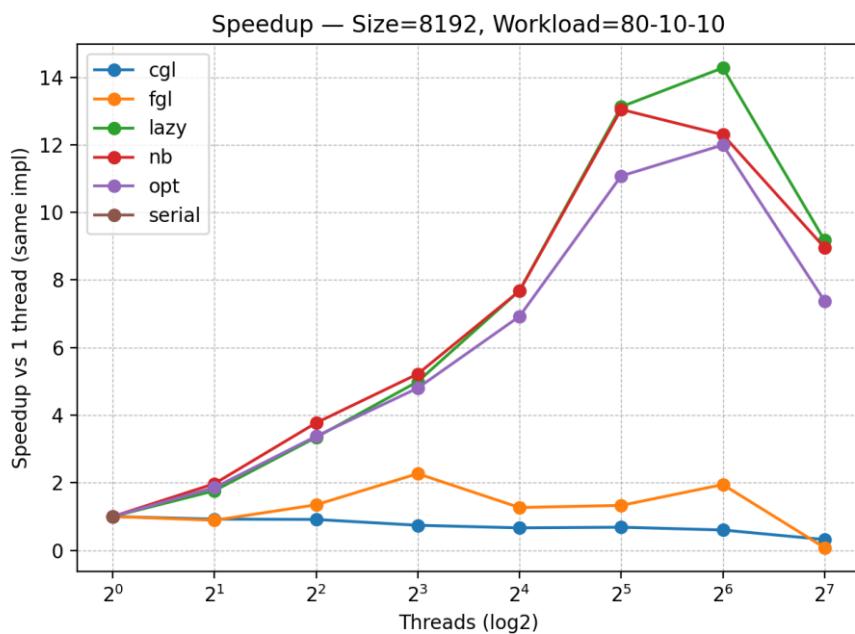
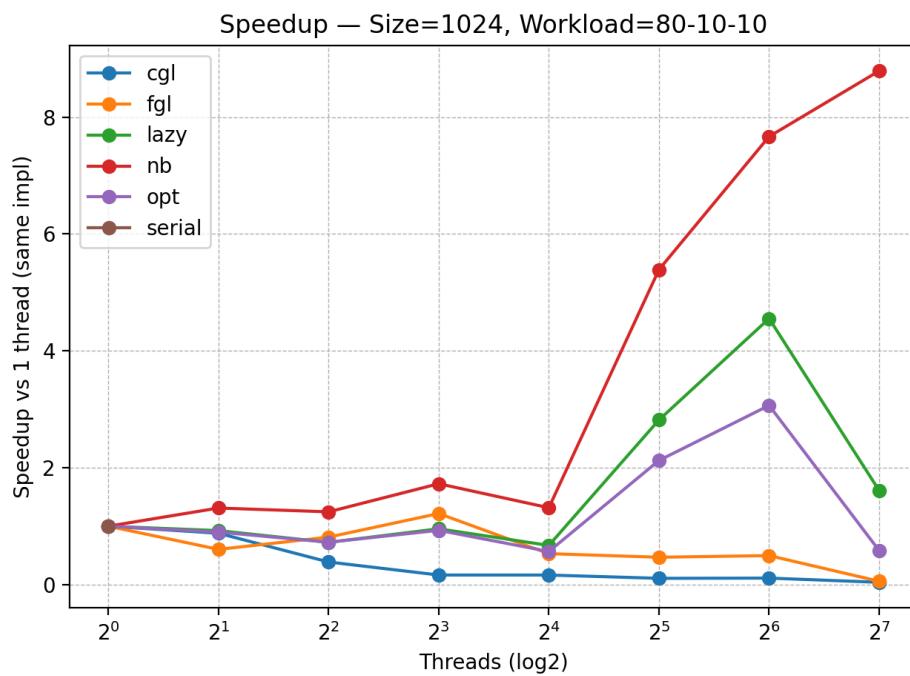
Οι υλοποιήσεις fine-grain και coarse-grain locking εμφανίζουν σαφώς χαμηλότερο throughput. Στην coarse-grain, όλες οι λειτουργίες σειριαποιούνται μέσω ενός καθολικού lock, ενώ στη fine-grain το κόστος των πολλαπλών lock/unlock γίνεται αισθητό ακόμη και με σχετικά περιορισμένο αριθμό ενημερώσεων.



Με τη μεγαλύτερη λίστα, το συνολικό throughput μειώνεται για όλες τις υλοποιήσεις, λόγω αυξημένου κόστους διάσχισης και χειρότερης χωρικής τοπικότητας μνήμης. Παρόλα αυτά, η lazy synchronization διατηρεί ξεκάθαρο προβάδισμα, παρουσιάζοντας την πιο σταθερή συμπεριφορά σε όλο το εύρος των νημάτων.

Η optimistic synchronization επηρεάζεται περισσότερο από το μεγαλύτερο μέγεθος λίστας, καθώς κάθε αποτυχημένο validation συνεπάγεται επαναδιάσχιση μεγαλύτερου τμήματος της δομής. Η non-blocking υλοποίηση συνεχίζει να κλιμακώνει έως ένα σημείο, αλλά ο κορεσμός εμφανίζεται νωρίτερα σε σχέση με το S=1024.

Speedup ως προς τον αριθμό νημάτων



Η ανάλυση του speedup δείχνει ότι:

- Η lazy synchronization παρουσιάζει την καλύτερη κλιμάκωση και στα δύο μεγέθη λίστας, με σχεδόν γραμμική αύξηση έως τα 32–64 νήματα.
- Η optimistic και η non-blocking υλοποίηση εμφανίζουν μέτριο speedup, το οποίο περιορίζεται καθώς αυξάνονται τα retries (λόγω validation ή CAS αποτυχιών).
- Οι locking-based υλοποιήσεις (coarse και fine-grain) παρουσιάζουν περιορισμένη επιτάχυνση, με τις καμπύλες speedup να παραμένουν χαμηλά.
- Σε όλα τα σχήματα, η μετάβαση από 64 σε 128 νήματα δεν οδηγεί σε ουσιαστική επιπλέον επιτάχυνση, γεγονός που αποδίδεται στη χρήση hyperthreading και oversubscription.

Επίδραση του μεγέθους της λίστας

Η αύξηση του μεγέθους της λίστας από 1024 σε 8192 στοιχεία:

- Μειώνει το απόλυτο throughput σε όλες τις υλοποιήσεις.
- Επηρεάζει δυσανάλογα τις optimistic και fine-grain υλοποιήσεις, όπου το κόστος αποτυχημένων προσπαθειών (validation ή locks) αυξάνεται με το μήκος της διάσχισης.

Παρόλα αυτά, η σχετική κατάταξη των υλοποιήσεων παραμένει σταθερή, με τη lazy synchronization να αποτελεί την πιο ανθεκτική επιλογή ως προς την αύξηση του μεγέθους της λίστας.

Συμπεράσματα για το workload 80-10-10

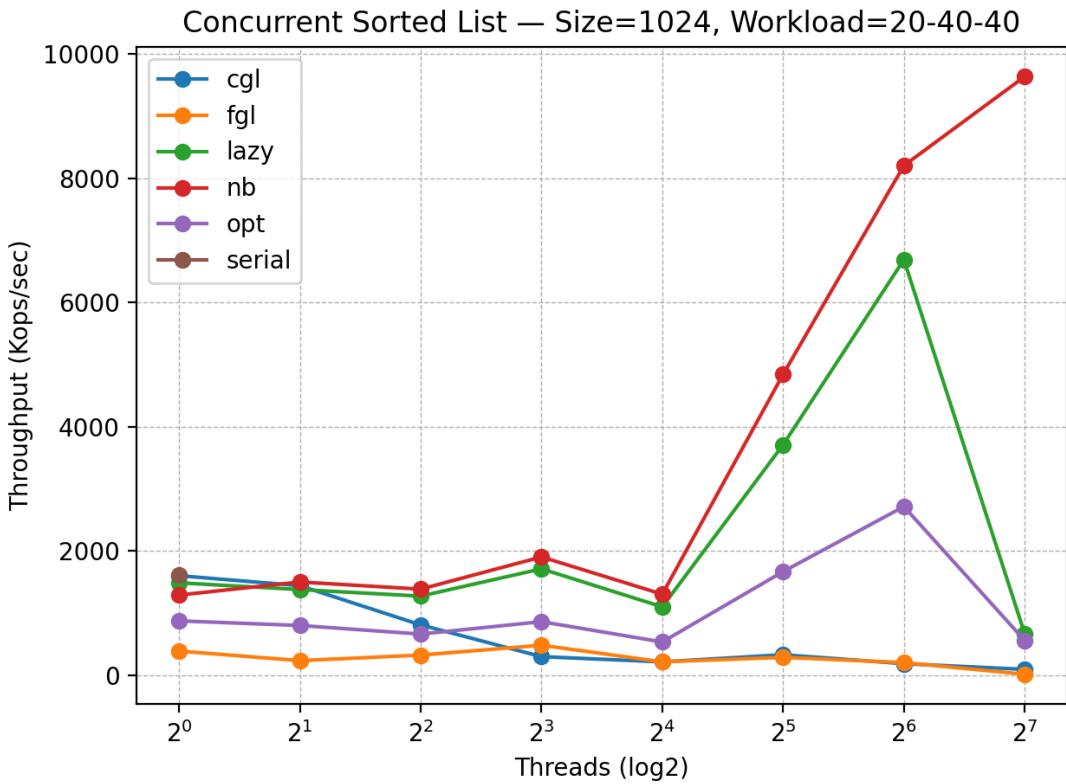
Από την ανάλυση του workload 80-10-10 προκύπτουν τα εξής:

- Η lazy synchronization αποτελεί την πιο αποδοτική και σταθερή υλοποίηση σε read-mostly σενάρια με περιορισμένα updates.
- Η optimistic synchronization λειτουργεί ικανοποιητικά, αλλά η απόδοσή της περιορίζεται από τα αποτυχημένα validations όσο αυξάνεται το concurrency.
- Η non-blocking υλοποίηση παρουσιάζει καλό scaling σε μέτριο αριθμό νημάτων, αλλά εμφανίζει κορεσμό σε υψηλό contention.
- Οι coarse-grain και fine-grain locking υλοποιήσεις υστερούν σημαντικά, επιβεβαιώνοντας ότι το blocking και το lock overhead επηρεάζουν αρνητικά την απόδοση ακόμη και όταν τα updates είναι σχετικά λίγα.

Γ. Workload 20-40-40 (κυρίως ενημερώσεις)

Στο workload 20-40-40, μόνο το 20% των λειτουργιών είναι contains(), ενώ το 80% αφορά ενημερώσεις (add() και remove()). Πρόκειται για ένα update-dominated σενάριο, στο οποίο το contention στη δομή δεδομένων είναι έντονο και η αποδοτικότητα των μηχανισμών συγχρονισμού παίζει καθοριστικό ρόλο.

Σε αντίθεση με τα read-mostly workloads, εδώ αναδεικνύεται το κόστος των locks, των αποτυχημένων validations και των επαναλαμβανόμενων atomic retries.

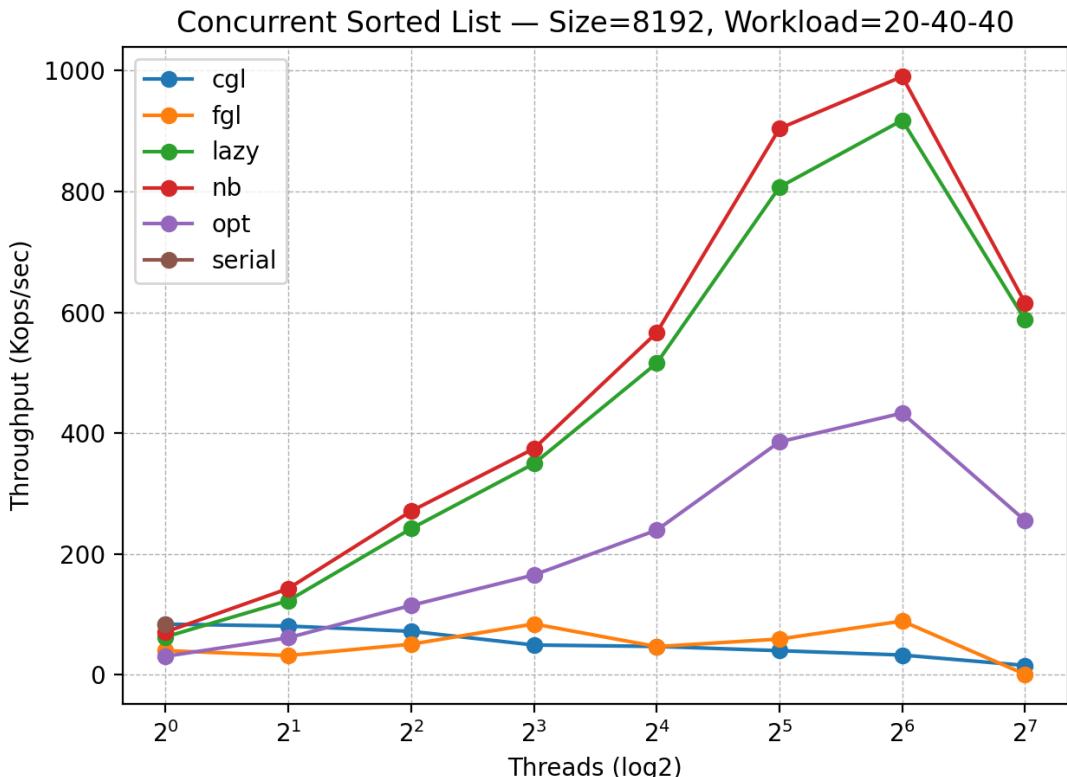


Για μικρό μέγεθος λίστας, η non-blocking (lock-free) υλοποίηση επιτυγχάνει το υψηλότερο throughput, παρουσιάζοντας σαφή υπεροχή σε μεγάλο εύρος αριθμού νημάτων. Το throughput αυξάνεται έντονα έως τα 64 νήματα, γεγονός που υποδηλώνει ότι η αποφυγή locks επιτρέπει μεγαλύτερο βαθμό ταυτόχρονης προόδου ακόμη και υπό υψηλό contention.

Η lazy synchronization ακολουθεί σε απόδοση, με καλή κλιμάκωση έως τα 32–64 νήματα. Παρότι χρησιμοποιεί locks στις ενημερώσεις, ο διαχωρισμός λογικής και φυσικής διαγραφής μειώνει τη διάρκεια κατοχής locks και περιορίζει τις συγκρούσεις.

Η optimistic synchronization παρουσιάζει αισθητά χαμηλότερο throughput. Η συχνότητα των αποτυχημένων validations αυξάνεται λόγω των πολλών updates, με αποτέλεσμα σημαντικό ποσοστό του χρόνου εκτέλεσης να αναλώνεται σε επαναλήψεις (retries).

Οι υλοποιήσεις fine-grain και coarse-grain locking εμφανίζουν τη χαμηλότερη απόδοση. Στην coarse-grain, όλες οι ενημερώσεις σειριοποιούνται, ενώ στη fine-grain το κόστος από τα πολλαπλά locks σε κάθε ενημέρωση επιβαρύνει έντονα το throughput.



Για μεγαλύτερο μέγεθος λίστας, το throughput μειώνεται σε όλες τις υλοποιήσεις, καθώς οι ενημερώσεις απαιτούν μεγαλύτερες διασχίσεις και αυξάνεται το memory contention.

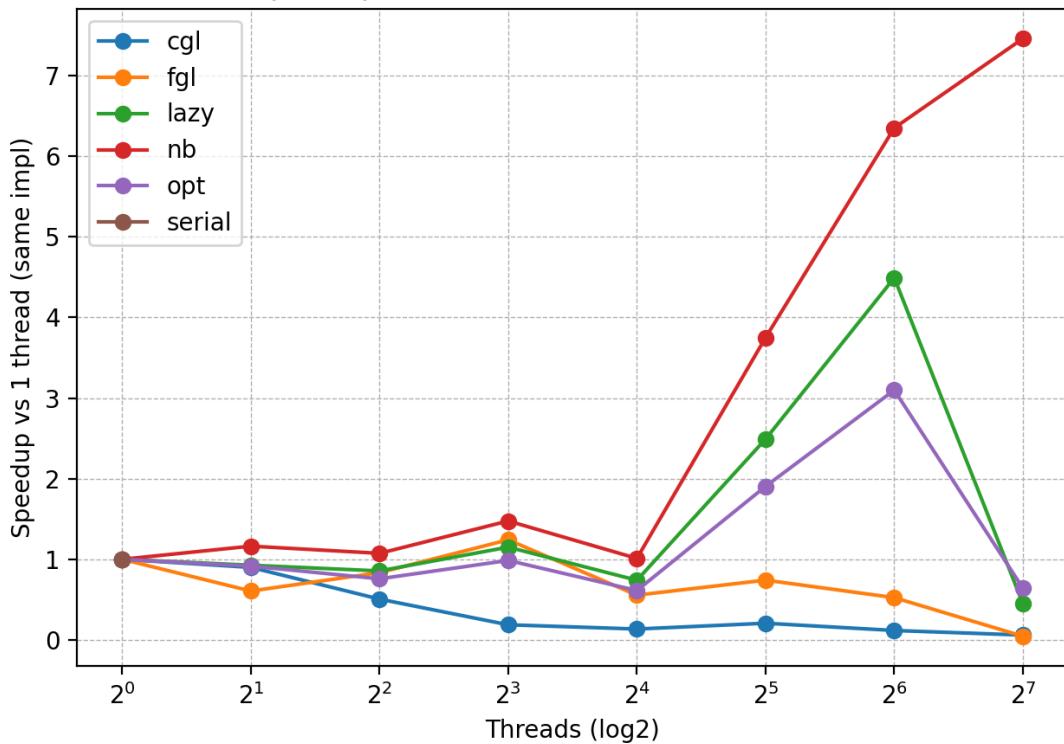
Η non-blocking υλοποίηση παραμένει η καλύτερη σε απόλυτους όρους, φτάνοντας σε μέγιστο throughput γύρω στα 32–64 νήματα, πριν εμφανίσει πτώση στα 128 νήματα. Η πτώση αυτή αποδίδεται στον συνδυασμό αυξημένου CAS contention και hyperthreading/oversubscription.

Η lazy synchronization διατηρεί σταθερά καλή απόδοση, αν και υστερεί ελαφρώς έναντι της non-blocking σε αυτό το έντονα update-heavy σενάριο. Η optimistic synchronization επηρεάζεται ακόμη περισσότερο από το αυξημένο μήκος της λίστας, καθώς κάθε αποτυχημένο validation συνεπάγεται επαναδιάσχιση μεγαλύτερου τμήματος της δομής.

Speedup ως προς τον αριθμό νημάτων

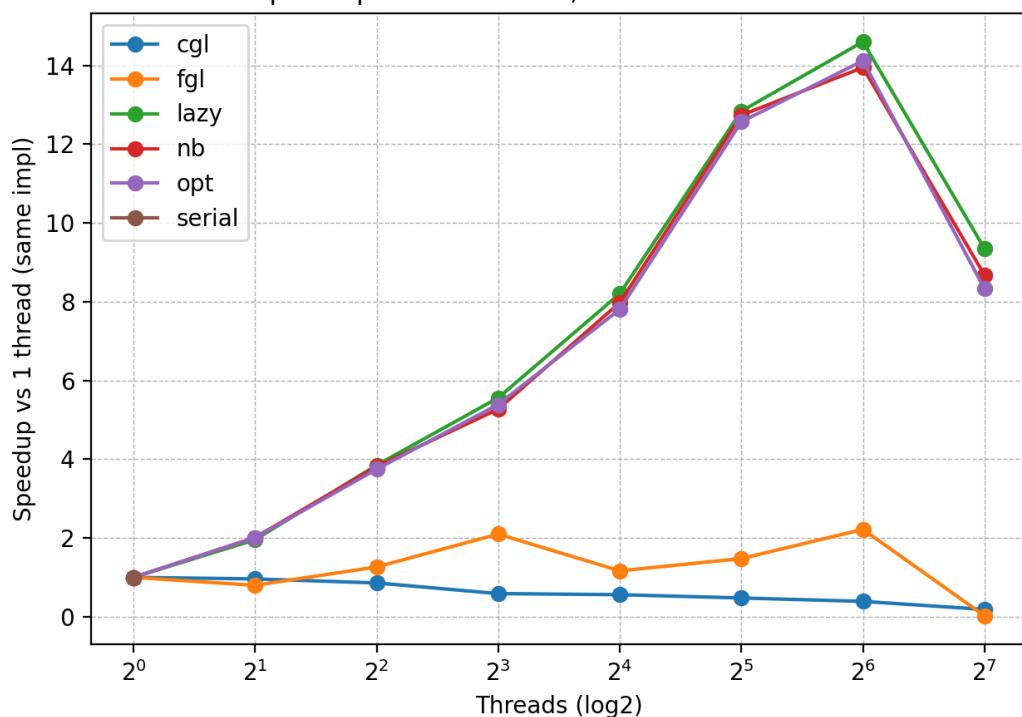
Η ανάλυση του speedup αποκαλύπτει σημαντικές διαφορές στη δυνατότητα κλιμάκωσης

Speedup — Size=1024, Workload=20-40-40



Για $S = 1024$, η non-blocking υλοποίηση επιτυγχάνει τη μεγαλύτερη επιτάχυνση, με speedup που ξεπερνά κατά πολύ τις υπόλοιπες υλοποιήσεις. Η lazy synchronization παρουσιάζει επίσης αξιόλογο speedup, αν και σαφώς χαμηλότερο.

Speedup — Size=8192, Workload=20-40-40



Για $S = 8192$, τόσο η non-blocking όσο και η lazy εμφανίζουν μέγιστο speedup περίπου στα 32–64 νήματα, με αισθητή πτώση στα 128 νήματα.

Οι coarse-grain και fine-grain locking υλοποιήσεις παρουσιάζουν speedup μικρότερο της μονάδας σε αρκετά σημεία, υποδεικνύοντας ότι η προσθήκη νημάτων όχι μόνο δεν επιταχύνει, αλλά επιβαρύνει την εκτέλεση λόγω έντονου contention.

Επίδραση του μεγέθους της λίστας

Η σύγκριση μεταξύ $S=1024$ και $S=8192$ δείχνει ότι το μεγαλύτερο μέγεθος λίστας:

- Μειώνει το throughput σε όλες τις υλοποιήσεις.
- Εντείνει ιδιαίτερα το κόστος των retries σε optimistic και non-blocking προσεγγίσεις.
- Καθιστά πιο εμφανές το πλεονέκτημα της lazy synchronization, η οποία περιορίζει τη διάρκεια κρίσιμων τμημάτων.

Παρότι το absolute throughput μειώνεται, η σχετική κατάταξη των υλοποιήσεων παραμένει παρόμοια, με τις non-blocking και lazy να υπερέχουν σε σχέση με τις locking-based λύσεις.

Συμπεράσματα για το workload 20-40-40

Από την ανάλυση του update-heavy workload προκύπτουν τα εξής:

- Η non-blocking (lock-free) υλοποίηση επιτυγχάνει την υψηλότερη απόδοση, ιδιαίτερα σε μικρό και μεσαίο μέγεθος λίστας, επιβεβαιώνοντας το πλεονέκτημα της απουσίας locks σε περιβάλλον έντονων ενημερώσεων.
- Η lazy synchronization αποτελεί τον καλύτερο συμβιβασμό μεταξύ απόδοσης και σταθερότητας, ειδικά όταν αυξάνεται το μέγεθος της λίστας.
- Η optimistic synchronization υποφέρει σημαντικά λόγω συχνών αποτυχημένων validations και δεν ενδείκνυται για update-dominated workloads.
- Οι coarse-grain και fine-grain locking υλοποιήσεις παρουσιάζουν πολύ περιορισμένη κλιμάκωση και αποτελούν τις λιγότερο αποδοτικές επιλογές στο συγκεκριμένο σενάριο.

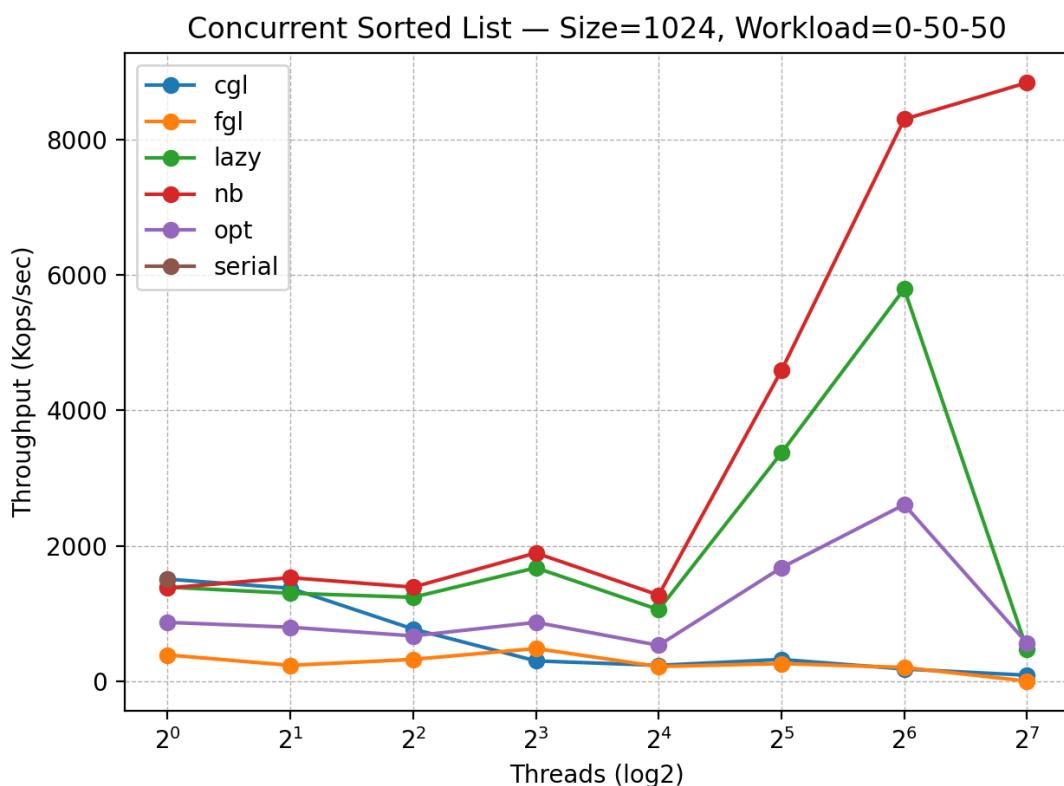
Η πτώση της απόδοσης στα 128 νήματα οφείλεται κυρίως σε αρχιτεκτονικούς περιορισμούς (hyperthreading και oversubscription), σε συνδυασμό με αυξημένο contention σε atomic ή locking μηχανισμούς.

Δ. Workload 0-50-50 (μόνο ενημερώσεις)

Στο workload 0-50-50 όλες οι λειτουργίες είναι ενημερώσεις (add() και remove()), χωρίς καθόλου αναζητήσεις. Πρόκειται για το πιο απαιτητικό σενάριο από πλευράς συγχρονισμού, καθώς κάθε λειτουργία τροποποιεί τη δομή της λίστας και συνεπώς δημιουργείται έντονο contention τόσο σε locks όσο και σε atomic operations.

To workload αυτό αναδεικνύει καθαρά τις διαφορές μεταξύ blocking, optimistic και lock-free προσεγγίσεων.

Throughput ως προς τον αριθμό νημάτων

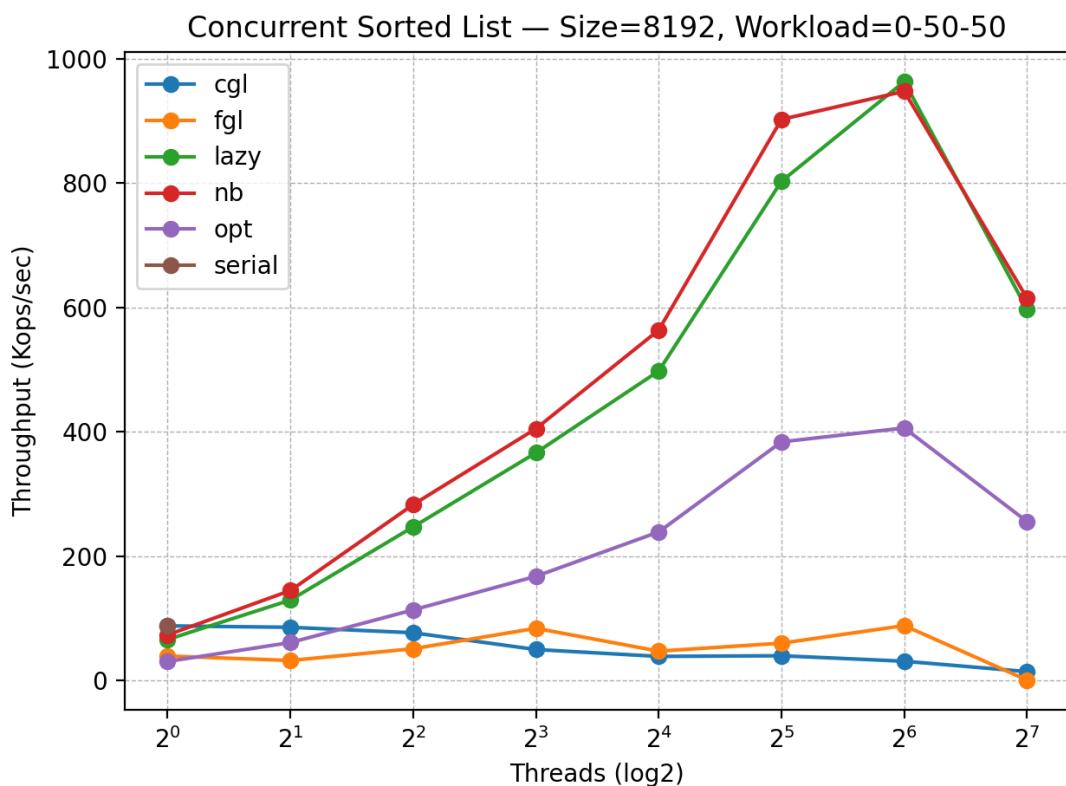


Για μικρό μέγεθος λίστας, η non-blocking (lock-free) υλοποίηση επιτυγχάνει το υψηλότερο throughput σε όλο το εύρος των νημάτων. Το throughput αυξάνεται σημαντικά έως τα 64 νήματα, όπου και παρατηρείται η μέγιστη απόδοση. Η απουσία locks επιτρέπει στα νήματα να προοδεύουν χωρίς να μπλοκάρουν μεταξύ τους, ακόμη και όταν όλες οι λειτουργίες είναι ενημερώσεις.

Η lazy synchronization ακολουθεί σε απόδοση, παρουσιάζοντας καλή κλιμάκωση έως τα 32–64 νήματα. Ο διαχωρισμός της λογικής από τη φυσική διαγραφή μειώνει το contention στα κρίσιμα τμήματα, αν και το κόστος των locks παραμένει αισθητό σε σύγκριση με την lock-free προσέγγιση.

Η optimistic synchronization εμφανίζει μέτριο throughput. Τα συχνά updates οδηγούν σε αποτυχημένα validations και retries, τα οποία περιορίζουν την απόδοση, αν και παραμένει σαφώς ανώτερη από τις locking-based υλοποιήσεις.

Οι fine-grain και coarse-grain locking υλοποιήσεις παρουσιάζουν τη χαμηλότερη απόδοση. Στην coarse-grain, όλες οι ενημερώσεις σειριοποιούνται, ενώ στη fine-grain το κόστος από τα πολλαπλά locks σε κάθε ενημέρωση επιβαρύνει έντονα το throughput.



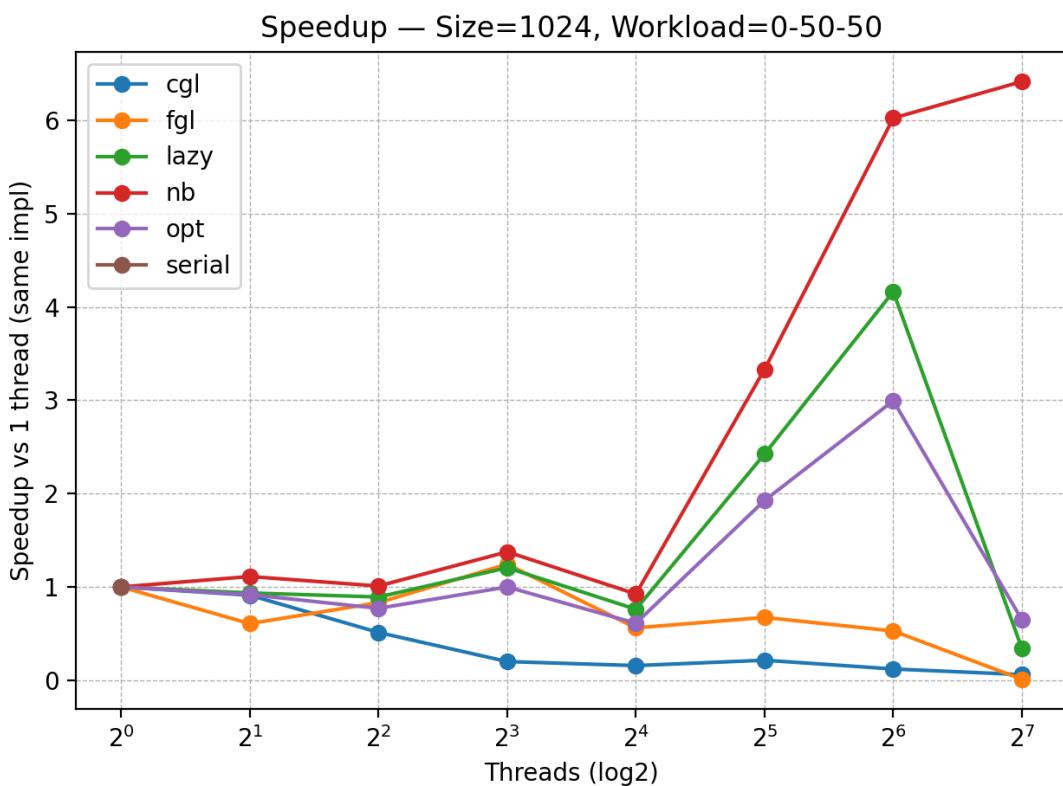
Με μεγαλύτερη λίστα, το throughput μειώνεται για όλες τις υλοποιήσεις, λόγω αυξημένου κόστους διάσχισης και μεγαλύτερης πιθανότητας συγκρούσεων.

Η non-blocking υλοποίηση παραμένει η ταχύτερη, φτάνοντας σε μέγιστο throughput γύρω στα 32–64 νήματα, πριν εμφανίσει πτώση στα 128 νήματα. Η πτώση αυτή αποδίδεται στον συνδυασμό έντονου CAS contention και oversubscription.

Η lazy synchronization παρουσιάζει παρόμοια ποιοτική συμπεριφορά, αλλά με χαμηλότερο απόλυτο throughput σε σχέση με τη non-blocking. Η optimistic synchronization επηρεάζεται ιδιαίτερα από το αυξημένο μέγεθος της λίστας, καθώς κάθε αποτυχημένο validation συνεπάγεται επαναδιάσχιση μεγαλύτερου τμήματος της δομής.

Speedup ως προς τον αριθμό νημάτων

Η ανάλυση του speedup επιβεβαιώνει τα συμπεράσματα από το throughput:

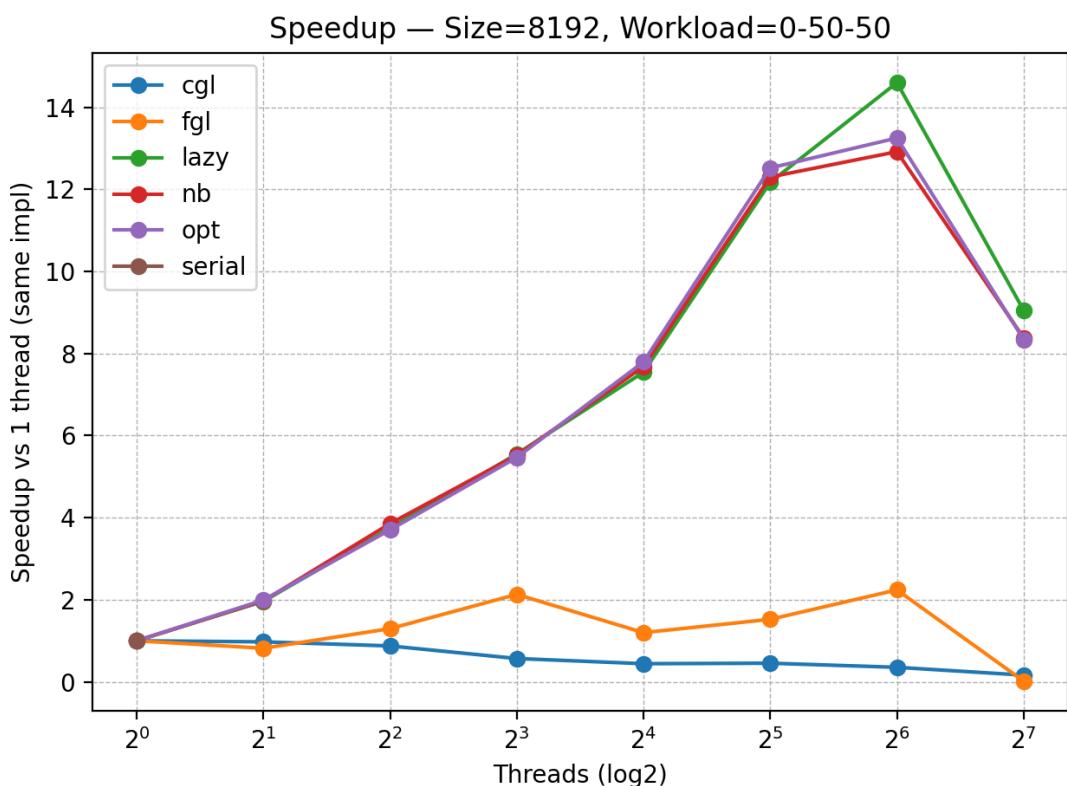


Για $S = 1024$, η non-blocking υλοποίηση παρουσιάζει τη μεγαλύτερη επιτάχυνση, ξεπερνώντας όλες τις άλλες υλοποιήσεις σε όλο το εύρος των νημάτων.

Η lazy synchronization επιτυγχάνει αξιόλογο speedup, αν και χαμηλότερο από τη non-blocking, λόγω του κόστους των locks στις ενημερώσεις.

Η optimistic synchronization εμφανίζει περιορισμένο speedup, καθώς τα retries ακυρώνουν μέρος του οφέλους από την παράλληλη εκτέλεση.

Οι coarse-grain και fine-grain locking υλοποιήσεις παρουσιάζουν speedup μικρότερο της μονάδας σε αρκετές περιπτώσεις, υποδεικνύοντας ότι η αύξηση του αριθμού νημάτων επιβαρύνει την εκτέλεση.



Για $S = 8192$, όλες οι καμπύλες speedup εμφανίζουν κορεσμό ή πτώση στα 128 νήματα, γεγονός που αποδίδεται στη χρήση hyperthreading και oversubscription σε συνδυασμό με έντονο contention.

Επίδραση του μεγέθους της λίστας

Η αύξηση του μεγέθους της λίστας από 1024 σε 8192 στοιχεία:

- Μειώνει σημαντικά το throughput σε όλες τις υλοποιήσεις.
- Εντείνει το κόστος των retries τόσο στις optimistic όσο και στις non-blocking υλοποιήσεις.
- Καθιστά πιο εμφανές το πλεονέκτημα της lazy synchronization έναντι των locking-based λύσεων, λόγω της μείωσης του χρόνου κατοχής locks.

Παρά τη μείωση των απόλυτων τιμών, η σχετική κατάταξη των υλοποιήσεων παραμένει σταθερή, με τις non-blocking και lazy να υπερέχουν.

Συμπεράσματα για το workload 0-50-50

Από την ανάλυση του αμιγώς update-heavy workload προκύπτουν τα εξής:

- Η non-blocking (lock-free) υλοποίηση αποτελεί την πιο αποδοτική επιλογή, επιτυγχάνοντας το υψηλότερο throughput και speedup.
- Η lazy synchronization προσφέρει τον καλύτερο συμβιβασμό μεταξύ απόδοσης και σταθερότητας, ειδικά σε μεγαλύτερα μεγέθη λίστας.
- Η optimistic synchronization δεν ενδείκνυται για workloads με αποκλειστικά ενημερώσεις, λόγω συχνών αποτυχημένων validations.
- Οι coarse-grain και fine-grain locking υλοποιήσεις αποτυγχάνουν να κλιμακώσουν και εμφανίζουν έντονη υποβάθμιση της απόδοσης.

4. Γενικό Συμπέρασμα

Από τη μελέτη των διαφορετικών workloads προκύπτει ότι η απόδοση και η κλιμάκωση μιας ταυτόχρονης ταξινομημένης λίστας εξαρτώνται άμεσα από το ποσοστό ενημερώσεων και τον μηχανισμό συγχρονισμού. Οι υλοποιήσεις με coarse-grain και fine-grain locking παρουσιάζουν περιορισμένη κλιμάκωση λόγω blocking και αυξημένου lock overhead, ανεξάρτητα από το workload. Η optimistic synchronization αποδίδει καλά μόνο σε read-heavy σενάρια, αλλά υποβαθμίζεται σημαντικά όταν αυξάνονται οι ενημερώσεις λόγω συχνών retries.

Η lazy synchronization εμφανίζει τη πιο σταθερή και ισορροπημένη συμπεριφορά σε όλα τα workloads, ενώ η non-blocking (lock-free) υλοποίηση επιτυγχάνει την υψηλότερη απόδοση σε update-heavy σενάρια, με κόστος αυξημένο CAS contention σε υψηλό αριθμό νημάτων. Τέλος, η εμφάνιση κορεσμού ή πτώσης της απόδοσης στα 64–128 νήματα αποδίδεται κυρίως σε αρχιτεκτονικούς περιορισμούς (hyperthreading και oversubscription) και όχι στους ίδιους τους αλγορίθμους.

**Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Δεκέμβριος 2025**