

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
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

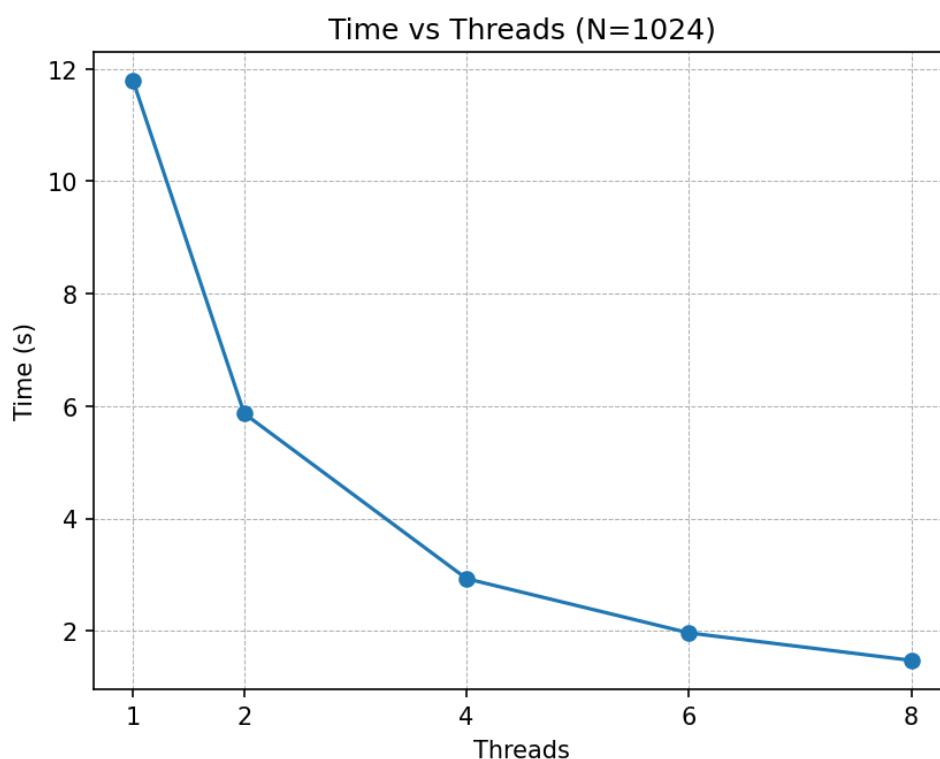
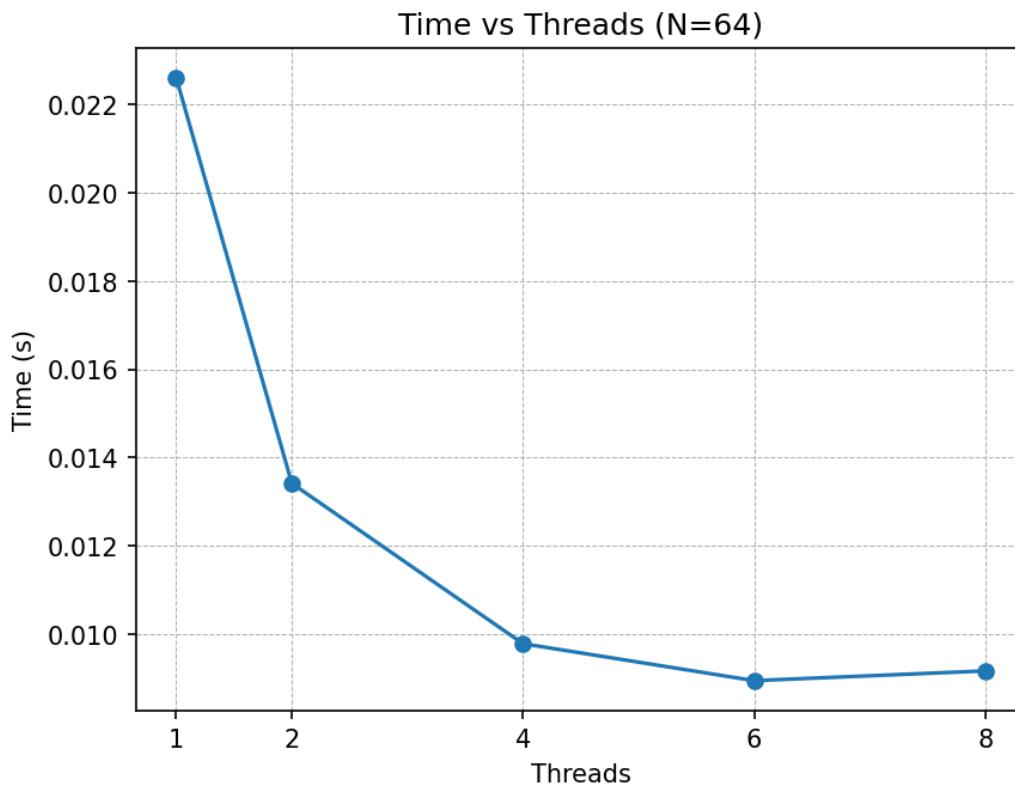
Τα αποτελέσματα αυτά αποτελούν τα δείγματα αναφοράς, με βάση τα οποία θα καταστρώσουμε τα ζητούμενα διαγράμματα (χρόνου και speedup) του επόμενου ερωτήματος. Επισημαίνουμε ότι:

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

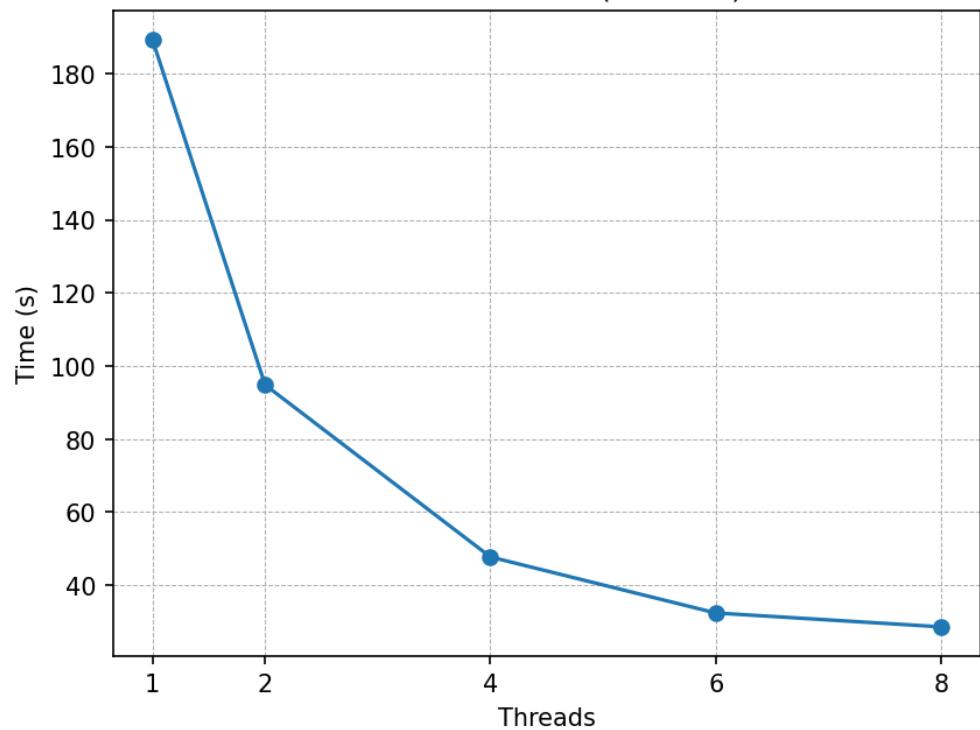
■ Ενότητα 1.4.5 – Τα Διαγράμματα

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

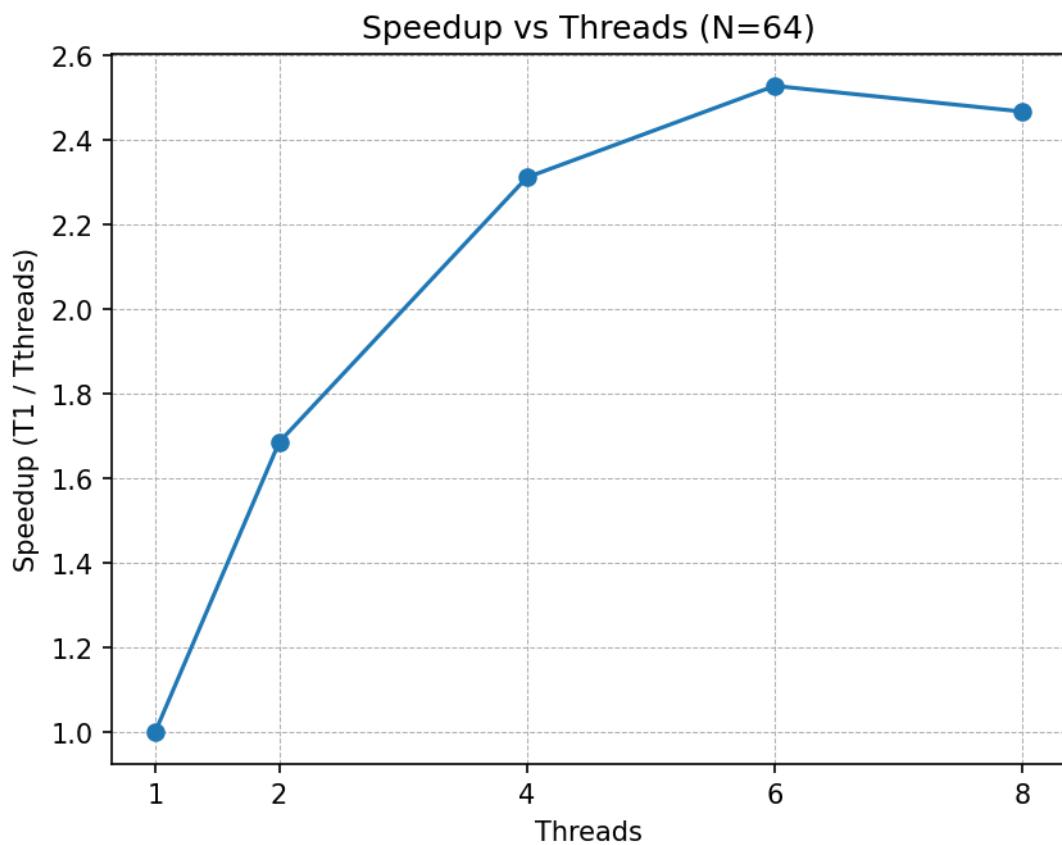
Διαγράμματα Χρόνου



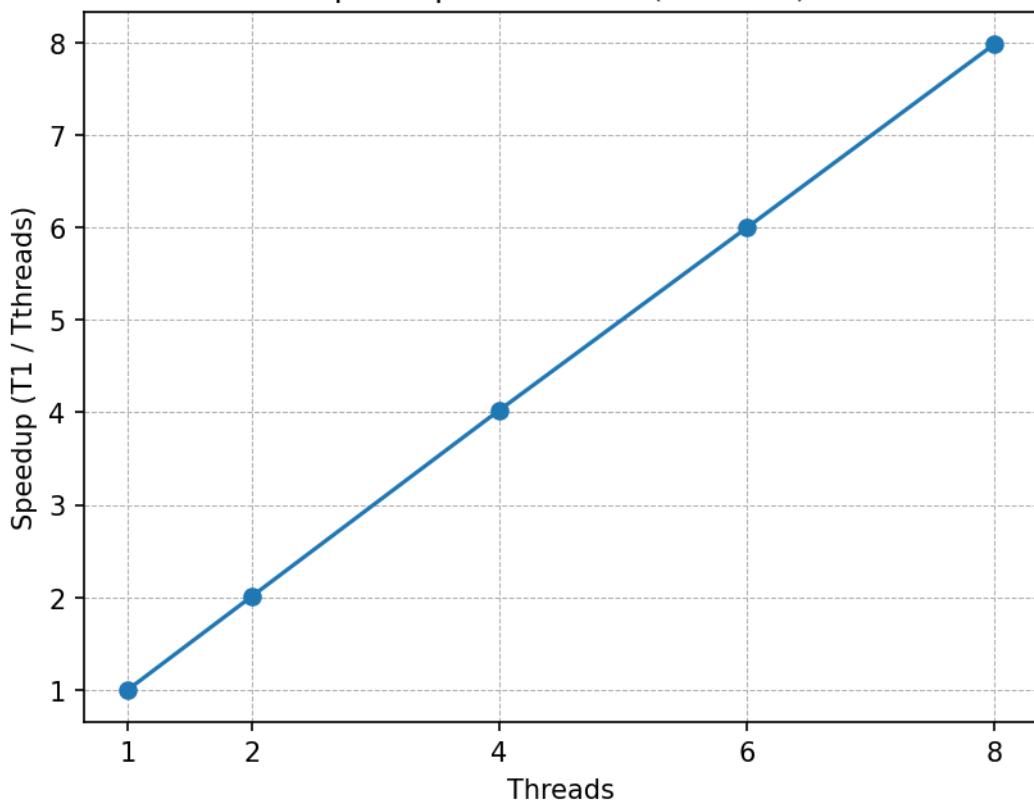
Time vs Threads (N=4096)



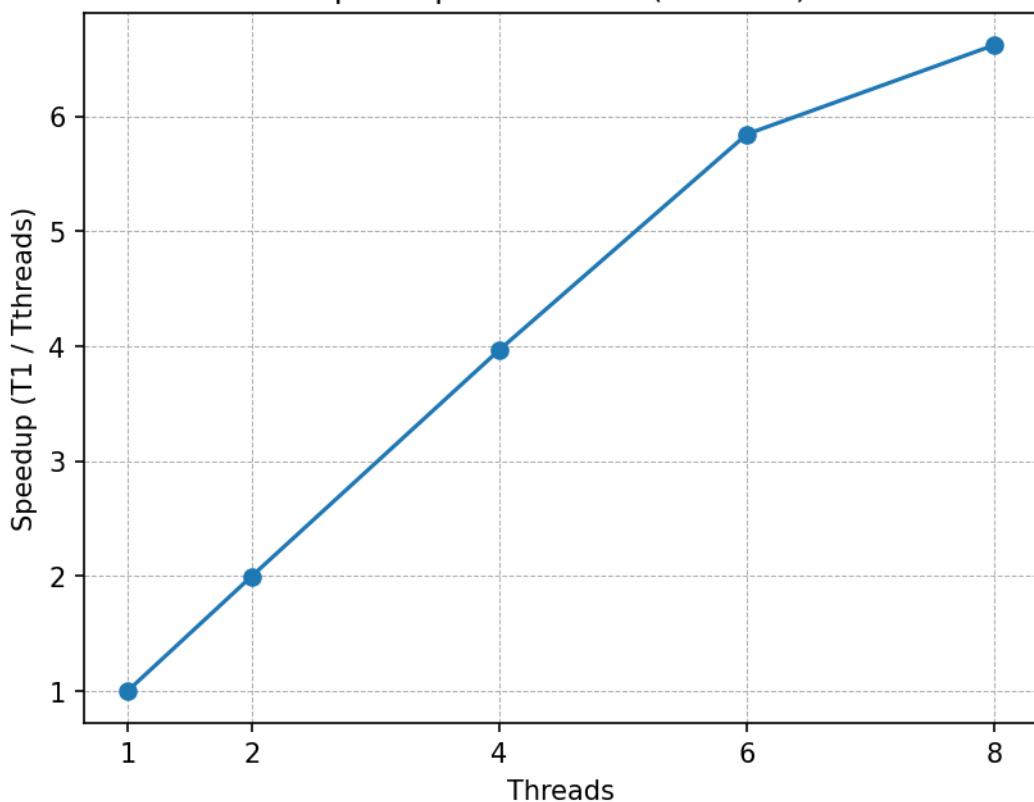
Διαγράμματα Speedup



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             {

```

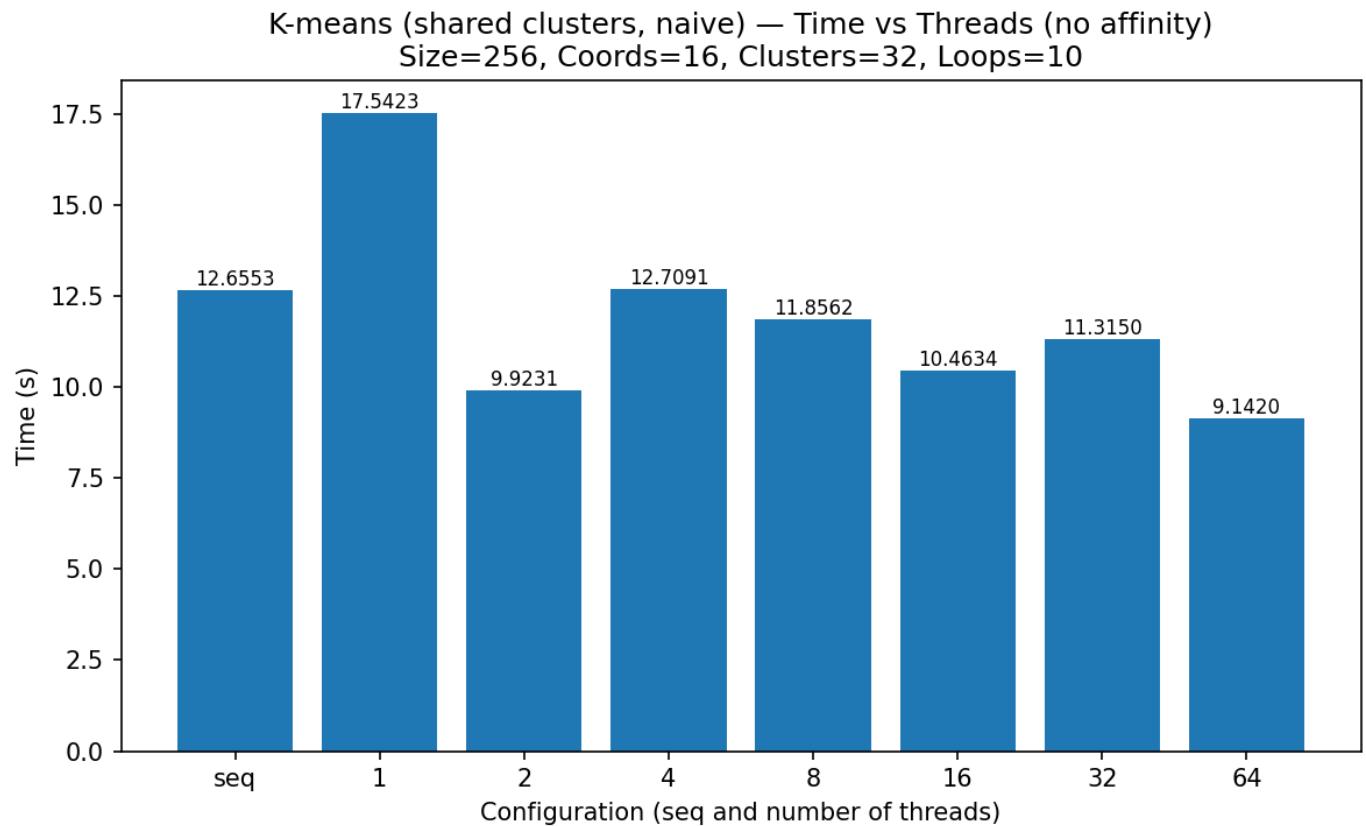
```
104 #pragma omp atomic // protect update on shared "delta" variable
105         delta += 1.0;
106     }
107
108     // assign the membership to object i
109     membership[i] = index;
110
111 // update new cluster centers : sum of objects located within
112 /*
113 * TODO: protect update on shared "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

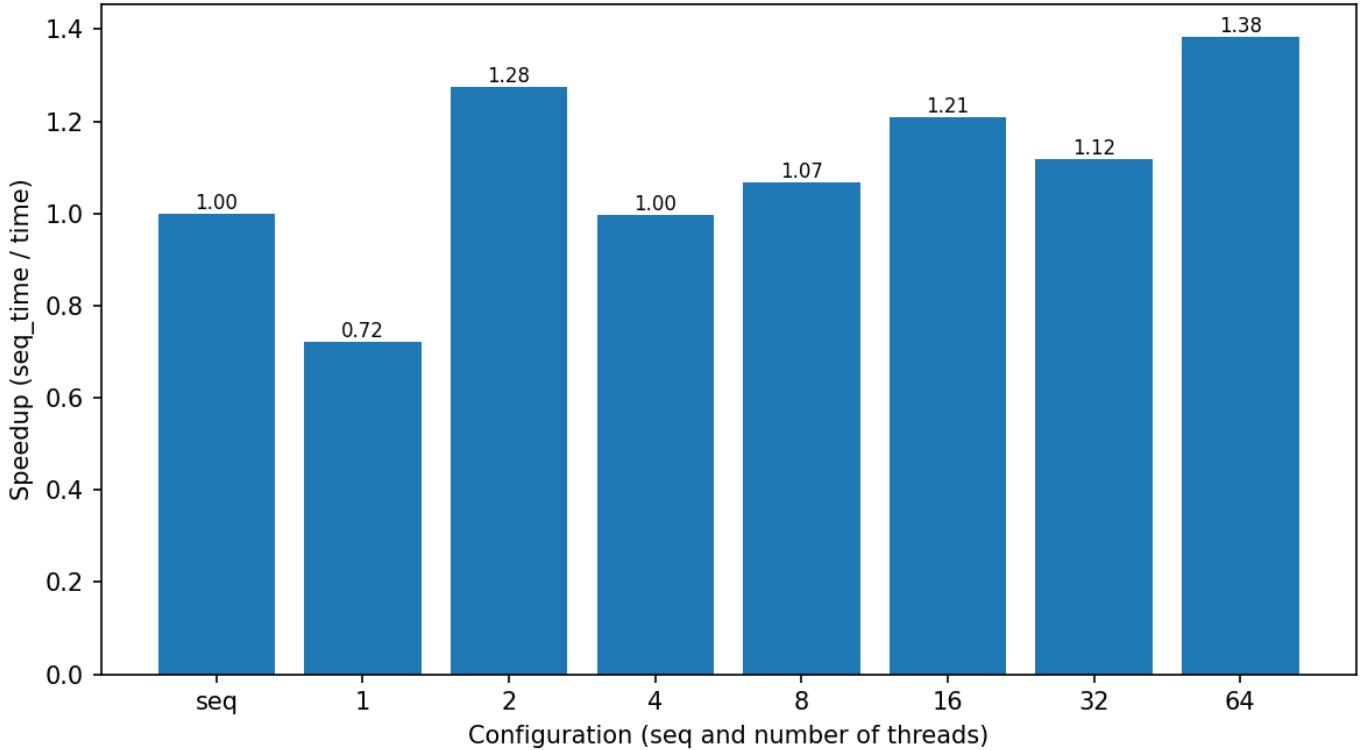
Το παραπάνω παράλληλο πρόγραμμα (omp_naive_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και χωρίς affinity. Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

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:



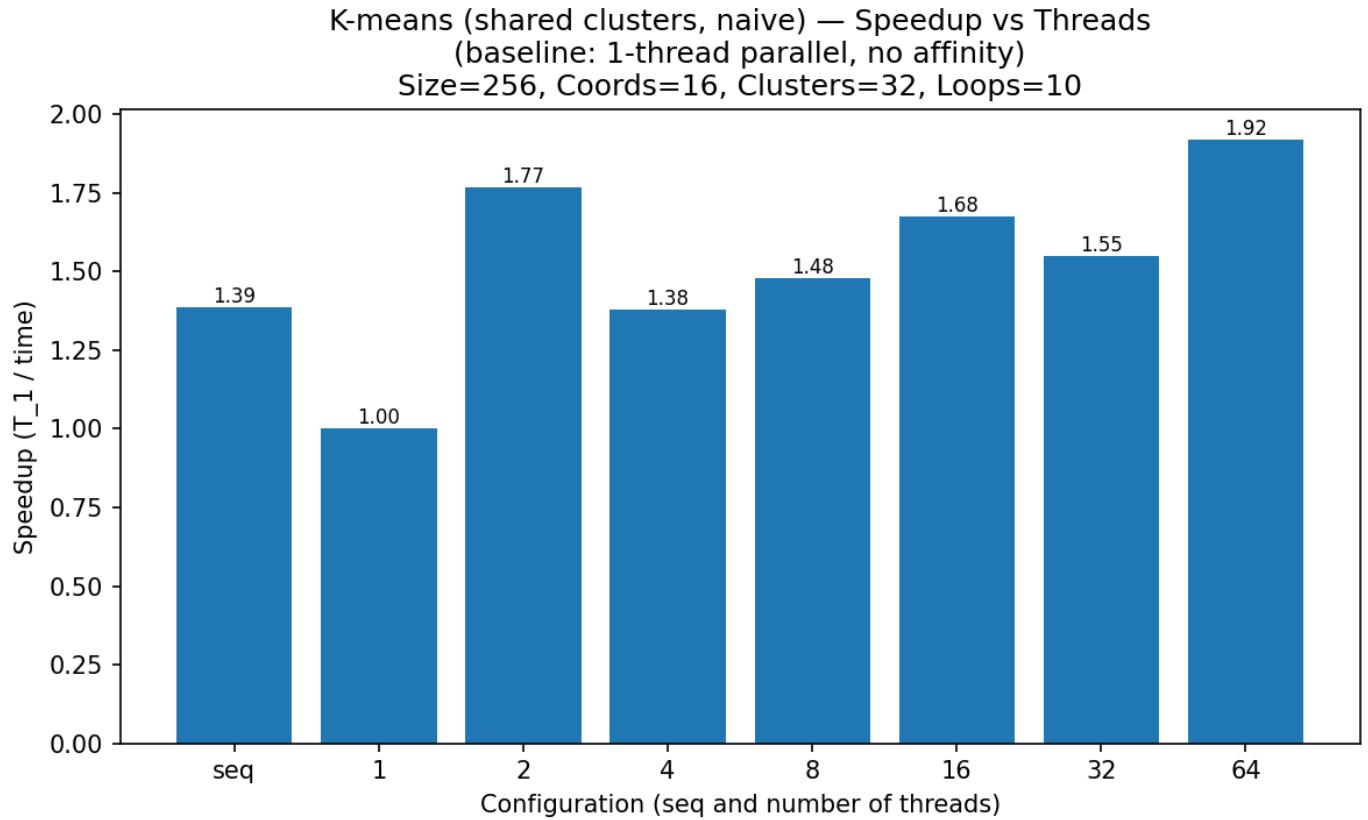
K-means (shared clusters, naive) — Speedup vs Threads (no affinity)
 Size=256, Coords=16, Clusters=32, Loops=10



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:



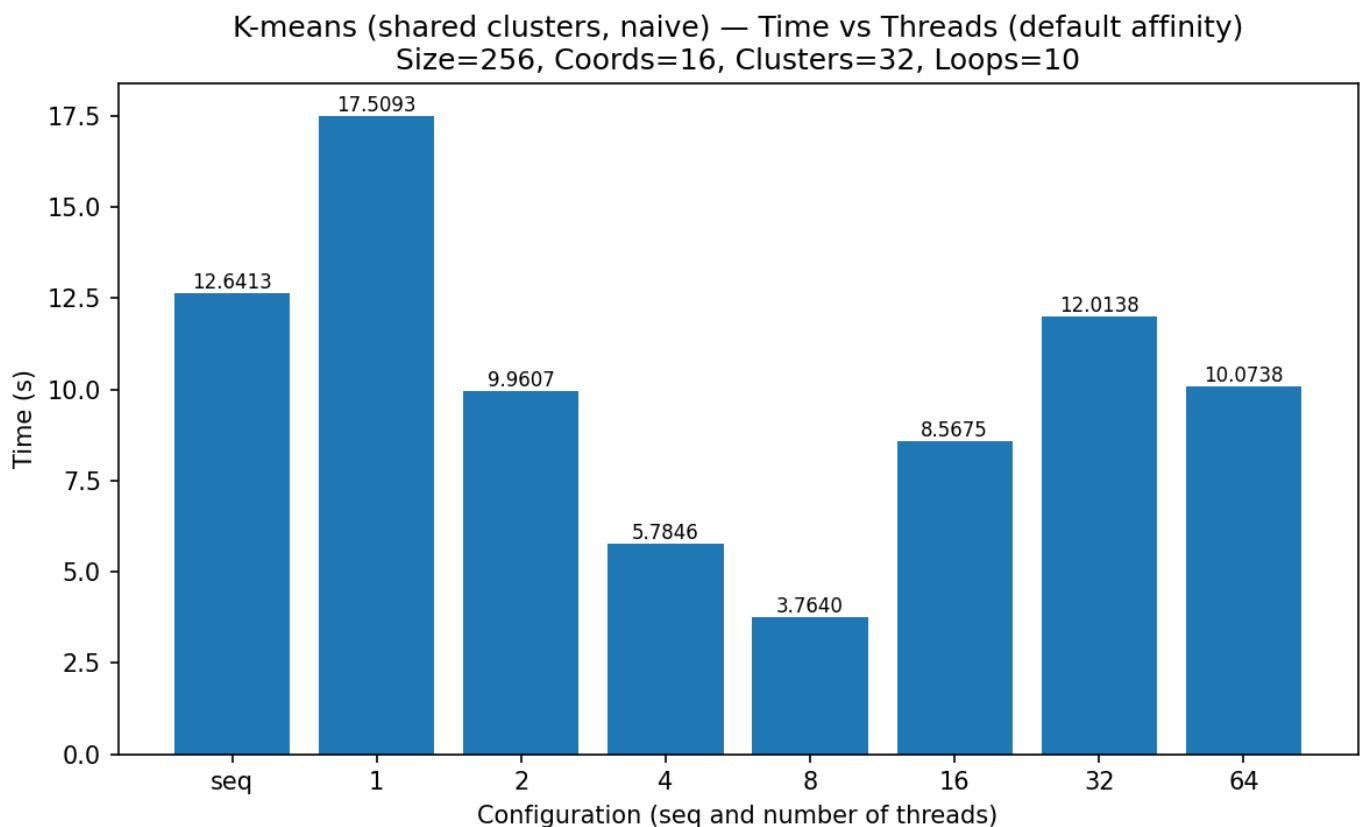
Από το παραπάνω διάγραμμα παρατηρούμε ότι, αν θεωρήσουμε ως βάση το παράλληλο πρόγραμμα με 1 νήμα, όλες οι εκτελέσεις με περισσότερα νήματα εμφανίζουν πλέον speedup μεγαλύτερο της μονάδας. Ενδεικτικά, για 2 νήματα το κέρδος είναι περίπου $1.8\times$ σε σχέση με το 1-thread (σχεδόν γραμμικό και το μόνο 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

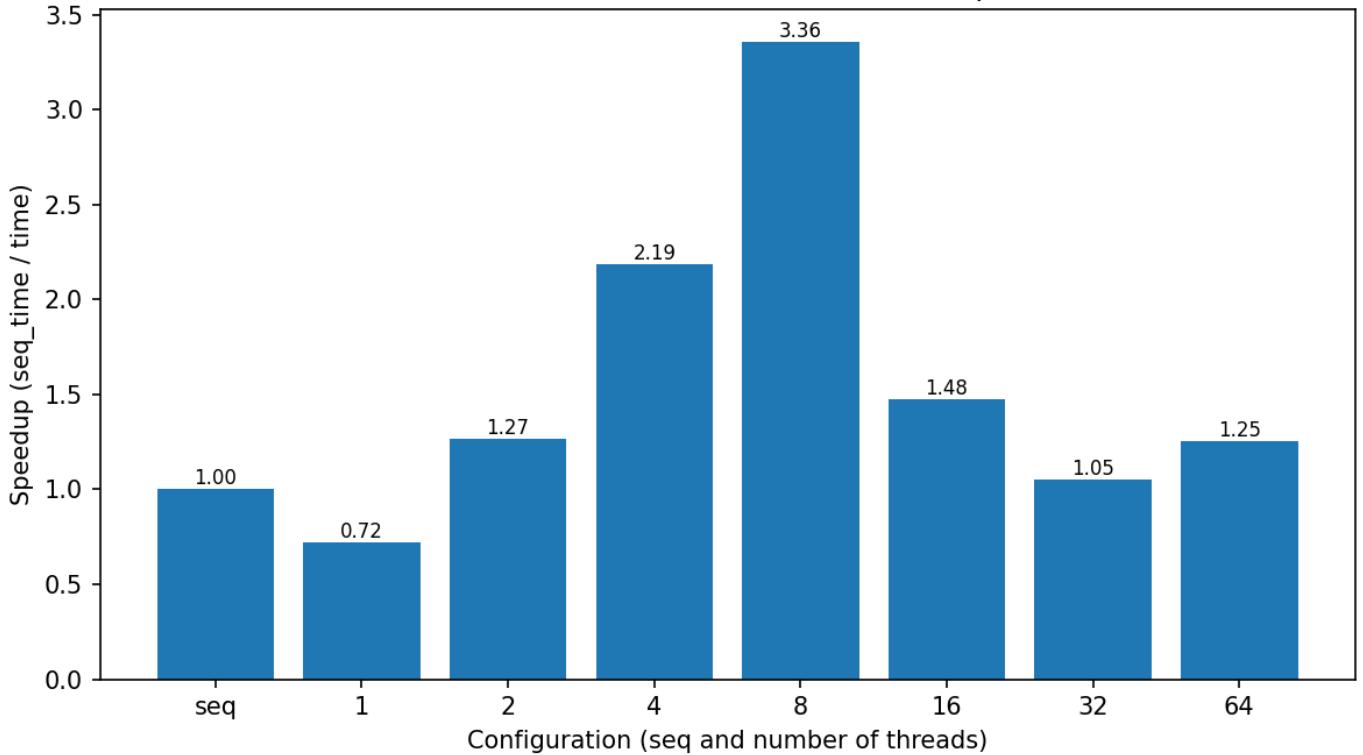
Το παραπάνω παράλληλο πρόγραμμα (omp_naive_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64, αλλά αυτή τη φορά με affinity. Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

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

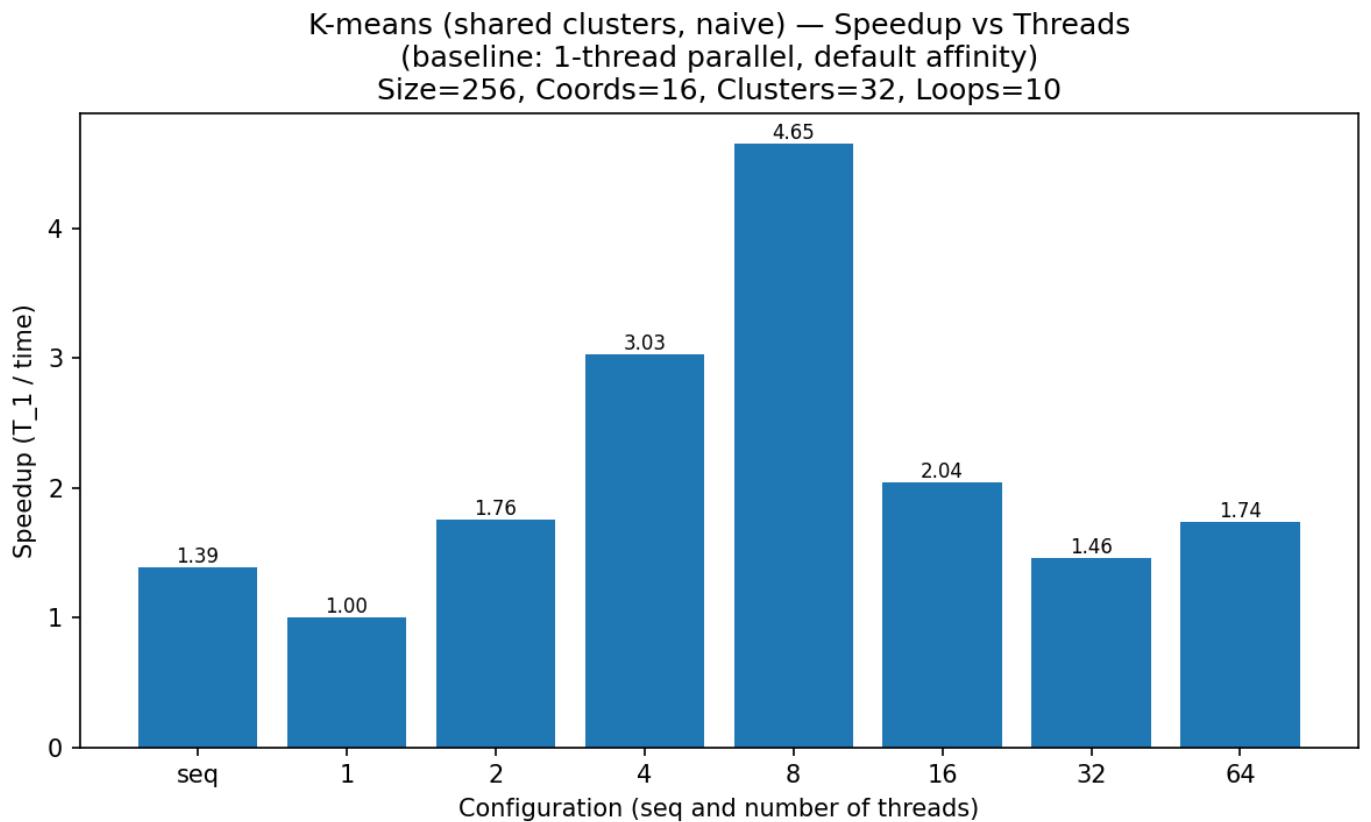


Παρατηρούμε ότι με ενεργοποιημένο το default affinity η συμπεριφορά της naive shared υλοποίησης βελτιώνεται σημαντικά σε σχέση με την περίπτωση χωρίς affinity. Ο χρόνος εκτέλεσης μειώνεται πολύ καλά οριακά γραμμικά ως προς το παράλληλο πρόγραμμα με 1 νήμα) μέχρι τα 8 νήματα (από ~17.5s στο 1 νήμα σε ~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     {

```

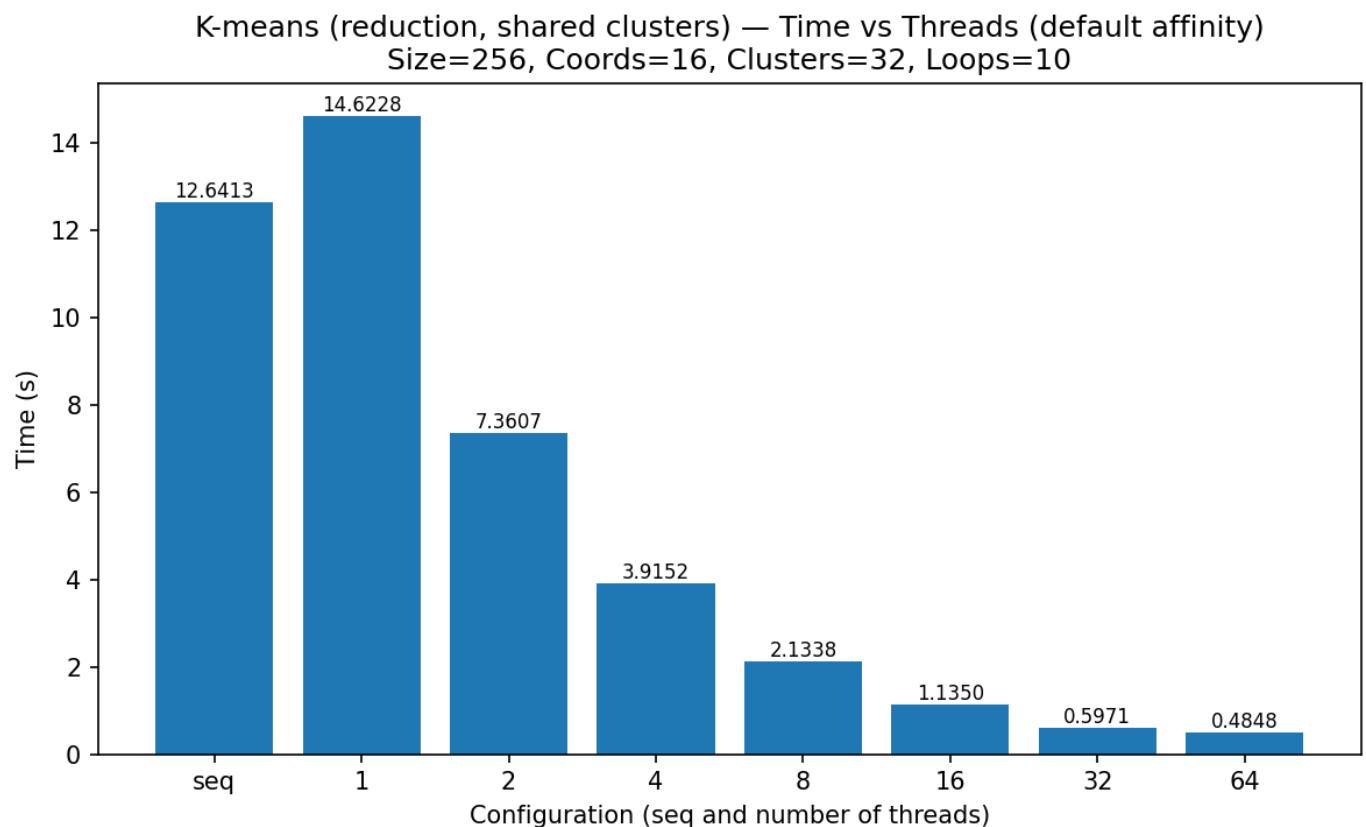
```
201     free(local_newClusterSize[k]);
202     free(local_newClusters[k]);
203 }
204 free(newClusters);
205 free(newClusterSize);
206 }
207
208 }
```

1. Παραλληλοποίηση για το αρχικό grid size και διαγράμματα

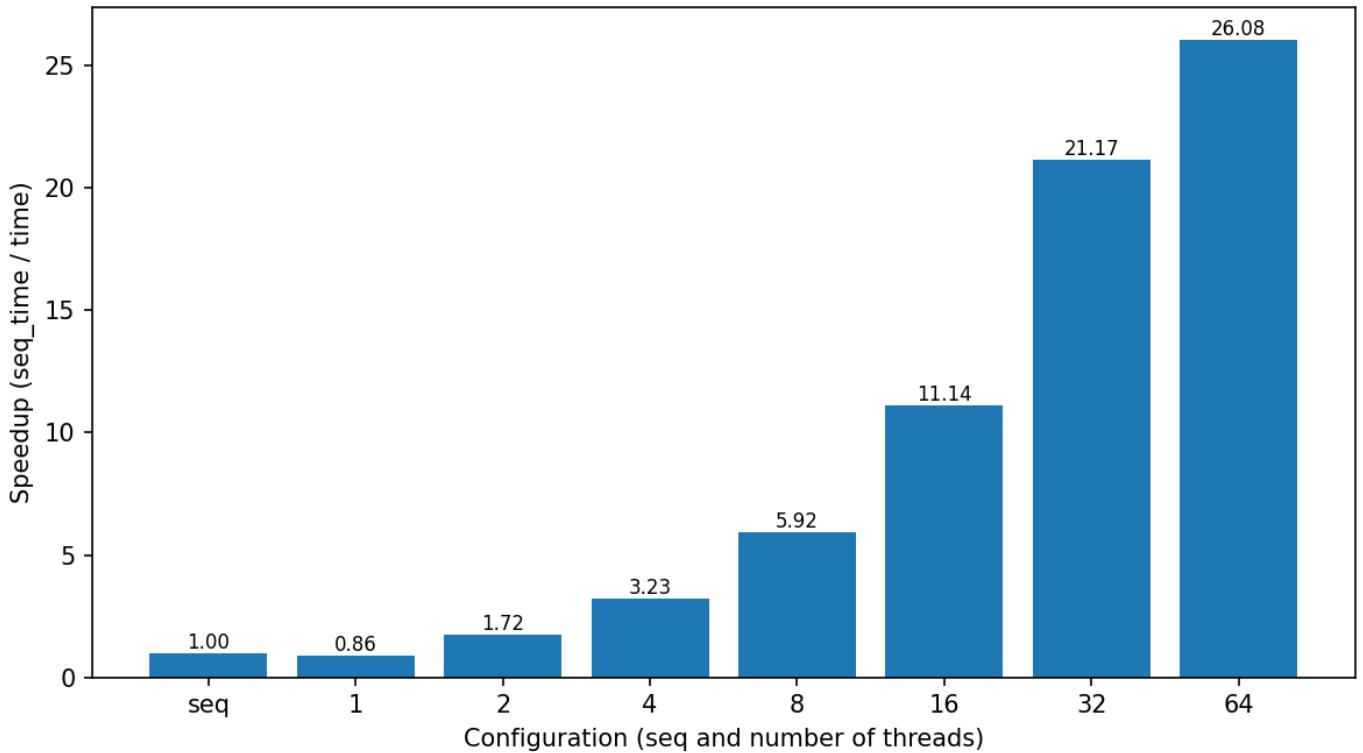
Το παραπάνω παράλληλο πρόγραμμα (omp_reduction_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και με affinity (όπως και πριν). Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

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

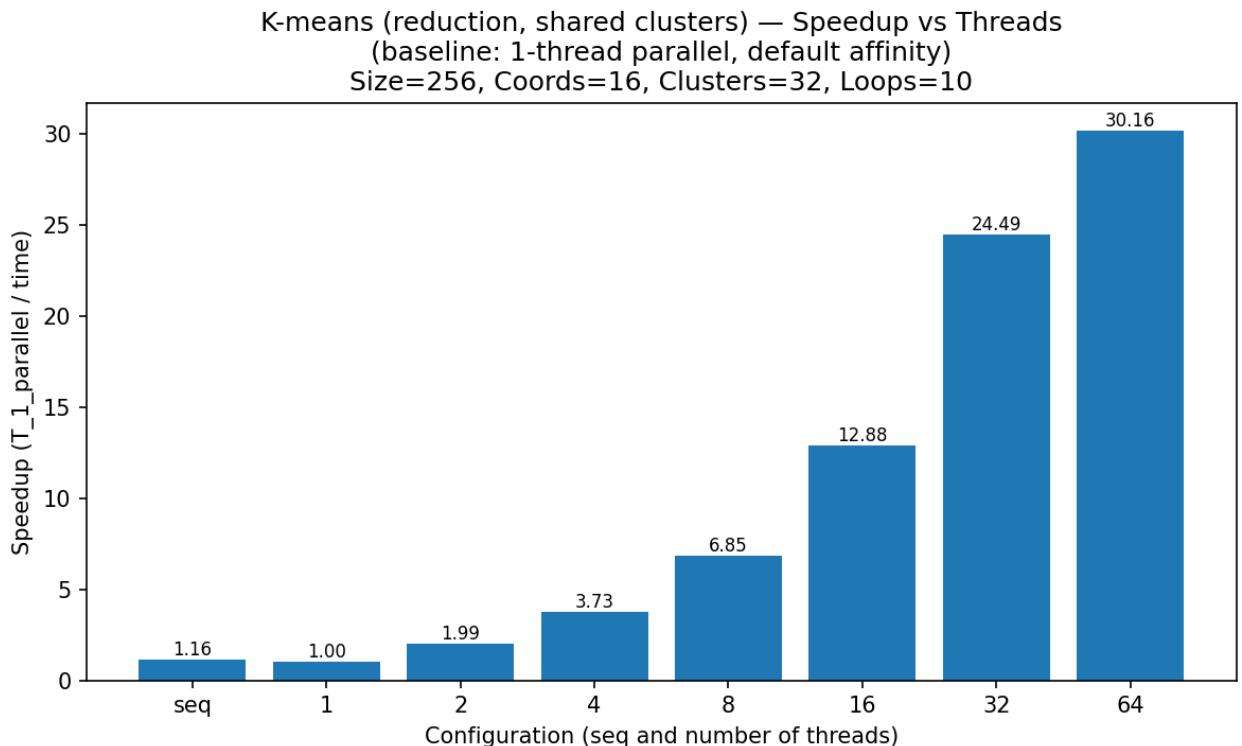


Από τα δύο πρώτα διαγράμματα παρατηρούμε ότι η έκδοση με copied clusters και reduction κλιμακώνει πλέον πολύ ικανοποιητικά σε σχέση με τη naive shared προσέγγιση. Ο σειριακός χρόνος είναι περίπου 12.6s, ενώ η παράλληλη εκτέλεση με 1 νήμα παραμένει λίγο χειρότερη (~14.6s), λόγω του parallel overhead του 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

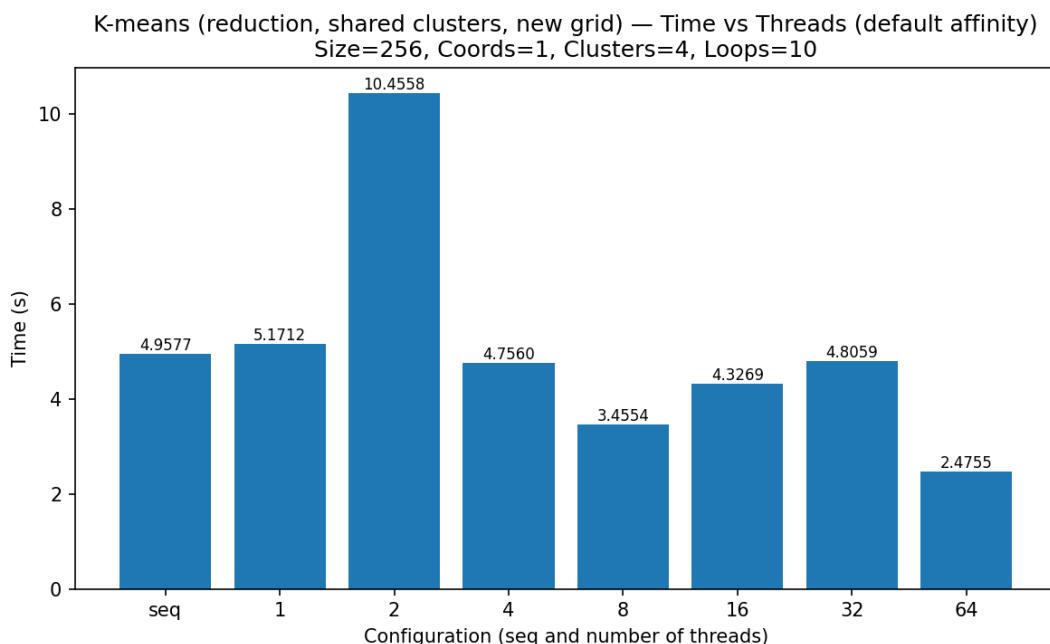
bandwidth και το μικρό σειριακό τμήμα του κώδικα (νόμος του Amdahl), ενώ το synchronization bottleneck έχει ουσιαστικά εξαλειφθεί.

2. Παραλληλοποίηση για το μειωμένο grid size και διαγράμματα

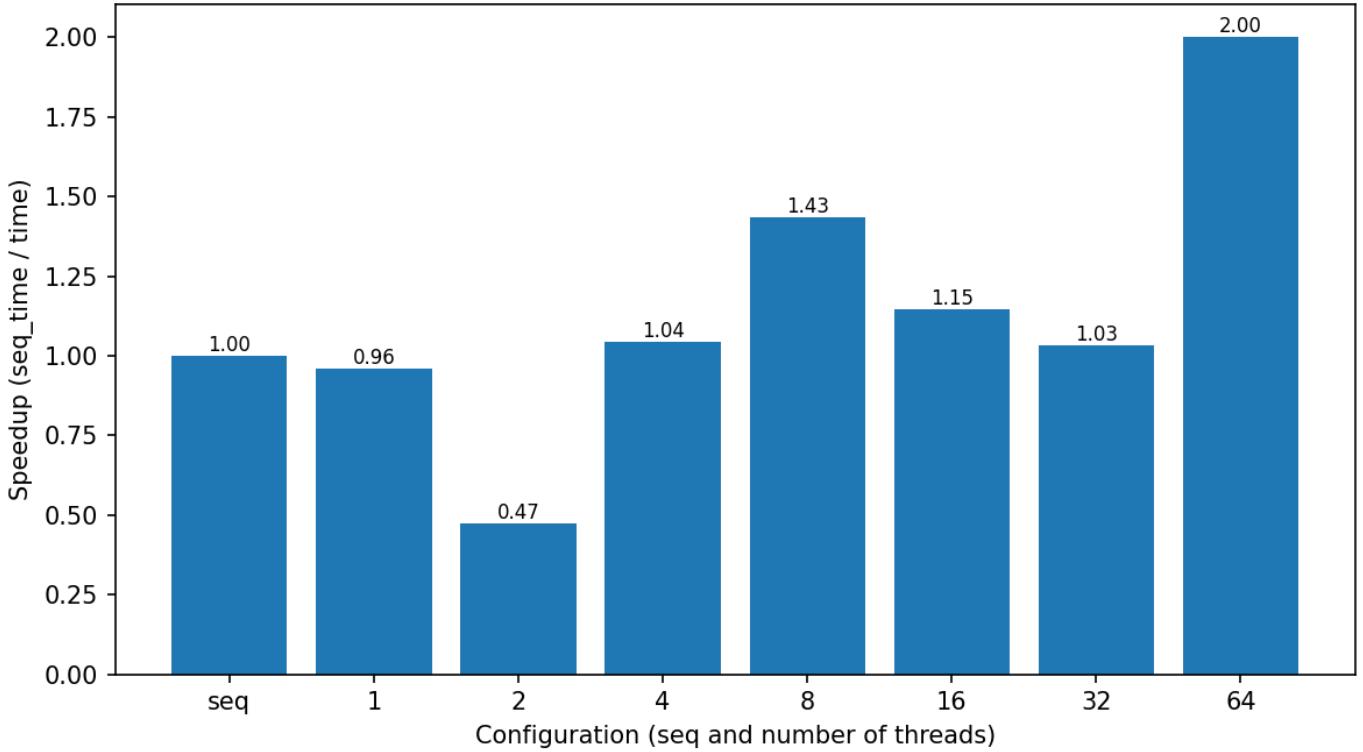
Το παραπάνω παράλληλο πρόγραμμα (omp_reduction_kmeans.c) έτρεξε για τις παραμέτρους: {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}, threads = {1, 2, 4, 8, 16, 32, 64} και με affinity (όπως και πριν). Τα αποτελέσματα που προέκυψαν παρουσιάζονται στον παρακάτω πίνακα:

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

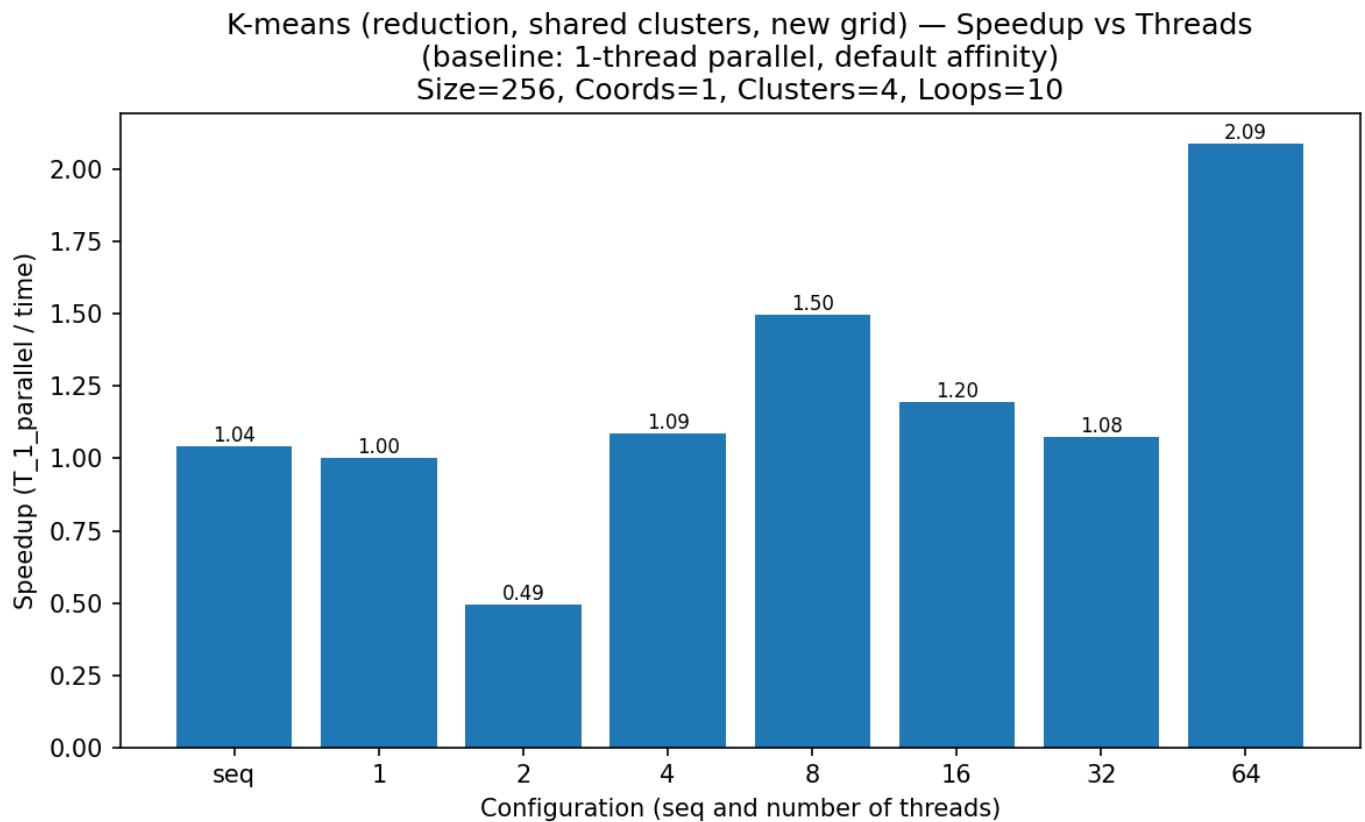


Από τα δύο πρώτα διαγράμματα για το μειωμένο grid παρατηρούμε ότι η συμπεριφορά της έκδοσης με reduction είναι σαφώς χειρότερη από αυτή στο αρχικό, πιο «υπολογιστικά βαρύ» (compute intensive) grid (<{256,16,32,10}). Εδώ ο σειριακός χρόνος είναι περίπου 4.96s και η παράλληλη εκτέλεση με 1 νήμα λίγο χειρότερη (~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     break;
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);
*/
270
```

- **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

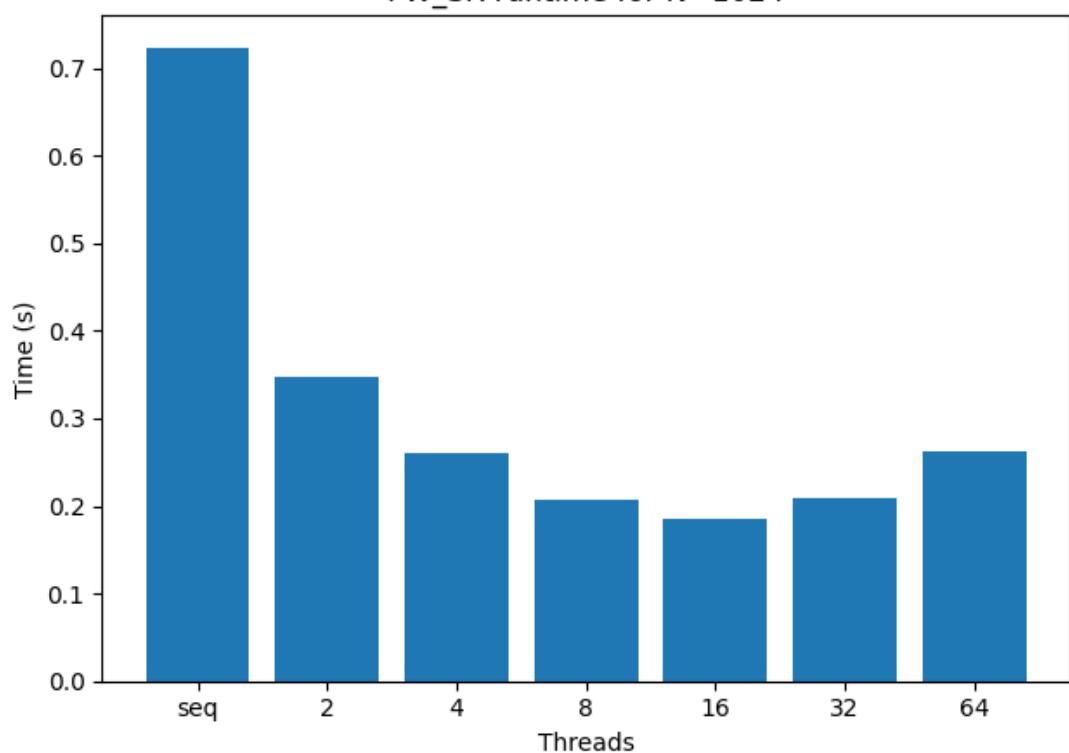
N	Tseq (T=1)	T=2	T=4	T=8	T=16	T=32	T=64
1024	0.7237	0.3467	0.2604	0.2078	0.1850	0.2097	0.2627
2048	5.1359	2.6523	1.9513	1.1305	0.7720	0.7496	0.9577
4096	43.4211	21.7275	14.3879	7.9806	4.4654	3.0423	3.1596

Συντελεστής επιτάχυνσης (speedup)

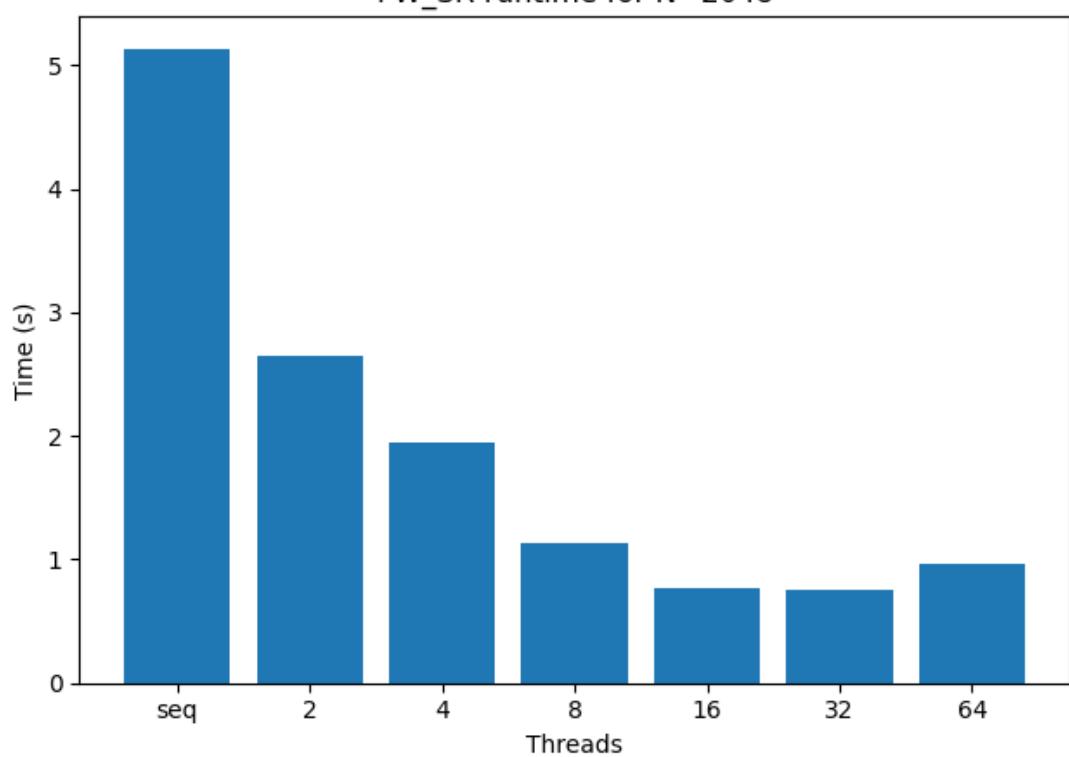
N	T=2	T=4	T=8	T=16	T=32	T=64	Max S
1024	2.09	2.78	3.48	3.91	3.45	2.75	3.91
2048	1.94	2.63	4.54	6.65	6.85	5.36	6.85
4096	2.00	3.02	5.44	9.72	14.27	13.74	14.27

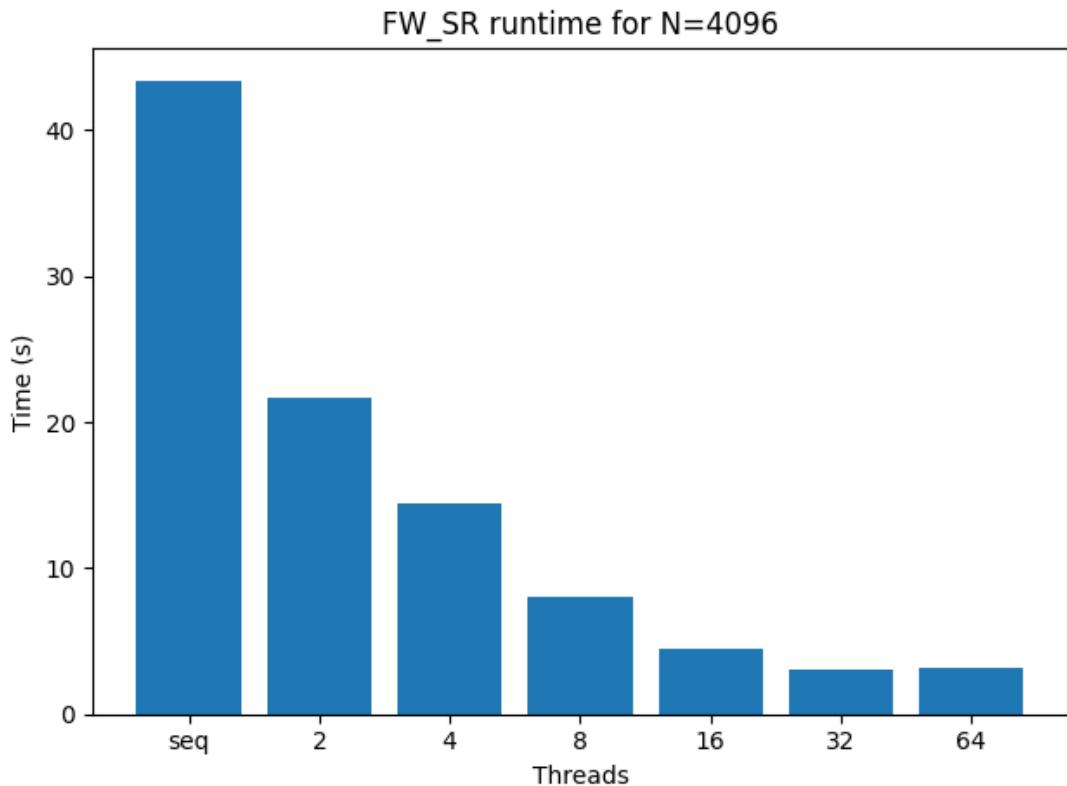
Ακολουθούν και τα barplots για τα runs των διαφορετικών N:

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:

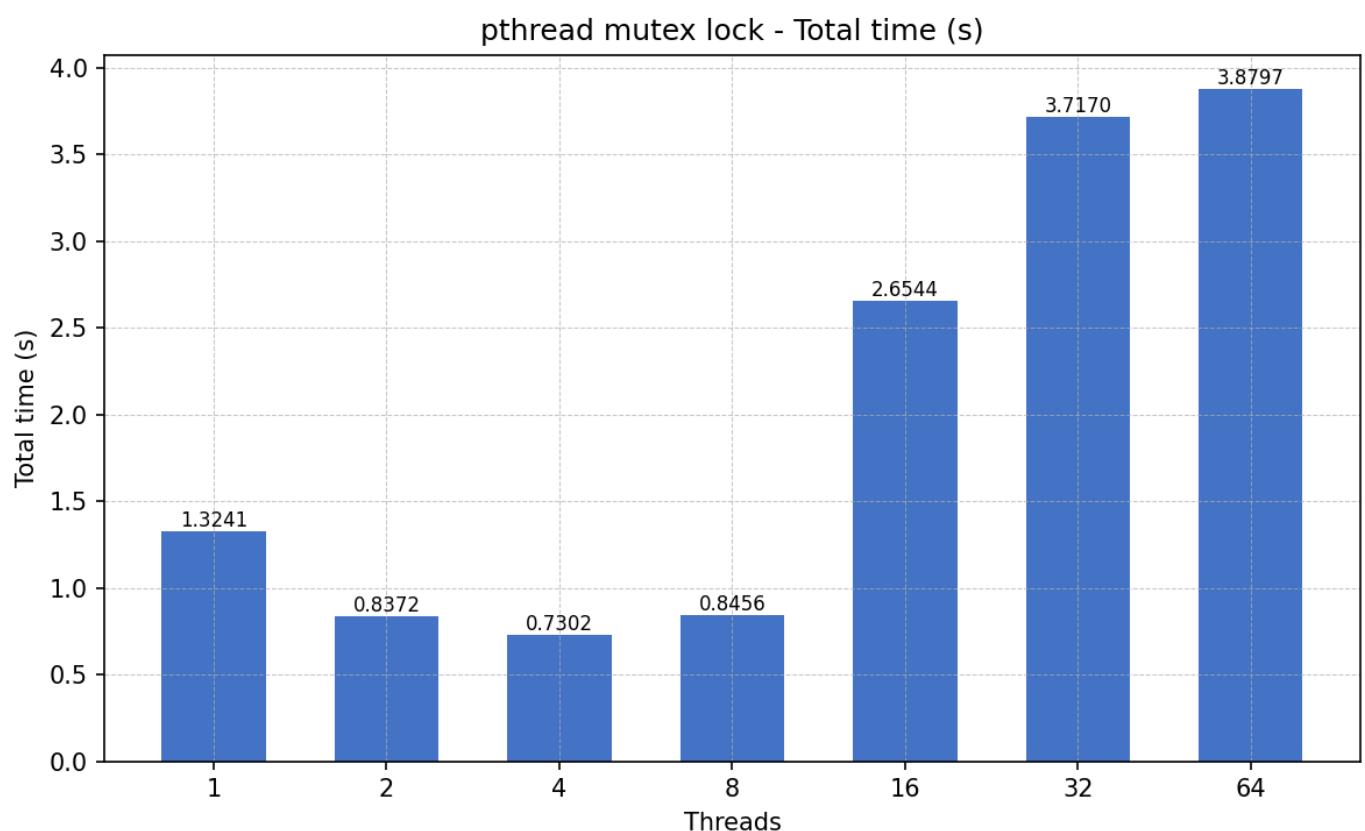
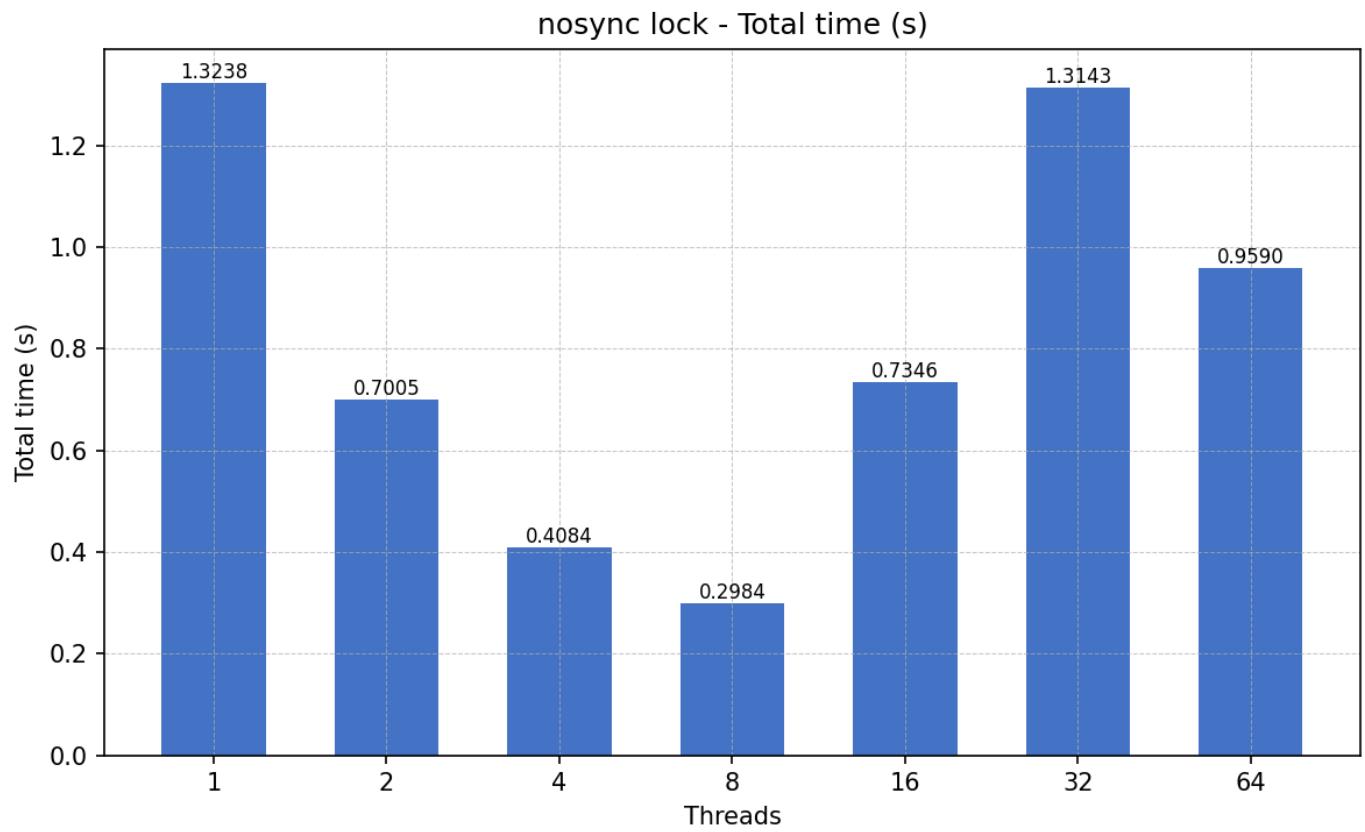
Threads	Total Time
1	1.3182
2	0.7995
4	0.4884
8	0.4319
16	1.2522
32	1.6519
64	1.4726

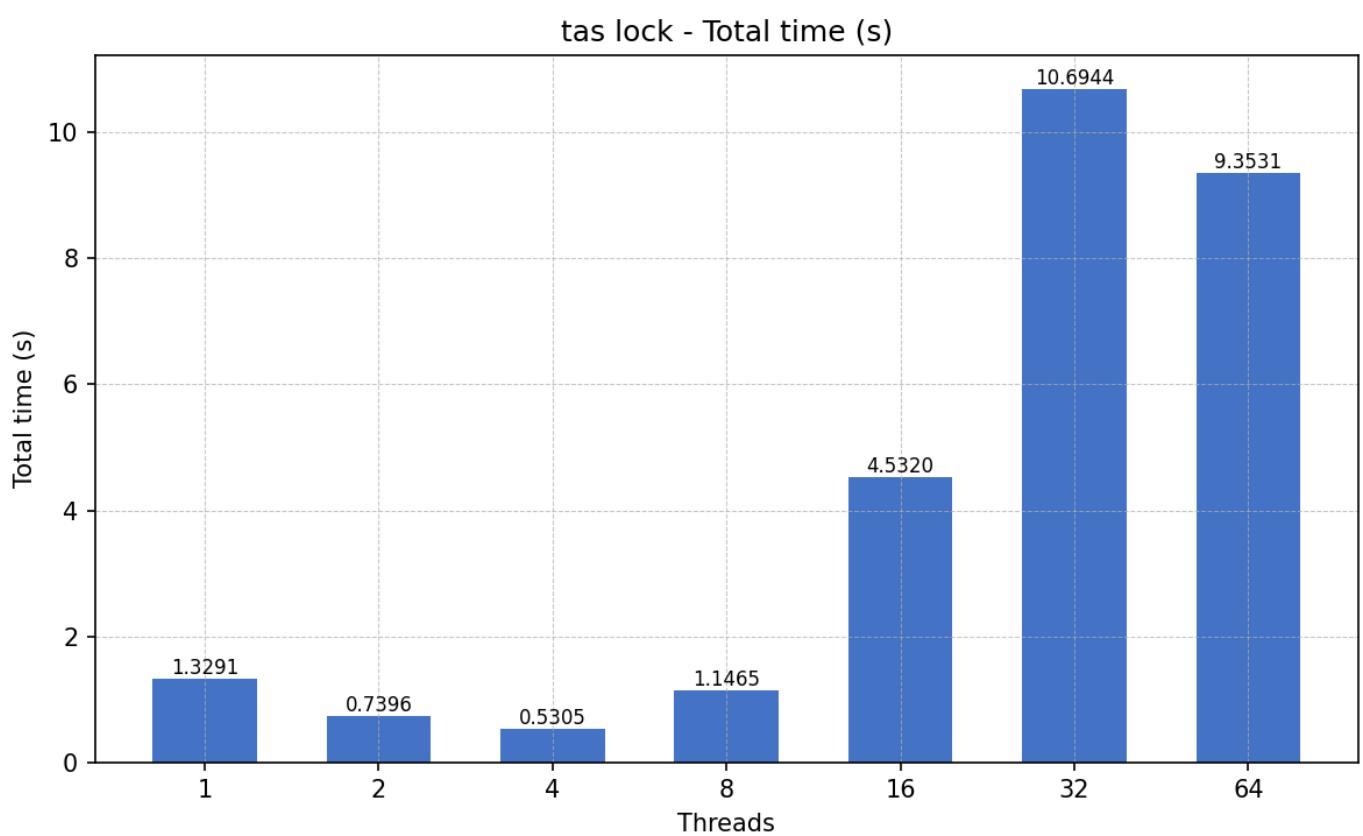
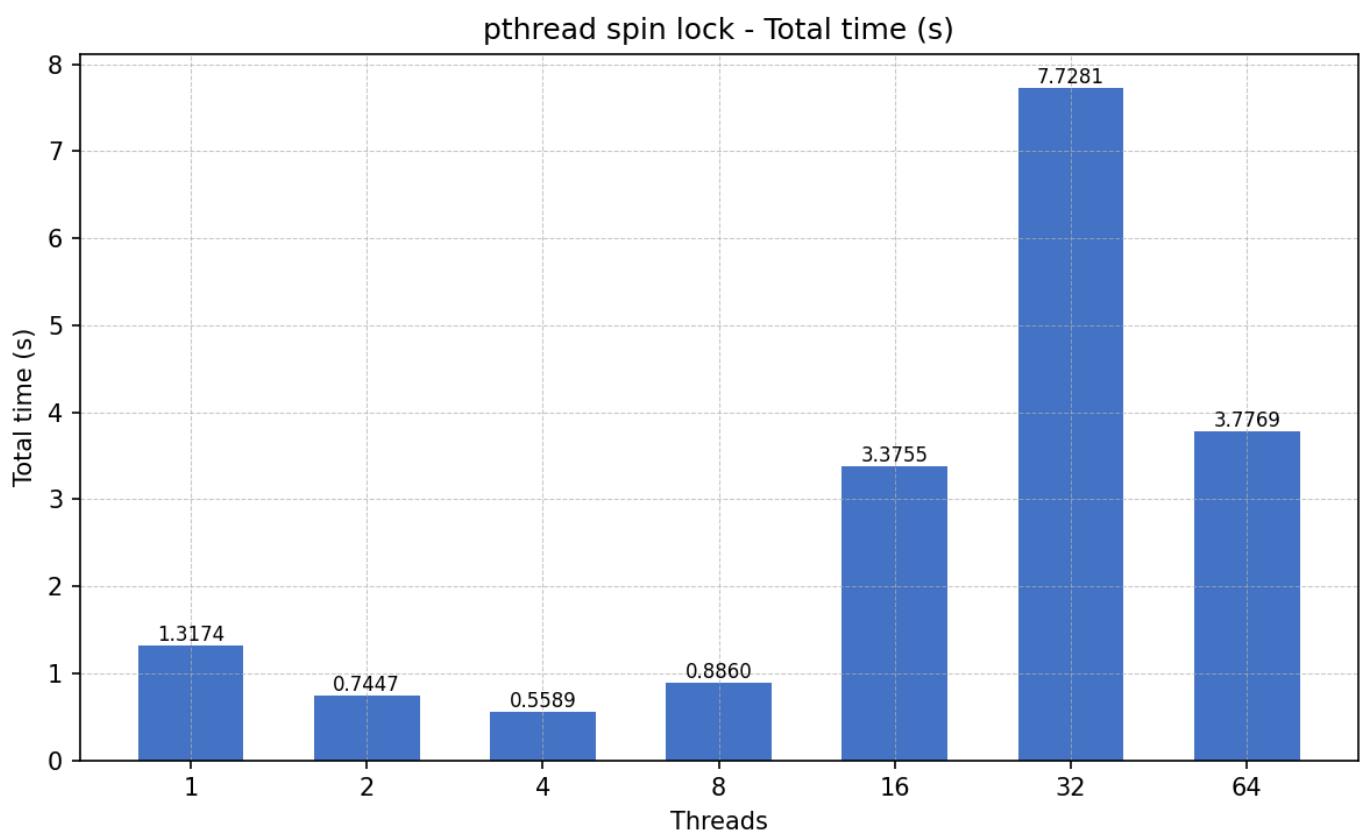
critical:

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

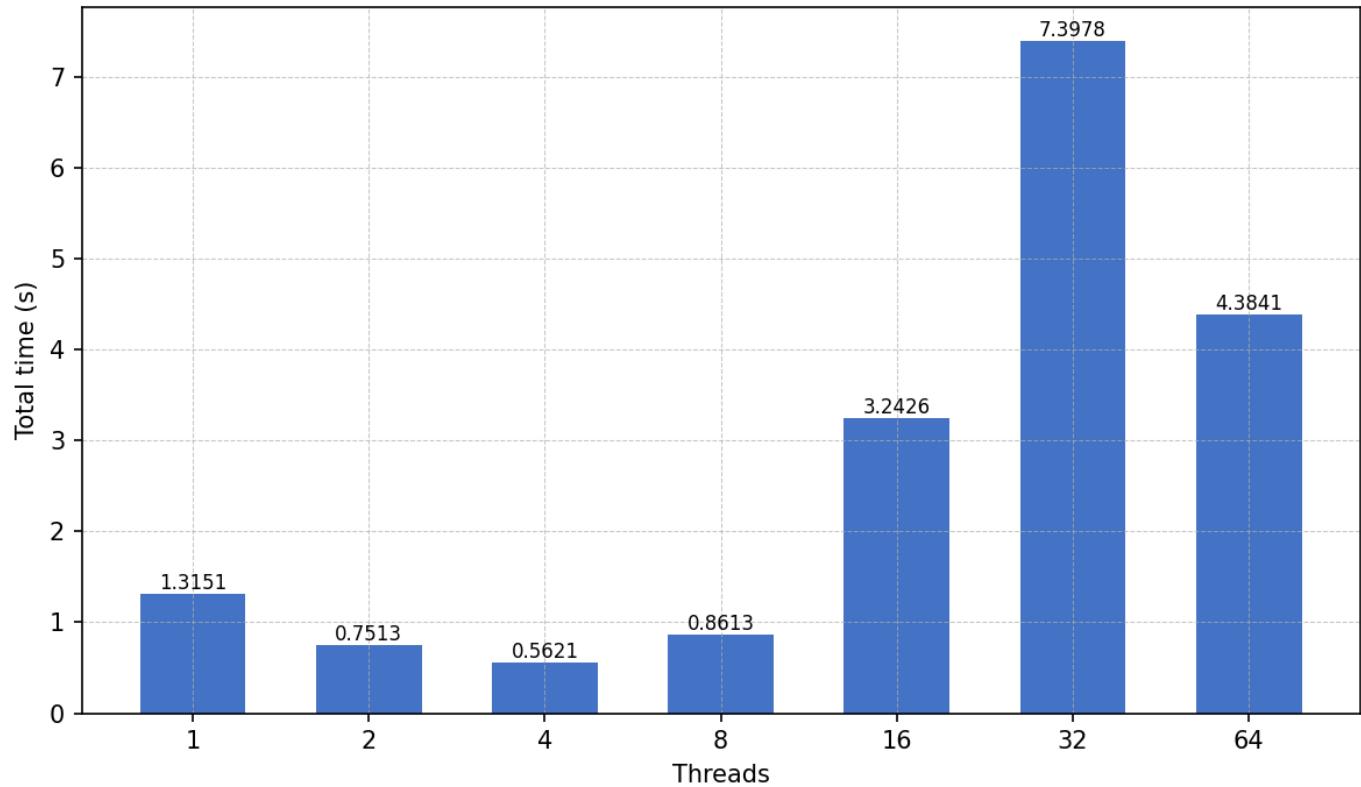
Για να είναι πιο ευδιάκριτες οι διαφορές μεταξύ των locks, παρουσιάζουμε τα αποτελέσματα και σε διαγράμματα. Στα διαγράμματα που ακολουθούν απεικονίζουμε τον χρόνο εκτέλεσης σε συνάρτηση με τον αριθμό νημάτων, για κάθε υλοποίηση κλειδώματος, καθώς και ένα συγκεντρωτικό διάγραμμα για όλους τους τύπους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

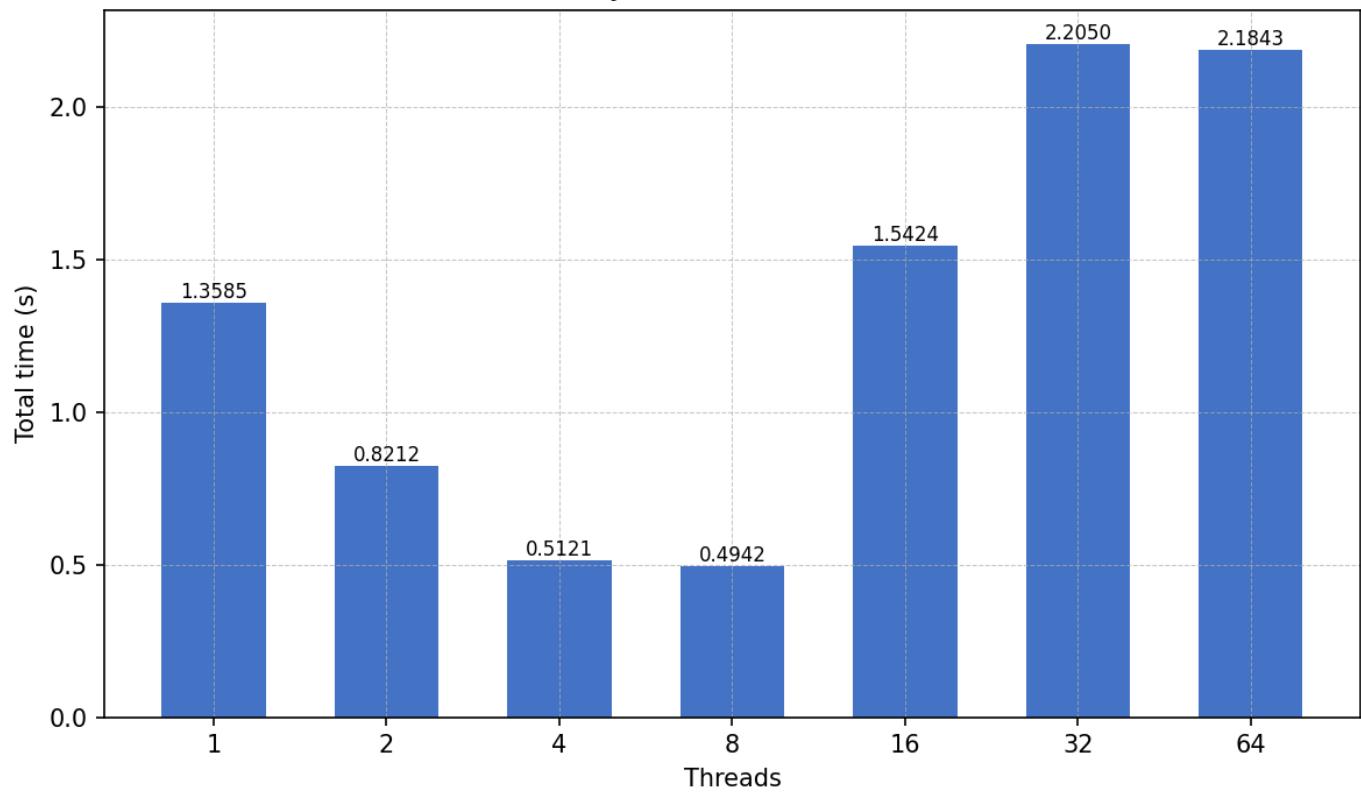


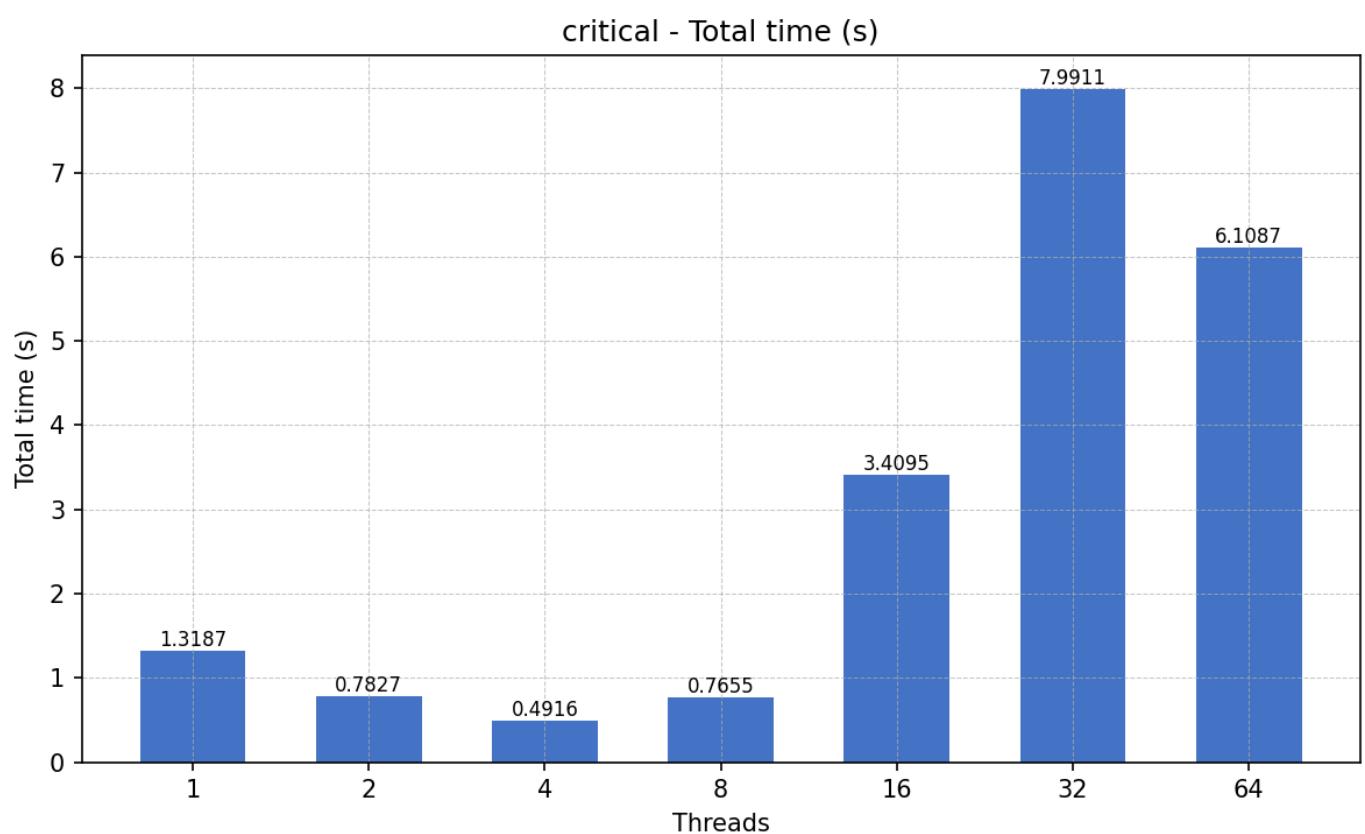
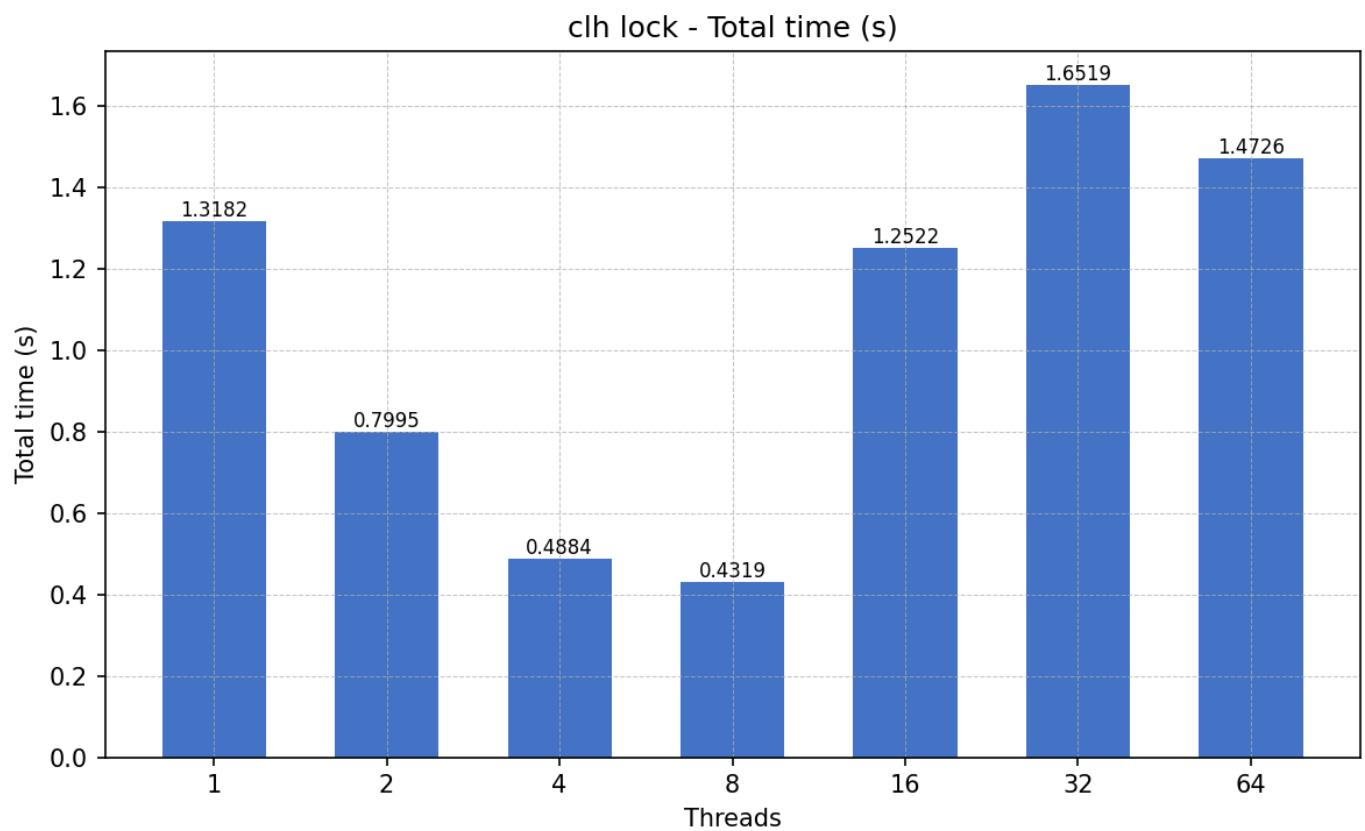


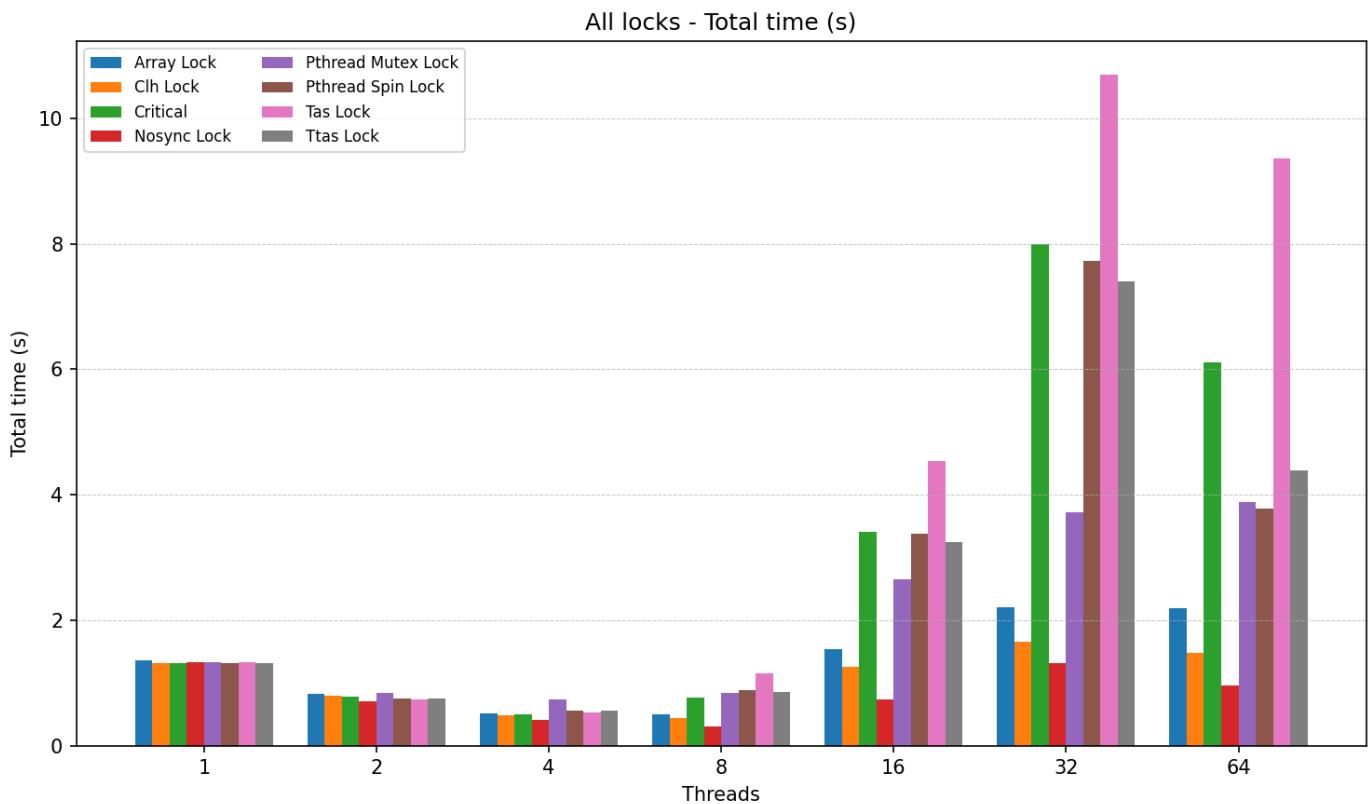
ttas lock - Total time (s)



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

```

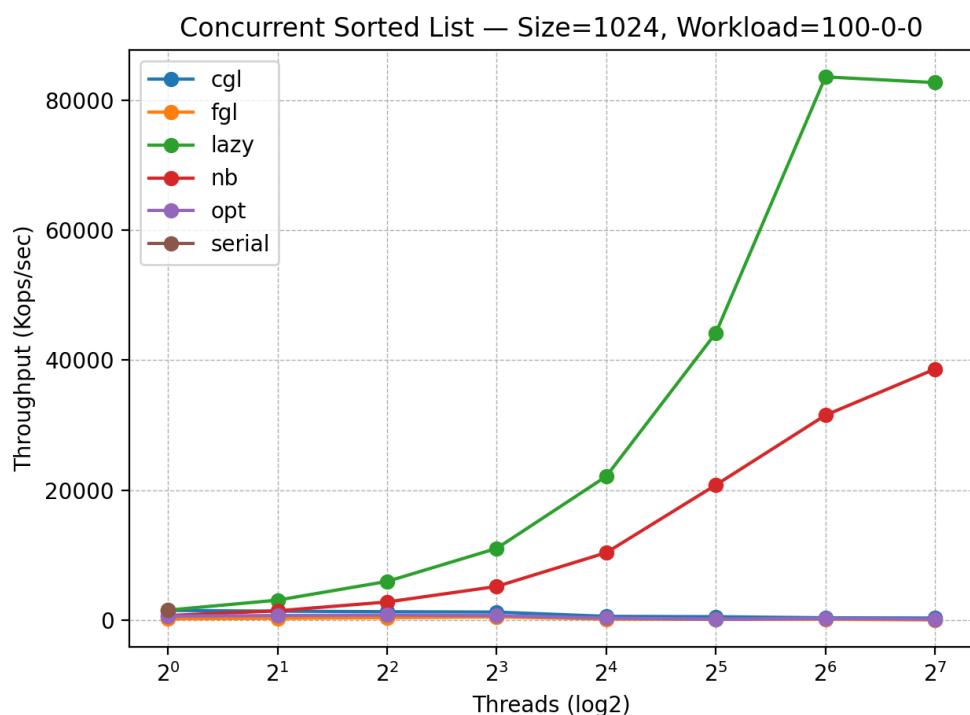
3. Αποτελέσματα και Σχολιασμός

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)

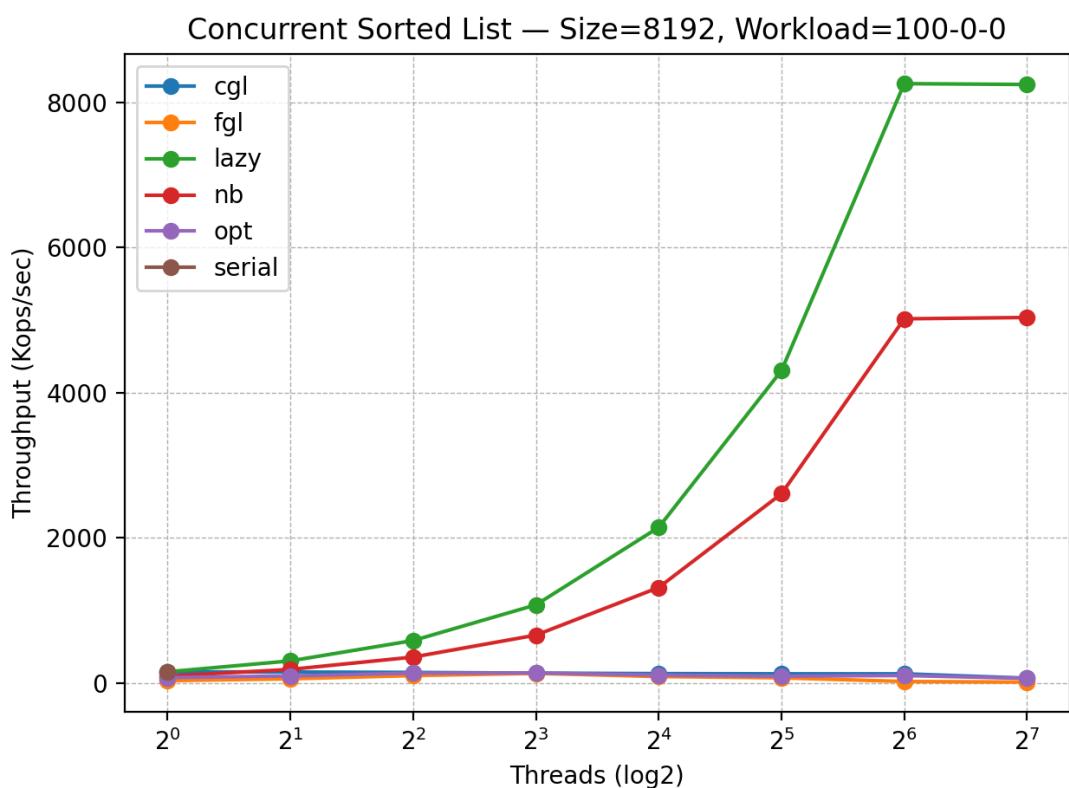
Στο συγκεκριμένο workload όλες οι λειτουργίες είναι contains(), χωρίς καμία εισαγωγή ή διαγραφή. Πρόκειται επομένως για ένα αμιγώς read-only σενάριο, στο οποίο δεν εμφανίζονται συγκρούσεις εγγραφών και η απόδοση εξαρτάται κυρίως από το αν και πόσο blocking απαιτεί η υλοποίηση για τις αναζητήσεις και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



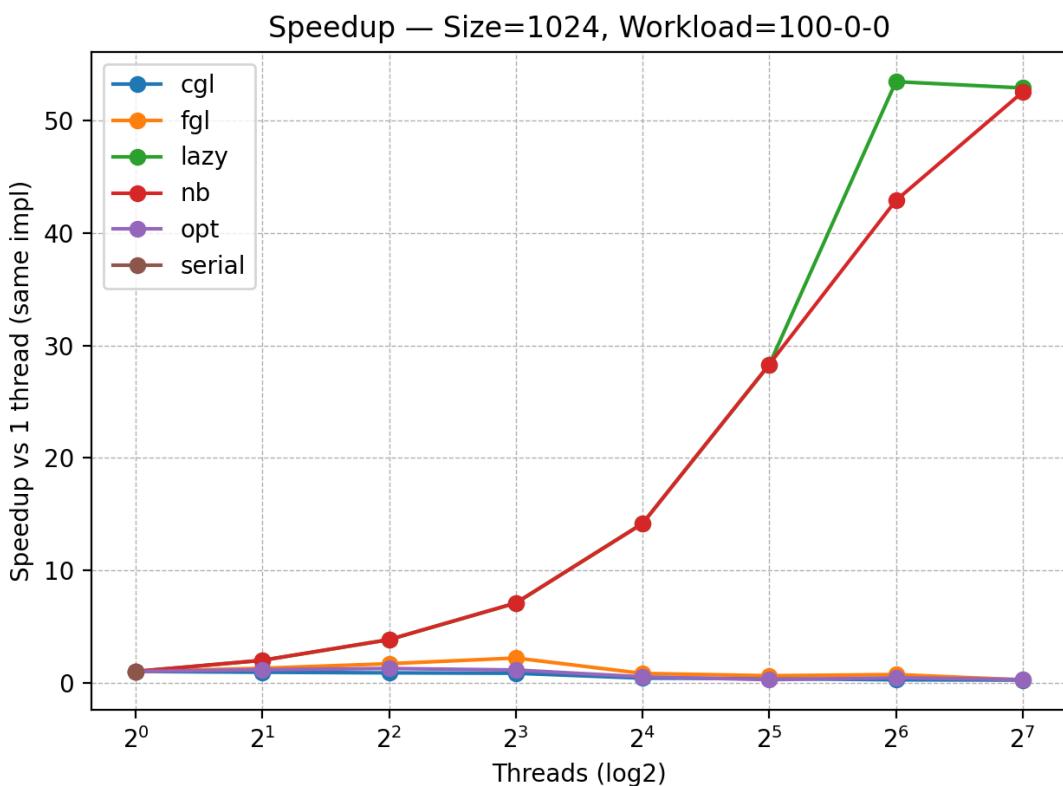
Για μεγαλύτερο μέγεθος λίστας, η ποιοτική εικόνα παραμένει ίδια, αλλά όλες οι υλοποιήσεις παρουσιάζουν σημαντικά χαμηλότερο throughput. Η αύξηση του

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

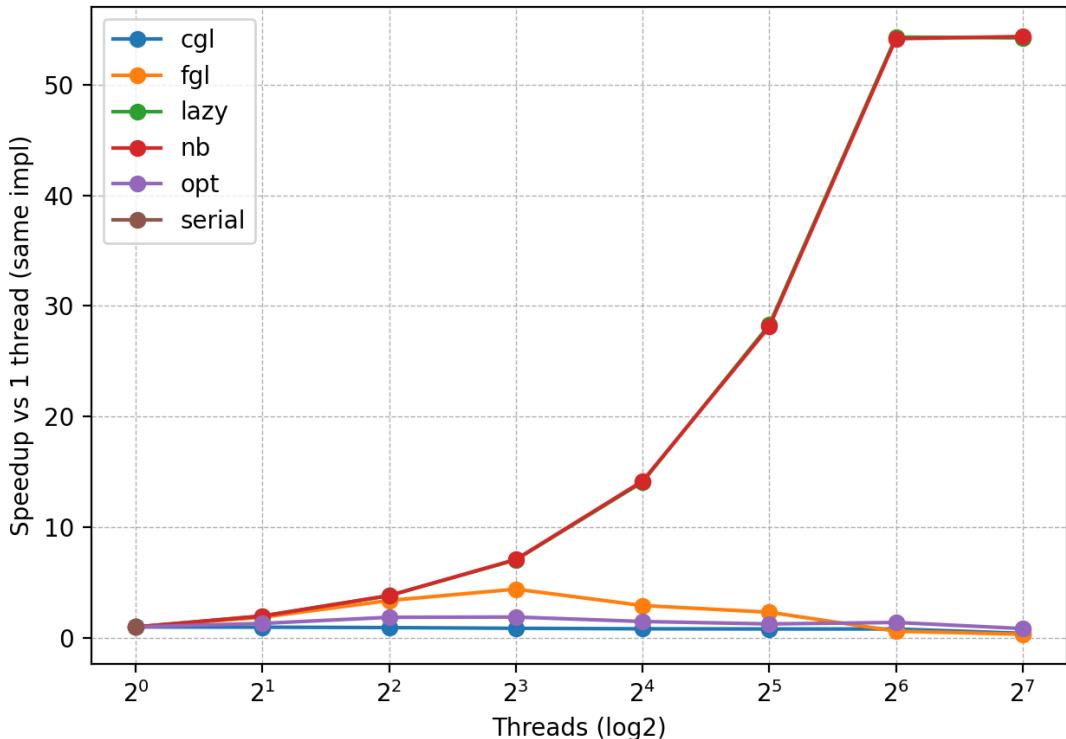
Παρόλα αυτά, οι lazy και non-blocking υλοποιήσεις εξακολουθούν να υπερέχουν σημαντικά έναντι των locking-based προσεγγίσεων, επιβεβαιώνοντας ότι η αποφυγή blocking είναι καθοριστικός παράγοντας απόδοσης σε read-only workloads.

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:

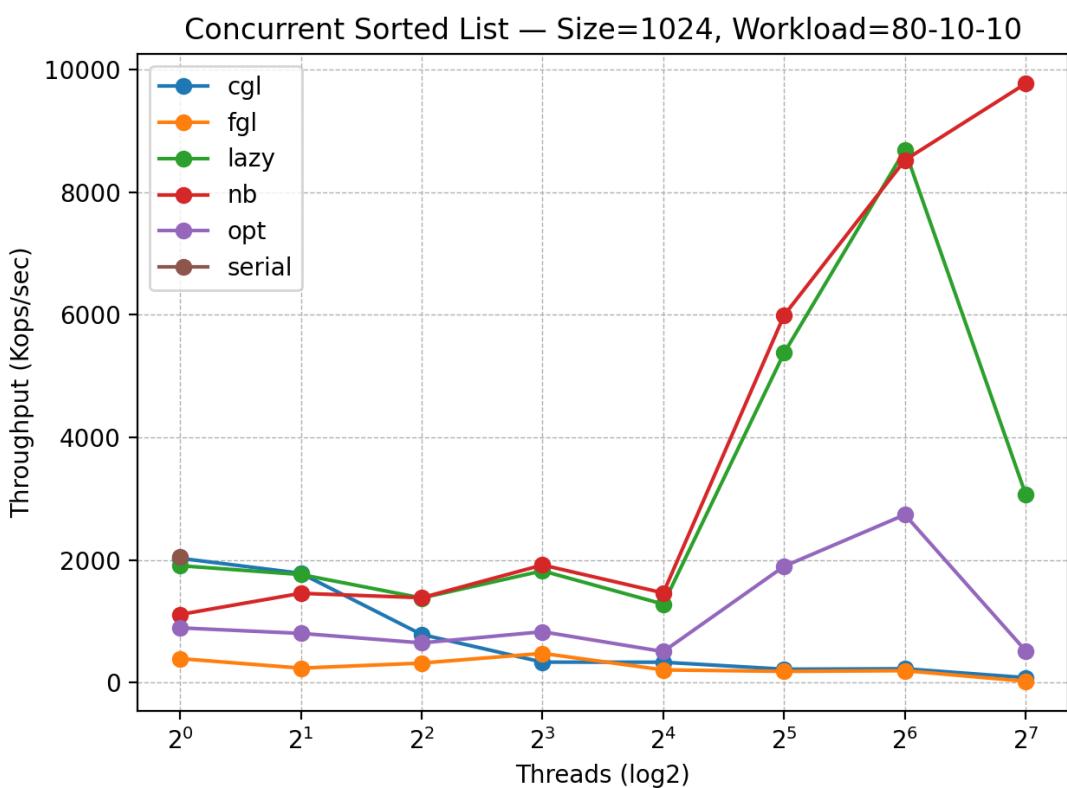
- Οι lazy synchronization και non-blocking υλοποιήσεις είναι οι πλέον κατάλληλες, καθώς οι αναζητήσεις εκτελούνται με ελάχιστο ή μηδενικό blocking.
- Η coarse-grain locking υλοποίηση δεν κλιμακώνεται, λόγω της πλήρους σειριοποίησης των λειτουργιών.
- Οι fine-grain και optimistic υλοποιήσεις παρουσιάζουν 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

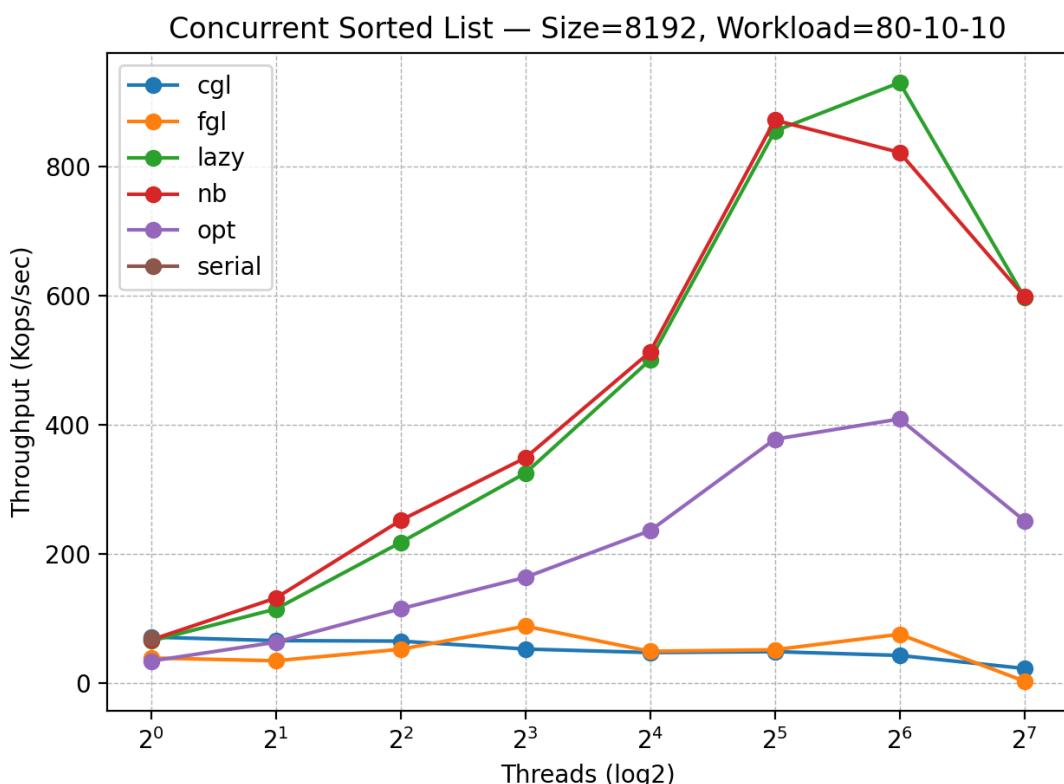


Για μικρό μέγεθος λίστας, η `lazy synchronization` εμφανίζει την καλύτερη συνολική απόδοση. Το throughput αυξάνεται σταθερά μέχρι τα 32–64 νήματα και στη συνέχεια παρουσιάζει ελαφρά σταθεροποίηση. Η καλή αυτή συμπεριφορά

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.

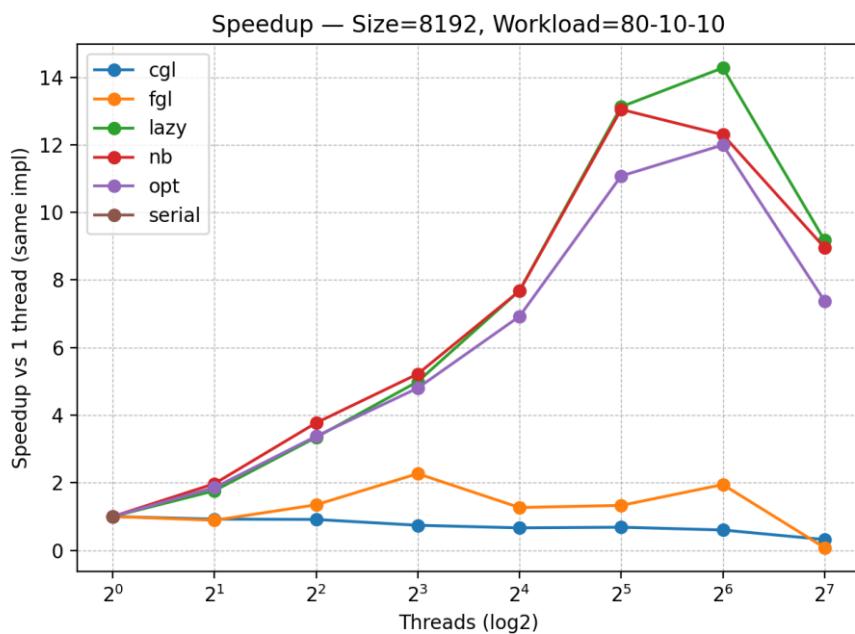
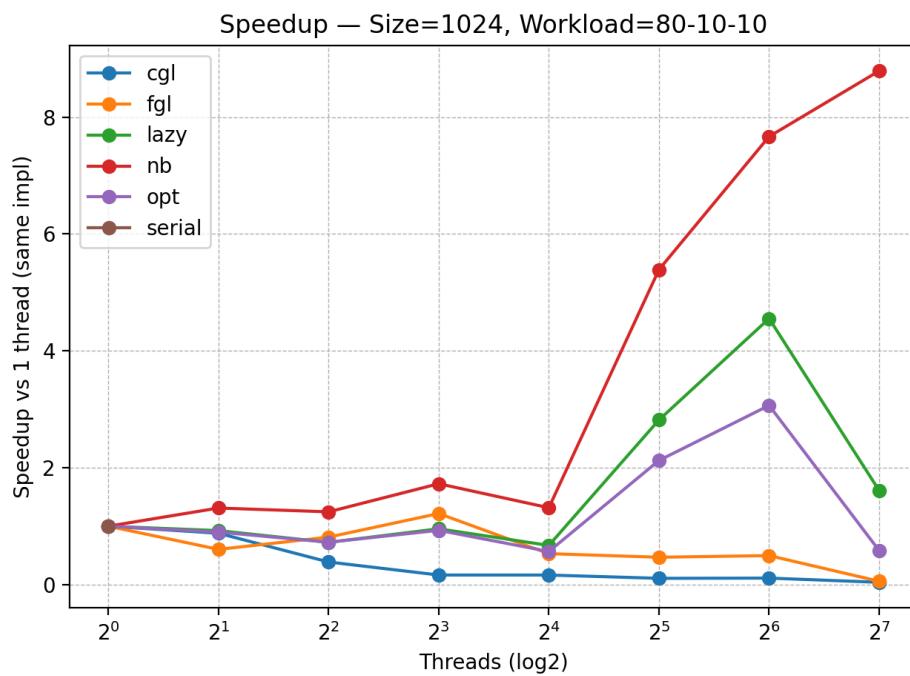
Οι υλοποιήσεις fine-grain και coarse-grain locking εμφανίζουν σαφώς χαμηλότερο throughput. Στην coarse-grain, όλες οι λειτουργίες σειριαποιούνται μέσω ενός καθολικού lock, ενώ στη fine-grain το κόστος των πολλαπλών lock/unlock γίνεται αισθητό ακόμη και με σχετικά περιορισμένο αριθμό ενημερώσεων.



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

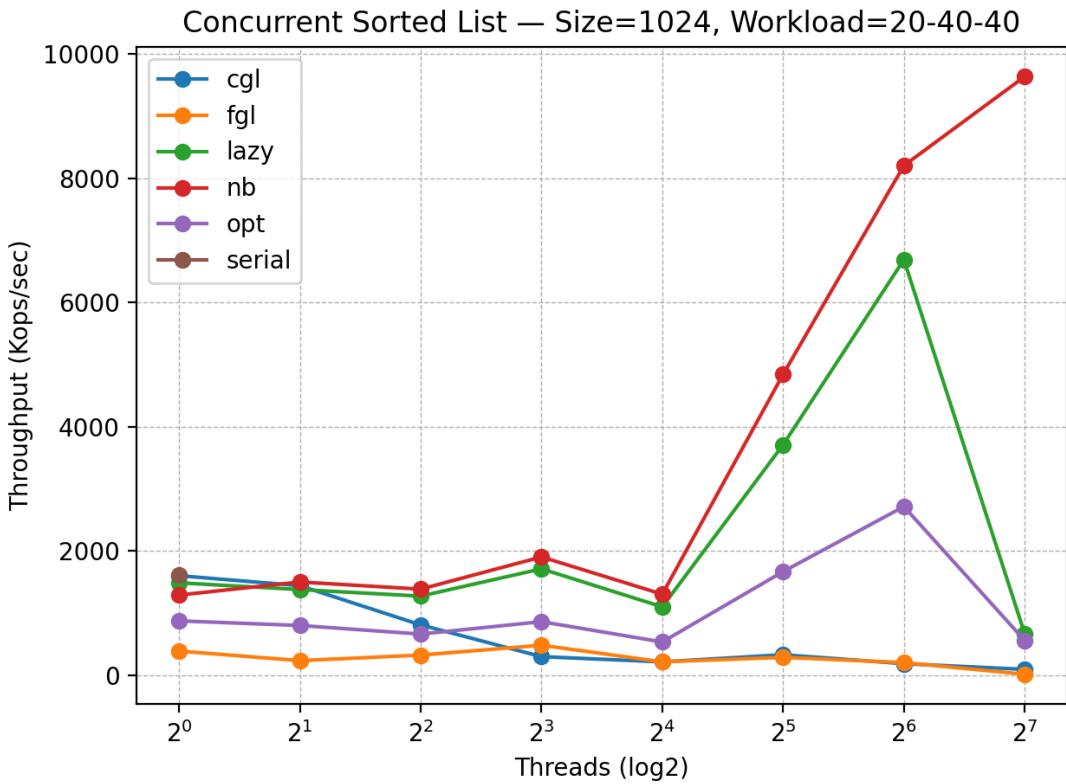
Από την ανάλυση του workload 80-10-10 προκύπτουν τα εξής:

- 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.

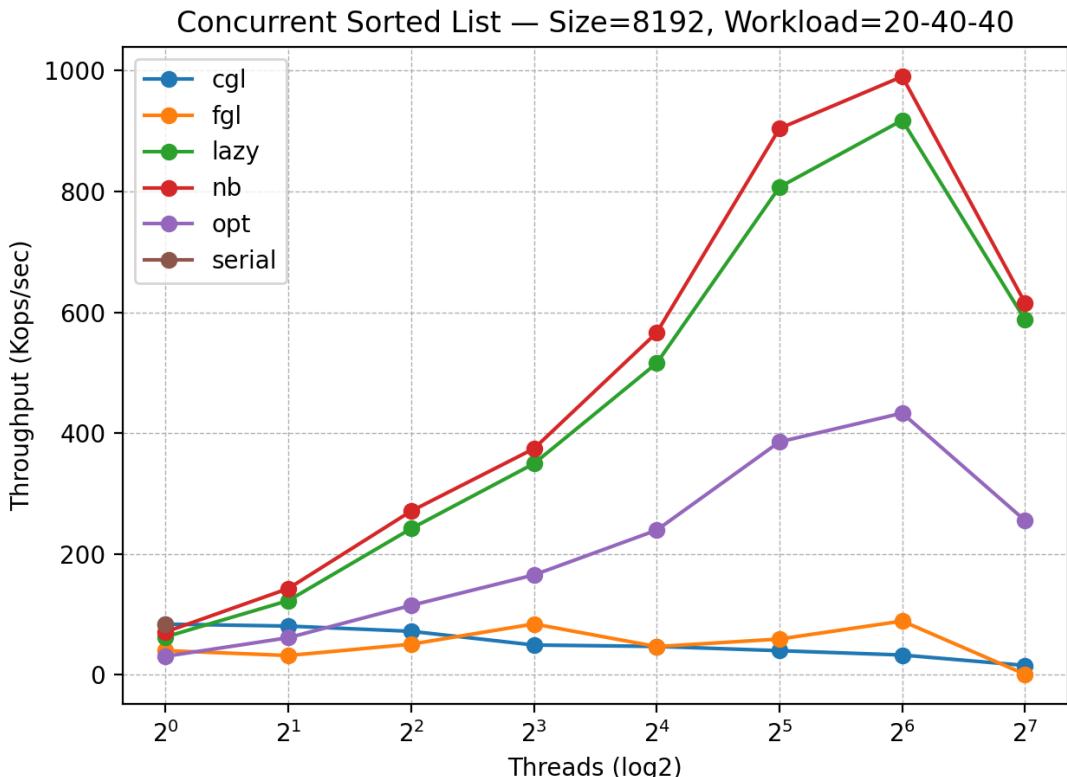


Για μικρό μέγεθος λίστας, η non-blocking (lock-free) υλοποίηση επιτυγχάνει το υψηλότερο throughput, παρουσιάζοντας σαφή υπεροχή σε μεγάλο εύρος αριθμού νημάτων. Το throughput αυξάνεται έντονα έως τα 64 νήματα, γεγονός που υποδηλώνει ότι η αποφυγή locks επιτρέπει μεγαλύτερο βαθμό ταυτόχρονης προόδου ακόμη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

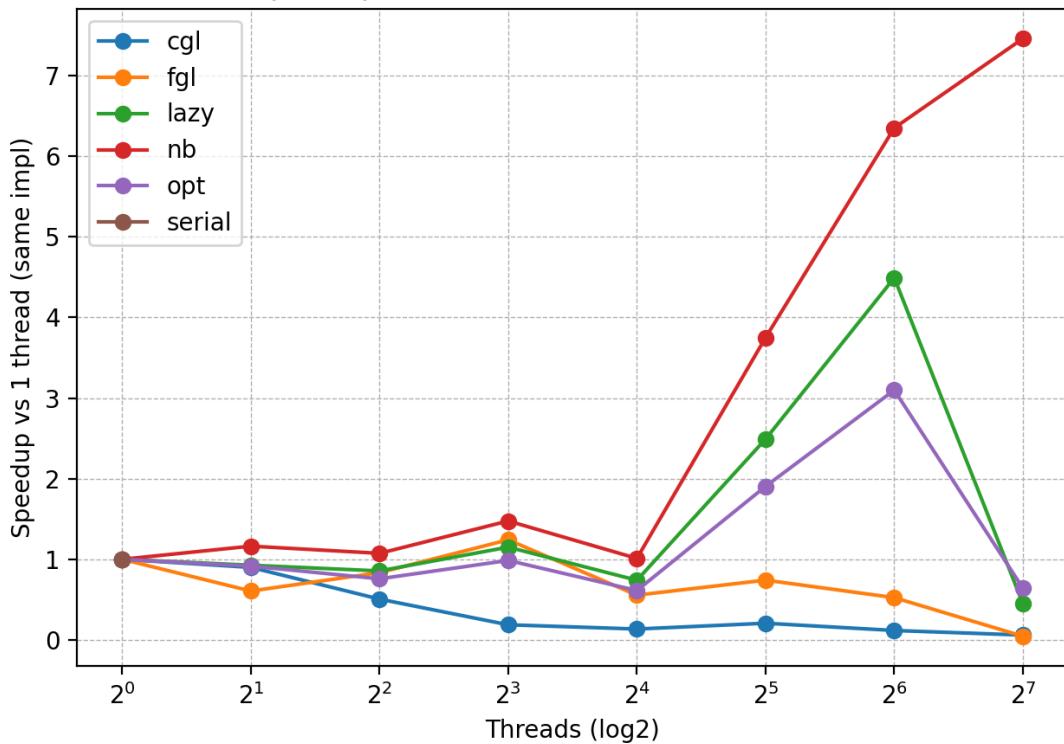
Η non-blocking υλοποίηση παραμένει η καλύτερη σε απόλυτους όρους, φτάνοντας σε μέγιστο throughput γύρω στα 32–64 νήματα, πριν εμφανίσει πτώση στα 128 νήματα. Η πτώση αυτή αποδίδεται στον συνδυασμό αυξημένου CAS contention και hyperthreading/oversubscription.

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

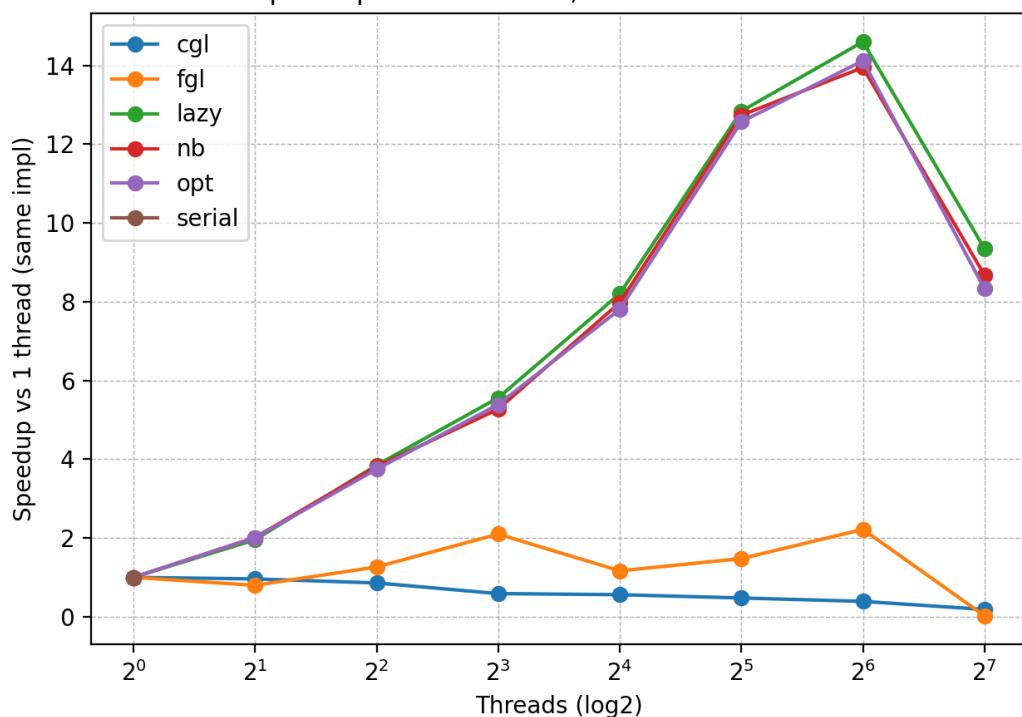
Η ανάλυση του speedup αποκαλύπτει σημαντικές διαφορές στη δυνατότητα κλιμάκωσης

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.

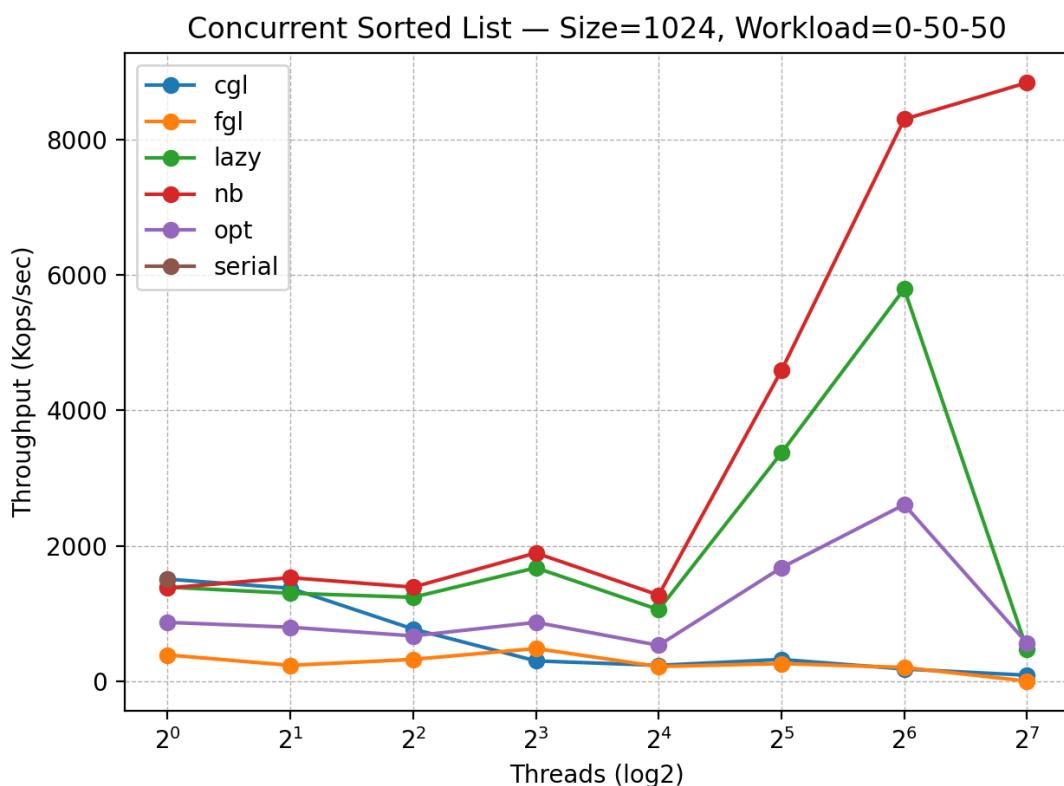
Η πτώση της απόδοσης στα 128 νήματα οφείλεται κυρίως σε αρχιτεκτονικούς περιορισμούς (hyperthreading και oversubscription), σε συνδυασμό με αυξημένο contention σε atomic ή locking μηχανισμούς.

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

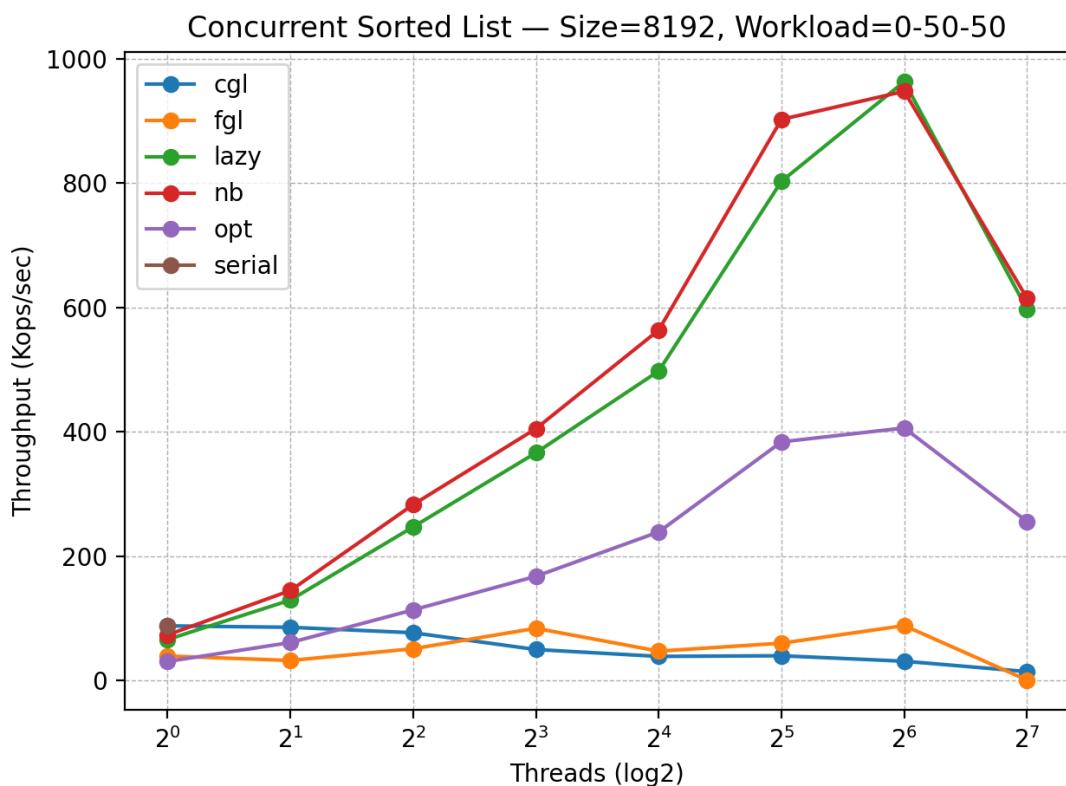


Για μικρό μέγεθος λίστας, η non-blocking (lock-free) υλοποίηση επιτυγχάνει το υψηλότερο throughput σε όλο το εύρος των νημάτων. Το throughput αυξάνεται σημαντικά έως τα 64 νήματα, όπου και παρατηρείται η μέγιστη απόδοση. Η απουσία locks επιτρέπει στα νήματα να προοδεύουν χωρίς να μπλοκάρουν μεταξύ τους,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



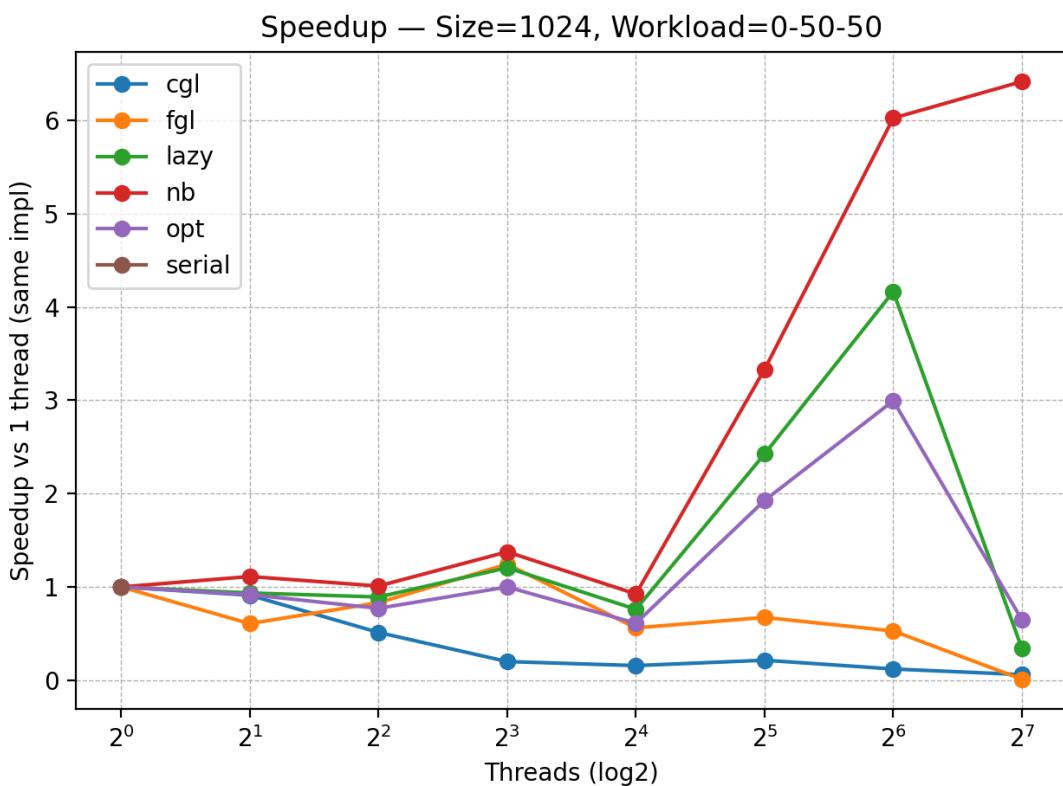
Με μεγαλύτερη λίστα, το throughput μειώνεται για όλες τις υλοποιήσεις, λόγω αυξημένου κόστους διάσχισης και μεγαλύτερης πιθανότητας συγκρούσεων.

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:

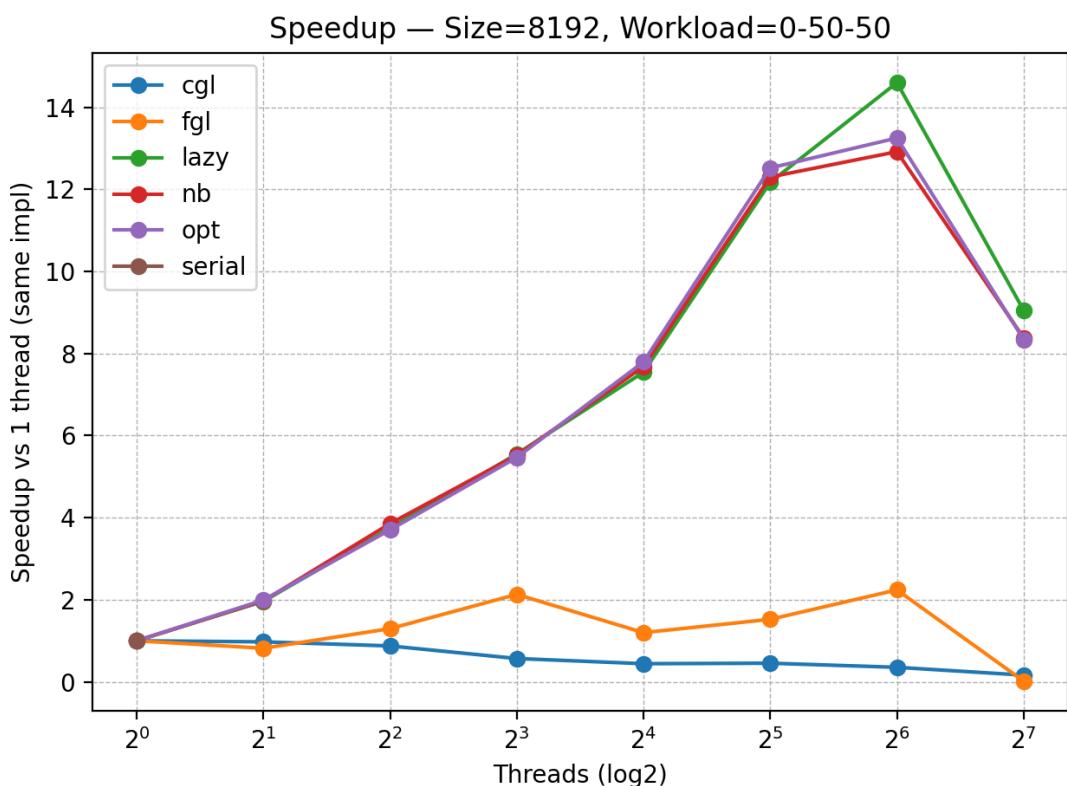


Για $S = 1024$, η non-blocking υλοποίηση παρουσιάζει τη μεγαλύτερη επιτάχυνση, ξεπερνώντας όλες τις άλλες υλοποιήσεις σε όλο το εύρος των νημάτων.

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**

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 4^{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: 18.01.2026

▪ Εισαγωγή

Σκοπός της άσκησης είναι η παραλληλοποίηση και η βελτιστοποίηση του αλγορίθμου K-means σε επεξεργαστές γραφικών NVIDIA, μέσω CUDA. Η υλοποίηση βασίζεται στο μοντέλο CPU-GPU: η CPU (host) προετοιμάζει τα δεδομένα, εκκινεί τα kernels στην GPU (device) και (ανάλογα το πρόγραμμα) εκτελεί μέρος του αλγορίθμου και διαχειρίζεται μεταφορές μνήμης. Η άσκηση συγκρίνει 4 διαδοχικές εκδόσεις του αλγορίθμου, οι οποίες στοχεύουν στην ανάδειξη και αξιολόγηση κλασικών παραγόντων επίδοσης GPU: προσπελάσεις global memory/coalescing, αξιοποίηση shared memory, κόστος atomic operations και overhead επικοινωνίας host-device.

Σύμφωνα με τα ζητούμενα της άσκησης, δουλέψαμε και παραγάγαμε 4 εκδόσεις του αλγορίθμου K-means:

1. Naive (cuda_kmeans_naive.cu): η GPU υπολογίζει μόνο την ανάθεση στο κοντινότερο cluster ανά αντικείμενο. Η ενημέρωση των κέντρων (update_centroids) γίνεται στην CPU, με μεταφορές host \leftrightarrow device ανά επανάληψη.
2. Transpose (cuda_kmeans_transpose.cu): αναδιάταξη δεδομένων σε column-major transpose μορφή για βελτιωμένο memory coalescing στις προσπελάσεις της GPU (κοντινά threads διαβάζουν γειτονικές διευθύνσεις).
3. Shared (cuda_kmeans_shared.cu): επιπλέον φόρτωση των cluster centers στη shared memory ανά block, ώστε οι επαναλαμβανόμενες αναγνώσεις των clusters κατά τον υπολογισμό αποστάσεων να γίνονται από ταχύτερη on-chip μνήμη.
4. All-GPU (cuda_kmeans_all_gpu.cu): πλήρες offload και του update_centroids στη GPU. Η συσσώρευση (sums/counts) υλοποιείται με atomics, αναδεικνύοντας το κόστος συγχρονισμού/contention ως πιθανό bottleneck.

Οι παραπάνω αυτές εκδοχές είναι αυτές που θα αναλύσουμε και θα συγκρίνουμε στη συνέχεια.

▪ Ενότητα 3.1 – Naive Version

A. Εισαγωγή

Η «naive» έκδοση μεταφέρει στη GPU μόνο το πιο υπολογιστικά βαρύ βήμα του K-means: την ανάθεση κάθε αντικειμένου στο κοντινότερο κέντρο ενός cluster. Η ενημέρωση των κέντρων (update_centroids: αθροίσματα/πλήθη/μέσοι όροι) παραμένει στην CPU. Έτσι, σε κάθε επανάληψη εκτελούνται:

- Host → Device: αντιγραφή των τρεχόντων cluster centers στη GPU,
- GPU kernel: υπολογισμός nearest cluster για κάθε object και ενημέρωση membership/delta,
- Device → Host: αντιγραφή membership και delta πίσω στην CPU,
- CPU: update_centroids, παραγωγή νέων cluster centers για το επόμενο loop.

Η αρχική αντιγραφή του συνόλου των objects (dataset) προς τη GPU γίνεται μία φορά πριν το while-loop (initialization) και δεν αποτελεί μέρος του «per-loop» breakdown.

Ο κώδικας του αρχείου μας (cuda_kmeans_naive.cu) παρατίθεται ακολούθως:

a5/cuda_kmeans_naive.cu

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "kmeans.h"
5 #include "alloc.h"
6 #include "error.h"
7
8 #ifdef __CUDACC__
9 inline void checkCuda(cudaError_t e) {
10     if (e != cudaSuccess) {
11         // cudaGetErrorString() isn't always very helpful. Look up the error
12         // number in the cudaError enum in driver_types.h in the CUDA includes
13         // directory for a better explanation.
14         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
15     }
16 }
17
18 inline void checkLastCudaError() {
19     checkCuda(cudaGetLastError());
20 }
21 #endif
22
23 __device__ int get_tid() {
24     return blockIdx.x * blockDim.x + threadIdx.x;
25 }
26
27 /* square of Euclid distance between two multi-dimensional points */
28 __host__ __device__ inline static
29 double euclid_dist_2(int numCoords,
30                      int numObjs,
31                      int numClusters,
32                      double *objects,      // [numObjs][numCoords]
33                      double *clusters,     // [numClusters][numCoords]
34                      int objectId,
35                      int clusterId) {
36     int i;
37     double ans = 0.0;
38
39     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
clusters*/
40     for (i = 0; i < numCoords; i++) {
41         double objectVal = objects[objectId * numCoords + i];
42         double clusterVal = clusters[clusterId * numCoords + i];
43
44         double diff = objectVal - clusterVal;
45         ans += diff * diff;
46     }
47
48     return (ans);
49 }
50
51 __global__ static
```

```
52 void find_nearest_cluster(int numCoords,
53                           int numObjs,
54                           int numClusters,
55                           double *objects,           // [numObjs][numCoords]
56                           double *deviceClusters,    // [numClusters][numCoords]
57                           int *deviceMembership,      // [numObjs]
58                           double *devdelta) {
59
60     /* Get the global ID of the thread. */
61     int tid = get_tid();
62
63     if (tid < numObjs) {
64         int index, i;
65         double dist, min_dist;
66
67         /* find the cluster id that has min distance to object */
68         index = 0;
69
70         min_dist = euclid_dist_2(numCoords, numObjs, numClusters,
71                               objects, deviceClusters,
72                               tid, index);
73
74         for (i = 1; i < numClusters; i++) {
75
76             dist = euclid_dist_2(numCoords, numObjs, numClusters,
77                               objects, deviceClusters,
78                               tid, i);
79             /* no need square root */
80             if (dist < min_dist) { /* find the min and its array index */
81                 min_dist = dist;
82                 index = i;
83             }
84         }
85
86         if (deviceMembership[tid] != index) {
87
88             atomicAdd(devdelta, 1.0);
89         }
90
91         /* assign the deviceMembership to object objectId */
92         deviceMembership[tid] = index;
93     }
94 }
95
96 //
97 // -----
98 // DATA LAYOUT
99 //
100 // objects        [numObjs][numCoords]
101 // clusters       [numClusters][numCoords]
102 // newClusters    [numClusters][numCoords]
103 // deviceObjects   [numObjs][numCoords]
104 // deviceClusters  [numClusters][numCoords]
105 // -----
```

```

106 //                                                 */
107 /* return an array of cluster centers of size [numClusters][numCoords]      */
108 void kmeans_gpu(double *objects,          /* in: [numObjs][numCoords] */
109                  int numCoords,        /* no. features */
110                  int numObjs,         /* no. objects */
111                  int numClusters,      /* no. clusters */
112                  double threshold,     /* % objects change membership */
113                  long loop_threshold,   /* maximum number of iterations */
114                  int *membership,      /* out: [numObjs] */
115                  double *clusters,       /* out: [numClusters][numCoords] */
116                  int blockSize) {
117
118     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
119     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
120     gpu_time = 0.0;
121
122     int loop_iterations = 0;
123     int i, j, index, loop = 0;
124     int *newClusterSize; /* [numClusters]: no. objects assigned in each
125                           new cluster */
126
127     double delta = 0, *dev_delta_ptr;           /* % of objects change their clusters */
128     double **newClusters = (double **) calloc_2d(numClusters, numCoords, sizeof(double));
129
130     double *deviceObjects;
131     double *deviceClusters;
132     int *deviceMembership;
133
134     printf("\n|-----Naive GPU Kmeans-----|\n\n");
135
136
137     /* initialize membership[] */
138     for (i = 0; i < numObjs; i++) membership[i] = -1;
139
140     /* need to initialize newClusterSize and newClusters[0] to all 0 */
141     newClusterSize = (int *) calloc(numClusters, sizeof(int));
142     assert(newClusterSize != NULL);
143
144     timing = wtime() - timing;
145     printf("t_alloc: %lf ms\n\n", 1000 * timing);
146     timing = wtime();
147
148     const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
149     numObjs;
150
151     const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
152     numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
153
154     const unsigned int clusterBlockSharedDataSize = 0;
155
156     checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
157     checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
158     checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
159     checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
160
161     timing = wtime() - timing;
162     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
163     timing = wtime();
164
165

```

```
157 checkCuda(cudaMemcpy(deviceObjects, objects,
158                     numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
159 checkCuda(cudaMemcpy(deviceMembership, membership,
160                     numObjs * sizeof(int), cudaMemcpyHostToDevice));
161 timing = wtime() - timing;
162 printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
163 timing = wtime();
164
165 do {
166     timing_internal = wtime();
167
168     /* GPU part: calculate new memberships */
169
170     timing_transfers = wtime();
171
172     checkCuda(cudaMemcpy(deviceClusters, clusters,
173                         numClusters * numCoords * sizeof(double),
174                         cudaMemcpyHostToDevice));
175
176     transfers_time += wtime() - timing_transfers;
177
178     checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
179
180     //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
181     //shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
182     taSize/1000);
183     timing_gpu = wtime();
184     find_nearest_cluster
185     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
186         (numCoords, numObjs, numClusters,
187          deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
188
189     cudaDeviceSynchronize();
190     checkLastCudaError();
191     gpu_time += wtime() - timing_gpu;
192     //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
193
194     timing_transfers = wtime();
195
196     checkCuda(cudaMemcpy(membership, deviceMembership,
197                         numObjs * sizeof(int),
198                         cudaMemcpyDeviceToHost));
199
200     checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
201                         sizeof(double),
202                         cudaMemcpyDeviceToHost));
203
204     transfers_time += wtime() - timing_transfers;
205
206     /* CPU part: Update cluster centers*/
207     timing_cpu = wtime();
208     for (i = 0; i < numObjs; i++) {
209         /* find the array index of nestest cluster center */
210         index = membership[i];
```

```

209
210     /* update new cluster centers : sum of objects located within */
211     newClusterSize[index]++;
212     for (j = 0; j < numCoords; j++)
213         newClusters[index][j] += objects[i * numCoords + j];
214     }
215
216     /* average the sum and replace old cluster centers with newClusters */
217     for (i = 0; i < numClusters; i++) {
218         for (j = 0; j < numCoords; j++) {
219             if (newClusterSize[i] > 0)
220                 clusters[i * numCoords + j] = newClusters[i][j] / newClusterSize[i];
221             newClusters[i][j] = 0.0; /* set back to 0 */
222         }
223         newClusterSize[i] = 0; /* set back to 0 */
224     }
225
226     delta /= numObjs;
227     //printf("delta is %f - ", delta);
228     loop++;
229     //printf("completed loop %d\n", loop);
230     cpu_time += wtime() - timing_cpu;
231
232     timing_internal = wtime() - timing_internal;
233     if (timing_internal < timer_min) timer_min = timing_internal;
234     if (timing_internal > timer_max) timer_max = timing_internal;
235 } while (delta > threshold && loop < loop_threshold);
236
237 timing = wtime() - timing;
238 printf("\nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t"
239         "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n-----|\n",
240         loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
241         1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
242
243 char outfile_name[1024] = {0};
244 sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_C1-%d.csv",
245         numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
246 FILE *fp = fopen(outfile_name, "a+");
247 if (!fp) error("Filename %s did not open successfully, no logging performed\n",
outfile_name);
248 fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "Naive", blockSize, timing / loop, timer_min,
timer_max);
249 fclose(fp);
250 checkCuda(cudaFree(deviceObjects));
251 checkCuda(cudaFree(deviceClusters));
252 checkCuda(cudaFree(deviceMembership));
253
254 free(newClusters[0]);
255 free(newClusters);
256 free(newClusterSize);
257
258 return;

```

259 | }

260 |

261 |

B. Υλοποίηση και Ορθότητα

(α) Υπολογισμός απόστασης (euclid dist 2)

Υλοποιείται η τετραγωνική Ευκλείδεια απόσταση σε μορφή row-major (naive έκδοση):

$$d^2(x, c) = \sum_{i=1}^n (x_i - c_i)^2$$

με indexing objects[objectId*numCoords + i], clusters[clusterId*numCoords + i]. Αυτή η προσέγγιση ακολουθήθηκε για τη naive μορφή δεδομένων.

(β) Kernel find_nearest_cluster: αντιστοίχιση threads σε objects

Κάθε thread αντιστοιχεί σε ένα object μέσω global thread id:

$$tid = blockIdx.x * blockDim.x + threadIdx.x.$$

Ο αριθμός blocks ορίζεται ως $\text{ceil}(\text{numObjs} / \text{block_size})$, ώστε να καλύπτονται όλα τα objects, και γίνεται έλεγχος ορίων ($tid < \text{numObjs}$).

(γ) Υπολογισμός delta με atomics

Η μεταβλητή delta μετρά πόσα objects άλλαξαν cluster σε μία επανάληψη. Στο kernel, αν το νέο clusterId διαφέρει από το παλιό membership[tid], γίνεται atomicAdd(devdelta, 1).

Η επιλογή atomics είναι σωστή για αποφυγή race conditions, αλλά αποτελεί και κλασικό σημείο bottleneck (contention) όταν πολλά threads ενημερώνουν την ίδια global μεταβλητή. Δηλαδή, τα atomics επιτυγχάνουν ορθότητα, αλλά όχι απαραίτητα επίδοση, και συχνά αντικαθίστανται από reduction patterns.

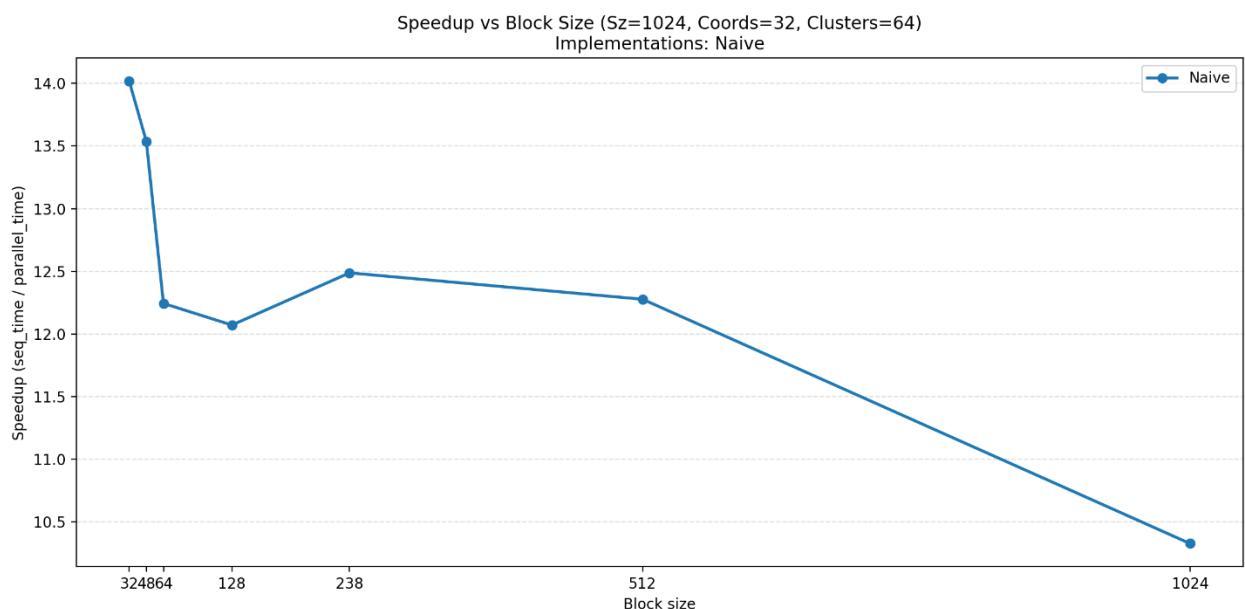
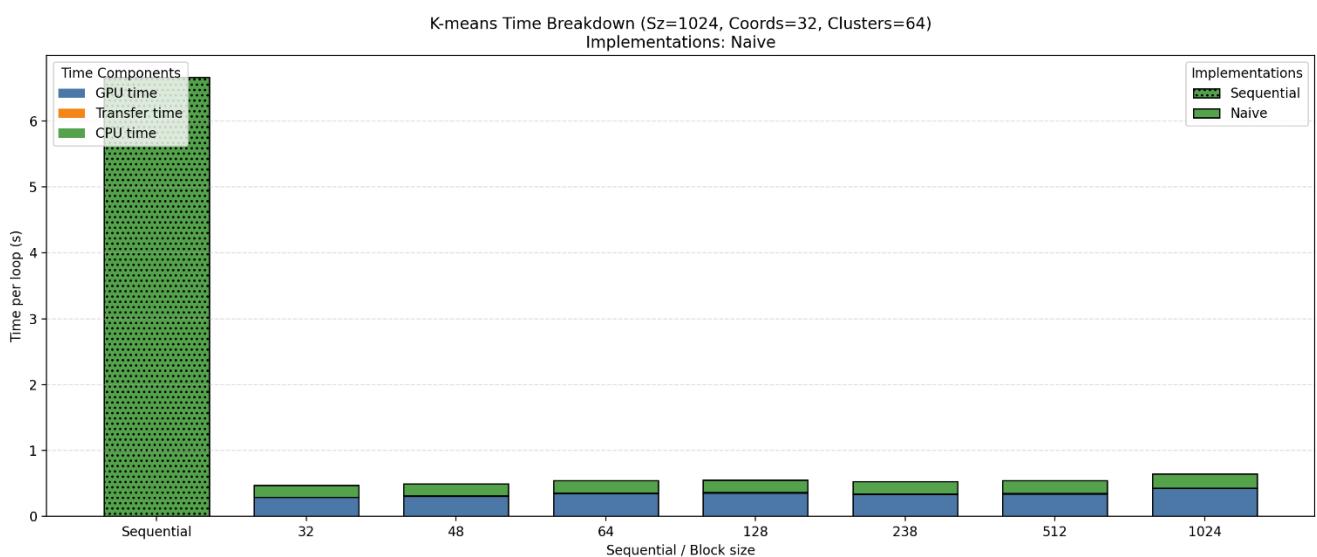
(δ) Timers (breakdown CPU / GPU / Transfers)

Στη naïve έκδοση καταγράφονται τρεις συνιστώσες χρόνου ανά loop:

- GPU time: χρόνος εκτέλεσης του kernel,
- Transfers time: χρόνος αντιγραφών host \leftrightarrow device που γίνονται μέσα στο loop,
- CPU time: χρόνος update_centroids στην CPU.

Επισημαίνεται ότι το «μεγάλο» H \rightarrow D copy του dataset (objects) γίνεται πριν την επανάληψη και μετριέται ξεχωριστά (άρα δεν εμφανίζεται στο per-loop transfers_avg).

Γ. Παρουσίαση Διαγραμμάτων



Δ. Ερμηνεία Διαγραμμάτων

(1) Speedup vs Block Size

Παρατηρείται μέγιστο speedup για μικρά block sizes (π.χ. 32–48) και σταδιακή υποβάθμιση για πολύ μεγάλα block sizes (έως 1024). Η συμπεριφορά αυτή είναι αναμενόμενη βάσει θεωρίας:

- Με μικρότερα blocks, ο scheduler μπορεί να διατηρεί περισσότερα resident blocks/warps ανά SM, αυξάνοντας την ικανότητα απόκρυψης latency (occupancy/latency hiding).
- Με πολύ μεγάλα blocks, μειώνεται ο αριθμός blocks που χωρούν ταυτόχρονα σε ένα SM (λόγω ορίων threads/SM ή πόρων όπως registers), άρα μειώνονται τα ενεργά warps και η GPU δυσκολεύεται να κρύψει memory latency. Επιπλέον, η naïve πρόσβαση σε global μνήμη (objects/clusters) κάνει την επίδοση πιο ευαίσθητη σε occupancy.

(2) Time Breakdown

To breakdown δείχνει ότι:

- Ο συνολικός χρόνος ανά loop της naïve έκδοσης είναι πολύ μικρότερος από το sequential baseline, άρα επιτυγχάνεται σημαντικό speedup.
- Το GPU time είναι η κυρίαρχη συνιστώσα (όπως αναμενόταν, αφού το assignment είναι το κύριο υπολογιστικό μέρος).
- Τα transfer times φαίνονται μηδαμηνά για Coords=32 και αυτό είναι λογικό: μέσα στο loop μεταφέρονται κυρίως (i) τα clusters (πολύ μικρά, ~KB) και (ii) το membership (μεγαλύτερο, αλλά όχι συγκρίσιμο με το 1GB dataset). Αντίθετα, η αρχική αντιγραφή του dataset προς τη GPU (1GB) γίνεται εκτός loop και δεν συμπεριλαμβάνεται στο transfers_avg του breakdown. Ωστόσο, το membership είναι $O(N)$ ανά επανάληψη (Device \rightarrow Host) και μπορεί να γίνει σημαντικό όταν το πλήθος objects N μεγαλώνει, ιδιαίτερα στο Coords=2 όπου για ίδιο Size προκύπτει πολύ μεγαλύτερο N. Άρα, το «μικρά transfers» ισχύει εδώ, για Coords=32, και όχι γενικά.

E. Συμπεράσματα

Η παίνε παραλληλοποίηση επιβεβαιώνει ότι το «assignment step» είναι κατάλληλο για GPU (data-parallel, ανεξάρτητος υπολογισμός ανά object), προσφέροντας υψηλό speedup. Ωστόσο, παραμένουν δύο εγγενή όρια:

- Επικοινωνία και CPU work ανά iteration (clusters/membership transfers + update_centroids στην CPU),
- Atomics για το delta (πιθανό contention).

Το K-means δεν είναι «ιδανικός» πυρήνας GPU ως συνολικός αλγόριθμος, αλλά περιέχει ένα τμήμα που είναι ιδιαίτερα κατάλληλο. Συγκεκριμένα, το βήμα ανάθεσης (assignment: για κάθε object υπολογισμός απόστασης από όλα τα clusters και επιλογή του ελάχιστου) είναι έντονα data-parallel, με ανεξάρτητη εργασία ανά object και μεγάλη παραλληλία, άρα ταιριάζει πολύ καλά στο SIMT μοντέλο των GPUs. Ωστόσο, η συνολική δομή του K-means είναι επαναληπτική και απαιτεί συγχρονισμό μεταξύ επαναλήψεων, ενώ το update των κέντρων είναι reduction/accumulation (sums & counts) και συχνά επιβαρύνεται από atomics και μη ευνοϊκές προσπελάσεις μνήμης. Στη παίνε υλοποίησή μας, επιπλέον, μέρος του κόστους παραμένει εκτός GPU (CPU update + μεταφορές membership/centroids ανά loop), άρα η επίδοση δεν εξαρτάται μόνο από το kernel αλλά και από επικοινωνία/overhead.

Αυτά αποτελούν και το κίνητρο για τις επόμενες εκδόσεις: βελτίωση προσπελάσεων global μνήμης (transpose/coalescing), επαναχρησιμοποίηση δεδομένων μέσω shared memory, και στη συνέχεια πλήρες offload (all-gpu) για μείωση CPU/transfer overhead.

▪ Ενότητα 3.2 – Transpose Version

A. Εισαγωγή

Η έκδοση Transpose στοχεύει αποκλειστικά στη βελτιστοποίηση των προσπελάσεων της global μνήμης στην GPU. Στην naïve έκδοση, τα threads ενός warp που επεξεργάζονται διαδοχικά objects προσπελαύνουν τα δεδομένα με «stride» ως προς τις συντεταγμένες (row-major layout), οδηγώντας σε μη coalesced accesses και αυξημένο αριθμό memory transactions. Η Transpose έκδοση αλλάζει τη διάταξη των δεδομένων σε column-based (transpose) μορφή, έτσι ώστε για κάθε συντεταγμένη i, τα 32 threads ενός warp να διαβάζουν συνεχόμενες διευθύνσεις μνήμης (coalescing), μειώνοντας δραστικά το κόστος πρόσβασης στη global memory και άρα τον χρόνο του kernel.

Ο κώδικας του αρχείου μας (cuda_kmeans_transpose.cu) παρατίθεται ακολούθως:

a5/cuda_kmeans_transpose.cu

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "kmeans.h"
5 #include "alloc.h"
6 #include "error.h"
7
8 #ifdef __CUDACC__
9 inline void checkCuda(cudaError_t e) {
10     if (e != cudaSuccess) {
11         // cudaGetErrorString() isn't always very helpful. Look up the error
12         // number in the cudaError enum in driver_types.h in the CUDA includes
13         // directory for a better explanation.
14         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
15     }
16 }
17
18 inline void checkLastCudaError() {
19     checkCuda(cudaGetLastError());
20 }
21 #endif
22
23 __device__ int get_tid() {
24     return blockIdx.x * blockDim.x + threadIdx.x;
25 }
26
27 /* square of Euclid distance between two multi-dimensional points using column-base format */
28 __host__ __device__ inline static
29 double euclid_dist_2_transpose(int numCoords,
30                                int numObjs,
31                                int numClusters,
32                                double *objects,      // [numCoords][numObjs]
33                                double *clusters,     // [numCoords][numClusters]
34                                int objectId,
35                                int clusterId) {
36     int i;
37     double ans = 0.0;
38
39     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
clusters, but for column-base format!!! */
40     for (i = 0; i < numCoords; i++) {
41         double objectVal = objects[i * numObjs + objectId];
42         double clusterVal = clusters[i * numClusters + clusterId];
43
44         double diff = objectVal - clusterVal;
45         ans += diff * diff;
46     }
47
48     return (ans);
49 }
50

```

```
51 __global__ static
52 void find_nearest_cluster(int numCoords,
53                           int numObjs,
54                           int numClusters,
55                           double *objects,           // [numCoords][numObjs]
56                           double *deviceClusters,    // [numCoords][numClusters]
57                           int *membership,          // [numObjs]
58                           double *devdelta) {
59
60     /* Get the global ID of the thread. */
61     int tid = get_tid();
62
63     if (tid < numObjs) {
64         int index, i;
65         double dist, min_dist;
66
67         /* find the cluster id that has min distance to object */
68         index = 0;
69
70         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
71                                           objects, deviceClusters,
72                                           tid, index);
73
74         for (i = 1; i < numClusters; i++) {
75
76             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
77                                             objects, deviceClusters,
78                                             tid, i);
79
80             /* no need square root */
81             if (dist < min_dist) { /* find the min and its array index */
82                 min_dist = dist;
83                 index = i;
84             }
85         }
86
87         if (membership[tid] != index) {
88
89             atomicAdd(devdelta, 1.0);
90
91         /* assign the deviceMembership to object objectId */
92         membership[tid] = index;
93     }
94
95     //
96     // -----
97     // DATA LAYOUT
98     //
99     // objects      [numObjs][numCoords]
100    // clusters     [numClusters][numCoords]
101    // dimObjects   [numCoords][numObjs]
102    // dimClusters  [numCoords][numClusters]
103    // newClusters  [numCoords][numClusters]
104    // deviceObjects [numCoords][numObjs]
```

```

105 // deviceClusters [numCoords][numClusters]
106 // -----
107 //
108 /* return an array of cluster centers of size [numClusters][numCoords] */
109 void kmeans_gpu(double *objects,      /* in: [numObjs][numCoords] */
110                 int numCoords,    /* no. features */
111                 int numObjs,     /* no. objects */
112                 int numClusters, /* no. clusters */
113                 double threshold, /* % objects change membership */
114                 long loop_threshold, /* maximum number of iterations */
115                 int *membership,  /* out: [numObjs] */
116                 double *clusters, /* out: [numClusters][numCoords] */
117                 int blockSize) {
118     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
119     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
gpu_time = 0.0;
120     int loop_iterations = 0;
121     int i, j, index, loop = 0;
122     int *newClusterSize; /* [numClusters]: no. objects assigned in each
new cluster */
123     double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
124
125     /* TODO: Transpose dims */
126     double **dimObjects = (double **) calloc_2d(numCoords, numObjs, sizeof(double));
//calloc_2d(..., numCoords, numObjs)
127     double **dimClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
//calloc_2d(..., numCoords, numClusters)
128     double **newClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
//calloc_2d(..., numCoords, numClusters)
129
130     double *deviceObjects;
131     double *deviceClusters;
132     int *deviceMembership;
133
134     printf("\n|-----Transpose GPU Kmeans-----|\n\n");
135
136     // TODO: Copy objects given in [numObjs][numCoords] layout to new
137     // [numCoords][numObjs] layout
138     for (i=0 ; i < numObjs; i++){
139         for (j=0; j<numCoords; j++){
140             dimObjects[j][i]=objects[i*numCoords + j];
141         }
142     }
143
144     /* pick first numClusters elements of objects[] as initial cluster centers*/
145     for (i = 0; i < numCoords; i++) {
146         for (j = 0; j < numClusters; j++) {
147             dimClusters[i][j] = dimObjects[i][j];
148         }
149     }
150
151     /* initialize membership[] */
152     for (i = 0; i < numObjs; i++) membership[i] = -1;
153
154

```

```
155 /* need to initialize newClusterSize and newClusters[0] to all 0 */
156 newClusterSize = (int *) calloc(numClusters, sizeof(int));
157 assert(newClusterSize != NULL);
158
159 timing = wtime() - timing;
160 printf("t_alloc: %lf ms\n\n", 1000 * timing);
161 timing = wtime();
162
163 const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
164 numObjs;
165 const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
166 numThreadsPerClusterBlock;
167 const unsigned int clusterBlockSharedDataSize = 0;
168
169 checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
170 checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
171 checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
172 checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
173 timing = wtime() - timing;
174 printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
175 timing = wtime();
176
177 checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
178                      numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
179 checkCuda(cudaMemcpy(deviceMembership, membership,
180                      numObjs * sizeof(int), cudaMemcpyHostToDevice));
181 timing = wtime() - timing;
182 printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
183 timing = wtime();
184
185 do {
186     timing_internal = wtime();
187
188     /* GPU part: calculate new memberships */
189
190     timing_transfers = wtime();
191     // TODO: Copy clusters to deviceClusters
192     checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
193                          numClusters * numCoords * sizeof(double),
194                          cudaMemcpyHostToDevice));
195
196     transfers_time += wtime() - timing_transfers;
197
198     checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
199
200     //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
201     //shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
202     taSize/1000);
203     timing_gpu = wtime();
204     find_nearest_cluster
205     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
206         (numCoords, numObjs, numClusters,
207          deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
```

```
205     cudaDeviceSynchronize();
206     checkLastCudaError();
207     gpu_time += wtime() - timing_gpu;
208     //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
209
210     timing_transfers = wtime();
211
212     checkCuda(cudaMemcpy(membership, deviceMembership,
213                         numObjs * sizeof(int),
214                         cudaMemcpyDeviceToHost));
215
216     checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
217                         sizeof(double),
218                         cudaMemcpyDeviceToHost));
219     transfers_time += wtime() - timing_transfers;
220
221     /* CPU part: Update cluster centers*/
222
223     timing_cpu = wtime();
224     for (i = 0; i < numObjs; i++) {
225         /* find the array index of nestest cluster center */
226         index = membership[i];
227
228         /* update new cluster centers : sum of objects located within */
229         newClusterSize[index]++;
230         for (j = 0; j < numCoords; j++)
231             newClusters[j][index] += objects[i * numCoords + j];
232     }
233
234     /* average the sum and replace old cluster centers with newClusters */
235     for (i = 0; i < numClusters; i++) {
236         for (j = 0; j < numCoords; j++) {
237             if (newClusterSize[i] > 0)
238                 dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
239             newClusters[j][i] = 0.0; /* set back to 0 */
240         }
241         newClusterSize[i] = 0; /* set back to 0 */
242     }
243
244     delta /= numObjs;
245     //printf("delta is %f - ", delta);
246     loop++;
247     //printf("completed loop %d\n", loop);
248     cpu_time += wtime() - timing_cpu;
249
250     timing_internal = wtime() - timing_internal;
251     if (timing_internal < timer_min) timer_min = timing_internal;
252     if (timing_internal > timer_max) timer_max = timing_internal;
253 } while (delta > threshold && loop < loop_threshold);
254
255 /*TODO: Update clusters using dimClusters. Be carefull of layout!!!
clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */
256 for (i = 0; i < numClusters; i++) {
257     for (j = 0; j < numCoords; j++) {
```

```
258         clusters[i * numCoords + j] = dimClusters[j][i];
259     }
260 }
261
262 timing = wtime() - timing;
263 printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n\n|-----|\n",
264     loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
265     1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
266
267
268 char outfile_name[1024] = {0};
269 sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_C1-%d.csv",
270         numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
271 FILE *fp = fopen(outfile_name, "a+");
272 if (!fp) error("Filename %s did not open successfully, no logging performed\n",
outfile_name);
273 fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "Transpose", blockSize, timing / loop, timer_min,
timer_max);
274 fclose(fp);
275
276 checkCuda(cudaFree(deviceObjects));
277 checkCuda(cudaFree(deviceClusters));
278 checkCuda(cudaFree(deviceMembership));
279
280 free(dimObjects[0]);
281 free(dimObjects);
282 free(dimClusters[0]);
283 free(dimClusters);
284 free(newClusters[0]);
285 free(newClusters);
286 free(newClusterSize);
287
288 return;
289 }
290
291 }
```

B. Υλοποίηση και Ορθότητα

Η λογική του αλγορίθμου παραμένει ίδια: η GPU εκτελεί το assignment (membership) και η CPU εκτελεί το update_centroids. Η αλλαγή είναι καθαρά στη δομή δεδομένων:

- Αντί για `objects[object][coord]`, δημιουργείται `dimObjects[coord][object]`.
- Αντί για `clusters[cluster][coord]`, δημιουργείται `dimClusters[coord][cluster]`.

Επισημαίνουμε τα εξής σημεία:

(α) Νέα συνάρτηση απόστασης euclid dist 2 transpose

Υπολογίζεται η ίδια Ευκλείδεια απόσταση, αλλά με indexing που ευνοεί coalescing:

`objects[i*numObjs + objectId]` και `clusters[i*numClusters + clusterId]`.

Έτσι, για σταθερό i , τα threads του warp διαβάζουν συνεχόμενα `objects` (`objectId` διαδοχικά), άρα οι αναγνώσεις είναι coalesced.

(β) Μετασχηματισμός των δεδομένων (transpose) πριν την επανάληψη

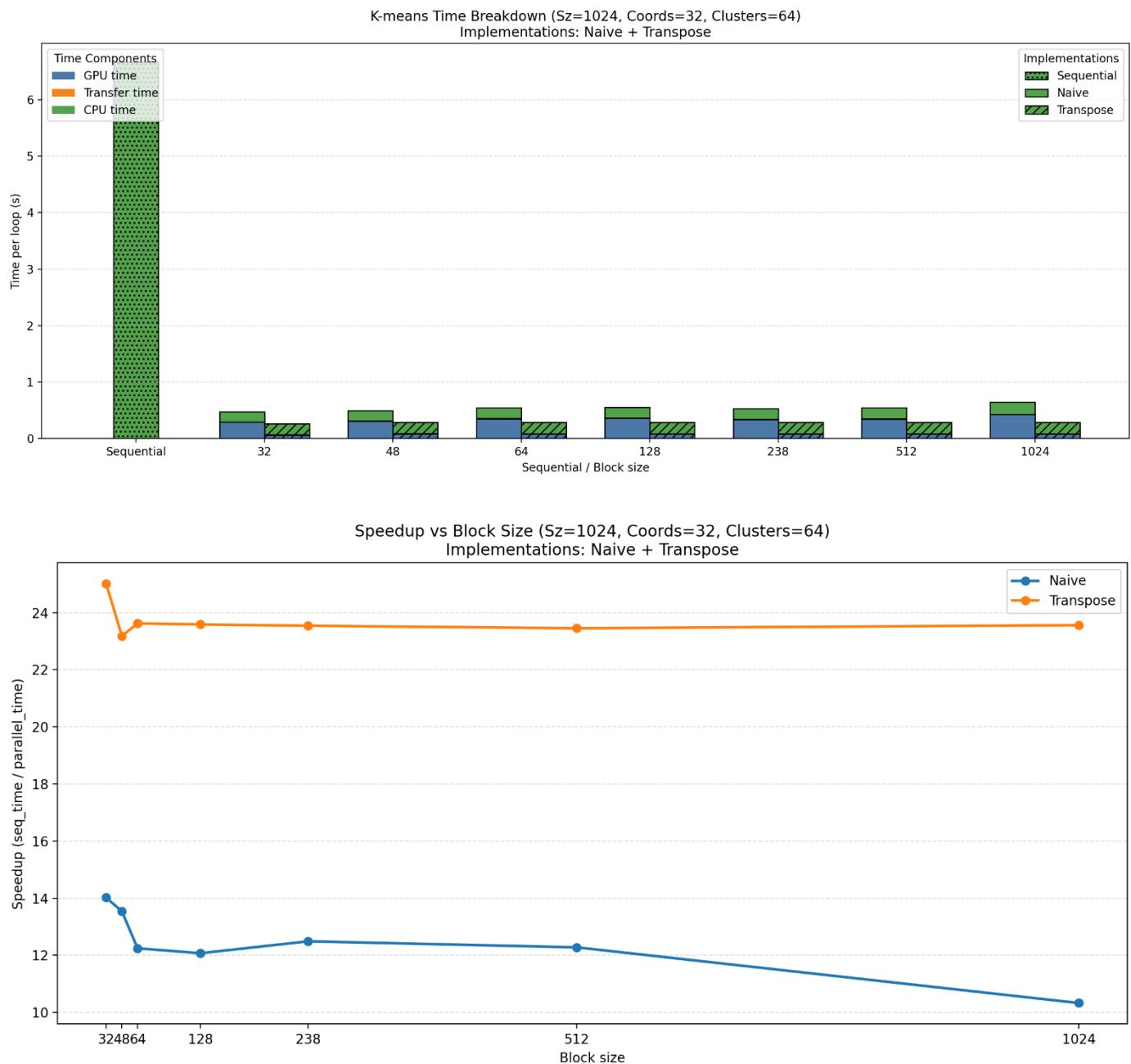
Πριν ξεκινήσει το loop, ο host κατασκευάζει τον `dimObjects` πίνακα από το αρχικό row-major `objects`. Αυτό είναι κόστος προεπεξεργασίας (εκτός loop) και δεν επηρεάζει το per-loop breakdown.

(γ) Ενημέρωση clusters με transpose μορφή

Στο CPU `update_centroids`, τα αθροίσματα/μέσοι όροι ενημερώνονται στη μορφή `dimClusters[coord][cluster]` ώστε το επόμενο H \rightarrow D copy να διατηρεί τη coalesced διάταξη. Στο τέλος γίνεται back-transform σε `clusters[cluster][coord]` μόνο για λόγους συμβατότητας/εκτύπωσης.

Γενικά, η Transpose έκδοση είναι αριθμητικά ισοδύναμη με τη Naive (ίδια μετρική απόστασης, ίδια διαδικασία ανάθεσης/ενημέρωσης), αλλά με διαφορετική διάταξη στη μνήμη. Επομένως, αναμένουμε τα ίδια clusters (εντός floating-point διαφορών) και το ίδιο κριτήριο σύγκλισης· η διαφορά αφορά αποκλειστικά την επίδοση λόγω memory access pattern.

Γ. Παρουσίαση Διαγραμμάτων



Δ. Ερμηνεία Διαγραμμάτων

(1) Speedup vs Block Size

Η Transpose έκδοση παρουσιάζει σημαντικά υψηλότερο speedup από τη Naive (περίπου 23–25 έναντι ~10–14), και μάλιστα με σχετικά «επίπεδη» συμπεριφορά ως προς το block size. Αυτό είναι αναμενόμενο, καθώς:

- Με coalesced προσπελάσεις, μειώνονται τα global memory transactions ανά warp, άρα αυξάνεται το effective bandwidth και μειώνεται ο χρόνος kernel.
- Όταν το κύριο bottleneck είναι οι προσπελάσεις μνήμης, η βελτίωση στο memory access pattern έχει μεγαλύτερη επίδραση από μικρο-βελτιστοποιήσεις scheduling/occupancy μέσω block size, με αποτέλεσμα πιο σταθερή καμπύλη.

Στη Transpose έκδοση το block_size παίζει σαφώς μικρότερο ρόλο σε σχέση με τη Naive, όπως φαίνεται από τη σχεδόν επίπεδη καμπύλη speedup. Ο λόγος είναι ότι με το transpose πετυχαίνουμε coalesced προσπελάσεις στη global μνήμη (τα threads ενός warp διαβάζουν συνεχόμενες διευθύνσεις για κάθε συντεταγμένη), άρα μειώνεται δραστικά το κόστος memory transactions και το kernel γίνεται λιγότερο ευαίσθητο σε αλλαγές occupancy/scheduling που προκαλεί το block_size. Εφόσον το block_size είναι πολλαπλάσιο του 32 (warp size) και διατηρεί επαρκή ενεργά warps ανά SM, η απόδοση παραμένει σχεδόν σταθερή. Μόνο σε ακραία μεγέθη blocks ενδέχεται να εμφανιστεί μικρή πτώση (π.χ. λόγω μειωμένων resident blocks/warps ανά SM ή αυξημένων απαιτήσεων πόρων), αλλά συνολικά το κυρίαρχο κέρδος στη Transpose προέρχεται από το βελτιωμένο memory access pattern και όχι από την επιλογή block_size.

(2) Time Breakdown (GPU / Transfers / CPU)

To breakdown δείχνει ότι η κύρια μείωση χρόνου προέρχεται από το GPU time (kernel). Αυτό είναι ακριβώς το αναμενόμενο αποτέλεσμα της βελτιστοποίησης coalescing: δεν αλλάζουμε τις μεταφορές ανά loop ούτε το CPU update_centroids, αλλά μειώνουμε δραστικά τον χρόνο της φάσης assignment στη GPU.

Τα transfer times παραμένουν χαμηλά στο συγκεκριμένο σενάριο (Coords=32), διότι εντός loop μεταφέρεται:

- Host→Device: clusters (μικρό μέγεθος),
- Device→Host: membership + delta.

Η αρχική αντιγραφή των objects (1GB) γίνεται εκτός loop και δεν περιλαμβάνεται στο per-loop transfer χρόνο.

E. Σύγκριση Αποτελεσμάτων (με Naive)

1. Κύριο εύρημα: Η Transpose έκδοση επιτυγχάνει $\sim 1.7\times\text{--}2\times$ καλύτερο speedup από τη Naive, παρότι ο αλγόριθμος παραμένει ο ίδιος.
2. Η βελτίωση δεν οφείλεται σε περισσότερους υπολογισμούς στη GPU, αλλά σε καθαρά αρχιτεκτονικό λόγο: καλύτερη αξιοποίηση του memory subsystem μέσω coalescing (32-thread warps → συνεχόμενες διευθύνσεις → λιγότερα transactions).
3. Η συνιστώσα CPU (update_centroids) παραμένει πρακτικά η ίδια, άρα το συνολικό κέρδος έρχεται από τη μείωση του kernel time.
4. Η εξάρτηση από block size είναι μικρότερη σε σχέση με τη Naive, επειδή η Transpose μειώνει το memory overhead και σταθεροποιεί την απόδοση.

ΣΤ. Συμπεράσματα

Η Transpose έκδοση επιβεβαιώνει ότι η διάταξη των δεδομένων στη μνήμη μπορεί να είναι καθοριστική για την απόδοση σε GPU. Με την αναδιάταξη σε column-based μορφή πετυχαίνουμε coalesced global memory accesses κατά τον υπολογισμό αποστάσεων, μειώνοντας αισθητά τον χρόνο του kernel και αυξάνοντας το speedup. Αυτό αποτελεί το φυσικό επόμενο βήμα μετά τη Naive προσέγγιση και δημιουργεί τη βάση για την επόμενη βελτιστοποίηση (Shared), όπου στοχεύουμε επιπλέον στη μείωση των επαναλαμβανόμενων αναγνώσεων clusters μέσω shared memory (on-chip reuse).

- **Ενότητα 3.3 – Shared Version**

A. Εισαγωγή

Η έκδοση Shared επεκτείνει την βελτιστοποίηση που εισαγάγαμε στην Transpose και στοχεύει στη μείωση των επαναλαμβανόμενων αναγνώσεων των cluster centers από την global μνήμη. Στο K-means, για κάθε object υπολογίζονται αποστάσεις από όλα τα clusters. Άρα, τα ίδια cluster centers επαναχρησιμοποιούνται πολλές φορές από τα threads ενός block. Με τη φόρτωσή τους στη shared memory (on-chip), μειώνουμε σημαντικά το global memory traffic και επιταχύνουμε το assignment kernel.

Ο κώδικας του αρχείου μας (cuda_kmeans_shared.cu) παρατίθεται ακολούθως:

a5/cuda_kmeans_shared.cu

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "kmeans.h"
5 #include "alloc.h"
6 #include "error.h"
7
8 #ifdef __CUDACC__
9 inline void checkCuda(cudaError_t e) {
10     if (e != cudaSuccess) {
11         // cudaGetErrorString() isn't always very helpful. Look up the error
12         // number in the cudaError enum in driver_types.h in the CUDA includes
13         // directory for a better explanation.
14         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
15     }
16 }
17
18 inline void checkLastCudaError() {
19     checkCuda(cudaGetLastError());
20 }
21 #endif
22
23 __device__ int get_tid() {
24     return blockIdx.x * blockDim.x + threadIdx.x;
25 }
26
27 /* square of Euclid distance between two multi-dimensional points using column-base format */
28 __host__ __device__ inline static
29 double euclid_dist_2_transpose(int numCoords,
30                                int numObjs,
31                                int numClusters,
32                                double *objects,      // [numCoords][numObjs]
33                                double *clusters,     // [numCoords][numClusters]
34                                int objectId,
35                                int clusterId) {
36     int i;
37     double ans = 0.0;
38
39     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
clusters, but for column-base format!!! */
40     for (i = 0; i < numCoords; i++) {
41         double objectVal = objects[i * numObjs + objectId];
42         double clusterVal = clusters[i * numClusters + clusterId];
43
44         double diff = objectVal - clusterVal;
45         ans += diff * diff;
46     }
47
48     return (ans);
49 }
50
```

```

51 __global__ static
52 void find_nearest_cluster(int numCoords,
53                           int numObjs,
54                           int numClusters,
55                           double *objects,           // [numCoords][numObjs]
56                           double *deviceClusters,    // [numCoords][numClusters]
57                           int *deviceMembership,     // [numObjs]
58                           double *devdelta) {
59     extern __shared__ double shmemClusters[];
60
61     // TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
62     int tid_in_block = threadIdx.x;      // Το ID του νήματος μέσα στο Block
63     int block_size = blockDim.x;         // Πόσα νήματα έχει το Block
64     int total_cluster_doubles = numClusters * numCoords; // Συνολικά νούμερα προς αντιγραφή
65
66     // Κάθε νήμα αντιγράφει όσα στοιχεία του αναλογούν (με βήμα block_size)
67     for (int k = tid_in_block; k < total_cluster_doubles; k += block_size) {
68         shmemClusters[k] = deviceClusters[k];
69     }
70
71     /* Συγχρονισμός (BARRIER) */
72
73     __syncthreads();
74
75     /* Get the global ID of the thread. */
76     int tid = get_tid();
77
78     /* TODO: Maybe something is missing here... should all threads run this? */
79     if (tid < numObjs) {
80         int index, i;
81         double dist, min_dist;
82
83         /* find the cluster id that has min distance to object */
84         index = 0;
85         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using
clusters in shmem*/
86
87
88         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
89                                           objects, shmemClusters,
90                                           tid, index);
91
92         for (i = 1; i < numClusters; i++) {
93             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
94                                           objects, shmemClusters,
95                                           tid, i);
96
97             /* no need square root */
98             if (dist < min_dist) { /* find the min and its array index */
99                 min_dist = dist;
100                index = i;
101            }
102        }
103    }

```

```

104     if (deviceMembership[tid] != index) {
105         /* TODO: Maybe something is missing here... is this write safe? */
106         atomicAdd(devdelta, 1.0);
107     }
108
109     /* assign the deviceMembership to object objectId */
110     deviceMembership[tid] = index;
111 }
112 }
113
114 //
115 // -----
116 // DATA LAYOUT
117 //
118 // objects      [numObjs][numCoords]
119 // clusters     [numClusters][numCoords]
120 // dimObjects   [numCoords][numObjs]
121 // dimClusters  [numCoords][numClusters]
122 // newClusters  [numCoords][numClusters]
123 // deviceObjects [numCoords][numObjs]
124 // deviceClusters [numCoords][numClusters]
125 //
126 //
127 /* return an array of cluster centers of size [numClusters][numCoords]      */
128 void kmeans_gpu(double *objects,          /* in: [numObjs][numCoords] */
129                  int numCoords,    /* no. features */
130                  int numObjs,     /* no. objects */
131                  int numClusters, /* no. clusters */
132                  double threshold, /* % objects change membership */
133                  long loop_threshold, /* maximum number of iterations */
134                  int *membership, /* out: [numObjs] */
135                  double *clusters, /* out: [numClusters][numCoords] */
136                  int blockSize) {
137     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;
138     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
gpu_time = 0.0;
139     int loop_iterations = 0;
140     int i, j, index, loop = 0;
141     int *newClusterSize; /* [numClusters]: no. objects assigned in each
new cluster */
142     double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
143     /* TODO: Copy me from transpose version*/
144     double **dimObjects = (double **) calloc_2d(numCoords, numObjs, sizeof(double));
//calloc_2d(...)->[numCoords][numObjs]
145     double **dimClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
//calloc_2d(...)->[numCoords][numClusters]
146     double **newClusters = (double **) calloc_2d(numCoords, numClusters, sizeof(double));
//calloc_2d(...)->[numCoords][numClusters]
147
148     double *deviceObjects;
149     double *deviceClusters;
150     int *deviceMembership;
151
152     printf("\n|-----Shared GPU Kmeans-----|\n\n");

```

```
154
155     /* TODO: Copy me from transpose version*/
156     for (i=0 ; i < numObjs; i++){
157         for (j=0; j<numCoords; j++){
158             dimObjects[j][i]=objects[i*numCoords + j];
159         }
160     }
161
162     /* pick first numClusters elements of objects[] as initial cluster centers*/
163     for (i = 0; i < numCoords; i++) {
164         for (j = 0; j < numClusters; j++) {
165             dimClusters[i][j] = dimObjects[i][j];
166         }
167     }
168
169     /* initialize membership[] */
170     for (i = 0; i < numObjs; i++) membership[i] = -1;
171
172     /* need to initialize newClusterSize and newClusters[0] to all 0 */
173     newClusterSize = (int *) calloc(numClusters, sizeof(int));
174     assert(newClusterSize != NULL);
175
176     timing = wtime() - timing;
177     printf("t_alloc: %lf ms\n\n", 1000 * timing);
178     timing = wtime();
179     const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
180     numObjs;
181     const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
182     numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
183
184     /* Define the shared memory needed per block.
185      - BEWARE: We can overrun our shared memory here if there are too many
186      clusters or too many coordinates!
187      - This can lead to occupancy problems or even inability to run.
188      - Your exercise implementation is not requested to account for that (e.g. always
189      assume deviceClusters fit in shmemClusters */
190     const unsigned int clusterBlockSharedDataSize = numClusters*numCoords*sizeof(double);
191
192     cudaDeviceProp deviceProp;
193     int deviceNum;
194     cudaGetDevice(&deviceNum);
195     cudaGetDeviceProperties(&deviceProp, deviceNum);
196
197     if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock) {
198         error("Your CUDA hardware has insufficient block shared memory to hold all cluster
199         centroids\n");
200     }
201
202     checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
203     checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
204     checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
205     checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
206
207     timing = wtime() - timing;
```

```
204     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
205     timing = wtime();
206
207     checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
208                          numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
209     checkCuda(cudaMemcpy(deviceMembership, membership,
210                         numObjs * sizeof(int), cudaMemcpyHostToDevice));
211     timing = wtime() - timing;
212     printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
213     timing = wtime();
214
215 do {
216     timing_internal = wtime();
217
218     /* GPU part: calculate new memberships */
219
220     timing_transfers = wtime();
221     // TODO: Copy clusters to deviceClusters
222     checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
223                          numClusters * numCoords * sizeof(double),
224                          cudaMemcpyHostToDevice));
225
226     transfers_time += wtime() - timing_transfers;
227
228     checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
229
230     timing_gpu = wtime();
231     //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
232     shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize/1000);
233     find_nearest_cluster
234     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
235         (numCoords, numObjs, numClusters,
236          deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
237
238     cudaDeviceSynchronize();
239     checkLastCudaError();
240     gpu_time += wtime() - timing_gpu;
241     //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
242
243     timing_transfers = wtime();
244
245     checkCuda(cudaMemcpy(membership, deviceMembership,
246                          numObjs * sizeof(int),
247                          cudaMemcpyDeviceToHost));
248
249     checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
250                         sizeof(double),
251                         cudaMemcpyDeviceToHost));
252
253     transfers_time += wtime() - timing_transfers;
254
255     /* CPU part: Update cluster centers*/
```

```

256
257     timing_cpu = wtime();
258     for (i = 0; i < numObjs; i++) {
259         /* find the array index of nestest cluster center */
260         index = membership[i];
261
262         /* update new cluster centers : sum of objects located within */
263         newClusterSize[index]++;
264         for (j = 0; j < numCoords; j++)
265             newClusters[j][index] += objects[i * numCoords + j];
266     }
267
268     /* average the sum and replace old cluster centers with newClusters */
269     for (i = 0; i < numClusters; i++) {
270         for (j = 0; j < numCoords; j++) {
271             if (newClusterSize[i] > 0)
272                 dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
273             newClusters[j][i] = 0.0; /* set back to 0 */
274         }
275         newClusterSize[i] = 0; /* set back to 0 */
276     }
277
278     delta /= numObjs;
279     //printf("delta is %f - ", delta);
280     loop++;
281     //printf("completed loop %d\n", loop);
282     cpu_time += wtime() - timing_cpu;
283
284     timing_internal = wtime() - timing_internal;
285     if (timing_internal < timer_min) timer_min = timing_internal;
286     if (timing_internal > timer_max) timer_max = timing_internal;
287 } while (delta > threshold && loop < loop_threshold);
288
289 /*TODO: Update clusters using dimClusters. Be carefull of layout!!!
clusters[numClusters][numCoords] vs dimClusters[numCoords][numClusters] */
290 for (i = 0; i < numClusters; i++) {
291     for (j = 0; j < numCoords; j++) {
292         clusters[i * numCoords + j] = dimClusters[j][i];
293     }
294 }
295
296 timing = wtime() - timing;
297 printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
ms\n\t-> t_loop_max = %lf ms\n\t"
298         "-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
ms\n\n|-----|\n",
299         loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
300         1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
301
302 char outfile_name[1024] = {0};
303 sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
304         numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
305 FILE *fp = fopen(outfile_name, "a+");

```

```
306 if (!fp) error("Filename %s did not open successfully, no logging performed\n",
307     outfile_name);
308     fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "Shmem", blockSize, timing / loop, timer_min,
309     timer_max);
310     fclose(fp);
311
312     checkCuda(cudaFree(deviceObjects));
313     checkCuda(cudaFree(deviceClusters));
314     checkCuda(cudaFree(deviceMembership));
315
316     free(dimObjects[0]);
317     free(dimObjects);
318     free(dimClusters[0]);
319     free(dimClusters);
320     free(newClusters[0]);
321     free(newClusters);
322     free(newClusterSize);
323
324     return;
325 }
```

B. Υλοποίηση και Ορθότητα

Η δομή δεδομένων παραμένει transpose (dimObjects[coord][obj], dimClusters[coord][cluster]) για coalescing. Η βασική αλλαγή είναι ότι στον kernel:

- Τα cluster centers αντιγράφονται μια φορά ανά block από global σε shared memory.
- Όλοι οι υπολογισμοί απόστασης χρησιμοποιούν πλέον τη shared μνήμη για τα clusters.

Επισημαίνουμε τα εξής σημεία:

(α) Δυναμική shared memory και αντιγραφή clusters

Χρησιμοποιείται extern __shared__ double shmemClusters[] και αντιγράφεται ολόκληρος ο πίνακας dimClusters (numCoords*numClusters στοιχεία) στη shared memory, με «striding» ως προς threadIdx (k += blockDim.x). Έτσι η φόρτωση μοιράζεται σε threads και γίνεται μία φορά ανά block.

(β) Συγχρονισμός (__syncthreads)

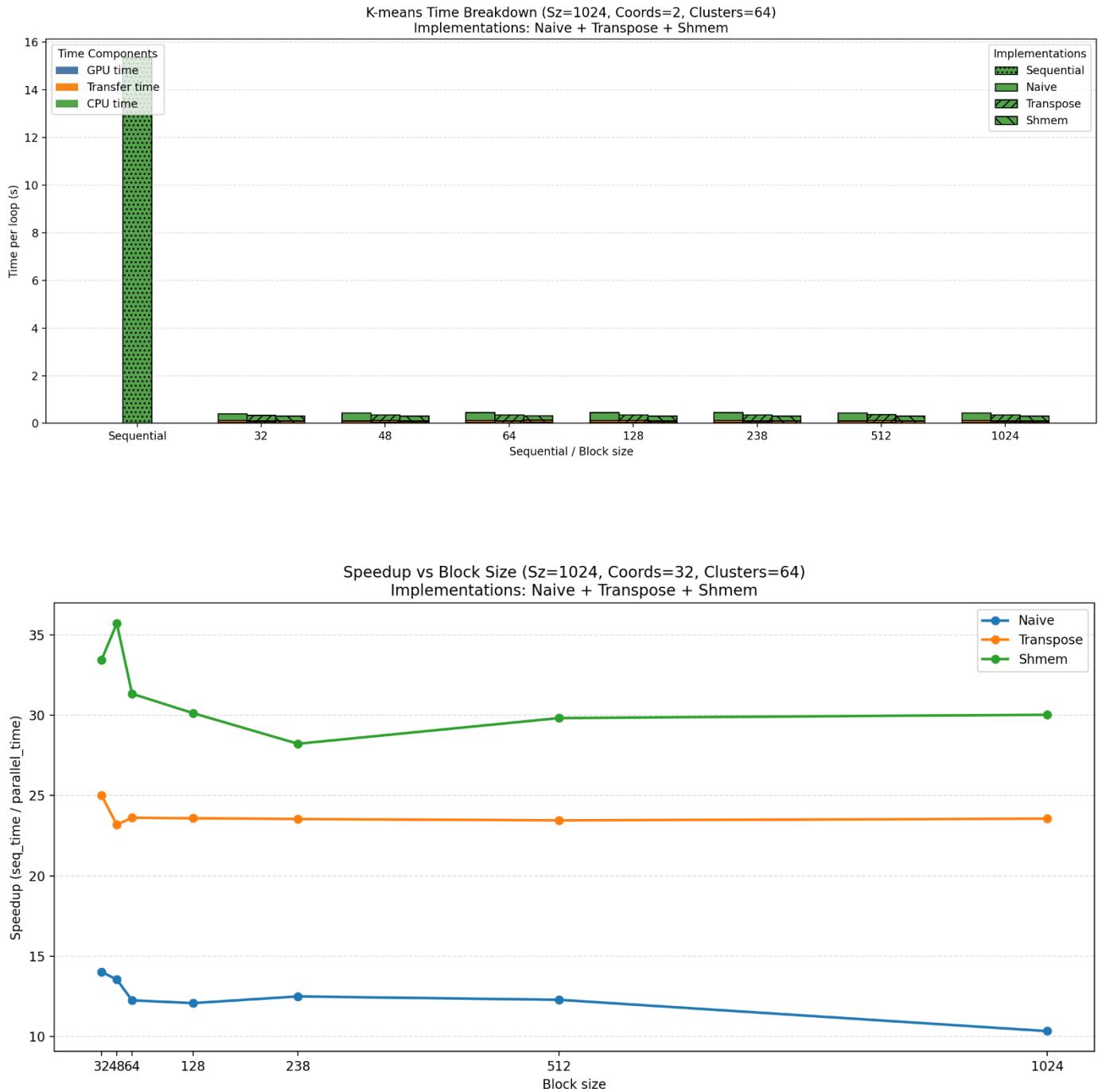
Μετά τη φόρτωση στη shared, γίνεται __syncthreads() ώστε να εξασφαλιστεί ότι όλα τα threads του block βλέπουν πλήρως γραμμένα τα δεδομένα πριν ξεκινήσουν τους υπολογισμούς αποστάσεων. Αυτό είναι απαραίτητο για ορθότητα (διαφορετικά κάποια threads θα διάβαζαν μη αρχικοποιημένες τιμές).

(γ) Έλεγχος διαθέσιμης shared μνήμης ανά block

Στον host υπολογίζεται το απαιτούμενο shared size = numClusters * numCoords * sizeof(double) και ελέγχεται έναντι της ιδιότητας sharedMemPerBlock της συσκευής. Αν το όριο ξεπεραστεί, η έκδοση δεν μπορεί να τρέξει (σωστό safeguard, καθώς το shared memory είναι περιορισμένος πόρος).

Η έκδοση Shared είναι αριθμητικά ισοδύναμη με Transpose/Naive: δεν αλλάζει ο ορισμός απόστασης ούτε το κριτήριο ανάθεσης. Αλλάζει μόνο η τοποθέτηση των cluster centers (shared αντί global), άρα αναμένουμε ίδια αποτελέσματα σύγκλισης (εντός floating-point διαφορών), με χαμηλότερο χρόνο kernel.

Γ. Παρουσίαση Διαγραμμάτων



Δ. Ερμηνεία Διαγραμμάτων

(1) Speedup vs Block Size

Η Shared υπερέχει σαφώς, με speedup περίπου 28–36, έναντι ~23–25 της Transpose και ~10–14 της Naive. Η βελτίωση είναι αναμενόμενη: ενώ η Transpose μειώνει τα global transactions μέσω coalescing, η Shared μειώνει και το πλήθος των global loads για τα clusters, αφού κάθε block φέρνει τα clusters μία φορά και τα επαναχρησιμοποιεί σε όλους τους distance υπολογισμούς.

(2) Time Breakdown (GPU / Transfers / CPU)

Το breakdown δείχνει ότι το κύριο κέρδος της Shared προέρχεται από περαιτέρω μείωση του GPU time (kernel). Οι μεταφορές (clusters H→D, membership+delta D→H) και ο CPU χρόνος (update_centroids) παραμένουν ουσιαστικά παρόμοια με Transpose/Naive, άρα η επιτάχυνση οφείλεται σχεδόν αποκλειστικά στη βελτίωση του memory access/reuse εντός του kernel.

Ρόλος του block_size στη Shared

Σε αντίθεση με τη Transpose (όπου η εξάρτηση από block_size ήταν μικρή), στη Shared το block_size μπορεί να επηρεάζει περισσότερο την απόδοση, επειδή η shared memory εισάγει πρόσθετους περιορισμούς στους resident πόρους ανά SM:

- Κάθε block δεσμεύει σταθερό shared size (εδώ: `numClusters*numCoords*sizeof(double)`), άρα ο μέγιστος αριθμός blocks/SM μπορεί να περιοριστεί από τη διαθέσιμη shared μνήμη, μειώνοντας occupancy (active warps/SM).
- Με πολύ μεγάλα blocks, περιοριζόμαστε επιπλέον από το όριο threads/SM, άρα μπορεί να μειωθούν ταυτόχρονα resident blocks και warps, και να αυξηθεί η ευαισθησία σε latency.

Συνεπώς, παρατηρείται συνήθως ένα ιδανικό σημείο (small block sizes) όπου συνδυάζονται αρκετά active warps και χαμηλό global traffic, ενώ σε ακραία μεγέθη blocks η απόδοση μπορεί να σταθεροποιείται ή να πέφτει.

E. Σύγκριση Αποτελεσμάτων (με Naive + Transpose)

1. Από Naive → Transpose: μεγάλο κέρδος λόγω coalescing (μείωση global memory transactions).
2. Από Transpose → Shared: επιπλέον μεγάλο κέρδος, διότι μειώνουμε τις επαναλαμβανόμενες αναγνώσεις clusters από global (on-chip reuse). Το κέρδος εμφανίζεται κυρίως ως περαιτέρω μείωση του GPU time, ενώ CPU και transfers παραμένουν περίπου σταθερά.
3. Η Shared εμφανίζει μεγαλύτερη (αλλά λογική) εξάρτηση από block_size σε σχέση με Transpose, λόγω των πόρων shared memory/occupancy.

ΣΤ. Συμπεράσματα

Η Shared έκδοση επιβεβαιώνει τη βασική αρχή βελτιστοποίησης GPU: πέρα από το coalescing, η επαναχρησιμοποίηση «hot» δεδομένων στη shared memory μπορεί να μειώσει δραστικά το global memory traffic και να επιταχύνει σημαντικά memory-bound kernels όπως το assignment του K-means. Για το συγκεκριμένο σενάριο (Coords=32, Clusters=64), η Shared είναι η καλύτερη από τις τρεις εκδόσεις, με το κέρδος να προέρχεται κυρίως από τη μείωση του χρόνου kernel και δευτερευόντως από επιλογές block_size που επηρεάζουν occupancy.

■ Ενότητα 3.4 – Σύγκριση Υλοποιήσεων/Bottleneck Analysis

A. Εισαγωγή

Στο σημείο αυτό συγκρίνουμε τις τρεις υλοποιήσεις (Naive, Transpose, Shared) και εντοπίζουμε το bottleneck χρησιμοποιώντας τα δεδομένα χρόνου ανά επανάληψη (GPU kernel / transfers CPU↔GPU / CPU update). Υπενθυμίζουμε ότι τα transfers που μετράμε εδώ αφορούν τις αντιγραφές μέσα στο loop (π.χ. clusters H→D και membership+delta D→H) και όχι την αρχική μεταφορά του dataset προς τη GPU, η οποία γίνεται μία φορά πριν ξεκινήσουν οι επαναλήψεις.

B. Σύγκριση Υλοποιήσεων/Bottleneck Analysis

1. Ποιο bottleneck περιορίζει την επίδοση (Sz=1024MB, Coords=32, Clusters=64);

Από τα διαγράμματα για Coords=32, το αποτέλεσμα είναι ξεκάθαρο:

- Naive → Transpose: η κύρια μείωση χρόνου έρχεται από το GPU time, επειδή το transpose βελτιώνει το memory coalescing (λιγότερα global memory transactions ανά warp).
- Transpose → Shared: το GPU time μειώνεται περαιτέρω, επειδή τα cluster centers επαναχρησιμοποιούνται από shared memory (on-chip) αντί για επαναλαμβανόμενες αναγνώσεις από global.

Μετά τις δύο αυτές βελτιστοποιήσεις, όμως, παρατηρείται ότι το συνολικό κέρδος δεν αυξάνεται αναλογικά (ιδανικά, δηλαδή δεν κλιμακώνει τέλεια) με τη μείωση του GPU time. Ο λόγος είναι ότι πλέον αρχίζουν να κυριαρχούν/να γίνονται συγκρίσιμα τα μη-επιταχυνόμενα τμήματα:

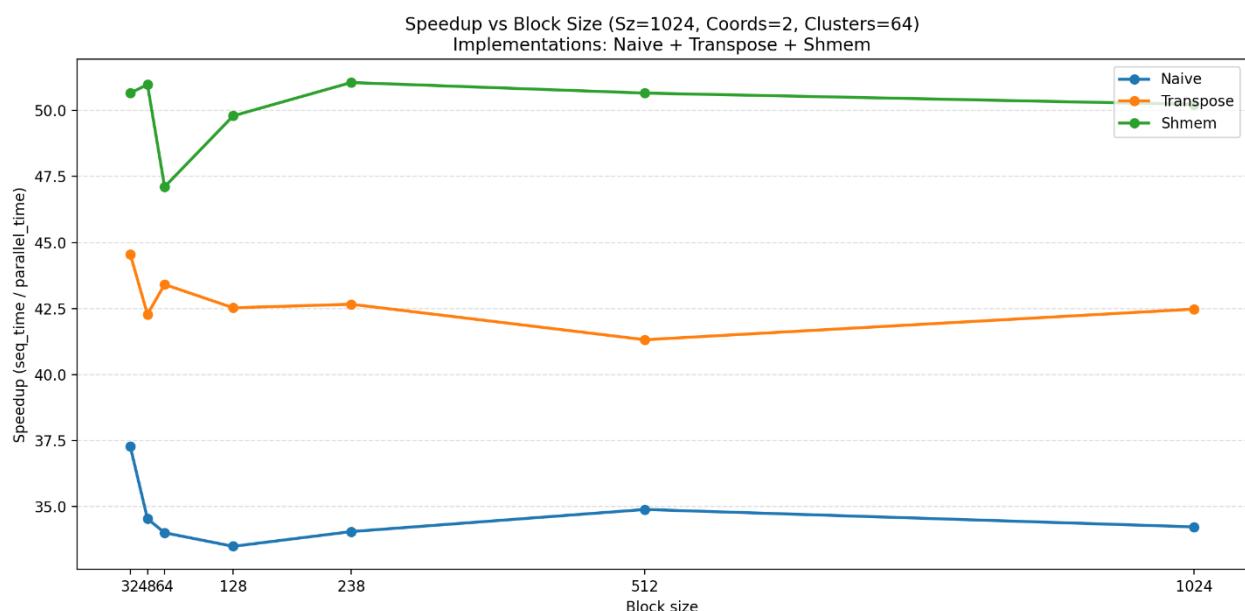
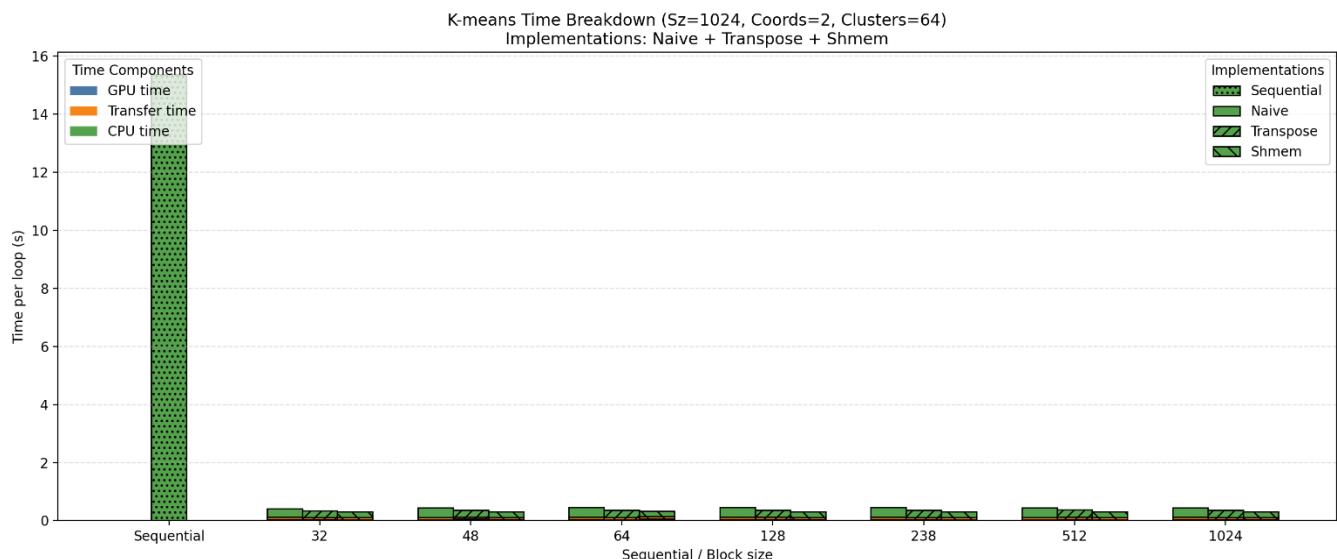
- CPU time (update_centroids): παραμένει σειριακό στην CPU στις τρεις εκδόσεις και θέτει όριο βάσει του Amdahl (όσο μικραίνει το GPU kernel, τόσο μεγαλύτερο ποσοστό του συνολικού χρόνου καταλαμβάνει το CPU update).

- Transfers time: για Coords=32 είναι σχετικά μικρό, αλλά είναι σταθερό overhead ανά επανάληψη (ιδίως η επιστροφή του membership), το οποίο δεν μειώνεται από τις βελτιστοποιήσεις μέσα στον kernel.

Συνεπώς, για Coords=32 το bottleneck «μετατοπίζεται» σταδιακά: αρχικά είναι κυρίως η απόδοση του GPU kernel (Naive), ενώ στις βελτιστοποιημένες εκδόσεις (Transpose/Shared) το όριο τίθεται ολοένα περισσότερο από το CPU update και το σταθερό κόστος επικοινωνίας ανά loop.

2. Τι αλλάζει για Coords=2 και είναι η προσέγγιση Shared κατάλληλη για arbitrary configs;

Αρχικά, παρουσιάζουμε τα διαγράμματα με 2 συντεταγμένες ακολούθως:



Με σταθερό dataset size (1024MB), όταν μειώνουμε τις διαστάσεις από 32 σε 2, ο αριθμός objects αυξάνεται περίπου κατά 16 \times (numObjs \propto 1/numCoords). Αυτό έχει δύο κρίσιμες συνέπειες:

1. Αυξάνεται δραστικά το μέγεθος του membership που πρέπει να επιστρέψει στη CPU σε κάθε επανάληψη ($D \rightarrow H$), άρα ο χρόνος transfers γίνεται πολύ πιο σημαντικός σε σχέση με το Coords=32.
2. Ταυτόχρονα, ο υπολογισμός απόστασης ανά object γίνεται ελαφρύτερος (μόνο 2 συντεταγμένες), άρα το GPU kernel έχει μικρότερο arithmetic work ανά element και η συνολική εκτέλεση τείνει να γίνεται λιγότερο compute-bound και πιο overhead/communication sensitive.

Τα διαγράμματα για Coords=2 επιβεβαιώνουν αυτή τη μετατόπιση: ενώ η Shared παραμένει η ταχύτερη υλοποίηση (Shared > Transpose > Naive), η διαφορά μεταξύ των GPU εκδόσεων προκύπτει πλέον κυρίως από σχετικά μικρότερες βελτιώσεις στο GPU time, επειδή ένα μεγαλύτερο ποσοστό του συνολικού χρόνου ανά loop ανήκει στις μεταφορές (και στο CPU update). Με άλλα λόγια, όταν το bottleneck είναι η επικοινωνία (membership transfer) και το σειριακό update, οι βελτιστοποιήσεις εντός του kernel έχουν περιορισμένο χώρο να αποδώσουν.

Γ. Συμπέρασμα για arbitrary configs

Η τεχνική shared memory για τα clusters είναι γενικά αποδοτική όταν:

- το (numClusters \times numCoords) χωράει σε shared ανά block, και
- υπάρχει αρκετή επαναχρησιμοποίηση/υπολογιστικό έργο ανά φόρτωση (ώστε το κόστος φόρτωσης + `__syncthreads` να αποσβεστεί).

Ωστόσο, δεν είναι καθολική αλήθεια για όλα τα configs: σε περιπτώσεις όπως Coords=2, όπου αυξάνεται έντονα το communication overhead (membership $D \rightarrow H$) και μειώνεται το arithmetic intensity του distance computation, το συνολικό bottleneck μετακινείται εκτός kernel. Τότε η Shared εξακολουθεί να βοηθά (μειώνει το GPU time), αλλά το συνολικό speedup περιορίζεται κυρίως από transfers και CPU update, δηλαδή από τμήματα που η Shared δεν μπορεί να βελτιώσει.

- **Full-Offload (All-GPU) Version**

A. Εισαγωγή

Στην έκδοση Full-Offload (All-GPU) μεταφέρουμε ολόκληρο το iterative μέρος του K-means στη GPU: όχι μόνο το assignment (εύρεση κοντινότερου cluster για κάθε object), αλλά και το update των centroids (συσσώρευση sums/counts και υπολογισμός νέων κέντρων). Στόχος είναι να εξαλειφθούν (i) το CPU load ανά επανάληψη (update_centroids στην CPU) και (ii) οι μεγάλες μεταφορές CPU↔GPU μέσα στο loop (ιδίως το D2H membership), ώστε το bottleneck να περιοριστεί στον καθαρό GPU υπολογισμό.

Ο κώδικας του αρχείου μας (cuda_kmeans_all_gpu.cu) παρατίθεται ακολούθως:

a5/cuda_kmeans_all_gpu.cu

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "kmeans.h"
5 #include "alloc.h"
6 #include "error.h"
7
8 #ifdef __CUDACC__
9 inline void checkCuda(cudaError_t e)
10 {
11     if (e != cudaSuccess)
12     {
13         // cudaGetErrorString() isn't always very helpful. Look up the error
14         // number in the cudaError enum in driver_types.h in the CUDA includes
15         // directory for a better explanation.
16         error("CUDA Error %d: %s\n", e, cudaGetErrorString(e));
17     }
18 }
19
20 inline void checkLastCudaError()
21 {
22     checkCuda(cudaGetLastError());
23 }
24 #endif
25
26 __device__ int get_tid()
27 {
28     return blockIdx.x * blockDim.x + threadIdx.x;
29 }
30
31 /* square of Euclid distance between two multi-dimensional points using column-base format
 */
32 __host__ __device__ inline static double euclid_dist_2_transpose(int numCoords,
33                                                               int numObjs,
34                                                               int numClusters,
35                                                               double *objects, //
36                                                               [numCoords][numObjs]
37                                                               double *clusters, //
38                                                               [numCoords][numClusters]
39                                                               int objectId,
40                                                               int clusterId)
41 {
42     int i;
43     double ans = 0.0;
44
45     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
46     clusters, but for column-base format!!! */
47     for (i = 0; i < numCoords; i++)
48     {
49         double objectVal = objects[i * numObjs + objectId];
50         double clusterVal = clusters[i * numClusters + clusterId];
```

```

49     double diff = objectVal - clusterVal;
50     ans += diff * diff;
51 }
52
53 return (ans);
54 }
55
56 __global__ static void find_nearest_cluster(int numCoords,
57                                             int numObjs,
58                                             int numClusters,
59                                             double *deviceObjects, // [numCoords]
60                                             [numObjs]
61                                             /*
62 TODO: If you choose to do (some of) the new centroid calculation here, you will need some
63 extra parameters here (from "update_centroids").
64 */
65                                             int *devicenewClusterSize,
66                                             double *devicenewClusters, // [numCoords]
67                                             [numClusters]
68                                             double *deviceClusters, // [numCoords]
69                                             int *deviceMembership, // [numObjs]
70                                             double *devdelta)
71 {
72     extern __shared__ double shmemClusters[];
73     // TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
74     int tid_in_block = threadIdx.x; // To ID του νήματος μέσα στο Block
75     int block_size = blockDim.x; // Πόσα νήματα έχει το Block
76     int total_cluster_doubles = numClusters * numCoords; // Συνολικά νούμερα προς αντιγραφή
77
78     // Κάθε νήμα αντιγράφει όσα στοιχεία του αναλογούν (με βήμα block_size)
79     for (int k = tid_in_block; k < total_cluster_doubles; k += block_size)
80     {
81         shmemClusters[k] = deviceClusters[k];
82     }
83
84     /* Συγχρονισμός (BARRIER) */
85
86     __syncthreads();
87
88     /* Get the global ID of the thread. */
89     int tid = get_tid();
90
91     /* TODO: Maybe something is missing here... should all threads run this? */
92     if (tid < numObjs)
93     {
94         int index, i;
95         double dist, min_dist;
96
97         /* find the cluster id that has min distance to object */
98         index = 0;
99         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using
100            clusters in shmem*/

```

```
97
98     min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
99                                         deviceObjects, shmemClusters,
100                                        tid, index);
101
102    for (i = 1; i < numClusters; i++)
103    {
104        dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters,
105                                         deviceObjects, shmemClusters,
106                                         tid, i);
107
108        /* no need square root */
109        if (dist < min_dist)
110        { /* find the min and its array index */
111            min_dist = dist;
112            index = i;
113        }
114    }
115
116    if (deviceMembership[tid] != index)
117    {
118        /* TODO: Maybe something is missing here... is this write safe? */
119        atomicAdd(devdelta, 1.0);
120    }
121
122    /* assign the deviceMembership to object objectId */
123    deviceMembership[tid] = index;
124
125    /* TODO: additional steps for calculating new centroids in GPU? */
126
127    atomicAdd(&devicenewClusterSize[index], 1);
128
129    for (int j = 0; j < numCoords; j++)
130    {
131        // Διαβάζουμε την τιμή του αντικειμένου (Coordinate j, Object tid)
132        double objVal = deviceObjects[j * numObjs + tid];
133
134        // Προσθέτουμε στο άθροισμα (Coordinate j, Cluster index)
135        atomicAdd(&devicenewClusters[j * numClusters + index], objVal);
136    }
137 }
138 }
139
140 __global__ static void update_centroids(int numCoords,
141                                         int numClusters,
142                                         int *devicenewClusterSize, // [numClusters]
143                                         double *devicenewClusters, // [numCoords]
144                                         [numClusters]
145                                         [numClusters])
146 {
147     /* Κάθε νήμα αναλαμβάνει ΜΙΑ τιμή (double) του πίνακα clusters.
148     Συνολικά νήματα = numCoords * numClusters
149     */
150 }
```

```

149 int tid = get_tid();
150 int total_elements = numCoords * numClusters;
151
152 if (tid < total_elements)
153 {
154     // Αποκαδικοποίηση του 1D tid σε 2D (Coordinate, Cluster)
155     // Layout: [numCoords][numClusters] --> index = coord * numClusters + cluster
156     int clusterId = tid % numClusters;
157     // int coordId = tid / numClusters; // Δεν το χρειαζόμαστε άμεσα για τον υπολογισμό,
αλλά για το reset
158
159     int count = devicenewClusterSize[clusterId];
160
161     // Υπολόγισε το νέο κέντρο (Average)
162     if (count > 0)
163     {
164         double sum = devicenewClusters[tid];
165         deviceClusters[tid] = sum / count;
166     }
167     // Av count == 0, κρατάμε την παλιά τιμή (ή δεν κάνουμε τίποτα), όπως και στον CPU
κώδικα
168
169     // RESET για τον επόμενο γύρο (Πολύ σημαντικό!)
170     // Μηδενίζουμε το άθροισμα που μόλις χρησιμοποιήσαμε
171     devicenewClusters[tid] = 0.0;
172 }
173 }
174
175 //
176 // -----
177 // DATA LAYOUT
178 //
179 // objects      [numObjs][numCoords]
180 // clusters     [numClusters][numCoords]
181 // dimObjects   [numCoords][numObjs]
182 // dimClusters  [numCoords][numClusters]
183 // newClusters  [numCoords][numClusters]
184 // deviceObjects [numCoords][numObjs]
185 // deviceClusters [numCoords][numClusters]
186 //
187 //
188 /* return an array of cluster centers of size [numClusters][numCoords] */
189 void kmeans_gpu(double *objects,    /* in: [numObjs][numCoords] */
190                 int numCoords,    /* no. features */
191                 int numObjs,     /* no. objects */
192                 int numClusters, /* no. clusters */
193                 double threshold,/* % objects change membership */
194                 long loop_threshold,/* maximum number of iterations */
195                 int *membership, /* out: [numObjs] */
196                 double *clusters, /* out: [numClusters][numCoords] */
197                 int blockSize)
198 {
199     double timing = wtime(), timing_internal, timer_min = 1e42, timer_max = 0;

```

```

200  double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
201  gpu_time = 0.0;
202  int loop_iterations = 0;
203  int i, j, index, loop = 0;
204  double delta = 0, *dev_delta_ptr; /* % of objects change their clusters */
205  /* TODO: Copy me from transpose version*/
206  double **dimObjects = (double **)calloc_2d(numCoords, numObjs, sizeof(double));      // 
207  calloc_2d(...) -> [numCoords][numObjs]
208  double **dimClusters = (double **)calloc_2d(numCoords, numClusters, sizeof(double)); // 
209  calloc_2d(...) -> [numCoords][numClusters]
210  double **newClusters = (double **)calloc_2d(numCoords, numClusters, sizeof(double));
211
212  printf("\n|-----Full-offload GPU Kmeans-----|\n\n");
213
214  /* TODO: Copy me from transpose version*/
215  for (i = 0; i < numObjs; i++)
216  {
217      for (j = 0; j < numCoords; j++)
218      {
219          dimObjects[j][i] = objects[i * numCoords + j];
220      }
221  }
222
223  double *deviceObjects;
224  double *deviceClusters, *devicenewClusters;
225  int *deviceMembership;
226  int *devicenewClusterSize; /* [numClusters]: no. objects assigned in each new cluster */
227
228  /* pick first numClusters elements of objects[] as initial cluster centers*/
229  for (i = 0; i < numCoords; i++)
230  {
231      for (j = 0; j < numClusters; j++)
232      {
233          dimClusters[i][j] = dimObjects[i][j];
234      }
235  }
236
237
238  /* initialize membership[] */
239  for (i = 0; i < numObjs; i++)
240      membership[i] = -1;
241
242  timing = wtime() - timing;
243  printf("t_alloc: %lf ms\n\n", 1000 * timing);
244  timing = wtime();
245  const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize :
246  numObjs;
247  const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
248  numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
249
250  /* Define the shared memory needed per block.
251     - BEWARE: We can overrun our shared memory here if there are too many
252       clusters or too many coordinates!
253     - This can lead to occupancy problems or even inability to run.

```

```
248     - Your exercise implementation is not requested to account for that (e.g. always
249     assume deviceClusters fit in shmemClusters */
250
251     const unsigned int clusterBlockSharedDataSize = numClusters * numCoords *
252     sizeof(double);
253
254     cudaDeviceProp deviceProp;
255     int deviceNum;
256     cudaGetDevice(&deviceNum);
257     cudaGetDeviceProperties(&deviceProp, deviceNum);
258
259     if (clusterBlockSharedDataSize > deviceProp.sharedMemPerBlock)
260     {
261         error("Your CUDA hardware has insufficient block shared memory to hold all cluster
262         centroids\n");
263     }
264
265     checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
266     checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
267     checkCuda(cudaMalloc(&devicenewClusters, numClusters * numCoords * sizeof(double)));
268     checkCuda(cudaMalloc(&devicenewClusterSize, numClusters * sizeof(int)));
269     checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
270     checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
271
272     timing = wtime() - timing;
273     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
274     timing = wtime();
275
276     checkCuda(cudaMemcpy(deviceObjects, dimObjects[0],
277                         numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
278     checkCuda(cudaMemcpy(deviceMembership, membership,
279                         numObjs * sizeof(int), cudaMemcpyHostToDevice));
280     checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
281                         numClusters * numCoords * sizeof(double), cudaMemcpyHostToDevice));
282     checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));
283     free(dimObjects[0]);
284
285     timing = wtime() - timing;
286     printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
287     timing = wtime();
288
289     do
290     {
291         timing_internal = wtime();
292         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
293         checkCuda(cudaMemset(devicenewClusterSize, 0, numClusters * sizeof(int)));
294         timing_gpu = wtime();
295         // printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
296         // shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDa-
297         taSize/1000);
298         // TODO: change invocation if extra parameters needed
299         find_nearest_cluster<<<numClusterBlocks, numThreadsPerClusterBlock,
300         clusterBlockSharedDataSize>>>(numCoords, numObjs, numClusters,
301
302         deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters, deviceMembership,
303         dev_delta_ptr);
```

```
295     cudaDeviceSynchronize();
296     checkLastCudaError();
297
298     gpu_time += wtime() - timing_gpu;
299
300     // printf("Kernels complete for itter %d, updating data in CPU\n", loop);
301
302     timing_transfers = wtime();
303     // TODO: Copy dev_delta_ptr to &delta
304     checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
305     transfers_time += wtime() - timing_transfers;
306
307     const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ?
308 blockSize : numCoords * numClusters;           /* TODO: can use different blocksize here if
309 deemed better */
310     const unsigned int update_centroids_dim_sz = (numCoords * numClusters +
311 update_centroids_block_sz - 1) / update_centroids_block_sz; /* TODO: calculate dim for
312 "update_centroids" */
313     timing_gpu = wtime();
314     // TODO: use dim for "update_centroids" and fire it
315     update_centroids<<<update_centroids_dim_sz, update_centroids_block_sz, 0>>>(numCoords,
316 numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);
317     cudaDeviceSynchronize();
318     checkLastCudaError();
319     gpu_time += wtime() - timing_gpu;
320
321     timing_cpu = wtime();
322     delta /= numObjs;
323     // printf("delta is %f - ", delta);
324     loop++;
325     // printf("completed loop %d\n", loop);
326     cpu_time += wtime() - timing_cpu;
327
328     timing_internal = wtime() - timing_internal;
329     if (timing_internal < timer_min)
330         timer_min = timing_internal;
331     if (timing_internal > timer_max)
332         timer_max = timing_internal;
333 } while (delta > threshold && loop < loop_threshold);
334
335     checkCuda(cudaMemcpy(membership, deviceMembership,
336                           numObjs * sizeof(int), cudaMemcpyDeviceToHost));
337     checkCuda(cudaMemcpy(dimClusters[0], deviceClusters,
338                           numClusters * numCoords * sizeof(double), cudaMemcpyDeviceToHost));
339
340     for (i = 0; i < numClusters; i++)
341     {
342         for (j = 0; j < numCoords; j++)
343         {
344             clusters[i * numCoords + j] = dimClusters[j][i];
345         }
346     }
347 }
```

```
344     timing = wtime() - timing;
345     printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
346 ms\n\t-> t_loop_max = %lf ms\n\t-> t_cpu_avg = %lf ms\n\t-> t_gpu_avg = %lf ms\n\t-> t_transfers_avg = %lf
347 ms\n\n|-----|\n",
348         loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
349         1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
350
350     char outfile_name[1024] = {0};
351     sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
352             numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
353     FILE *fp = fopen(outfile_name, "a+");
354     if (!fp)
355         error("Filename %s did not open successfully, no logging performed\n", outfile_name);
356     fprintf(fp, "%s,%d,%lf,%lf,%lf\n", "All_GPU", blockSize, timing / loop, timer_min,
357             timer_max);
358     fclose(fp);
359
359     checkCuda(cudaFree(deviceObjects));
360     checkCuda(cudaFree(deviceClusters));
361     checkCuda(cudaFree(devicenewClusters));
362     checkCuda(cudaFree(devicenewClusterSize));
363     checkCuda(cudaFree(deviceMembership));
364
365     return;
366 }
367 }
```

B. Υλοποίηση και Ορθότητα

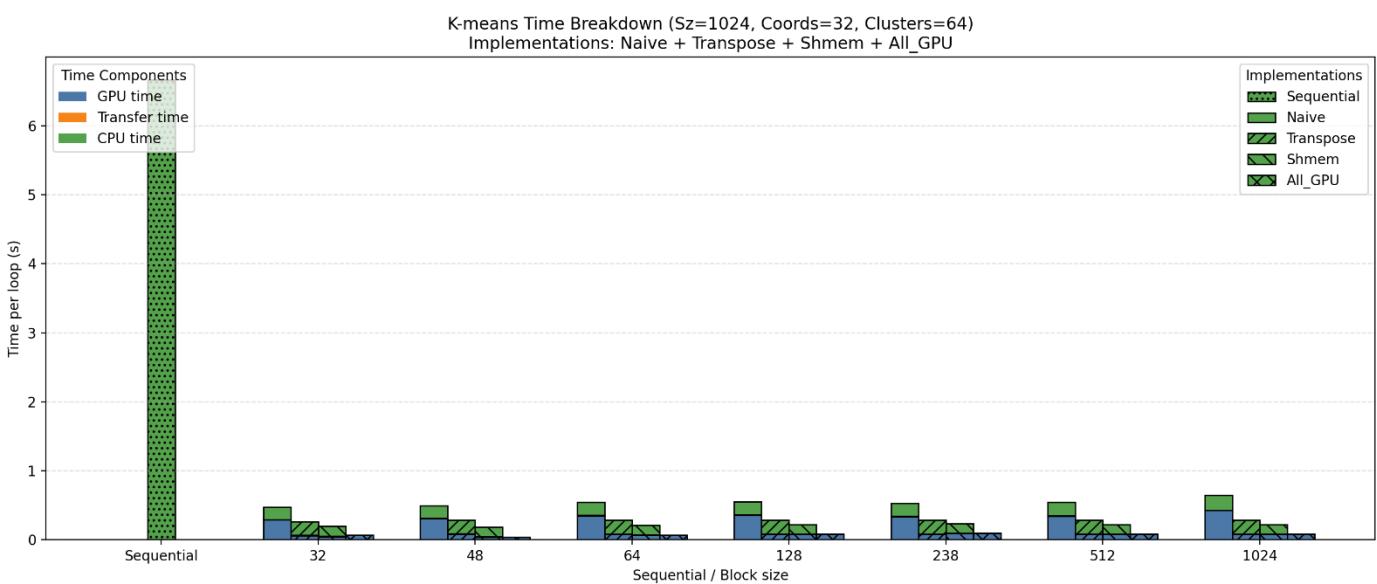
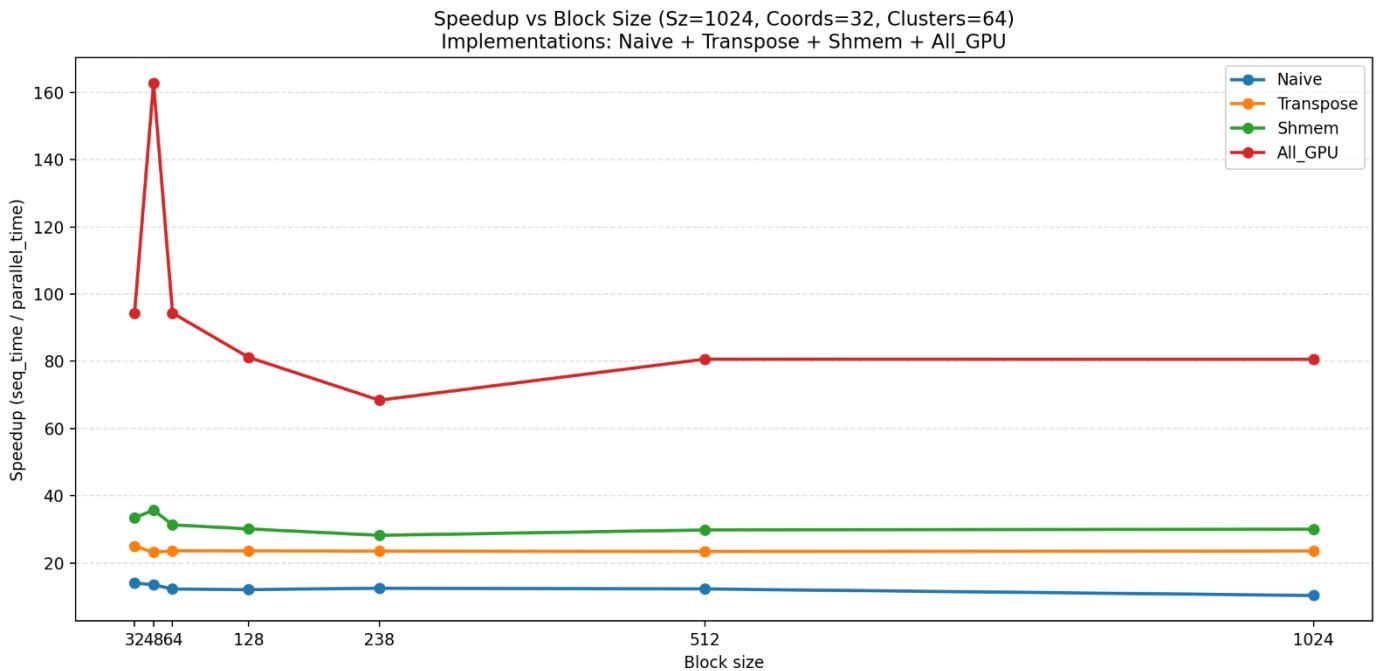
Η λογική της All-GPU υλοποίησης είναι να σπάσει το update_centroids σε βήματα που μπορούν να γίνουν ασφαλώς στη GPU χωρίς καθολικό barrier μέσα σε ένα kernel:

1. Μηδενισμός/αρχικοποίηση device arrays για newClusters_sums και newClusters_counts (και ό,τι άλλο χρειάζεται).
2. Kernel ανάθεσης (find_nearest_cluster): κάθε thread επεξεργάζεται ένα object, υπολογίζει αποστάσεις προς όλα τα clusters, ενημερώνει το membership και ταυτόχρονα συσσωρεύει τη συνεισφορά του object στο cluster που ανήκει.
 - Η συσσώρευση sums/counts γίνεται με atomics σε global μνήμη (atomicAdd σε counts και σε κάθε διάσταση του sum), ώστε να αποφευχθούν race conditions.
 - Τα cluster centers μπορούν να φορτωθούν ανά block στη shared memory (όπως στη shared έκδοση) ώστε οι επαναλαμβανόμενες αναγνώσεις κατά τον υπολογισμό αποστάσεων να γίνονται από on-chip μνήμη.
3. Kernel τελικοποίησης centroids: για κάθε cluster (και διάσταση) υπολογίζεται ο μέσος όρος (sum/count) και παράγονται τα νέα centers για το επόμενο iteration.
4. Για τον τερματισμό του while-loop, στον host επιστρέφει μόνο το delta (ή/και ελάχιστη μετα-πληροφορία). Έτσι, οι μεταφορές μέσα στο loop ελαχιστοποιούνται.

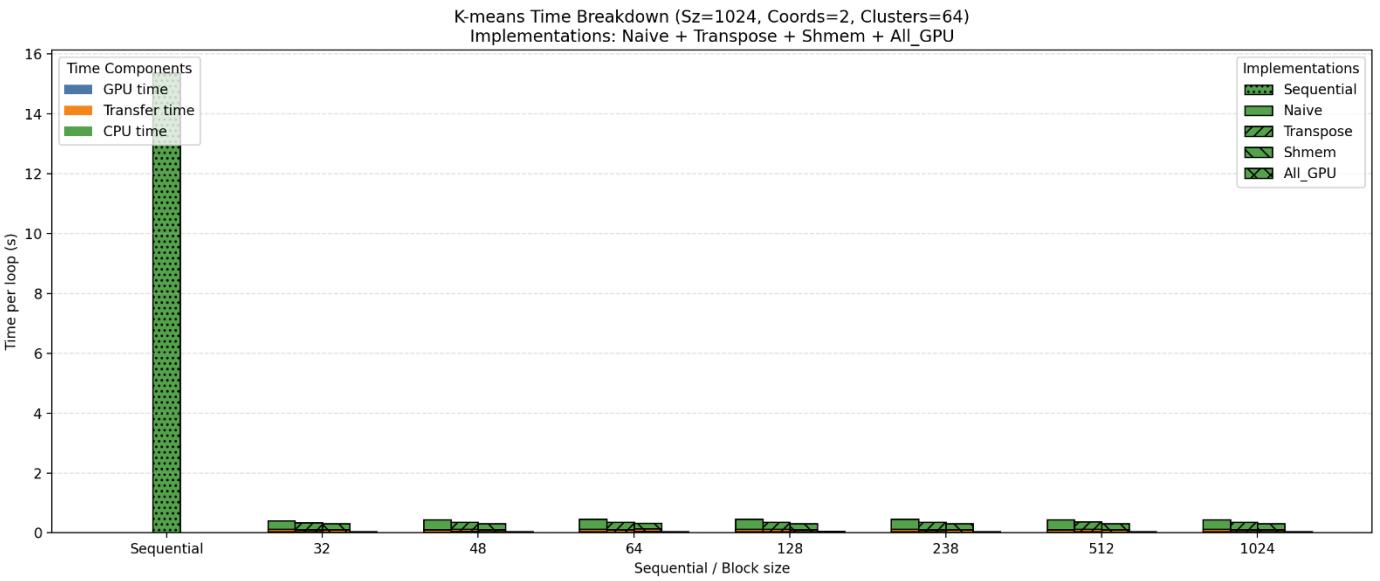
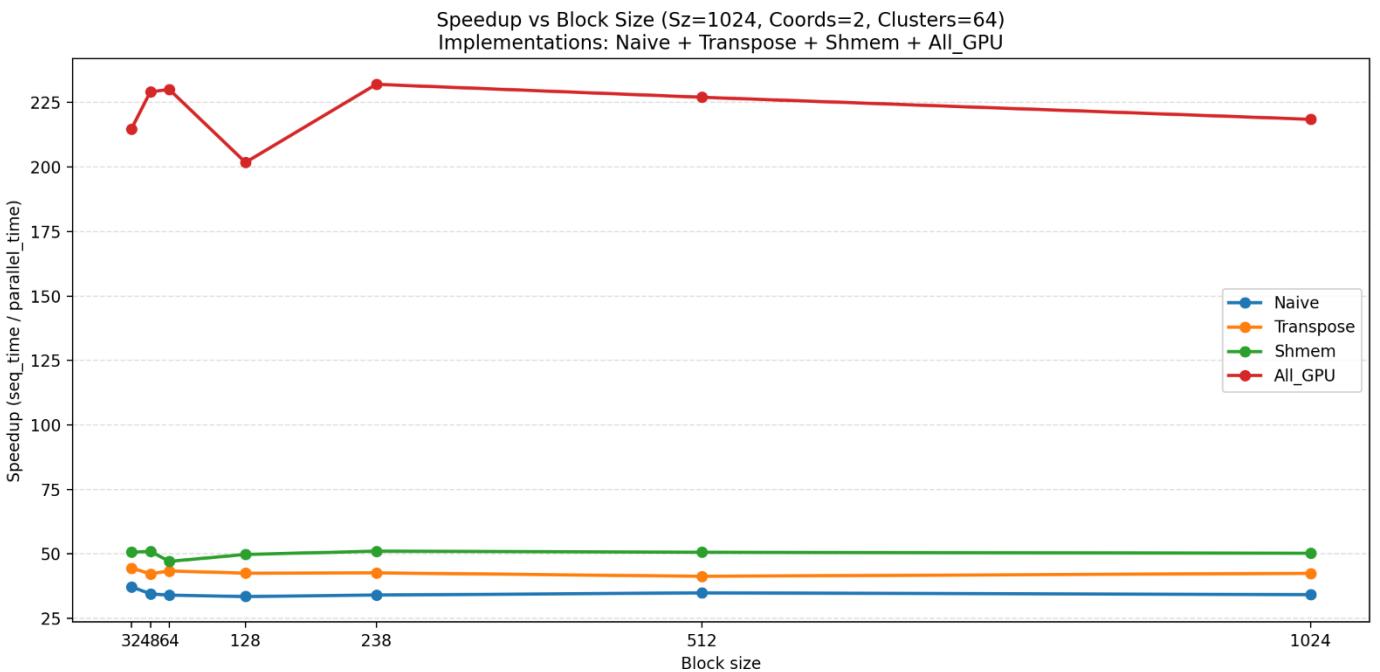
Η χρήση atomics εξασφαλίζει το σωστό αποτέλεσμα στα αθροίσματα, παρά το ταυτόχρονο update από πολλά threads. Ο απαιτούμενος καθολικός συγχρονισμός επιτυγχάνεται φυσικά με τη διάσπαση σε πολλαπλά kernels (τα kernels εκτελούνται σειριακά ως προς τη σειρά κλήσης τους), κάτι που αντικαθιστά την απουσία global barrier μέσα σε έναν kernel.

Γ. Παρουσίαση Διαγραμμάτων

(α) Configuration {1024,32,64,10}



(β) Configuration {1024,2,64,10}



Δ. Ερμηνεία Διαγραμμάτων – Απάντηση Ερωτημάτων (1) και (2)

(1) Επίδοση All-GPU σε σχέση με naive/transpose/shared (και για τα δύο configurations)

Τα διαγράμματα δείχνουν ότι η All-GPU υπερέχει σημαντικά έναντι όλων των προηγούμενων εκδόσεων και στα δύο configurations. Αυτό είναι αναμενόμενο, καθώς:

- Το CPU time μέσα στο loop (update_centroids στην CPU) πρακτικά μηδενίζεται.
- Το transfer time μέσα στο loop μειώνεται δραστικά, επειδή δεν απαιτείται πλέον αντιγραφή του membership πίσω στην CPU σε κάθε επανάληψη. Στον host επιστρέφει μόνο το delta, άρα οι per-iteration μεταφορές περιορίζονται σε πολύ μικρά δεδομένα (σε αντίθεση με τις naive/transpose/shared όπου υπάρχει O(N) Device→Host membership).

Άρα, το iterative μέρος παύει να είναι υβριδικό (GPU assignment + CPU update + transfers) και γίνεται σχεδόν αποκλειστικά GPU workload, το οποίο ταιριάζει καλύτερα στη φιλοσοφία throughput της GPU. Το νέο bottleneck προέρχεται κυρίως στον GPU χρόνο (distance computations + atomics για sums/counts).

Σημείωση: Η εκτέλεση έχει σχετικά μικρό warp divergence (κυρίως bounds checks και η απλή ενημέρωση membership), άρα το bottleneck προέρχεται κυρίως από global memory traffic και atomic contention, όχι από branching

(2) Παίζει διαφορετικό ρόλο το block_size και γιατί;

Ναι, στην All-GPU έκδοση το block_size επηρεάζει έντονα την επίδοση και αυτό φαίνεται καθαρά στα διαγράμματα (ιδίως στο Coords=32, όπου υπάρχει πολύ μεγάλη διακύμανση speedup ανά block size). Ο λόγος είναι ότι, αφού σχεδόν μηδενίζονται τα per-loop transfers και το CPU update, το συνολικό runtime καθορίζεται σχεδόν αποκλειστικά από καθαρά GPU φαινόμενα, τα οποία εξαρτώνται άμεσα από το block_size:

1. Occupancy / latency hiding: Το block_size καθορίζει πόσα blocks/warps μπορούν να είναι resident ανά SM. Με μεγαλύτερα blocks αυξάνονται οι απαιτήσεις σε threads/SM (και σε registers ανά block), άρα συχνά μειώνονται

τα ταυτόχρονα resident blocks/warps. Όταν μειωθούν τα active warps, η GPU κρύβει χειρότερα τη latency της global μνήμης και η επίδοση πέφτει.

2. Πίεση σε registers και shared: Στον assignment kernel κάθε thread κάνει σχετικά βαριά δουλειά (loop σε numClusters και numCoords). Αυτό τείνει να αυξάνει τα registers/thread. Όσο μεγαλώνει το block_size, το συνολικό register footprint/block μεγαλώνει και μπορεί να περιορίσει τα blocks/SM. Αν χρησιμοποιείται και shared caching για τα clusters, το shared ανά block είναι σταθερό, αλλά σε συνδυασμό με τα registers/threads μπορεί να κλειδώσει το occupancy.
3. Atomic contention στο update_centroids: Η All-GPU κάνει συσσώρευση sums/counts με atomics. Το block_size επηρεάζει πόσα threads πηγαίνουν ταυτόχρονα τους ίδιους counters/αθροίσματα (ιδίως όταν πολλά objects καταλήγουν στα ίδια clusters). Μεγαλύτερη ταυτόχρονη πίεση σε atomics οδηγεί σε serialization και απώλεια throughput, άρα μπορεί να εμφανίζεται ισχυρό sweet spot σε συγκεκριμένα block sizes. Θεωρητικά, για να μειωθεί το contention, μια κλασική τεχνική είναι block-level partial sums/counts σε shared memory (με reduction) και στη συνέχεια ένα μόνο atomicAdd ανά (block, cluster, coord) προς global μνήμη
4. Warp efficiency / μη ιδανικά block sizes: Επειδή η εκτέλεση γίνεται σε warps των 32 threads, block sizes που δεν είναι πολλαπλάσια του 32 δημιουργούν μερικώς γεμάτα warps (wasted lanes). Αυτό μπορεί να επιδεινώσει την αποδοτικότητα και να αλλάξει το ισοζύγιο occupancy–contention.

Συμπέρασμα: Σε All-GPU, το block_size δεν είναι δευτερεύον όπως μπορεί να φαινόταν σε Transpose-only σενάρια. Αντίθετα καθορίζει άμεσα το occupancy και το atomic contention (και άρα τον GPU χρόνο), οπότε εμφανίζονται έντονα βέλτιστα σημεία και απότομες μεταβολές στην επίδοση, ειδικά στο Coords=32 όπου αυξάνεται το έργο/νήμα και το πλήθος atomicAdds ανά object. Συνολικά, το block_size καθορίζει ένα trade-off ανάμεσα σε occupancy/latency hiding και σε contention/πόρους (registers/shared), οπότε εμφανίζεται φυσιολογικά sweet spot.

E. Είναι το update_centroids κατάλληλο για GPUs; Και γιατί η All-GPU διαφέρει τόσο σε επίδοση;

To update_centroids δεν είναι ιδανικό GPU kernel με την έννοια του τέλειου, ανεξάρτητου per-thread υπολογισμού: απαιτεί συνάθροιση (reduction) πολλών contributions σε κοινά arrays (sums/counts), άρα:

- Εισάγει συγχρονισμό μέσω atomics και contention (πολλά threads ενημερώνουν τα ίδια clusters), που μπορεί να περιορίσει το scaling.
- Περιλαμβάνει στάδια που απαιτούν καθολικό συγχρονισμό (π.χ. πρώτα να ολοκληρωθούν όλα τα sums/counts πριν γίνει η διαίρεση για τα νέα centroids), κάτι που μας αναγκάζει να το σπάσουμε σε πολλαπλά kernels.

Παρόλα αυτά, η All-GPU είναι πολύ ταχύτερη συνολικά, επειδή αφαιρεί τα προηγούμενα dominant bottlenecks:

- Δεν πληρώνουμε πλέον CPU χρόνο ανά iteration για update_centroids.
- Δεν πληρώνουμε πλέον μεγάλο D2H transfer του membership ανά iteration (ούτε το H2D των clusters σε κάθε γύρο).

Άρα, ακόμη κι αν το update_centroids στη GPU “δεν είναι τέλειο” και έχει atomic overhead, το συνολικό κέρδος από την εξάλειψη CPU+PCIe κόστους είναι πολύ μεγαλύτερο, με αποτέλεσμα την εντυπωσιακή αύξηση speedup έναντι naive transpose/shared.

ΣΤ. Τι διαφέρει μεταξύ των δύο configurations και πώς αιτιολογείται η διαφορά επίδοσης;

Το κρίσιμο σημείο είναι ότι το “Size=1024” αντιστοιχεί σε σταθερό συνολικό μέγεθος dataset, άρα αλλάζει ο αριθμός των objects όταν αλλάζει το Coords:

- Με Coords=2, κάθε object έχει πολύ λιγότερα bytes → έχουμε πολύ περισσότερα objects.
- Με Coords=32, κάθε object είναι “βαρύτερο” → έχουμε πολύ λιγότερα objects.

Αυτό επηρεάζει και τη σειριακή και την παράλληλη εκτέλεση, αλλά και το είδος bottleneck:

- Coords=2: τεράστιος αριθμός objects → πολύ υψηλός παραλληλισμός (η GPU γεμίζει εύκολα), και στις παλιές εκδόσεις υπήρχαν πολύ μεγάλα per-iteration transfers (membership), τα οποία η All-GPU εξαφανίζει. Έτσι βλέπουμε πολύ υψηλό speedup.
- Coords=32: λιγότερα objects → λιγότερος παραλληλισμός και μεγαλύτερη σημασία στα σταθερά/overhead κόστη (kernel launches, reset/finalize kernels). Επιπλέον, στην All-GPU αυξάνεται η δουλειά ανά object (περισσότερες διαστάσεις σε distance + περισσότερα atomicAdds ανά object για sums), οπότε ο GPU χρόνος ανεβαίνει και το speedup περιορίζεται σε σχέση με το Coords=2.

Z. Συμπεράσματα

Η Full-Offload (All-GPU) εκδοχή επιβεβαιώνει ότι η μεγαλύτερη πηγή απώλειας στις προηγούμενες υλοποιήσεις ήταν το CPU work + PCIe transfers μέσα στο iterative loop. Με το πλήρες offload, το πρόγραμμα γίνεται πραγματικά GPU-centric και το bottleneck μεταφέρεται κυρίως στον GPU χρόνο και ειδικά στο κόστος των atomics του update_centroids. Παρ' όλα αυτά, η συνολική επίδοση βελτιώνεται θεαματικά και η All-GPU αποτελεί το φυσικό επόμενο βήμα μετά τις βελτιώσεις πρόσβασης μνήμης (transpose) και επαναχρησιμοποίησης δεδομένων (shared).

▪ Γενικά Συμπεράσματα

Η συνολική εικόνα που προκύπτει είναι ότι η επίδοση δεν καθορίζεται μόνο από το πόσο γρήγορο είναι το kernel, αλλά από το πού βρίσκεται κάθε φορά το bottleneck (PCIe transfers, CPU τμήμα, global memory traffic, atomics). Έτσι, όσον αφορά τις 4 εκδόσεις (προγράμματα) που υλοποιήσαμε:

1. Στη naïve εκδοχή, το βασικό κέρδος προκύπτει από τη μεταφορά του assignment step στη GPU, όμως η συνολική επιτάχυνση περιορίζεται από το ότι παραμένουν σημαντικά κόστη εκτός GPU: (i) οι μεταφορές δεδομένων (ιδίως η μεταφορά του membership προς τον host σε κάθε επανάληψη, που είναι $O(N)$) και (ii) το update_centroids στην CPU. Έτσι, ακόμη κι αν το kernel βελτιωθεί, το speedup περιορίζεται λόγω Amdahl (μη παραλληλοποιημένο/μη offloaded μέρος).
2. Η transpose εκδοχή δείχνει καθαρά τη σημασία της διάταξης δεδομένων και της συν-αξιοποίησης της μνήμης (coalescing). Με το transposed layout, οι προσπελάσεις γίνονται πιο συνεκτικές (coalesced) και μειώνεται η σπατάλη bandwidth, με αποτέλεσμα αισθητή βελτίωση σε σχέση με τη naïve, ειδικά όταν το workload είναι memory-bound. Παράλληλα, ο ρόλος του block_size γίνεται πιο επηρεάζει περισσότερο τη GPU (occupancy/latency hiding), καθώς η διαφορά από transfers/CPU αρχίζει να μειώνεται.
3. Στη shared εκδοχή, η μεταφορά των centroids σε shared memory λειτουργεί ως user-managed cache και μειώνει περαιτέρω τα global reads, οδηγώντας σε επιπλέον επιτάχυνση όταν το configuration το επιτρέπει. Ωστόσο, η τεχνική δεν έρχεται χωρίς κόστος: περιορίζεται από τη διαθέσιμη shared memory και μπορεί να επηρεάσει το occupancy, άρα υπάρχει πρακτικό όριο ως προς τα K-Coords και το block_size. Το συμπέρασμα είναι ότι η shared memory δίνει κέρδος όταν υπάρχει επανάχρηση δεδομένων ανά block, αλλά απαιτεί προσεκτικό διάβασμα των resource constraints.
4. Η all-GPU εκδοχή επιβεβαιώνει ότι το μεγαλύτερο κέρδος έρχεται όταν εξαλειφθούν τα υβριδικά κομμάτια μέσα στο iterative loop. Αφαιρώντας το per-iteration Device→Host membership και μεταφέροντας και το update_centroids στη GPU, μειώνεται δραστικά το transfer/CPU bottleneck, και ο συνολικός χρόνος κυριαρχείται πλέον από καθαρά GPU κόστη. Σε αυτήν τη φάση, το block_size επηρεάζει κυρίως μέσω occupancy/latency hiding, πίεσης σε registers και (κυρίως στο update_centroids) μέσω atomic contention. Ειδικά στο update_centroids, τα atomics μπορούν να αποτελέσουν σημαντικό περιορισμό, κάτι που εξηγεί γιατί το

all-GPU δεν κλιμακώνει πάντα όσο ιδανικά θα περιμέναμε χωρίς πρόσθετες τεχνικές μείωσης contention (π.χ. block-level partial sums σε shared και λιγότερα atomics προς global).

Όσον αφορά τις συντεταγμένες και το block size:

1. Η σύγκριση Coords=32 με Coords=2 δείχνει ότι το ίδιο «Size» δεν συνεπάγεται ίδιο υπολογιστικό κόστος: με μικρότερο Coords προκύπτει πολύ μεγαλύτερο πλήθος points N, άρα αυξάνει έντονα το workload του assignment και το μέγεθος του membership ($O(N)$). Αυτό μεταβάλλει το bottleneck: στο Coords=2 είναι πολύ πιο εύκολο να κυριαρχήσουν bandwidth/atomics ή ακόμη και οι μεταφορές membership (στις υβριδικές εκδοχές), ενώ στο Coords=32 το προφίλ είναι πιο ισορροπημένο και οι per-iteration μεταφορές centroids είναι πράγματι αμελητέες.
2. Τέλος, από τη μελέτη του block_size προκύπτει ότι δεν υπάρχει μία σωστή τιμή: η βέλτιστη επιλογή είναι αποτέλεσμα trade-off ανάμεσα σε occupancy, latency hiding, register/shared pressure και contention. Γι' αυτό βλέπουμε sweet spots και όχι μονοτονικές τάσεις, ενώ οι πολύ μικρές ή πολύ μεγάλες τιμές μπορούν να υποβαθμίσουν την επίδοση (είτε λόγω χαμηλής αξιοποίησης είτε λόγω περιορισμού πόρων).

Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.
Ιανουάριος 2026

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

ΑΝΑΦΟΡΑ 5^{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: 19.01.2026

▪ Εισαγωγή

Στην παρούσα άσκηση μελετάμε την παραλληλοποίηση και τη βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης, αξιοποιώντας το πρότυπο MPI. Στόχος μας είναι να σχεδιάσουμε και να υλοποιήσουμε αποδοτικές κατανεμημένες εκδόσεις δύο προβλημάτων (K-means, 2D heat-transfer), να αξιολογήσουμε την κλιμακωσιμότητά τους σε πολλαπλές διεργασίες και να ερμηνεύσουμε τα αποτελέσματα με βάση τη θεωρία επικοινωνίας/συγχρονισμού σε συστήματα κατανεμημένης μνήμης, του μαθήματος.

Το πρώτο μέρος αφορά τον αλγόριθμο K-means. Εξετάζουμε πώς διαμοιράζονται τα δεδομένα (points) στις διεργασίες, πώς υπολογίζονται τοπικά οι συνεισφορές για τα νέα centroids και πώς συνδυάζονται μέσω συλλογικών επικοινωνιών (reduce/allreduce) ώστε να ενημερώνονται τα κέντρα ανά επανάληψη. Η ανάλυση εστιάζει τόσο στη λειτουργική ορθότητα όσο και στη συμπεριφορά χρόνου εκτέλεσης όταν αυξάνει ο αριθμός διεργασιών.

Το δεύτερο μέρος αφορά τη διάδοση θερμότητας (2D heat transfer). Σχεδιάζουμε κατανομή του πλέγματος (grid decomposition) και την απαιτούμενη ανταλλαγή οριακών τιμών (halo exchange) μεταξύ γειτονικών διεργασιών, καθώς και μηχανισμό ελέγχου σύγκλισης που βασίζεται σε global reductions. Παρουσιάζουμε μετρήσεις τόσο με ενεργό έλεγχο σύγκλισης όσο και με απενεργοποιημένη σύγκλιση για σταθερό αριθμό επαναλήψεων, ώστε να αξιολογηθεί καθαρά η κλιμάκωση και να απομονωθεί το κόστος επικοινωνίας.

Στην αναφορά καταγράφουμε το πώς υλοποιήθηκαν τα παραπάνω (βασικές σχεδιαστικές επιλογές, κλήσεις MPI, συγχρονισμός), και παρουσιάζουμε πειραματικά αποτελέσματα με διαγράμματα χρόνου και speedup για διαφορετικό αριθμό διεργασιών. Ιδιαίτερη έμφαση δίνεται στη σωστή μεθοδολογία χρονομέτρησης (διάκριση συνολικού χρόνου, χρόνου υπολογισμού και χρόνου επικοινωνίας), καθώς και στην ερμηνεία των τάσεων κλιμάκωσης με βάση έννοιες όπως συλλογικές επικοινωνίες, τοπολογίες/γειτονική επικοινωνία και κόστος συγχρονισμού.

Σημείωση: Έχουμε λάβει υπόψιν μας όλες τις υποδείξεις – διευκρινίσεις της άσκησης. Έτσι, οι χρόνοι που μετρήθηκαν και τα διαγράμματα συνιστούν τον μέσο όρο από τρία runs, όπως ζητούνταν (πιο σημαντική διευκρίνιση).

■ Ενότητα 4.1 – Αλγόριθμος K-means με MPI

A. Εισαγωγή

Στο πρώτο μέρος της άσκησης υλοποιούμε μια κατανεμημένη έκδοση του αλγορίθμου K-means, αξιοποιώντας το πρότυπο του MPI. Ο K-means είναι ένας επαναληπτικός αλγόριθμος ομαδοποίησης που:

1. αναθέτει κάθε σημείο (object) στο πλησιέστερο κέντρο συστάδας (centroid)
2. επανυπολογίζει τα centroids ως μέσο όρο των σημείων που τους ανήκουν
3. επαναλαμβάνει μέχρι σύγκλιση ή μέχρι μέγιστο αριθμό επαναλήψεων.

Στόχος μας είναι να παραλληλοποιήσουμε τη ροή σε αρχιτεκτονική και προγραμματιστικό μοντέλο κατανεμημένης μνήμης, ώστε κάθε διεργασία να επεξεργάζεται μόνο ένα τμήμα των δεδομένων και να συνεργάζεται με τις υπόλοιπες μέσω συλλογικών επικοινωνιών (collectives) για τον υπολογισμό των νέων centroids και του κριτηρίου σύγκλισης. Στην αναφορά παρουσιάζουμε:

- τον τρόπο κατανομής των δεδομένων (data decomposition) και την επιλογή των εντολών MPI,
- το πώς συνδυάζονται οι τοπικές συνεισφορές στις global (κοινές) μεταβλητές (global reductions),
- και (στη συνέχεια) τη μελέτη της κλιμάκωσης του προγράμματος, με μετρήσεις και διαγράμματα χρόνου και speedup, για διαφορετικό αριθμό διεργασιών.

B. Υλοποίηση

Η υλοποίηση του αλγορίθμου βασίζεται σε σαφή διάσπαση του προβλήματος ως προς τα δεδομένα (objects): κάθε rank κρατά μόνο το δικό του υποσύνολο σημείων, ενώ τα centroids παραμένουν κοινά (ίδια τιμή) σε όλα τα ranks, σε κάθε επανάληψη. Ειδικότερα, επισημαίνουμε τα εξής σημεία στην υλοποίησή μας:

1) Παραγωγή/Κατανομή δεδομένων (dataset_generation στο file_io.c)

- Ο συνολικός αριθμός σημείων υπολογίζεται από το dataset_size (MB), το numCoords και το sizeof(double).
- Η κατανομή γίνεται ισότιμα (block distribution) με χρήση της μεταβλητής remainder: κάθε rank παίρνει είτε $\lfloor N/P \rfloor$ είτε $\lfloor N/P \rfloor + 1$ αντικείμενα, ώστε να μοιράζεται ομοιόμορφα ο φόρτος όταν το N δεν διαιρείται ακριβώς με P . Αυτό αποφεύγει το έντονο load imbalance, ειδικά σε πολλά ranks.
- Το rank 0 δημιουργεί ντετερμινιστικά το dataset (rand_r με seed = i για κάθε object), ώστε να μπορούμε να αναπαράγουμε τα δεδομένα και τις μετρήσεις.
- Η κατανομή των δεδομένων γίνεται με MPI_Scatterv από το rank 0, χρησιμοποιώντας sendcounts/displs (objects × coords). Η χρήση Scatterv (αντί Scatter) είναι κρίσιμη, επειδή τα ranks μπορεί να λαμβάνουν διαφορετικό πλήθος αντικειμένων λόγω του remainder.

2) Αρχικοποίηση κέντρων (main.c)

- Τα αρχικά centroids επιλέγονται από τα πρώτα σημεία numClusters. Η επιλογή γίνεται μόνο από το rank 0 (για να υπάρχει μοναδική πηγή αρχικοποίησης) και στη συνέχεια διαχέεται σε όλα τα ranks με MPI_Bcast.
- Με αυτόν τον τρόπο όλα τα ranks ξεκινούν από τα ίδια centroids, άρα η εξέλιξη του αλγορίθμου παραμένει συνεπής.

3) Κύριος επαναληπτικός βρόχος K-means (kmeans.c)

Σε κάθε επανάληψη κάθε rank εκτελεί καθαρά τοπικό υπολογισμό πάνω στα δικά του objects:

- Ανάθεση (assignment): για κάθε local object, βρίσκουμε το πλησιέστερο centroid (εύρεση ελάχιστης τετραγωνικής Ευκλείδειας απόστασης) και ενημερώνουμε το membership[i].
- Τοπικές συσσωρεύσεις για update centroids:
 1. rank_newClusterSize[c]: πόσα τοπικά σημεία ανατέθηκαν στο cluster c,
 2. rank_newClusters[c, j]: άθροισμα των συντεταγμένων (sums) των τοπικών σημείων του cluster c.

Κατόπιν, συνδυάζουμε τα τοπικά αποτελέσματα στις global μεταβλητές, με reductions:

- MPI_Allreduce για rank_newClusters → newClusters (SUM) και rank_newClusterSize → newClusterSize (SUM). Η κύρια επιλογή εδώ ήταν η χρήση του Allreduce (και όχι Reduce) ώστε:
 1. να αποφεύγεται το bottleneck στο rank 0,
 2. και να έχουν όλα τα ranks άμεσα τα αθροίσματα/δεδομένα ώστε να ενημερώσουν τα centroids τοπικά, χωρίς επιπλέον επικοινωνία.
- Ενημέρωση centroids: για κάθε cluster c, αν newClusterSize[c] > 0, θέτουμε clusters[c, j] = newClusters[c, j] / newClusterSize[c].

4) Κριτήριο σύγκλισης

- Κάθε rank υπολογίζει rank_delta = πλήθος τοπικών αντικειμένων που άλλαξαν cluster σε αυτή την επανάληψη.
- Έπειτα κάνουμε MPI_Allreduce(rank_delta → delta, SUM) ώστε να έχουμε τον συνολικό αριθμό αλλαγών.
- Τέλος, το delta κανονικοποιείται ώστε να προκύψει το κλάσμα των αλλαγών και συγκρίνεται με το threshold, ενώ υπάρχει και ανώτατο όριο επαναλήψεων loop_threshold.

5) Συγκέντρωση αποτελεσμάτων membership (main.c)

- Το τελικό membership ανά αντικείμενο υπολογίζεται τοπικά σε κάθε rank (membership[rank_numObjs]).
- Για να σχηματιστεί το πλήρες διάνυσμα membership για όλα τα αντικείμενα στον root, χρησιμοποιούμε MPI_Gatherv (αντί Gather), ξανά λόγω άνισων μεγεθών (remainder).
- Τα recvcounts/displs υπολογίζονται στο rank 0 σύμφωνα με την ίδια πολιτική κατανομής και μεταδίδονται με MPI_Bcast, ώστε όλα τα ranks να καλέσουν το MPI_Gatherv.

Συνολικά, οι κρίσιμες σχεδιαστικές επιλογές ήταν:

1. Data decomposition ανά object (όχι ανά coords), ώστε κάθε rank να εκτελεί ανεξάρτητο, γραμμικό υπολογισμό πάνω σε συνεχόμενη μνήμη.
2. Χρήση Scatterv/Gatherv, για σωστή διαχείριση $N \bmod P$.
3. Χρήση Allreduce για τα κοινά μεγέθη (sums, counts, delta), ώστε να αποφεύγεται το master bottleneck και να κρατάμε τα centroids συγχρονισμένα χωρίς πρόσθετα broadcasts ανά επανάληψη.

Τα αρχεία αυτά (file_io.c, main.c και kmeans.c) που περιέχουν την υλοποίησή μας παρατίθενται ακολούθως:

a6/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 #include <mpi.h>
9
10 #include "kmeans.h"
11
12 double * dataset_generation(int numObjs, int numCoords, long *rank_numObjs)
13 {
14     double * objects = NULL, * rank_objects = NULL;
15     long i, j, k;
16
17     // Random values that will be generated will be between 0 and 10.
18     double val_range = 10;
19
20     int rank, size;
21     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
22     MPI_Comm_size(MPI_COMM_WORLD, &size);
23
24     /*
25      * TODO: Calculate number of objects that each rank will examine (*rank_numObjs)
26      */
27     *rank_numObjs = numObjs / size;
28     if (rank < numObjs % size) {
29         (*rank_numObjs)++;
30     }
31
32
33     /* allocate space for objects[][] and read all objects */
34     int sendcounts[size], displs[size];
35     if (rank == 0) {
36         objects = (typeof(objects)) malloc(numObjs * numCoords * sizeof(*objects));
37
38         /*
39          * TODO: Calculate sendcounts and displs, which will be used to scatter data to
40          * each rank.
41          * Hint: sendcounts: number of elements sent to each rank
42          *        displs: displacement of each rank's data
43          */
44
45         int count = 0;
46         int remainder = numObjs % size;
47
48         for (k = 0; k < size; k++) {
49
50             int k_numObjs = numObjs / size;
51             if (k < remainder) {
52                 k_numObjs++;
53             }
54
55             sendcounts[k] = k_numObjs;
56             displs[k] = count;
57             count += k_numObjs;
58         }
59
60         MPI_Scatterv(objects, sendcounts, displs, MPI_DOUBLE, 0, MPI_DOUBLE, rank,
61                      MPI_COMM_WORLD);
62     }
63 }
```

```
52         }
53
54         sendcounts[k] = k_numObjs * numCoords;
55         displs[k] = count;
56         count += sendcounts[k];
57     }
58 }
59
60 /*
61  * TODO: Broadcast the sendcounts and displs arrays to other ranks
62  */
63 MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
64 MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);
65
66
67 /* allocate space for objects[][] (for each rank separately) and read all objects */
68 rank_objects = (typeof(rank_objects)) malloc((*rank_numObjs) * numCoords *
69 sizeof(*rank_objects));
70
71 /* rank 0 will generate data for the objects array. This array will be used later to
72 scatter data to each rank. */
73 if (rank == 0) {
74     for (i=0; i<numObjs; i++)
75     {
76         unsigned int seed = i;
77         for (j=0; j<numCoords; j++)
78         {
79             objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) *
80             val_range;
81             if (_debug && i == 0)
82                 printf("object[i=%ld][j=%ld]=%f\n", i, j, objects[i*numCoords + j]);
83         }
84     }
85
86     /*
87      * TODO: Scatter objects to every rank. (hint: each rank may receive different number
88      * of objects)
89      */
90     MPI_Scatterv(objects, sendcounts, displs, MPI_DOUBLE, rank_objects, sendcounts[rank],
91 MPI_DOUBLE, 0, MPI_COMM_WORLD);
92
93     if (rank == 0)
94         free(objects);
95
96     return rank_objects;
97 }
```

```

a6/kmeans/kmeans.c

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

```

```

51         double * clusters)          /* out: [numClusters][numCoords] */
52 {
53     int i, j;
54     int index, loop=0;
55     double timing = 0;
56
57     /* Every variable has its "rank_" version, which is used to store local data,
58      * and its "new" version, which is used to store global data.
59      */
60     double rank_delta, delta = 0;           // fraction of objects whose clusters
change in each loop
61     int * rank_newClusterSize, * newClusterSize; // [numClusters]: no. objects assigned in
each new cluster
62     double * rank_newClusters, *newClusters;    // [numClusters][numCoords]
63
64     // Get rank of this process
65     int rank;
66     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
67
68     // initialize membership
69     for (i=0; i<numObjs; i++)
70         membership[i] = -1;
71
72     // initialize rank_newClusterSize and rank_newClusters to all 0
73     rank_newClusterSize = (typeof(rank_newClusterSize)) malloc(numClusters,
sizeof(*rank_newClusterSize));
74     rank_newClusters    = (typeof(rank_newClusters)) malloc(numClusters * numCoords,
sizeof(*rank_newClusters));
75     newClusterSize      = (typeof(newClusterSize)) malloc(numClusters,
sizeof(*newClusterSize));
76     newClusters         = (typeof(newClusters)) malloc(numClusters * numCoords,
sizeof(*newClusters));
77
78     timing = wtime();
79     do {
80         // before each loop, set cluster data to 0
81         for (i=0; i<numClusters; i++) {
82             for (j=0; j<numCoords; j++)
83                 rank_newClusters[i*numCoords + j] = 0.0;
84             rank_newClusterSize[i] = 0;
85         }
86
87         rank_delta = 0.0;
88
89         for (i=0; i<numObjs; i++) {
90             // find the array index of nearest cluster center
91             index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords],
clusters);
92
93             // if membership changes, increase rank_delta by 1
94             if (membership[i] != index)
95                 rank_delta += 1.0;
96
97             // assign the membership to object i

```

```

98         membership[i] = index;
99
100        // update new cluster centers : sum of objects located within
101        rank_newClusterSize[index]++;
102        for (j=0; j<numCoords; j++)
103            rank_newClusters[index*numCoords + j] += objects[i*numCoords + j];
104    }
105
106    /*
107     * TODO: Perform reduction of cluster data (rank_newClusters, rank_newClusterSize)
108     * from local arrays to shared.
109     */
110    MPI_Allreduce(rank_newClusters, newClusters, numClusters * numCoords, MPI_DOUBLE,
111    MPI_SUM, MPI_COMM_WORLD);
112    MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT, MPI_SUM,
113    MPI_COMM_WORLD);
114
115
116    // average the sum and replace old cluster centers with newClusters
117    for (i=0; i<numClusters; i++) {
118        if (newClusterSize[i] > 0) {
119            for (j=0; j<numCoords; j++) {
120                clusters[i*numCoords + j] = newClusters[i*numCoords + j] /
121                newClusterSize[i];
122            }
123        }
124    }
125
126    /*
127     * TODO: Perform reduction from rank_delta variable to delta variable, that will
128     * be used for convergence check.
129     */
130    MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
131
132
133    // Get fraction of objects whose membership changed during this loop. This is used
134    // as a convergence criterion.
135    delta /= numObjs;
136
137    loop++;
138    //printf("\r\tcompleted loop %d", loop);
139    //fflush(stdout);
140
141    } while (delta > threshold && loop < loop_threshold);
142
143    timing = wtime() - timing;
144    if (rank == 0) fprintf(stdout, "      nloops = %3d    (total = %7.4fs)  (per loop =
145    %7.4fs)\n", loop, timing, timing/loop);
146
147
148    free(rank_newClusters);
149    free(rank_newClusterSize);
150    free(newClusters);
151    free(newClusterSize);
152}
153

```

```

a6/kmeans/main.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>      /* getopt() */
8 #include <mpi.h>
9
10 int _debug;
11 #include "kmeans.h"
12
13 static void usage(char *argv0) {
14     char *help =
15         "Usage: %s [switches]\n"
16         "      -c num_clusters      : number of clusters (must be > 1)\n"
17         "      -s size                : size of examined dataset\n"
18         "      -n num_coords          : number of coordinates\n"
19         "      -t threshold            : threshold value (default : 0.001)\n"
20         "      -l loop_threshold       : iterations threshold (default : 10)\n"
21         "      -d                      : enable debug mode\n"
22         "      -h                      : print this help information";
23     fprintf(stderr, help, argv0);
24     exit(-1);
25 }
26
27 int main(int argc, char **argv)
28 {
29     long i, j, opt;
30     extern char* optarg;
31     extern int optind;
32
33     long    numClusters=0, numCoords=0, numObjs=0;
34     long    rank_numObjs=0;
35     int    * membership; // [rank_numObjs] this array will contain membership
information for this rank's objects
36     int    * tot_membership; // [numObjs]      this array will contain membership
information for all objects
37     double * objects;      // [numObjs * numCoords] data objects
38     double * clusters;     // [numClusters * numCoords] cluster center
39     double  dataset_size = 0, threshold;
40     long    loop_threshold;
41     double  io_timing_read;
42
43     /* some default values */
44     _debug        = 0;
45     threshold     = 0.001;
46     loop_threshold = 10;
47     numClusters   = 0;
48
49     while ( (opt = getopt(argc,argv,"n:t:l:c:s:dh")) != EOF) {
50         switch (opt) {

```

```

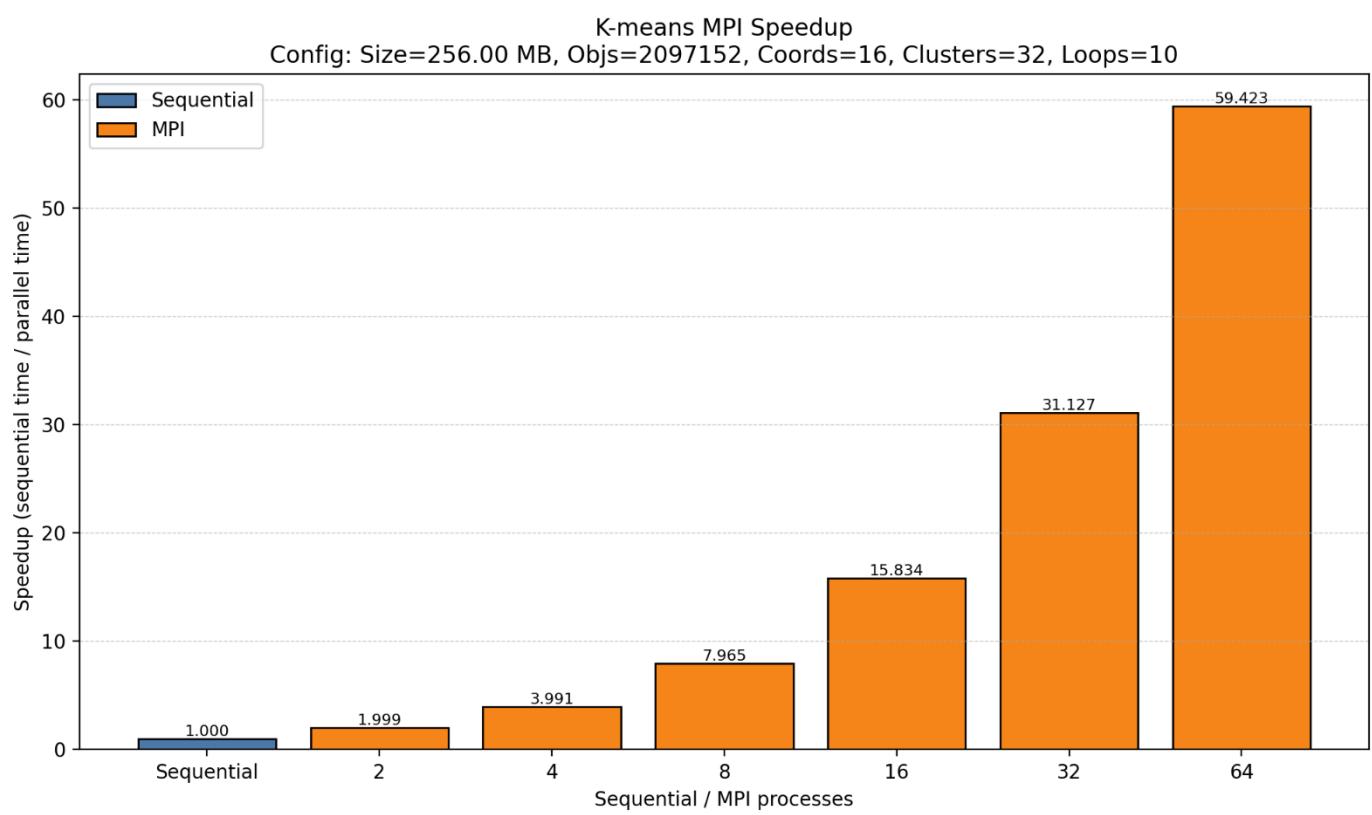
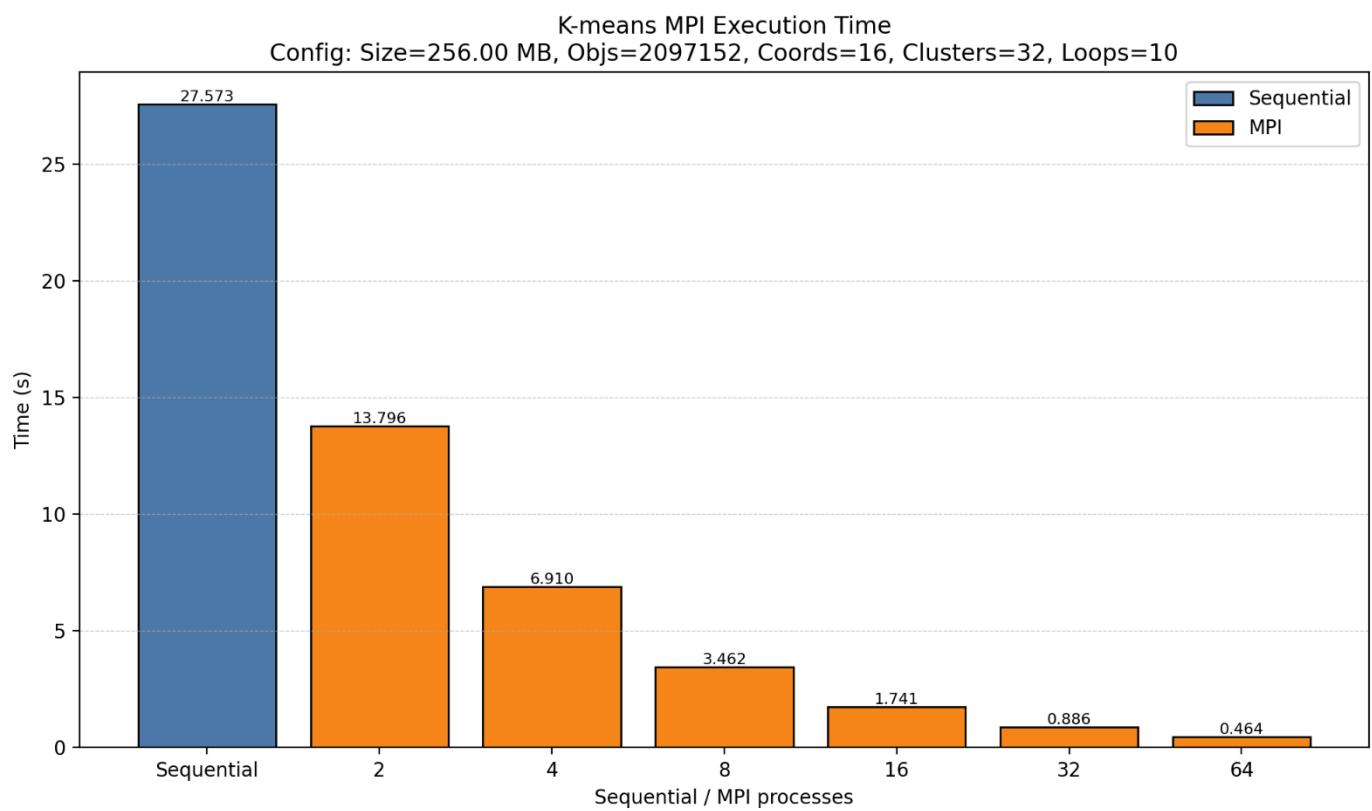
51     case 'c': numClusters = atol(optarg);
52         break;
53     case 't': threshold=atof(optarg);
54         break;
55     case 'l': loop_threshold=atol(optarg);
56         break;
57     case 's': dataset_size=atof(optarg);
58         break;
59     case 'n': numCoords=atol(optarg);
60         break;
61     case 'd': _debug = 1;
62         break;
63     case 'h':
64         default: usage(argv[0]);
65             break;
66     }
67 }
68 if (numClusters <= 1) {
69     usage(argv[0]);
70 }
71
72 int rank, size;
73 MPI_Init(&argc,&argv);
74 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
75 MPI_Comm_size(MPI_COMM_WORLD,&size);
76
77 numObjs = (dataset_size*1024*1024) / (numCoords*sizeof(double));
78
79 if (numObjs < numClusters) {
80     if (rank == 0) printf("Error: number of clusters must be larger than the number of
81 data points to be clustered.\n");
82     MPI_Finalize();
83     return 1;
84 }
85 if (rank == 0) printf("dataset_size = %.2f MB      numObjs = %ld      numCoords = %ld
86 numClusters = %ld\n", dataset_size, numObjs, numCoords, numClusters);
87
88 // Allocate space for clusters (coordinates of cluster centers)
89 clusters = (double*) malloc(numClusters * numCoords * sizeof(double));
90
91 // The first numClusters elements are selected as initial centers. Only rank 0 needs
92 // to calculate this, and later broadcast it to all ranks.
93 if (rank == 0) {
94     for (i=0; i<numClusters; i++)
95         for (j=0; j<numCoords; j++)
96             clusters[i*numCoords + j] = objects[i*numCoords + j];
97
98     // check initial cluster centers for repetition
99     if (check_repeated_clusters(numClusters, numCoords, clusters) == 0) {
100         printf("Error: some initial clusters are repeated. Please select distinct
initial centers\n");
101         MPI_Finalize();

```

```
101         return 1;
102     }
103     /*
104     printf("Initial cluster centers:\n");
105     for (i=0; i<numClusters; i++) {
106         printf("(0) clusters[%ld] = ",i);
107         for (j=0; j<numCoords; j++)
108             printf(" %6.6f", clusters[i*numCoords + j]);
109         printf("\n");
110     }
111 */
112 }
113 /*
114 * TODO: Broadcast initial cluster positions to all ranks
115 */
116 MPI_Bcast(clusters, numClusters * numCoords, MPI_DOUBLE, 0, MPI_COMM_WORLD);
117
118
119
120 // membership: the cluster id for each data object
121 membership = (int*) malloc(rank_numObjs * sizeof(int));
122 tot_membership = (int*) malloc(numObjs * sizeof(int));
123
124 // start the core computation
125 /*
126 * TODO: Fix number of objects that this kmeans function call will process
127 */
128 kmeans(objects, numCoords, rank_numObjs, numClusters, threshold, loop_threshold,
membership, clusters);
129
130 /*
131 if (rank == 0) {
132     printf("Final cluster centers:\n");
133     for (i=0; i<numClusters; i++) {
134         printf("clusters[%ld] = ",i);
135         for (j=0; j<numCoords; j++)
136             printf(" %6.6f ", clusters[i*numCoords + j]);
137         printf("\n");
138     }
139 }
140 */
141
142 // Gather membership information from all ranks to tot_membership
143 int recvcounts[size], displs[size];
144 if (rank == 0) {
145     /* TODO: Calculate recvcounts and displs, which will be used to gather data from
each rank.
146         * Hint: recvcounts: number of elements received from each rank
147         *       displs: displacement of each rank's data
148         */
149     int sum_disp = 0;
150     int remainder = numObjs % size;
151
152     for (j = 0; j < size; j++) {
```

```
153     // Υπολογισμός πόσα αντικείμενα περιμένουμε από το rank 'j'  
154     int j_numObjs = numObjs / size;  
155     if (j < remainder) {  
156         j_numObjs++;  
157     }  
158  
159     recvcounts[j] = j_numObjs; // Εδώ είναι σκέτα αντικείμενα (int)  
160     displs[j] = sum_disp;  
161     sum_disp += recvcounts[j];  
162 }  
163 }  
164  
165 /*  
166 * TODO: Broadcast the recvcounts and displs arrays to other ranks.  
167 */  
168 MPI_Bcast(recvcounts, size, MPI_INT, 0, MPI_COMM_WORLD);  
169 MPI_Bcast(displs, size, MPI_INT, 0, MPI_COMM_WORLD);  
170  
171 /*  
172 * TODO: Gather membership information from every rank. (hint: each rank may send  
173 different number of objects)  
174 */  
175 MPI_Gatherv(membership, rank_numObjs, MPI_INT, tot_membership, recvcounts, displs,  
MPI_INT, 0, MPI_COMM_WORLD);  
176  
177  
178 if (_debug && rank == 0)  
179     for (i = 0; i < numObjs; ++i)  
180         fprintf(stderr, "%d\n", tot_membership[i]);  
181  
182 free(objects);  
183 free(membership);  
184 free(tot_membership);  
185 free(clusters);  
186  
187 MPI_Finalize();  
188 return 0;  
189 }  
190 }
```

Γ. Μετρήσεις και Διαγράμματα



Δ. Συμπεράσματα Μετρήσεων

Από τα διαγράμματα χρόνου εκτέλεσης και speedup (Config: Size=256MB, Obs=2,097,152, Coords=16, Clusters=32, Loops=10) παρατηρούμε εξαιρετικά καλή κλιμάκωση του MPI K-means, σχεδόν γραμμική (ιδανική), σε όλο το εύρος διεργασιών μέχρι και 64. Ο σειριακός χρόνος είναι 27.573 s, ενώ με 2/4/8/16/32/64 διεργασίες ο χρόνος μειώνεται σε 13.796/6.910/3.462/1.741/0.886/0.464 s αντίστοιχα. Αυτό μεταφράζεται σε speedup 1.999x, 3.991x, 7.965x, 15.834x, 31.127x και 59.423x, δηλαδή μικρή απόκλιση από την ιδανική κλιμάκωση, ακόμα και σε 64 διεργασίες.

Η συμπεριφορά αυτή είναι πλήρως αναμενόμενη αν λάβουμε υπόψη τη δομή του αλγορίθμου και τη θεωρία του παραλληλισμού κατανεμημένης μνήμης, καθώς:

1) Κυριαρχία υπολογιστικού τμήματος

Το βασικό κόστος ανά επανάληψη στον K-means είναι η ανάθεση (assignment): για κάθε σημείο υπολογίζονται αποστάσεις από όλα τα centroids. Η πολυπλοκότητα ανά iteration είναι περίπου $O(N \cdot K \cdot d)$ (N objects, K clusters, d coords). Με τις παραμέτρους ($N \approx 2.1M$, $K=32$, $d=16$) το workload είναι πολύ μεγάλο και έτσι επιτυγχάνεται πολύ υψηλός παραλληλισμός ως προς τα objects. Με data decomposition ανά object, κάθε rank εκτελεί περίπου N/P ανεξάρτητους υπολογισμούς, άρα ο χρόνος υπολογισμού μειώνεται $\sim 1/P$.

2) Μικρός όγκος επικοινωνίας ανά iteration

Στο τέλος κάθε iteration απαιτείται συγχρονισμένη ενημέρωση των centroids και του κριτηρίου σύγκλισης. Αυτό υλοποιείται με συλλογικές επικοινωνίες (MPI_Allreduce) πάνω σε σχετικά λίγες και μικρές κοινές μεταβλητές:

- Τα αθροίσματα των νέων centroids έχουν μέγεθος $K \cdot d = 32 \cdot 16 = 512$ doubles (μερικά KB).
- Τα counts των clusters έχουν μέγεθος K (λίγες δεκάδες bytes/ints).
- Επιπλέον γίνεται ένα Allreduce για το delta (αριθμός αλλαγών membership).

Με βάση το κλασικό μοντέλο κόστους επικοινωνίας (latency/bandwidth, α-β model), το Allreduce έχει κόστος που αυξάνει περίπου με $\log(P)$ στα περισσότερα δέντρα/αλγορίθμους υλοποίησης. Εδώ όμως το μήνυμα είναι μικρό, οπότε (α) το

bandwidth cost είναι πολύ χαμηλό και (β) το πρόσθετο latency/synchronization overhead παραμένει μικρό σε σχέση με το τεράστιο compute phase που κλιμακώνει με $1/P$. Το αποτέλεσμα είναι ότι η επικοινωνία δεν κυριαρχεί τον υπολογισμό, άρα διατηρούμε υψηλή επίδοση.

3) Επίδραση συγχρονισμού και συλλογικών επικοινωνιών στην απόκλιση από το ιδανικό

Παρότι η κλιμάκωση είναι σχεδόν ιδανική, το speedup υπολείπεται ελαφρά του P όσο αυξάνουμε τις διεργασίες (π.χ. 64: 59.4 \times αντί 64 \times). Αυτό αποδίδεται κυρίως σε:

- Συλλογικές επικοινωνίες (Allreduce) που λειτουργούν ως φράγμα (implicit synchronization): κάθε iteration ολοκληρώνεται μόνο όταν φτάσουν όλα τα ranks στο ίδιο σημείο.
- Αυξανόμενο κόστος latency με το πλήθος διεργασιών (περισσότερα communication steps και πιθανή επικοινωνία όταν απλώνουμε τα ranks σε πολλούς κόμβους).
- Σταθερά κόστη ανά iteration (overheads) που δεν κλιμακώνονται με $1/P$ (π.χ. bookkeeping, ενημέρωση δομών, overhead MPI runtime).
- Στη λογική του νόμου του Amdahl, η συνολική επιτάχυνση περιορίζεται από οποιοδήποτε «μη παραλληλοποιήσιμο» ή μη κλιμακούμενο κομμάτι. Εδώ το σειριακό κλάσμα είναι εξαιρετικά μικρό, άρα ο μέγιστος παραλληλισμός είναι πολύ υψηλός - κάτι που επιβεβαιώνεται από το ότι η αποδοτικότητα παραμένει μεγάλη ακόμα και σε 64 διεργασίες (περίπου 93%: 59.4/64).

4) Load balance και κατανομή δεδομένων

Η κατανομή των objects έγινε ισοζυγισμένα (block distribution με remainder μέσω Scatterv), οπότε η διαφορά φορτίου ανά rank είναι το πολύ ένα object. Εφόσον το κύριο κόστος είναι ίδιο ανά object (υπολογισμός αποστάσεων προς όλα τα centroids), το load imbalance πρακτικά μηδενίζεται. Αυτό περιορίζει το φαινόμενο όπου κάποια ranks καθυστερούν και κρατούν πίσω τα υπόλοιπα στο Allreduce.

Συνολικά, τα αποτελέσματα δείχνουν ότι ο K-means στο συγκεκριμένο configuration είναι ιδιαίτερα κατάλληλος για strong scaling σε MPI: το μεγαλύτερο μέρος της εργασίας είναι data-parallel, η επικοινωνία ανά iteration είναι μικρού όγκου (κυρίως συλλογικά reductions), και η κατανομή φορτίου είναι καλά

ισοζυγισμένη. Η μικρή πτώση αποδοτικότητας σε μεγάλες τιμές P ερμηνεύεται από την αυξανόμενη σχετική συμμετοχή των σταθερών overheads και των συλλογικών επικοινωνιών (latency/synchronization), όπως προβλέπει η θεωρία των συστημάτων κατανεμημένης μνήμης.

E. Σύγκριση με OpenMP

Για τη σύγκριση με την υλοποίηση σε OpenMP και αρχιτεκτονική κοινής μνήμης, χρησιμοποιούμε τα αποτελέσματα της αναφοράς του OpenMP της άσκησης 2 (με τη δική μας σημειολογία, parlab05_report_a2_final.pdf) για τον K-means.

Σημείωση: Οι απόλυτοι σειριακοί χρόνοι των δύο υλοποιήσεων (MPI vs OpenMP) δεν είναι απαραίτητα απευθείας συγκρίσιμοι (διαφορετικός κώδικας/μεθοδολογία timing), όμως οι τάσεις κλιμάκωσης και τα bottlenecks είναι ξεκάθαρα.

1) OpenMP – Naive shared

Στην αφελή αυτή εκδοχή, οι ενημερώσεις των κοινόχρηστων πινάκων (newClusters, newClusterSize και delta) προστατεύονται με atomic operations. Αυτό δημιουργεί έντονο synchronization bottleneck: πολλά νήματα προσπαθούν να ενημερώσουν τα ίδια cache lines, προκαλώντας serialization.

- No affinity: Tseq=12.65 s, ενώ για 2/4/8/16/32/64 νήματα παίρνουμε 9.92/12.70/11.85/10.46/11.31/9.14 s (μη ικανοποιητική κλιμάκωση, ακόμη και χειρότερη από το σειριακό σε σημεία).
- Default affinity: Tseq=12.64 s, και για 4/8 νήματα βελτιώνεται (5.78/3.76 s), αλλά μετά χειροτερεύει (16/32/64: 8.56/12.01/10.07 s).

Η βελτίωση με affinity συμφωνεί με τη θεωρία NUMA: το pinning μειώνει τις remote προσπελάσεις και κρατά καλύτερο locality, όμως δεν λύνει το βασικό πρόβλημα της σχεδίασης (atomics σε shared δομές), άρα το contention παραμένει και περιορίζει την κλιμάκωση.

2) OpenMP – Copied clusters + reduction

Στη βελτιστοποιημένη αυτή εκδοχή, κάθε νήμα γράφει σε ιδιωτικά (thread-local) arrays (local_newClusters/local_newClusterSize) και στη συνέχεια γίνεται συνδυασμός των αποτελεσμάτων (reduction/merge). Έτσι:

- Αφαιρείται το συνεχές atomic contention από τον κυρίως βρόχο.
- Οι ενημερώσεις γίνονται κατά βάση τοπικές (καλύτερη cache locality) και ο συγχρονισμός περιορίζεται σε μικρό/συγκεντρωμένο τμήμα.

Αποτελέσματα: Tseq=12.64 s και χρόνοι 2/4/8/16/32/64:

7.36/3.92/2.13/1.14/0.60/0.48 s. Το speedup φτάνει $\sim 26.3\times$ στα 64 νήματα, με υψηλή αποδοτικότητα μέχρι τα 32 νήματα ($\sim 21.1\times$), και πτώση πέρα από εκεί, κάτι αναμενόμενο από shared-memory περιορισμούς (memory bandwidth saturation, αυξημένο overhead/NUMA effects).

3) MPI – Distributed-memory

Στη δική μας υλοποίηση, το workload διαμοιράζεται ανά object, ενώ η επικοινωνία ανά iteration είναι μικρού όγκου (κυρίως MPI_Allreduce για sums/counts/delta). Αυτό ταιριάζει ιδανικά στο μοντέλο latency/bandwidth: μικρά collectives (μερικά KB) με κόστος που αυξάνει αργά με P (συνήθως $\sim \log P$), ενώ το compute μειώνεται περίπου με $1/P$.

Μετρήσεις (MPI): Tseq=27.573 s και χρόνοι 2/4/8/16/32/64:

13.796/6.910/3.462/1.741/0.886/0.464 s, με speedup 1.999 \times έως 59.423 \times . Η αποδοτικότητα παραμένει πολύ υψηλή ακόμη και στα 64 ranks, επειδή:

- αποφεύγονται shared-memory contention/coherence bottlenecks (ιδιωτική μνήμη ανά process),
- και το communication volume για centroids/delta είναι μικρό σε σχέση με το compute $O(N \cdot K \cdot d)$.

Συμπεράσματα σύγκρισης

1. Το OpenMP μπορεί να είναι εξαιρετικά καλό εντός ενός κόμβου, αλλά απαιτεί προσεκτικό σχεδιασμό για να μειωθούν τα synchronization/NUMA bottlenecks, ιδίως η αποφυγή atomics σε hot paths και η χρήση thread-local δομών + reduction.
2. Η MPI κλιμακώνει πολύ καλύτερα σε πολλούς κόμβους, ειδικά όταν το πρόβλημα είναι compute-intense και η επικοινωνία ανά iteration είναι μικρού μεγέθους (όπως στον K-means με K=4 μικρό).
3. Πρακτικά, η θεωρία υποδεικνύει ως βέλτιστη γενίκευση σε clusters έναν υβριδικό σχεδιασμό: MPI μεταξύ κόμβων και OpenMP εντός κόμβου, ώστε να εκμεταλλεύμαστε και την τοπολογία του συστήματος και να περιορίζουμε τόσο το inter-node communication όσο και το intra-node contention.

▪ Ενότητα 4.2 – Διάδοση Θερμότητας σε 2 Διαστάσεις

A. Εισαγωγή

Στην ενότητα αυτή θα υλοποιήσουμε την επίλυση του δισδιάστατου προβλήματος διάδοσης θερμότητας, σε αρχιτεκτονική κατανεμημένης μνήμης, με χρήση του πρωτοκόλλου MPI. Μετά τη διακριτοποίηση του συνεχούς προβλήματος σε πλέγμα, η ενημέρωση των εσωτερικών κελιών οδηγεί σε ένα επαναληπτικό σχήμα τύπου Jacobi, όπου κάθε νέο κελί προκύπτει από τον μέσο όρο των τεσσάρων γειτονικών τιμών του (πάνω/κάτω/αριστερά/δεξιά). Η μέθοδος αυτή είναι ιδιαίτερα κατάλληλη για την ανάδειξη του παραλληλισμού, καθώς ο υπολογισμός κάθε κελιού εξαρτάται μόνο από τιμές της προηγούμενης επανάληψης και έτσι γίνεται πολύ compute intense.

Ειδικότερα, στόχος μας είναι:

- να κατανείμουμε το δισδιάστατο πλέγμα σε υποπεριοχές και να αναθέσουμε καθεμία από αυτές σε μία διεργασία MPI,
- να υλοποιήσουμε την απαραίτητη ανταλλαγή «halo/ghost» ορίων μεταξύ των γειτονικών διεργασιών, ώστε κάθε διεργασία να μπορεί να υπολογίζει σωστά τα κελιά που βρίσκονται κοντά στα όριά της,
- και να χρονομετρήσουμε τον χρόνο υπολογισμού και τον συνολικό χρόνο (υπολογισμοί + επικοινωνίες) όπως απαιτεί η άσκηση.

Λόγω περιορισμένου χρόνου, δεν υλοποιήθηκαν τα bonus ερωτήματα (Gauss–Seidel SOR και Red–Black SOR), αν και προσπαθήσαμε (όπως φαίνεται και στον scirouter), απλά απέτυχαν στην σύγκλιση και έτσι τα παραλείψαμε, καθώς θεωρήσαμε ότι δεν ήταν ορθά και το debugging ήταν αρκετά χρονοβόρο. Η υλοποίηση καλύπτει την εκδοχή Jacobi με δυνατότητα (μέσω compile-time flags) είτε ελέγχου σύγκλισης ανά C επαναλήψεις είτε εκτέλεσης για σταθερό πλήθος επαναλήψεων ($T=256$), σύμφωνα με τα ζητούμενα.

B. Υλοποίηση

Η υλοποίηση πραγματοποιείται στο `mpi_jacobi.c` και βασίζεται σε μια 2D κατανομή του πλέγματος, με καρτεσιανή τοπολογία MPI, ghost cells, παράγωγους τύπου δεδομένων και συλλογικές επικοινωνίες για σύγκλιση και χρονομέτρηση.

1) Ορίσματα – Διάταξη διεργασιών (process grid)

Το πρόγραμμα δέχεται ως είσοδο τις διαστάσεις του πλέγματος X, Y και τη διάταξη διεργασιών Px, Py. Έτσι, δημιουργείται ένα 2D Cartesian communicator (`MPI_Cart_create`) με μη περιοδικά όρια (`periods={0,0}`) και κάθε διεργασία αποκτά τις συντεταγμένες της (`MPI_Cart_coords`). Η καρτεσιανή αυτή τοπολογία διευκολύνει την εύρεση γειτόνων με `MPI_Cart_shift`, κάτι που ταιριάζει άμεσα στο μοτίβο επικοινωνίας της εξίσωσης θερμότητας.

2) Padding και τοπικές διαστάσεις

Για να καλυφθούν περιπτώσεις όπου οι διαστάσεις δεν διαιρούνται ακριβώς από το Px,Py, εφαρμόζεται padding:

- `local[i] = ceil(global[i] / grid[i])`
- `global_padded[i] = local[i] * grid[i]`

Έτσι όλες οι διεργασίες χειρίζονται ομοιόμορφες τοπικές διαστάσεις, ενώ τα επιπλέον padded κελιά δεν πρέπει να ενημερώνονται ως πραγματικό πεδίο και αντιμετωπίζονται με προσεκτικό ορισμό ορίων επανάληψης.

3) Δομές δεδομένων – Ghost cells

Κάθε διεργασία δεσμεύει δύο τοπικούς πίνακες `u_previous` και `u_current` διαστάσεων $(localX+2) \times (localY+2)$. Το +2 αντιστοιχεί σε ghost layers (μία γραμμή/στήλη γύρω από τον πραγματικό τοπικό υποπίνακα), όπου αποθηκεύονται οι τιμές που λαμβάνονται από τους γείτονες. Η Jacobi απαιτεί δύο πεδία (παλιό/νέο), ώστε οι ενημερώσεις να βασίζονται αποκλειστικά στην προηγούμενη επανάληψη.

4) Αρχικοποίηση και Scatter του global πεδίου

Το rank 0 δεσμεύει τον global πίνακα U (με padded διαστάσεις) και τον αρχικοποιεί (init2d) σύμφωνα με τις συνοριακές συνθήκες του προβλήματος. Η διανομή των blocks προς τις διεργασίες γίνεται με MPI_Scatterv, ώστε να σταλούν/ληφθούν οι 2D υποπίνακες χωρίς manual packing:

- `global_block`: MPI_Type_vector πάνω στο padded global (stride = `global_padded[1]`) και resized extent `sizeof(double)`, ώστε τα displacements να δίνονται σε double units.
- `local_block`: MPI_Type_vector πάνω στο local buffer (stride = `localY+2`) και resized, ώστε να γεμίζει το εσωτερικό τμήμα `u_previous[1][1]`.

Τα scattercounts=1 για κάθε διεργασία και τα scatteroffsets υπολογίζονται ώστε να δείχνουν στο σωστό (i,j) block του global πίνακα.

5) Εύρος υπολογισμού (αποφυγή global boundaries και padded κελιών)

Ορίζονται `i_min..i_max` και `j_min..j_max` ώστε:

- να μη γίνονται updates στα φυσικά όρια του συνολικού πλέγματος (στα borders κρατάμε τις συνοριακές συνθήκες),
- και στις διεργασίες που βρίσκονται τελευταίες σε κάθε διάσταση να μη γίνονται updates στα κελιά που αντιστοιχούν στο padding.

Έτσι, οι ενημερώσεις περιορίζονται αυστηρά στα πραγματικά εσωτερικά κελιά του domain.

6) Halo exchange

Για την ανταλλαγή ορίων χρησιμοποιούνται δύο communication datatypes:

- `row_type`: contiguous `localY` doubles (για αποστολή/λήψη ολόκληρης γραμμής),
- `col_type`: vector `localX` στοιχείων με stride (`localY+2`) (για αποστολή/λήψη στήλης).

Οι γείτονες (north/south/east/west) προκύπτουν μέσω MPI_Cart_shift. Σε κάθε iteration εκτελείται nonblocking ανταλλαγή (MPI_Isend/MPI_Irecv) των εξής:

- Η πρώτη εσωτερική γραμμή προς north και λήψη στο ghost row 0,
- Η τελευταία εσωτερική γραμμή προς south και λήψη στο ghost row `localX+1`,

- Η πρώτη εσωτερική στήλη προς west και λήψη στο ghost col 0,
- Η τελευταία εσωτερική στήλη προς east και λήψη στο ghost col localY+1,

και ολοκλήρωση με MPI_Waitall. Η χρήση nonblocking αποφεύγει deadlocks και προσφέρει καθαρό συμμετρικό μοτίβο επικοινωνίας μεταξύ γειτόνων.

7) Ενημέρωση Jacobi (υπολογιστικός πυρήνας)

Μετά την ανταλλαγή halo, ενημερώνεται το u_current στα εσωτερικά κελιά:

$$\begin{aligned} u_{\text{current}}[i][j] = & (u_{\text{previous}}[i-1][j] + u_{\text{previous}}[i+1][j]) + \\ & u_{\text{previous}}[i][j-1] + u_{\text{previous}}[i][j+1]) / 4 \end{aligned}$$

Χρησιμοποιείται swap των pointers ($u_{\text{previous}} \leftrightarrow u_{\text{current}}$) στην αρχή κάθε iteration, ώστε να αποφεύγονται ακριβές αντιγραφές πινάκων.

8) Έλεγχος σύγκλισης (προαιρετικός, με compile flag)

Όταν είναι ενεργό το TEST_CONV, τότε υπολογίζεται ανά C επαναλήψεις τοπικά το converged και στη συνέχεια γίνεται MPI_Allreduce με MPI_LAND πάνω στο CART_COMM. Έτσι, η σύγκλιση αποφασίζεται globally: το global_converged γίνεται true μόνο αν όλες οι διεργασίες έχουν συγκλίνει. Χωρίς TEST_CONV, εκτελούνται πάντα T=256 επαναλήψεις (όπως ορίζει το σενάριο χωρίς σύγκλιση).

9) Χρονομέτρηση και συλλογή αποτελεσμάτων

Ο συνολικός χρόνος μετράται γύρω από ολόκληρο τον επαναληπτικό βρόχο, ενώ ο χρόνος υπολογισμού μετριέται μόνο γύρω από το double for των ενημερώσεων. Στο τέλος γίνεται MPI_Reduce (MPI_MAX) τόσο για total όσο και για compute time, ώστε να κρατήσουμε το critical-path (η πιο αργή διεργασία καθορίζει τον συνολικό χρόνο). Τέλος, το αποτέλεσμα συγκεντρώνεται στο rank 0 με MPI_Gatherv (ίδια λογική blocks/offsets όπως στο Scatterv) για εκτύπωση του midpoint και (προαιρετικά) την αποθήκευση του πλήρους πλέγματος.

Το αρχείο αυτό (mpi_jacobi.c) που περιέχει την υλοποίησή μας παρατίθεται ακολούθως:

```

a6/heat_transfer/mpi/mpi_jacobi.c

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <sys/time.h>
5 #include "mpi.h"
6 #include "utils.h"
7
8 int main(int argc, char ** argv) {
9     int rank,size;
10    int global[2],local[2]; //global matrix dimensions and local matrix dimensions
11    int global_padded[2];   //padded global matrix dimensions
12    int grid[2];           //processor grid dimensions
13    int i,j,t;
14    int global_converged=0,converged=0; //flags for convergence
15    MPI_Datatype dummy;             //dummy datatype
16    double omega;                 //relaxation factor - useless for Jacobi
17
18    struct timeval tts,ttf,tcs,tcf; //Timers
19    double ttotal=0,tcomp=0,total_time,comp_time;
20    double t_conv=0.0;
21    double t1,t2;
22
23    double ** U, ** u_current, ** u_previous, ** swap;
24
25    MPI_Init(&argc,&argv);
26    MPI_Comm_size(MPI_COMM_WORLD,&size);
27    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
28
29    //----Read arguments----/
30    if (argc!=5) {
31        fprintf(stderr,"Usage: mpirun .... ./exec X Y Px Py");
32        exit(-1);
33    }
34    else {
35        global[0]=atoi(argv[1]);
36        global[1]=atoi(argv[2]);
37        grid[0]=atoi(argv[3]);
38        grid[1]=atoi(argv[4]);
39    }
40
41    //----Create 2D-cartesian communicator----/
42    MPI_Comm CART_COMM;
43    int periods[2]={0,0};
44    int rank_grid[2];
45
46    MPI_Cart_create(MPI_COMM_WORLD,2,grid,periods,0,&CART_COMM);
47    MPI_Cart_coords(CART_COMM,rank,2,rank_grid);
48
49    //----Compute local dimensions & Padding----/
50    for (i=0;i<2;i++) {
51        if (global[i]%grid[i]==0) {

```

```

52         local[i]=global[i]/grid[i];
53         global_padded[i]=global[i];
54     }
55     else {
56         local[i]=(global[i]/grid[i])+1;
57         global_padded[i]=local[i]*grid[i];
58     }
59 }
60
61 //Initialization of omega
62 omega=2.0/(1+sin(3.14/global[0]));
63
64 //----Allocate global 2D-domain----//
65 if (rank==0) {
66     U=allocate2d(global_padded[0],global_padded[1]);
67     init2d(U,global[0],global[1]);
68 }
69
70 //----Allocate local 2D-subdomains---//
71 u_previous=allocate2d(local[0]+2,local[1]+2);
72 u_current=allocate2d(local[0]+2,local[1]+2);
73
74 //----Datatypes Definition---//
75 MPI_Datatype global_block;
76 MPI_Type_vector(local[0],local[1],global_padded[1],MPI_DOUBLE,&dummy);
77 MPI_Type_create_resized(dummy,0,sizeof(double),&global_block);
78 MPI_Type_commit(&global_block);
79
80 MPI_Datatype local_block;
81 MPI_Type_vector(local[0],local[1],local[1]+2,MPI_DOUBLE,&dummy);
82 MPI_Type_create_resized(dummy,0,sizeof(double),&local_block);
83 MPI_Type_commit(&local_block);
84
85 //----Scatter parameters---/
86 int * scatteroffset, * scattercounts;
87 if (rank==0) {
88     scatteroffset=(int*)malloc(size*sizeof(int));
89     scattercounts=(int*)malloc(size*sizeof(int));
90     for (i=0;i<grid[0];i++)
91         for (j=0;j<grid[1];j++) {
92             scattercounts[i*grid[1]+j]=1;
93             scatteroffset[i*grid[1]+j]=(local[0]*local[1]*grid[1]*i+local[1]*j);
94         }
95     }
96
97 //----Scatter---/
98 MPI_Scatterv(rank == 0 ? &U[0][0] : NULL,
99                 scattercounts, scatteroffset, global_block,
100                &u_previous[1][1], 1, local_block,
101                0, MPI_COMM_WORLD);
102
103 // Init u_current
104 for (i = 1; i <= local[0]; i++)
105     for (j = 1; j <= local[1]; j++)

```

```

106         u_current[i][j] = u_previous[i][j];
107
108     if (rank==0)
109         free2d(U);
110
111     //----Communication Datatypes----//
112     MPI_Datatype row_type, col_type;
113     MPI_Type_contiguous(local[1], MPI_DOUBLE, &row_type);
114     MPI_Type_commit(&row_type);
115     MPI_Type_vector(local[0], 1, local[1] + 2, MPI_DOUBLE, &col_type);
116     MPI_Type_commit(&col_type);
117
118     //----Find Neighbors----//
119     int north, south, east, west;
120     MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
121     MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);
122
123     //---Define iteration ranges----//
124     int i_min,i_max,j_min,j_max;
125
126     i_min = 1;
127     i_max = local[0];
128     j_min = 1;
129     j_max = local[1];
130
131     if (rank_grid[0] == 0) i_min = 2;
132     if (rank_grid[0] == grid[0] - 1) {
133         i_max = local[0] - (global_padded[0] - global[0]) - 1;
134         if (global_padded[0] == global[0]) i_max = local[0] - 1;
135     }
136
137     if (rank_grid[1] == 0) j_min = 2;
138     if (rank_grid[1] == grid[1] - 1) {
139         j_max = local[1] - (global_padded[1] - global[1]) - 1;
140         if (global_padded[1] == global[1]) j_max = local[1] - 1;
141     }
142
143     //----Computational core----//
144     gettimeofday(&tts, NULL);
145
146 #ifdef TEST_CONV
147     for (t=0;t<T && !global_converged;t++) {
148 #endif
149 #ifndef TEST_CONV
150     #undef T
151     #define T 256
152     for (t=0;t<T;t++) {
153 #endif
154
155         // 1. Swap
156         swap = u_previous;
157         u_previous = u_current;
158         u_current = swap;
159

```

```

160     // 2. Communication
161     MPI_Request reqs[8];
162     MPI_Status stats[8];
163     int req_cnt = 0;
164
165     MPI_Isend(&u_previous[1][1], 1, row_type, north, 1, CART_COMM, &reqs[req_cnt++]);
166     MPI_Irecv(&u_previous[0][1], 1, row_type, north, 2, CART_COMM, &reqs[req_cnt++]);
167
168     MPI_Isend(&u_previous[local[0]][1], 1, row_type, south, 2, CART_COMM,
169     &reqs[req_cnt++]);
170     MPI_Irecv(&u_previous[local[0]+1][1], 1, row_type, south, 1, CART_COMM,
171     &reqs[req_cnt++]);
172
173     MPI_Isend(&u_previous[1][1], 1, col_type, west, 3, CART_COMM, &reqs[req_cnt++]);
174     MPI_Irecv(&u_previous[1][0], 1, col_type, west, 4, CART_COMM, &reqs[req_cnt++]);
175
176     MPI_Isend(&u_previous[1][local[1]], 1, col_type, east, 4, CART_COMM,
177     &reqs[req_cnt++]);
178     MPI_Irecv(&u_previous[1][local[1]+1], 1, col_type, east, 3, CART_COMM,
179     &reqs[req_cnt++]);
180
181     MPI_Waitall(req_cnt, reqs, stats);
182
183     // 3. Computation
184     gettimeofday(&tcs, NULL);
185     for (i = i_min; i <= i_max; i++) {
186         for (j = j_min; j <= j_max; j++) {
187             u_current[i][j] = (u_previous[i-1][j] + u_previous[i+1][j] +
188                                 u_previous[i][j-1] + u_previous[i][j+1]) / 4.0;
189         }
190     }
191     gettimeofday(&tcf, NULL);
192     tcomp += (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec - tcs.tv_usec) * 0.000001;
193
194     // 4. Convergence Check
195     #ifdef TEST_CONV
196     if (t % C == 0) {
197         converged = converge(u_previous, u_current, i_min, i_max, j_min, j_max);
198
199         t1=MPI_Wtime(); //
200         MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_LAND, CART_COMM);
201
202         t2=MPI_Wtime();
203         t_conv+=(t2-t1);
204     }
205     #endif
206
207     }
208     gettimeofday(&ttf,NULL);
209
210     ttotal=(ttf.tv_sec-tts.tv_sec)+(ttf.tv_usec-tts.tv_usec)*0.000001;
211
212     MPI_Reduce(&ttotal,&total_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
213     MPI_Reduce(&tcomp,&comp_time,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);

```

```
210 //----Gather results----/
211 if (rank==0) {
212     U=allocate2d(global_padded[0],global_padded[1]);
213 }
214
215
216 MPI_Gatherv(&u_current[1][1], 1, local_block,
217             rank == 0 ? &U[0][0] : NULL,
218             scattercounts, scatteroffset, global_block,
219             0, MPI_COMM_WORLD);
220
221 //----Printing results----/
222 if (rank==0) {
223     printf("Jacobi X %d Y %d Px %d Py %d Iter %d ComputationTime %lf TotalTime %lf
ConvergenceTime %lf midpoint %lf\n",
224     global[0],global[1],grid[0],grid[1],t,comp_time,total_time,t_conv,U[global[0]/2]
225     [global[1]/2]);
226
227 #ifdef PRINT_RESULTS
228     char * s=malloc(50*sizeof(char));
229     sprintf(s,"resJacobiMPI_%dx%d_%dx%d",global[0],global[1],grid[0],grid[1]);
230     fprintf2d(s,U,global[0],global[1]);
231     free(s);
232 #endif
233 }
234
235 // Free Datatypes before Finalize
236 MPI_Type_free(&row_type);
237 MPI_Type_free(&col_type);
238 MPI_Type_free(&global_block);
239 MPI_Type_free(&local_block);
240 if(rank==0){
241     free(scattercounts);
242     free(scatteroffset);
243 }
244
245 MPI_Finalize();
246 return 0;
247 }
```

■ 4.2.1 – Μετρήσεις με Έλεγχο Σύγκλισης

A. Εισαγωγή

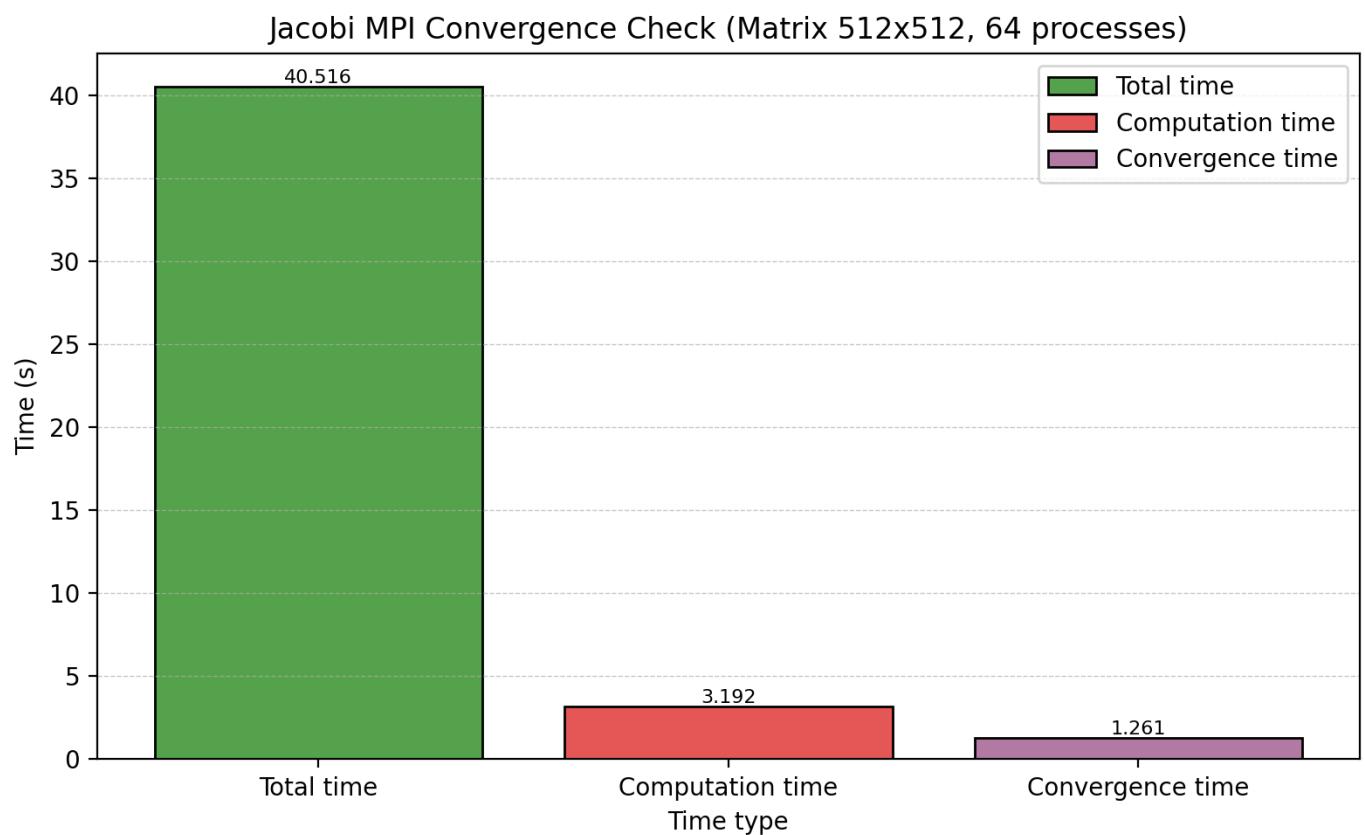
Στην υποενότητα αυτή, θα αξιολογήσουμε την υλοποίηση σε MPI της μεθόδου Jacobi για τη δισδιάστατη εξίσωση θερμότητας, όταν ο αλγόριθμος τερματίζει με βάση τον έλεγχο σύγκλισης. Σύμφωνα με την εκφώνηση, εκτελούμε το σενάριο για πλέγμα 512×512 με 64 MPI διεργασίες και καταγράφουμε ξεχωριστά:

- Συνολικό χρόνο: τον χρόνο του επαναληπτικού τμήματος, όπως καθορίζεται από την πιο αργή διεργασία (critical path).
- Χρόνο υπολογισμών: τον χρόνο που δαπανάται αποκλειστικά στον υπολογιστικό πυρήνα (ενημέρωση των εσωτερικών κελιών).
- Χρόνο ελέγχου σύγκλισης: τον χρόνο που αντιστοιχεί στον περιοδικό έλεγχο σύγκλισης (τοπικός υπολογισμός κριτηρίου + απόφαση μέσω συλλογικής επικοινωνίας).

Η θεωρητική σημασία του πειράματος είναι ότι ο Jacobi είναι ένας εύκολα παραλληλίσιμος αλγόριθμος (χρησιμοποιεί μόνο τιμές της προηγούμενης επανάληψης), όμως σε κατανεμημένη μνήμη απαιτεί ανταλλαγή ορίων σε κάθε iteration και επιπλέον επικοινωνίες για τον τελικό τερματισμό. Άρα, το συνολικό κόστος προκύπτει από την ισορροπία μεταξύ υπολογισμών (που μειώνονται με το πλήθος διεργασιών) και επικοινωνιών/συγχρονισμών (που δεν μειώνονται αντίστοιχα και συχνά αυξάνουν με την κλίμακα).

Σημείωση για τις bonus μεθόδους: η εκφώνηση ζητά σύγκριση και με Gauss–Seidel SOR και Red–Black SOR. Τις προσεγγίσαμε (σε επίπεδο σχεδίασης/δομής επικοινωνίας), αλλά δεν ολοκληρώσαμε την υλοποίησή τους λόγω χρόνου και δύσκολου debugging (αν και έχουμε σχεδόν όλο τον κώδικα στο scirouter), επομένως παραθέτουμε ποιοτική/θεωρητική σύγκριση για το τι θα αναμέναμε στην πράξη.

B. Μετρήσεις και Διαγράμματα



Γ. Συμπεράσματα

Για το σενάριο 512×512 με 64 διεργασίες, τα μετρούμενα αποτελέσματα είναι:

- Total time: 40.516 s
- Computation time: 3.192 s
- Convergence time: 1.261 s

1) Η εκτέλεση είναι έντονα communication/synchronization-bound

Παρατηρούμε ότι ο χρόνος υπολογισμών αποτελεί μικρό μέρος του συνολικού χρόνου: $3.192 / 40.516 \approx 7.9\%$ του total. Αντίστοιχα, ο χρόνος ελέγχου σύγκλισης είναι: $1.261 / 40.516 \approx 3.1\%$ του total. Το υπόλοιπο ($\sim 89\%$) αντιστοιχεί σε κόστη που δεν περιλαμβάνονται στον καθαρό υπολογισμό και στον μετρούμενο έλεγχο σύγκλισης: κυρίως halo exchanges (Isend/Irecv/Waitall), αναμονές/ανισορροπία (όλοι συγχρονίζονται ουσιαστικά ανά iteration), και γενικά MPI overheads ανά επανάληψη.

Αυτό είναι αναμενόμενο από τη θεωρία: όταν το local υποπλέγμα ανά διεργασία γίνεται μικρό (εδώ $512 \times 512 / 64 \approx 64 \times 64$ ανά process), ο λόγος επιφάνειας προς όγκο αυξάνει. Δηλαδή, τα δεδομένα που πρέπει να ανταλλαχθούν (περίμετρος του block) δεν μειώνονται τόσο γρήγορα όσο ο υπολογισμός (εμβαδό του block). Συνεπώς, το κόστος επικοινωνίας γίνεται συγκρίσιμο ή και κυρίαρχο.

2) Ο έλεγχος σύγκλισης προσθέτει παγκόσμιο συγχρονισμό (global reduction)

Παρότι το convergence time (1.261 s) δεν είναι το μεγαλύτερο κομμάτι, παραμένει μη αμελητέο, επειδή:

- ο έλεγχος απαιτεί μια καθολική απόφαση (όλες οι διεργασίες πρέπει να συμφωνήσουν ότι έχει επιτευχθεί σύγκλιση),
- άρα χρησιμοποιείται συλλογική επικοινωνία (π.χ. Allreduce) που επιβάλλει συγχρονισμό και κόστος latency που τυπικά αυξάνει με την κλίμακα (συχνά $\sim \log P$).

Πρακτικά, κάθε convergence check λειτουργεί σαν σημείο όπου η πιο αργή διεργασία καθορίζει την πρόοδο όλων (critical path). Αυτό είναι βασική αρχή στη θεωρία: είναι αποδοτική μέθοδος για μικρά μηνύματα, αλλά έχει ένα αναπόφευκτο (και συχνά μεγάλο) synchronization component.

3) Γιατί με 64 διεργασίες ο Jacobi μπορεί να έχει μεγάλο total time παρότι ο compute πυρήνας είναι μικρός

Η Jacobi είναι μεν εύκολα παραλληλοποιήσιμη, αλλά για να προχωρήσει κάθε iteration χρειάζεται ενημερωμένα halo από τους γείτονες. Αυτό σημαίνει ότι:

- υπάρχει επικοινωνία σε κάθε iteration (γειτονική ανταλλαγή γραμμών/στηλών),
- το κόστος της επικοινωνίας καθορίζεται έντονα από latency και όχι μόνο από bandwidth, ειδικά όταν τα blocks είναι μικρά,
- ενώ ταυτόχρονα ο υπολογισμός ανά διεργασία είναι μικρός και δεν κρύβει τον χρόνο επικοινωνίας.

Αποτέλεσμα: όσο μεγαλώνει το P για σταθερό πρόβλημα (strong scaling), το compute μειώνεται γρήγορα, αλλά η επικοινωνία/συγχρονισμοί γίνονται το bottleneck — κλασικό όριο strong scaling (Amdahl-like συμπεριφορά, αλλά εδώ κυρίως λόγω surface/volume και latency).

4) Θεωρητική σύγκριση με τις bonus μεθόδους (χωρίς υλοποίηση)

Παρότι δεν υλοποιήσαμε SOR/Red–Black SOR, η θεωρία υποδεικνύει τα εξής:

- Jacobi: πλήρως παράλληλος ανά iteration, αλλά συνήθως πιο αργή σύγκλιση (περισσότερες επαναλήψεις \rightarrow περισσότερα halo exchanges και περισσότερα convergence checks).
- Gauss–Seidel SOR: τείνει να συγκλίνει σε λιγότερες επαναλήψεις (άρα πιθανώς μικρότερο συνολικό communication volume), αλλά έχει ισχυρότερες εξαρτήσεις ενημέρωσης που δυσκολεύουν τον καθαρό παραλληλισμό.
- Red–Black SOR: προσφέρει πρακτικό συμβιβασμό, γιατί διατηρεί ταχύτερη σύγκλιση από Jacobi και επιτρέπει παράλληλη ενημέρωση ανά χρώμα (red/black), άρα είναι συχνά πιο κατάλληλο για παραλληλοποίηση από το κλασικό Gauss–Seidel.

Με βάση αυτό, αν οι bonus μέθοδοι ολοκληρώνονταν, θα αναμέναμε μικρότερο total time κυρίως επειδή θα απαιτούνταν λιγότερες επαναλήψεις (άρα λιγότερα halo exchanges και λιγότερα global convergence checks) — δηλαδή θα μειωνόταν το κυρίαρχο κομμάτι του κόστους.

Τελικό συμπέρασμα

Το πείραμα με 512×512 και 64 διεργασίες δείχνει καθαρά ότι, με ενεργό convergence check, η MPI Jacobi γίνεται κυρίως communication/synchronization-bound: ο καθαρός υπολογισμός είναι μικρό ποσοστό του total, ενώ το μεγαλύτερο μέρος προκύπτει από halo exchanges και τα αναπόφευκτα σημεία συγχρονισμού (γειτονική ανταλλαγή ανά iteration και περιοδικές συλλογικές επικοινωνίες για σύγκλιση). Αυτό συνάδει πλήρως με τη θεωρία των εφαρμογών σε κατανεμημένη μνήμη και εξηγεί γιατί οι μέθοδοι με ταχύτερη σύγκλιση (SOR, ιδιαίτερα Red–Black) είναι σημαντικές: μειώνουν τον αριθμό επαναλήψεων, άρα περιορίζουν το επικοινωνιακό κόστος που εδώ κυριαρχεί.

■ 4.2.2 – Μετρήσεις χωρίς Έλεγχο Σύγκλισης

A. Εισαγωγή

Στην υποενότητα αυτή θα μετρήσουμε την απόδοση της MPI υλοποίησης του 2D Jacobi για τη διάδοση θερμότητας, χωρίς κριτήριο σύγκλισης. Σύμφωνα με την εκφώνηση, ο αλγόριθμος εκτελείται για σταθερό πλήθος επαναλήψεων ($T=256$), ώστε όλες οι εκτελέσεις να έχουν ίδιο αριθμό βημάτων και οι συγκρίσεις να μην επηρεάζονται από διαφορετικό χρόνο τερματισμού.

Η απενεργοποίηση του convergence check έχει δύο στόχους:

- Απομονώνει το κόστος του βασικού υπολογισμού και των επικοινωνιών ανά επανάληψη, χωρίς πρόσθετο (καθολικό) συγχρονισμό τύπου Allreduce.
- Επιτρέπει καθαρή αξιολόγηση strong scaling: για κάθε μέγεθος πλέγματος (2048×2048 , 4096×4096 , 6144×6144) αυξάνουμε τον αριθμό MPI διεργασιών P και μετράμε τον χρόνο.

Μετρήσεις/Μετρικές (όπως ζητούνται):

- Speedup $S(P) = T_{seq} / TP$, όπου T_{seq} ο χρόνος της σειριακής εκτέλεσης για το ίδιο μέγεθος πλέγματος και TP ο χρόνος με P διεργασίες.
- Total time: χρόνος του επαναληπτικού τμήματος, λαμβάνοντας πρακτικά τον χειρότερο (critical path), αφού η συνολική πρόοδος καθορίζεται από την πιο αργή διεργασία.
- Computation time: χρόνος αποκλειστικά των ενημερώσεων χωρίς να μετρά την επικοινωνία/αναμονές.

Θεωρητικά, σε κάθε iteration κάθε διεργασία:

1. ανταλλάσσει halo με τους 4 γείτονες (πάνω/κάτω/αριστερά/δεξιά) και
2. υπολογίζει τοπικά το εσωτερικό της block.

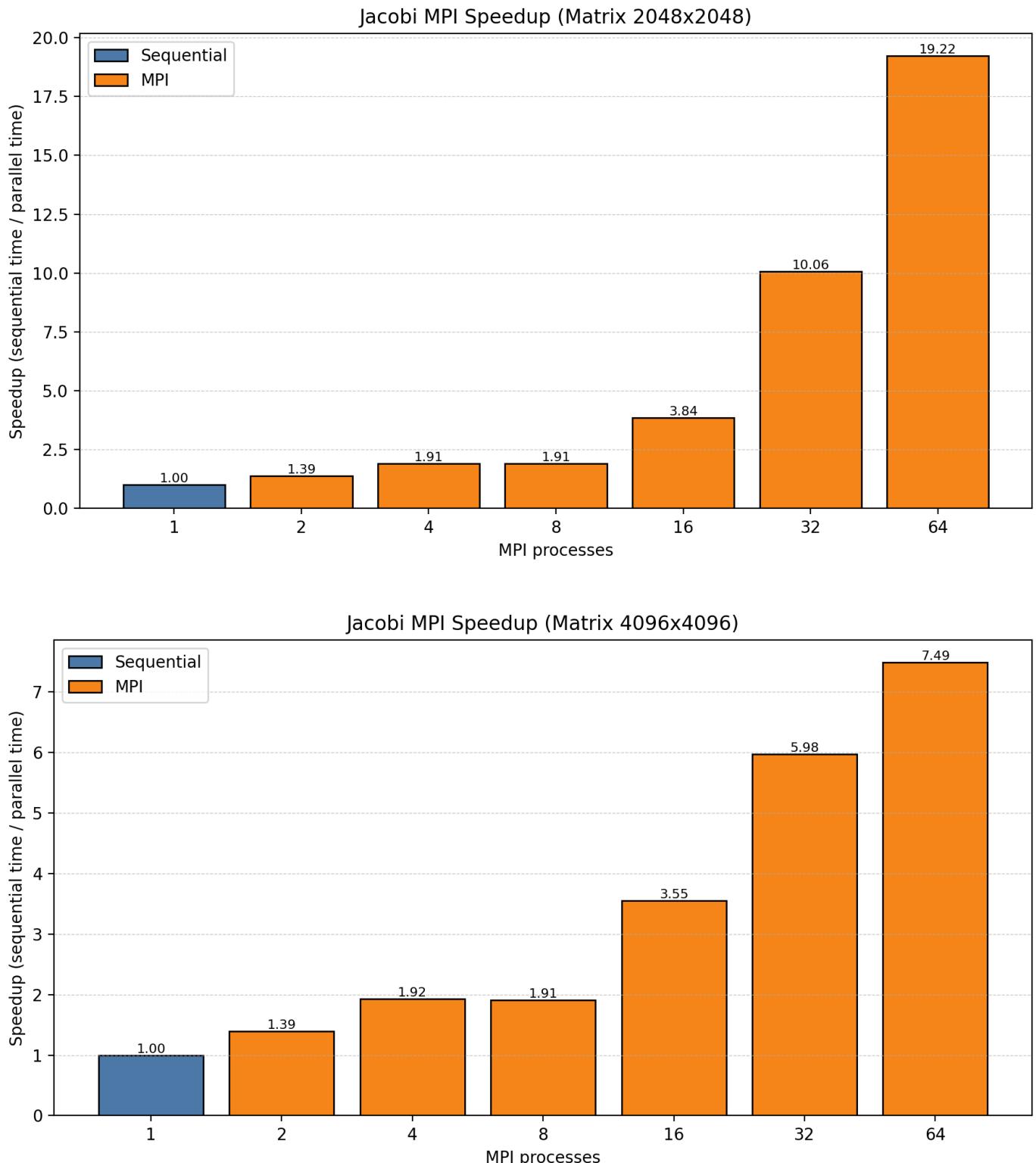
Άρα ο συνολικός χρόνος ανά iteration προσεγγίζεται από:

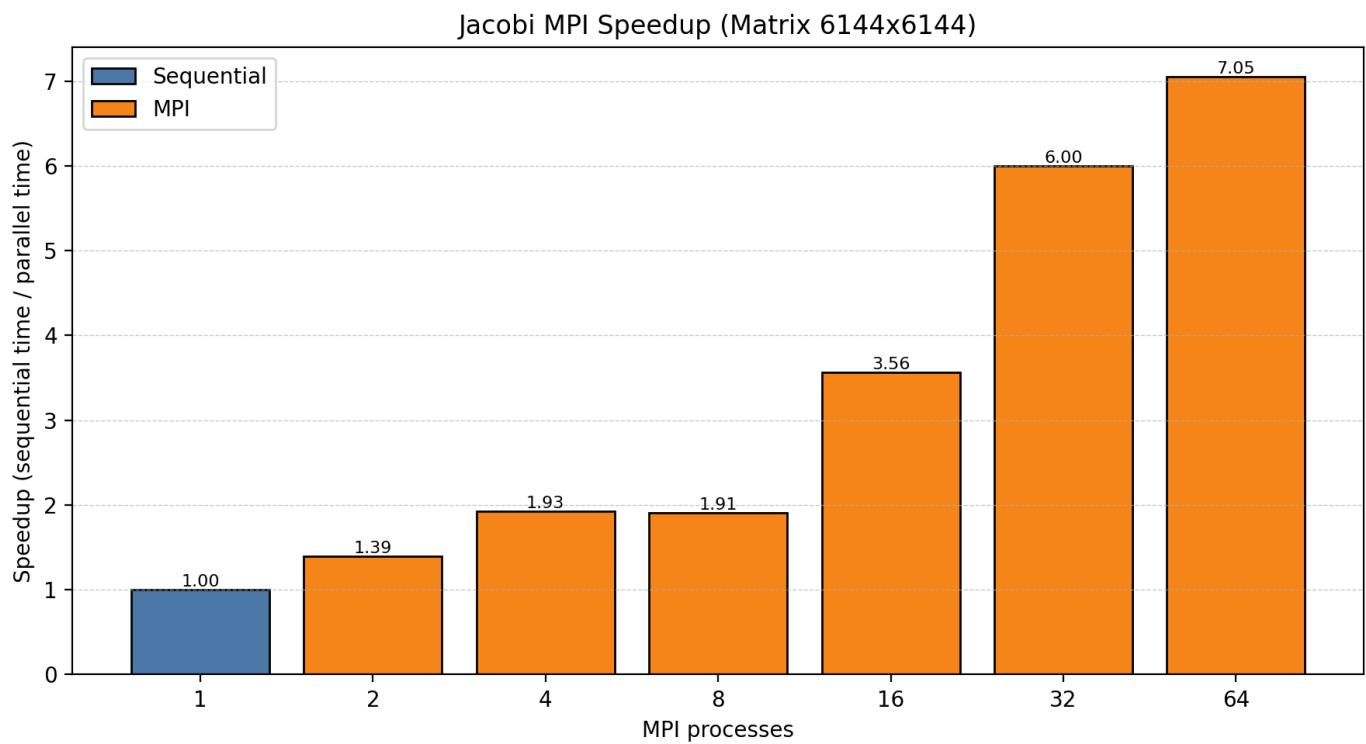
$T_{iter} \approx T_{comp} + T_{comm}$,

όπου T_{comm} περιλαμβάνει latency + bandwidth κόστος ανταλλαγών και τυχόν αναμονές (Waitall). Στην strong scaling περίπτωση (σταθερό πρόβλημα, αυξανόμενο P), το T_{comp} μειώνεται περίπου ως $O(1/P)$, ενώ το T_{comm} δεν μειώνεται αντίστοιχα και συχνά γίνεται το bottleneck λόγω αυξημένου surface/volume ratio και latency-dominated μικρών μηνυμάτων.

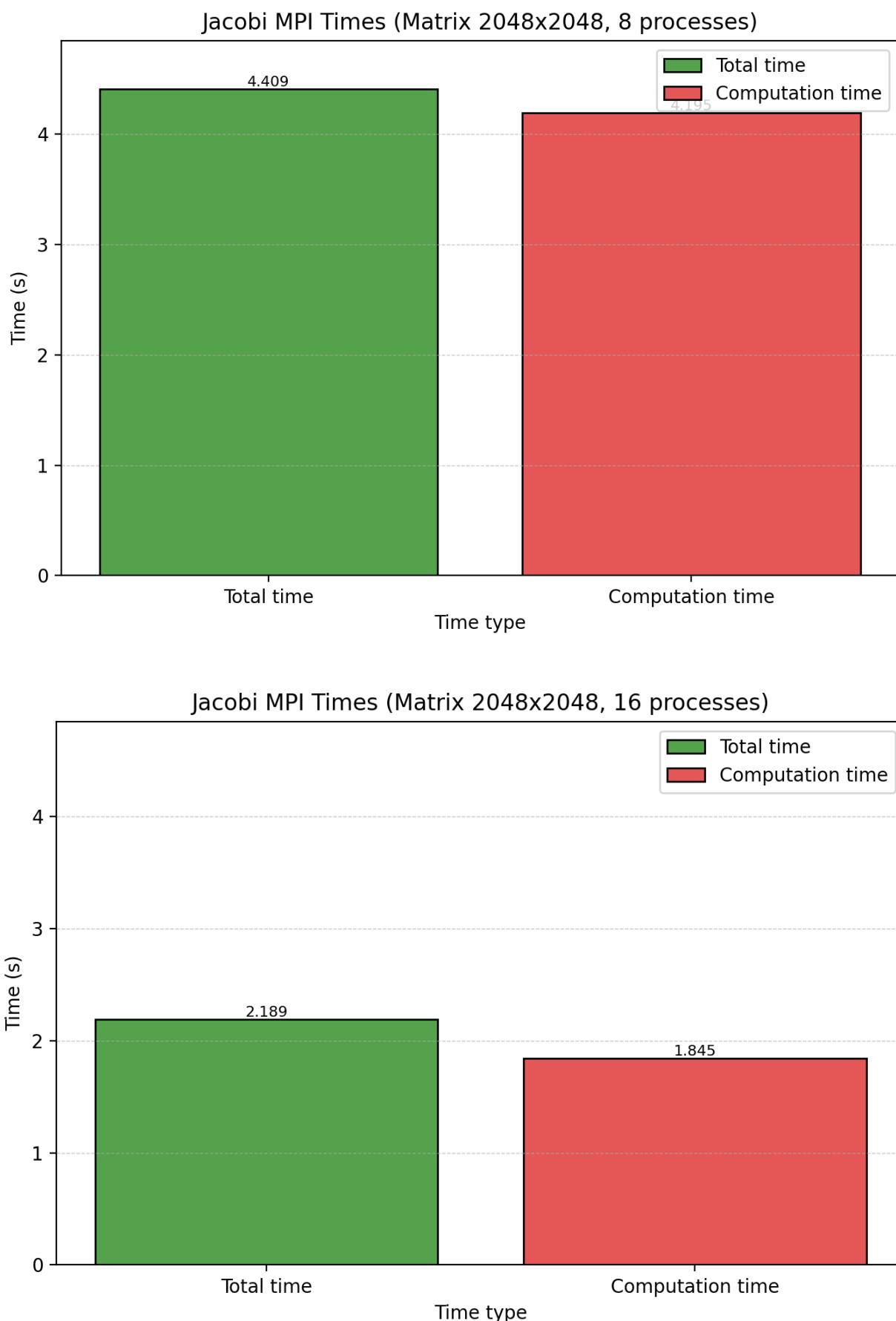
B. Μετρήσεις και Διαγράμματα

Διαγράμματα Επιτάχυνσης

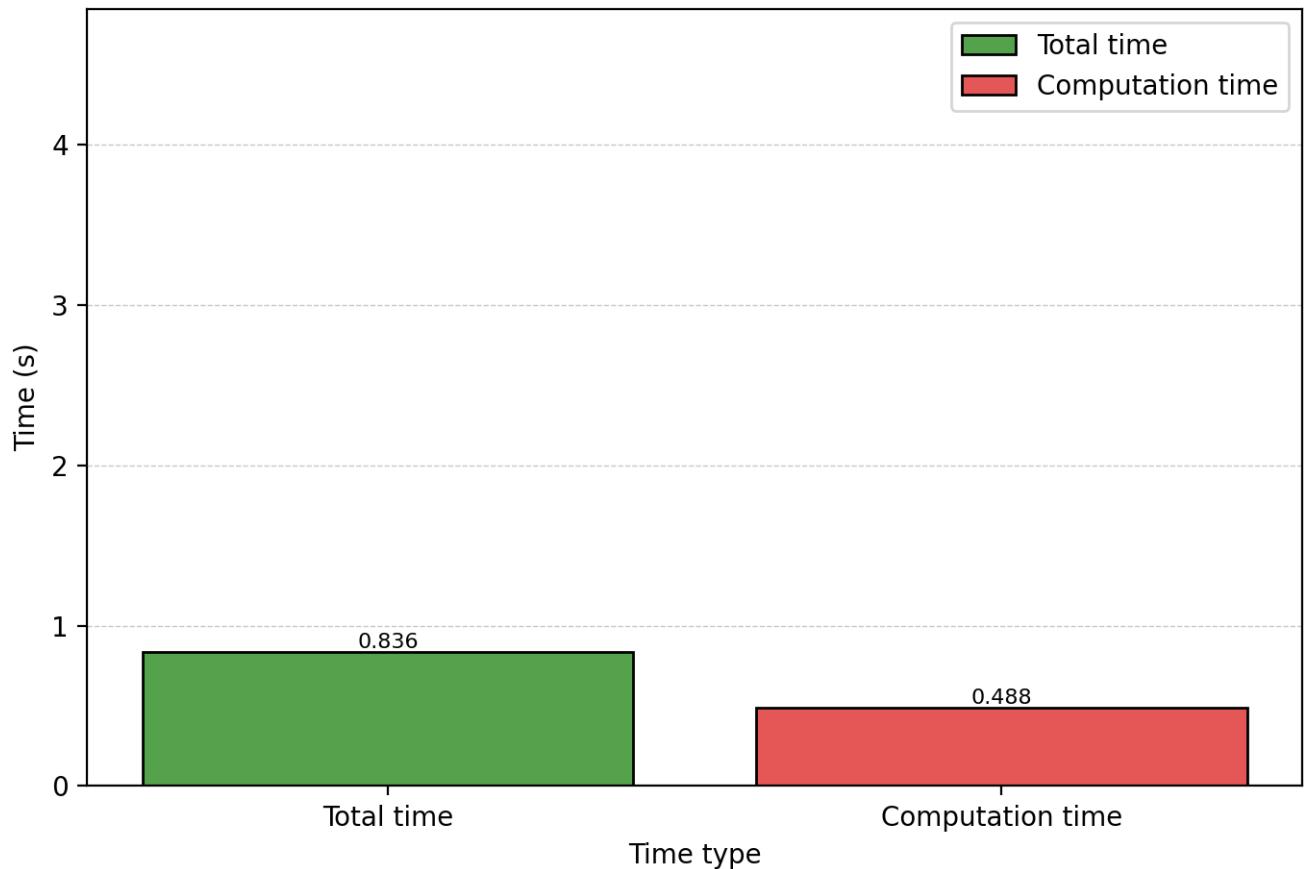




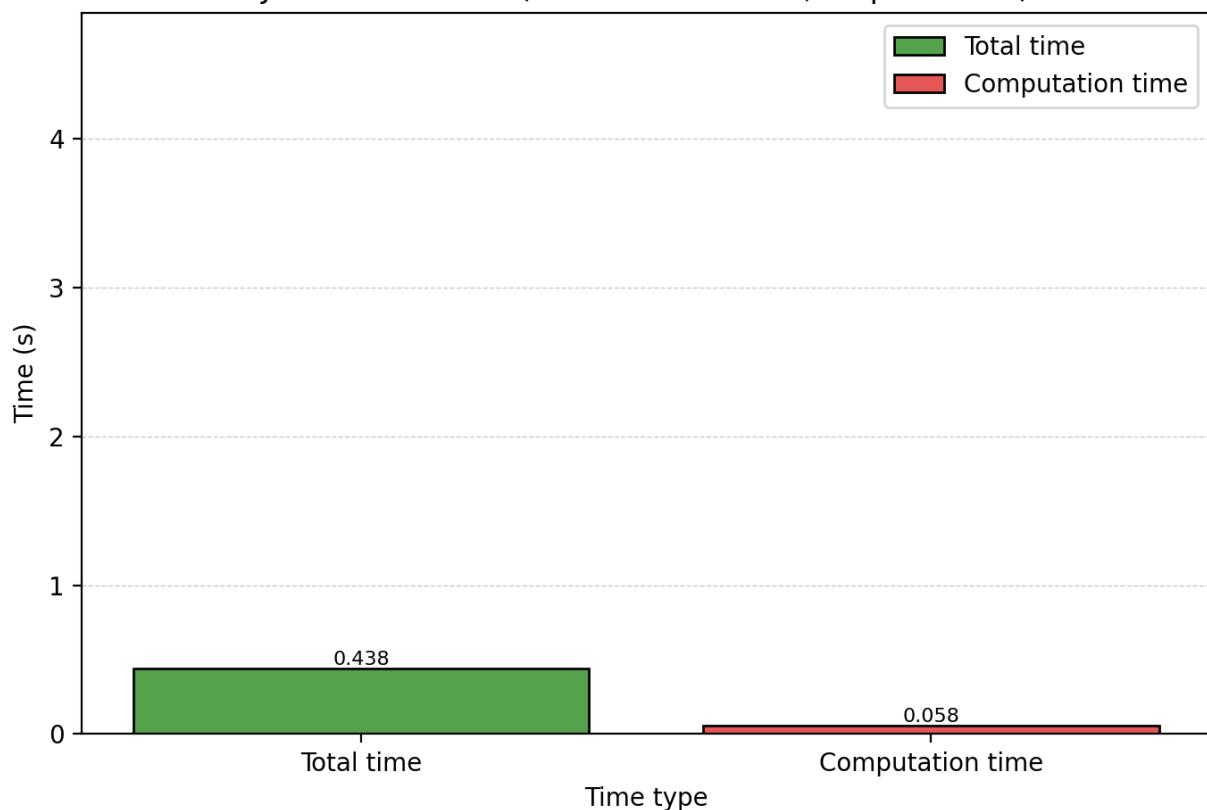
Διαγράμματα χρόνου – Πίνακας 2048x2048



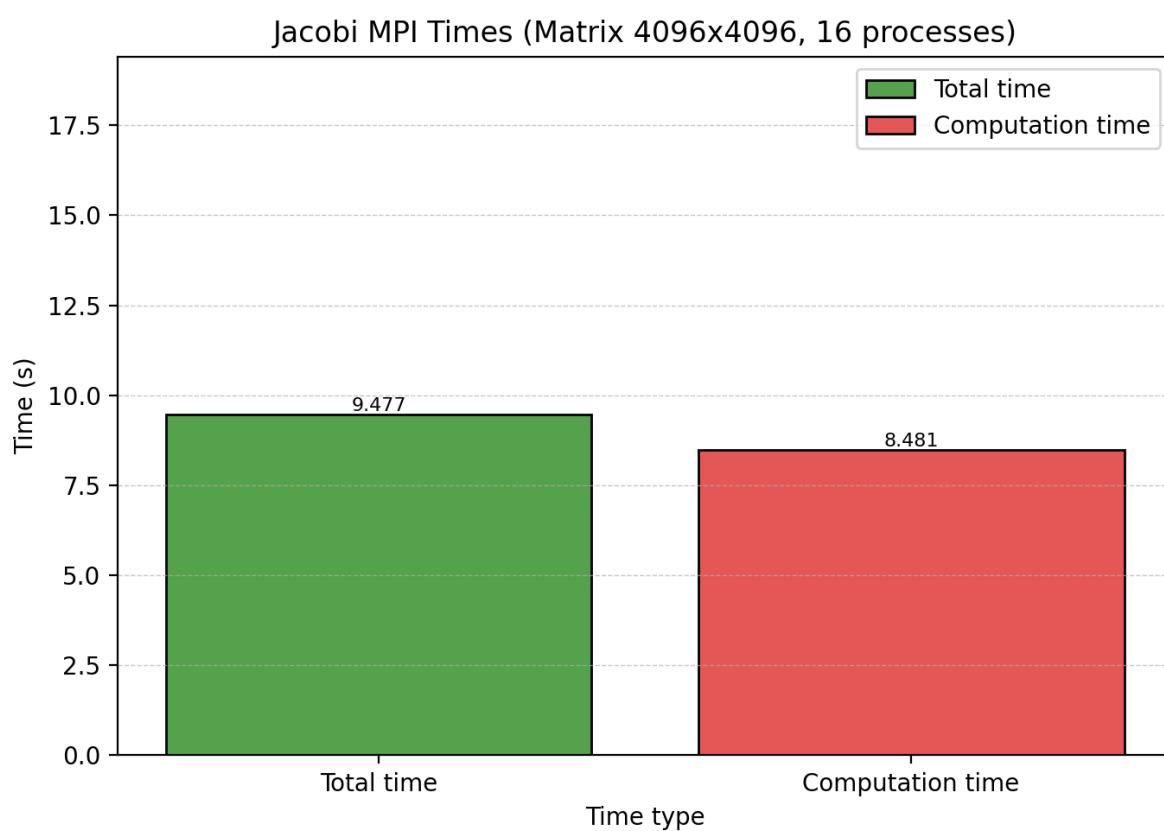
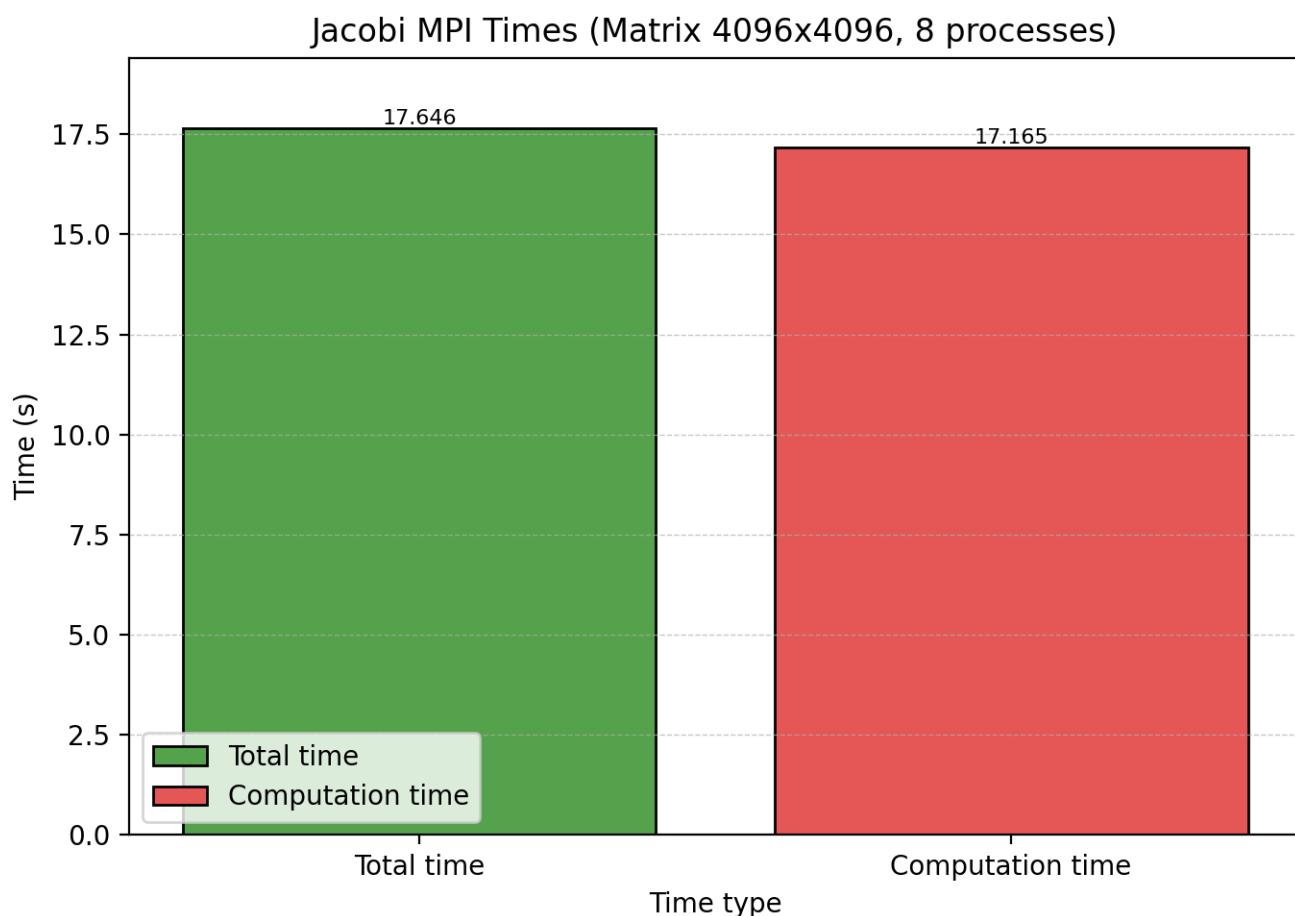
Jacobi MPI Times (Matrix 2048x2048, 32 processes)



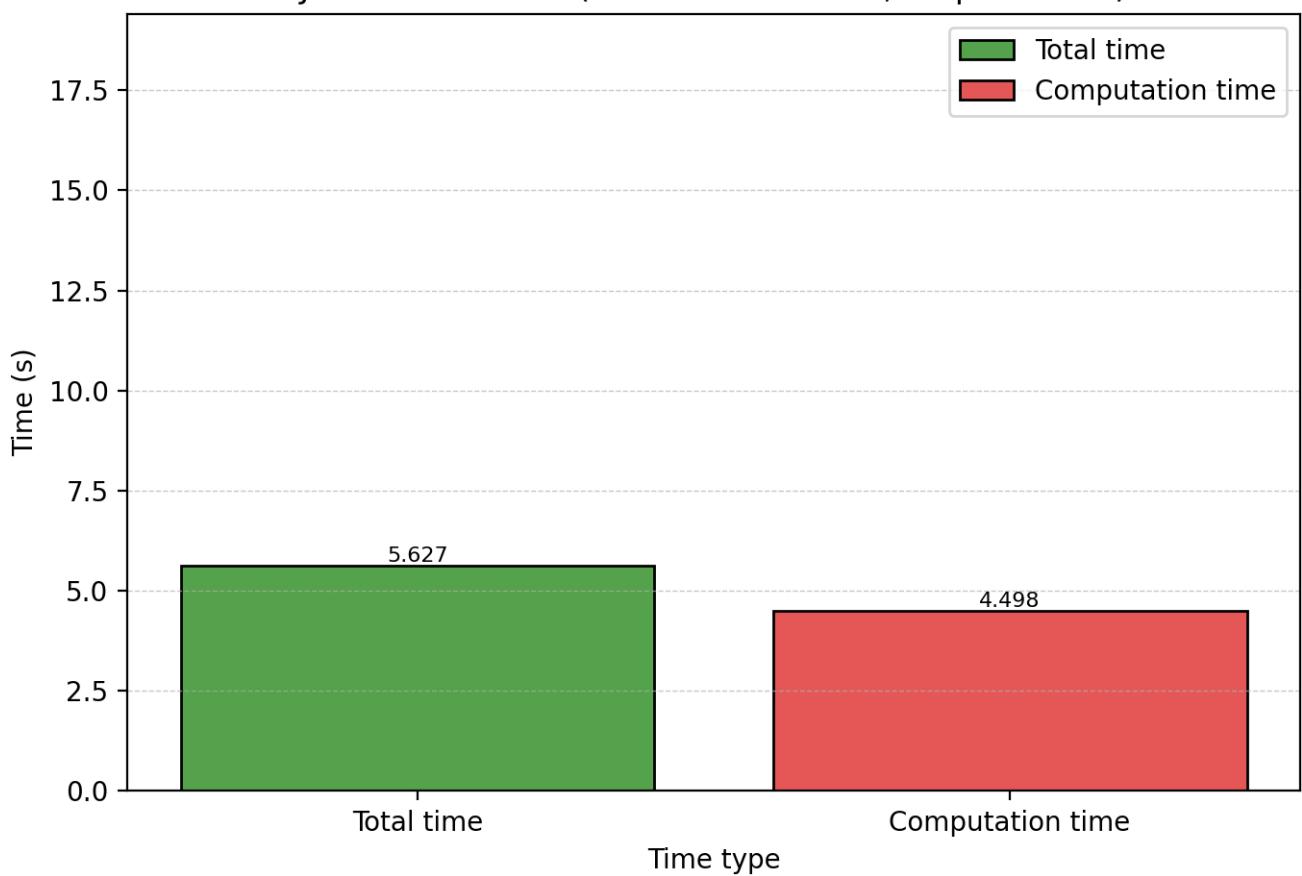
Jacobi MPI Times (Matrix 2048x2048, 64 processes)



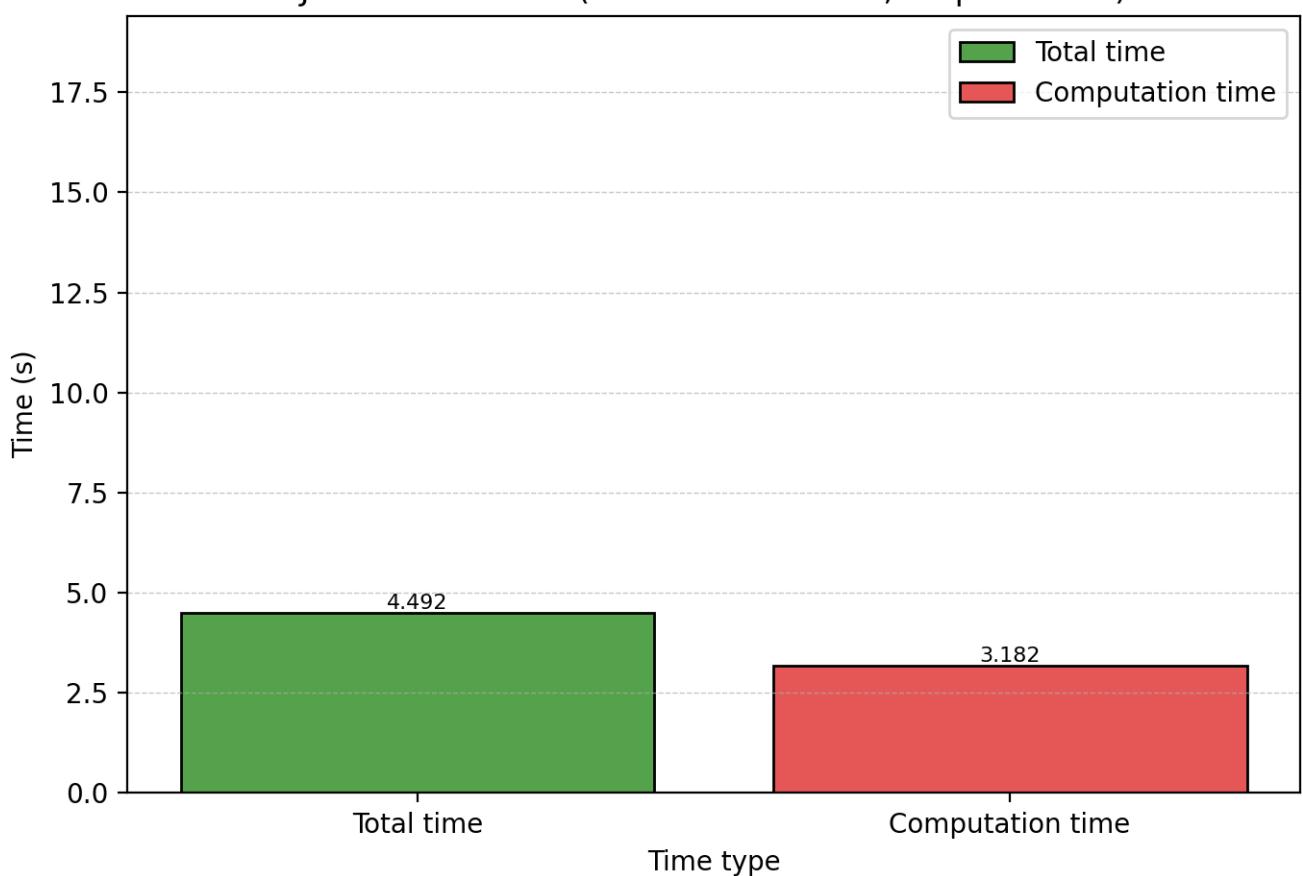
Διαγράμματα χρόνου – Πίνακας 4096x4096



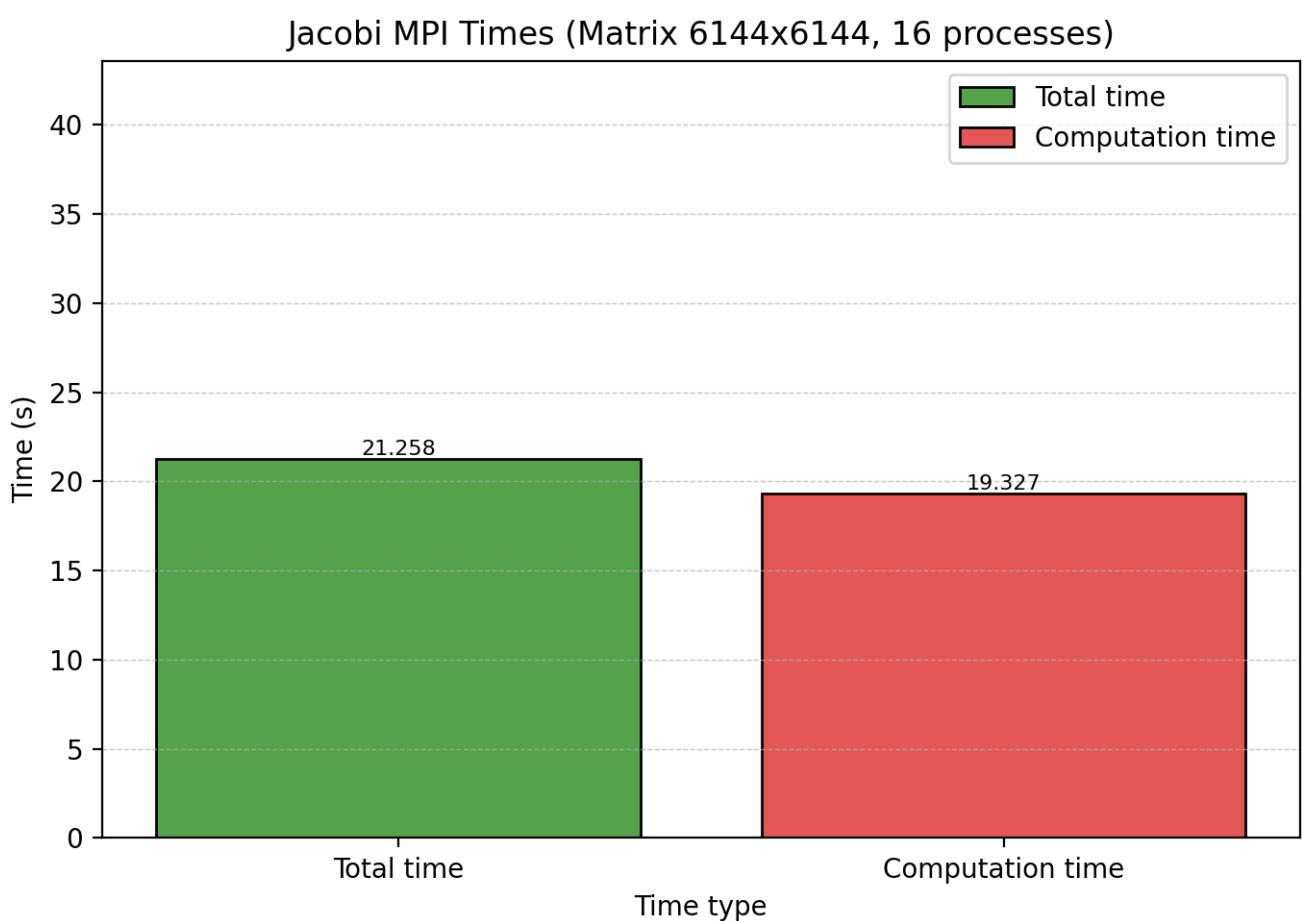
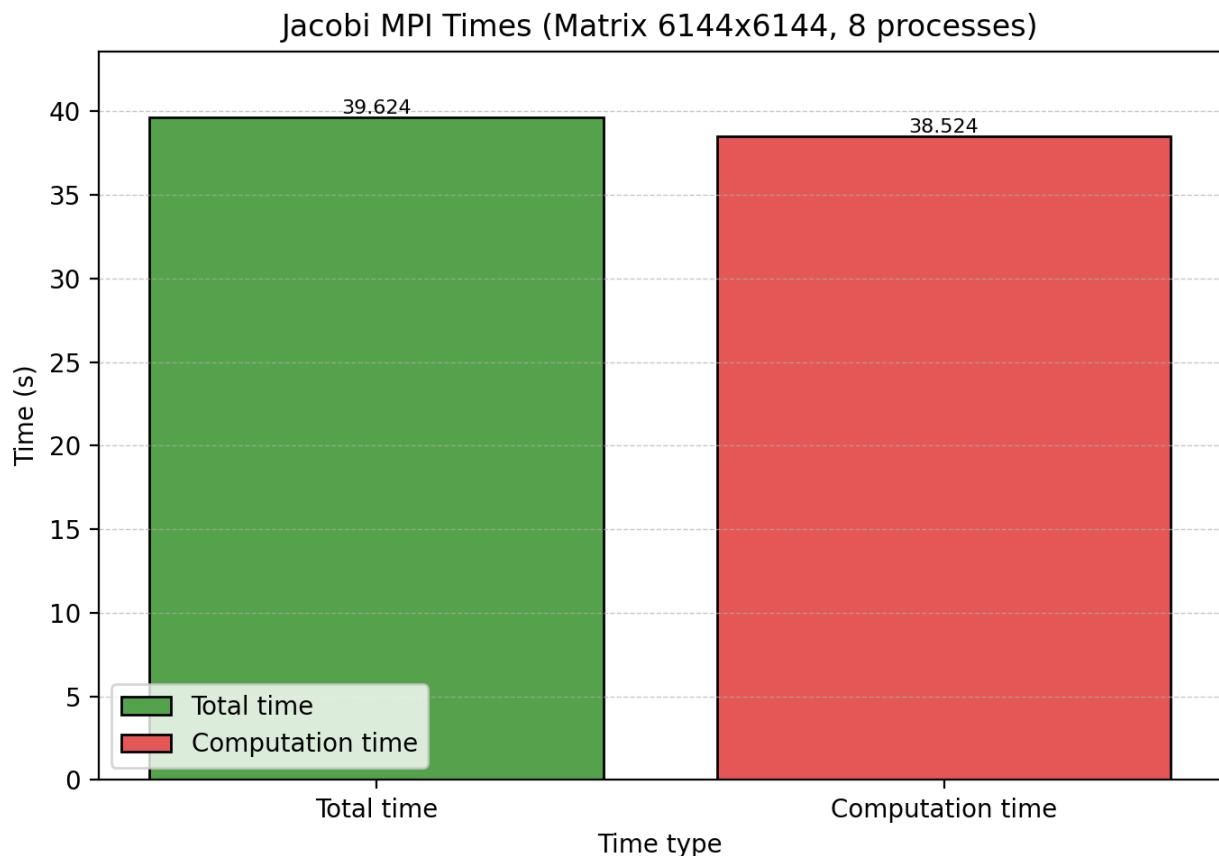
Jacobi MPI Times (Matrix 4096x4096, 32 processes)



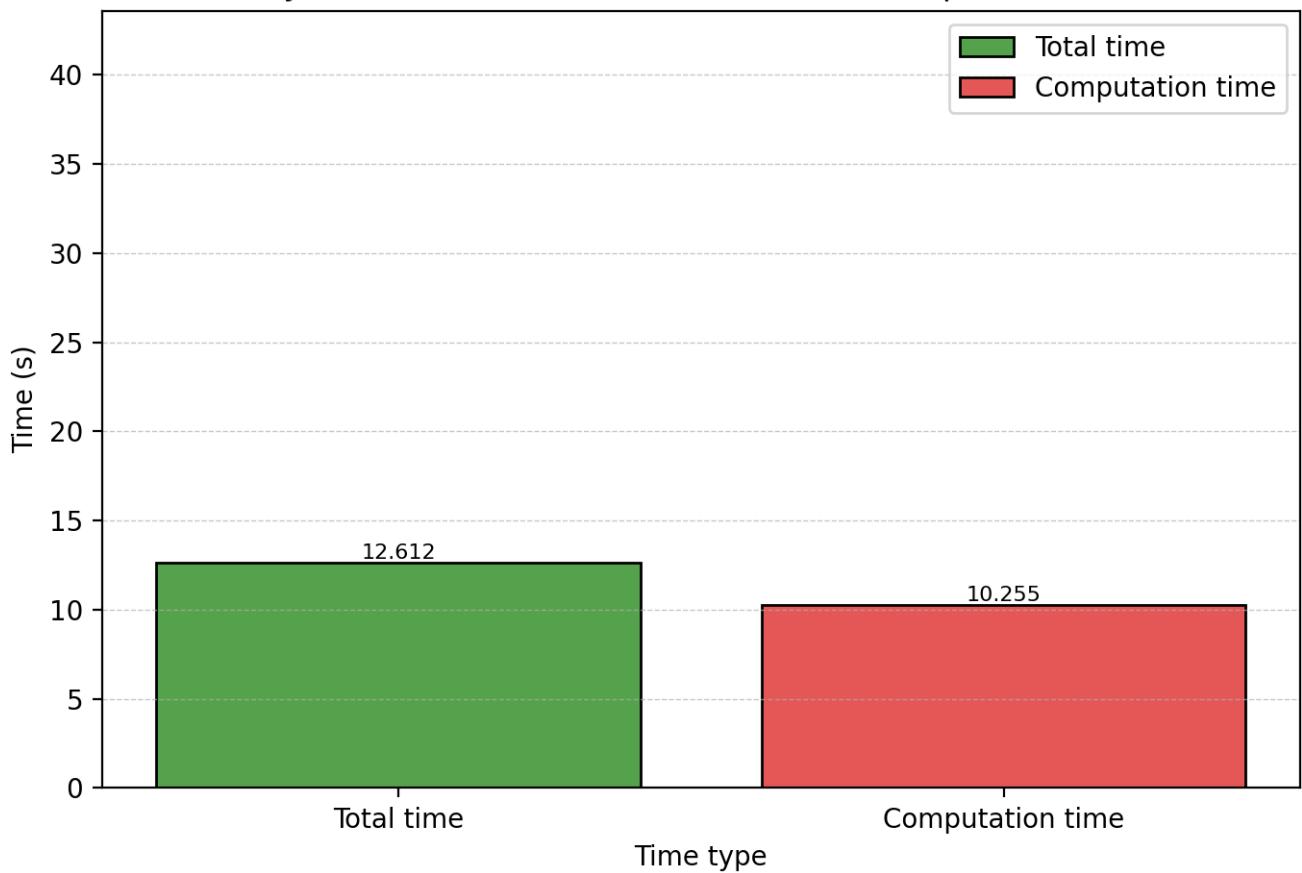
Jacobi MPI Times (Matrix 4096x4096, 64 processes)



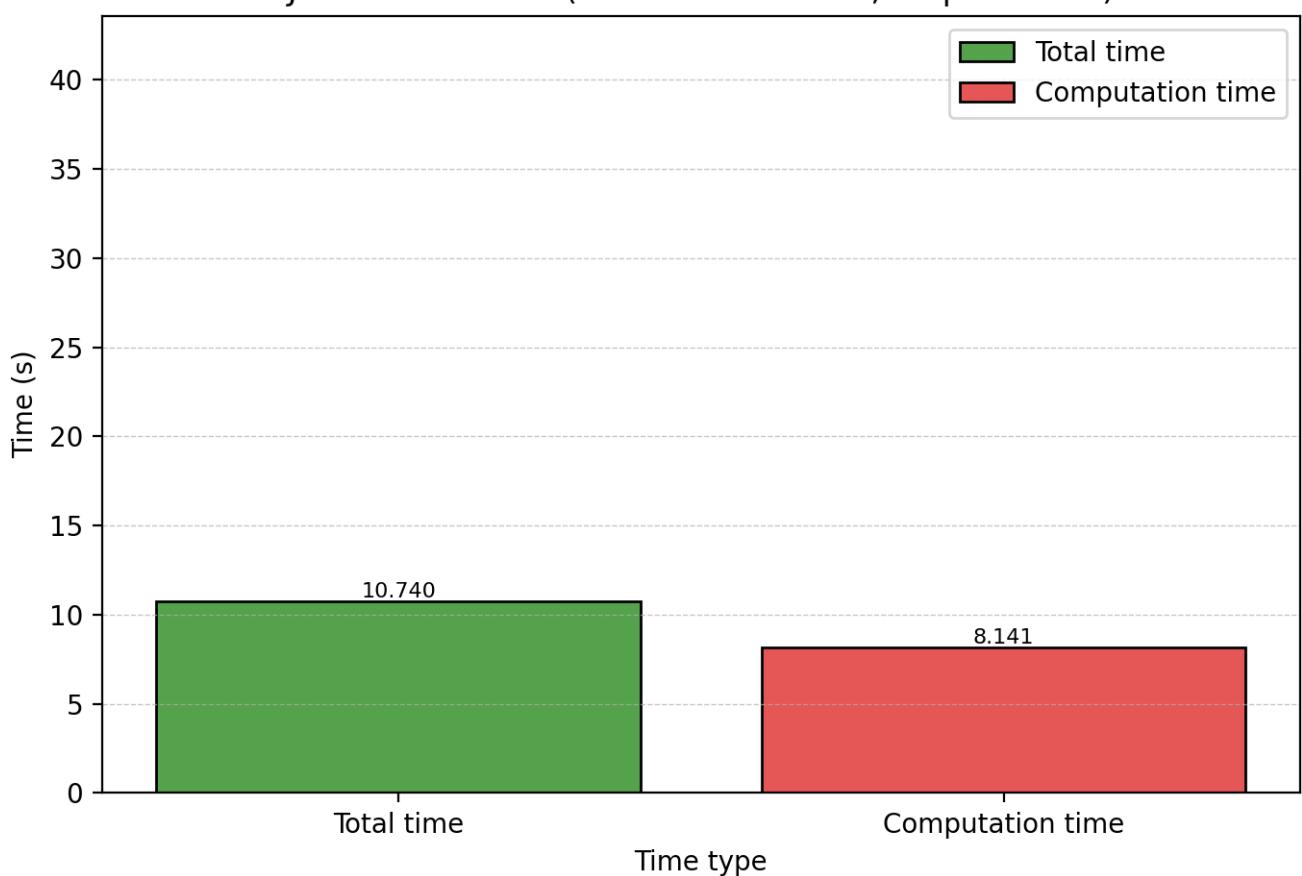
Διαγράμματα χρόνου – Πίνακας 4096x4096



Jacobi MPI Times (Matrix 6144x6144, 32 processes)



Jacobi MPI Times (Matrix 6144x6144, 64 processes)



Γ. Συμπεράσματα

Παρακάτω σχολιάζουμε συνολικά τα speedups και, με βάση τα διαγράμματα Total vs Computation time, τεκμηριώνουμε πού ξοδεύεται ο χρόνος και γιατί εμφανίζεται κορεσμός.

1) Speedup – Strong scaling τάση ανά μέγεθος πλέγματος

(α) Matrix 2048×2048

Speedup: 1.39 (P=2), 1.91 (P=4), 1.91 (P=8), 3.84 (P=16), 10.06 (P=32), 19.22 (P=64).

Παρατηρείται μικρό κέρδος μέχρι P=16, ενώ από P=32 και P=64 εμφανίζεται απότομη επιτάχυνση, με superlinear συμπεριφορά σε σχέση με τα μικρότερα P (δηλ. το speedup αυξάνει δυσανάλογα όταν διπλασιάζουμε τις διεργασίες). Αυτό μπορεί να εξηγηθεί από memory/cache effects: όσο αυξάνει το P, το τοπικό υποπλέγμα ανά διεργασία μικραίνει και χωράει καλύτερα σε caches (ή/και σε τοπικές NUMA περιοχές), μειώνοντας σημαντικά cache misses και πιέσεις στο memory bandwidth. Έτσι, δεν κερδίζουμε μόνο από τον καθαρό διαμοιρασμό του υπολογισμού, αλλά και από το ότι κάθε διεργασία εκτελεί έναν πιο cache-friendly υπολογισμό με χαμηλότερο κόστος πρόσβασης στη μνήμη, άρα ο χρόνος πέφτει περισσότερο από όσο θα περίμενε κανείς με απλή γραμμική κλιμάκωση. Παρόλα αυτά, το τελικό speedup 19.22 σε P=64 απέχει από το ιδανικό 64 (άρα το efficiency παραμένει χαμηλό), κάτι που δείχνει ότι το κόστος επικοινωνίας/συγχρονισμών και τα σταθερά overheads (halo exchanges, wait, πιθανό contention στο interconnect) εξακολουθούν να περιορίζουν τη συνολική κλιμάκωση, ειδικά σε τόσο μικρό πρόβλημα.

(β) Matrix 4096×4096

Speedup: 1.39 (P=2), 1.92 (P=4), 1.91 (P=8), 3.55 (P=16), 5.98 (P=32), 7.49 (P=64).

Στο μέγεθος 4096×4096 το speedup αυξάνει πιο ομαλά σε σχέση με το 2048×2048, δηλαδή βλέπουμε την αναμενόμενη τάση χωρίς απότομα άλματα: όσο αυξάνονται οι διεργασίες, ο χρόνος μειώνεται και το speedup μεγαλώνει. Αυτό είναι συμβατό με το ότι το πρόβλημα έχει πλέον αρκετό υπολογιστικό φορτίο ώστε το κόστος ανά iteration (υπολογισμός στο εσωτερικό) να παραμένει σημαντικό και να αποσβένει τα επικοινωνιακά overheads στα μικρότερα P.

Ωστόσο, μετά το $P=32$ εμφανίζεται σαφής κορεσμός: από 32 σε 64 διεργασίες κερδίζουμε σχετικά λίγο ($5.98 \rightarrow 7.49$). Η συμπεριφορά αυτή εξηγείται από τη Θεωρία: καθώς αυξάνεται το P , το τοπικό υποπλέγμα κάθε διεργασίας μικραίνει, άρα ο λόγος οριακών κελιών προς εσωτερικά αυξάνει και η σχετική σημασία του halo exchange μεγαλώνει. Επιπλέον, σε υψηλά P το κόστος της επικοινωνίας γίνεται όλο και πιο latency-dominated (πολλά μικρά μηνύματα/συχνές ανταλλαγές) και οι συγχρονισμοί (MPI_Waitall, implicit sync ανά iteration) επιβάλλουν έναν bulk-synchronous ρυθμό, όπου η συνολική πρόοδος περιορίζεται από τις καθυστερήσεις των πιο αργών διεργασιών και από πιθανό contention στο interconnect. Άρα, παρότι συνεχίζουμε να μειώνουμε τον καθαρό υπολογισμό ανά διεργασία, το μη-κλιμακούμενο μέρος (επικοινωνία + αναμονές) αρχίζει να κυριαρχεί και η επιπλέον παραλληλία μετά το $P=32$ αποδίδει φθίνον κέρδος. Γενικά, απέχουμε πολύ από την ιδανική (γραμμική) κλιμάκωση.

(γ) Matrix 6144×6144

Speedup: 1.39 ($P=2$), 1.93 ($P=4$), 1.91 ($P=8$), 3.56 ($P=16$), 6.00 ($P=32$), 7.05 ($P=64$).

Στο μεγαλύτερο πλέγμα 6144×6144 παρατηρούμε παρόμοια εικόνα με το 4096×4096 : το speedup αυξάνει αρχικά με τρόπο αναμενόμενο (strong scaling), αλλά μετά το $P=32$ εμφανίζεται καθαρός κορεσμός και στο $P=64$ καταλήγουμε σε τελικό speedup περίπου $7\times$ ($6.00 \rightarrow 7.05$ από 32 σε 64). Η βασική ερμηνεία είναι ότι, παρότι το μεγαλύτερο πλέγμα είναι πιο compute-intense (άρα ευνοεί γενικά τον παραλληλισμό στα μικρότερα P), σε υψηλές τιμές P το τοπικό block ανά διεργασία μικραίνει αρκετά ώστε να αυξηθεί ο λόγος επιφάνειας/όγκου και να ενισχυθεί η επίδραση της επικοινωνίας ορίων.

Συγκεκριμένα, η επικοινωνία halo ανά iteration παραμένει υποχρεωτική και συνοδεύεται από synchronization (Waitall), οπότε όσο μειώνεται ο υπολογισμός ανά διεργασία, το σταθερό/μη-κλιμακούμενο κόστος (latency των μικρών μηνυμάτων, κόστος δικτύου, και αναμονές λόγω slowest rank) αρχίζει να κυριαρχεί. Έτσι, η μετάβαση από 32 σε 64 διεργασίες δεν μπορεί να διπλασιάσει την απόδοση: το compute μειώνεται, αλλά το communication+sync δεν ακολουθεί, οδηγώντας σε φθίνον κέρδος. Με άλλα λόγια, στο 6144×6144 ο παραλληλισμός είναι καλύτερος από άποψη διαθέσιμου υπολογιστικού έργου, όμως στο πολύ υψηλό P μπαίνουμε στην περιοχή όπου το πρόβλημα περιορίζεται κυρίως από επικοινωνία και συγχρονισμούς, άρα εμφανίζεται ο ίδιος strong-scaling κορεσμός.

Συμπέρασμα speedup

Καθώς αυξάνεται το μέγεθος του πλέγματος, το πρόβλημα γίνεται συνολικά πιο compute-intense: ανά διεργασία αυξάνεται ο αριθμός αριθμητικών πράξεων που αντιστοιχεί σε κάθε ανταλλαγή ορίων (halo exchange), άρα βελτιώνεται ο λόγος υπολογισμού προς επικοινωνία. Αυτό θεωρητικά ευνοεί την κλιμάκωση, επειδή τα κόστη επικοινωνίας (latency + bandwidth) και οι συγχρονισμοί δεν μειώνονται με τον ίδιο ρυθμό όπως ο υπολογισμός όταν αυξάνει το P , ενώ σε μεγαλύτερα πλέγματα ο υπολογισμός παραμένει αρκετά βαρύς ώστε να αποσβένονται καλύτερα τα σταθερά επικοινωνιακά κόστη ανά iteration.

Συμπέρασμα από τα διαγράμματα: και στα τρία μεγέθη (2048×2048 , 4096×4096 , 6144×6144) παρατηρούμε strong scaling μέχρι ένα σημείο, δηλαδή η αύξηση των διεργασιών μειώνει τον χρόνο εκτέλεσης. Ωστόσο η κλιμάκωση δεν είναι γραμμική, κάτι αναμενόμενο από τη θεωρία strong scaling (Amdahl + communication overhead): υπάρχει ένα μη-κλιμακούμενο μέρος που σχετίζεται με ανταλλαγές ορίων, latency-dominated μηνύματα και αναμονές/συγχρονισμούς (π.χ. Waitall ανά iteration), το οποίο αποκτά ολοένα μεγαλύτερη σχετική βαρύτητα όσο το per-process υπολογιστικό φορτίο μειώνεται.

Για τα μεγαλύτερα πλέγματα (4096×4096 και 6144×6144) εμφανίζεται καθαρός κορεσμός από $P \geq 32$ και μετά (μικρό πρόσθετο κέρδος από $32 \rightarrow 64$). Η εικόνα αυτή ταιριάζει με τη θεωρία: όταν το τοπικό υποπλέγμα ανά διεργασία μικραίνει, αυξάνεται ο λόγος περιμέτρου προς εμβαδόν, άρα η σχετική επίδραση των halo exchanges και των συγχρονισμών μεγαλώνει και περιορίζει το επιπλέον όφελος της παραλληλίας. Με άλλα λόγια, ενώ ο υπολογισμός συνεχίζει να μειώνεται, το communication+sync δεν ακολουθεί, οδηγώντας σε φθίνον κέρδος.

Στο μικρότερο πλέγμα (2048×2048) παρατηρείται απότομη βελτίωση στα $P=32$ και $P=64$, με superlinear συμπεριφορά σε σχέση με τα μικρότερα P . Αυτό είναι συμβατό με τα cache/memory effects: το τοπικό block ανά διεργασία γίνεται αρκετά μικρό ώστε να χωράει καλύτερα σε caches και/ή να βελτιώνεται η NUMA locality, μειώνοντας δυσανάλογα τα cache misses και τις αργές προσπελάσεις στη μνήμη. Έτσι δεν κερδίζουμε μόνο από τον καθαρό επιμερισμό του compute, αλλά και από πιο αποδοτική εκτέλεση του ίδιου υπολογισμού λόγω καλύτερης iεραρχίας μνήμης. Παρόλα αυτά, το speedup παραμένει αρκετά κάτω από το ιδανικό P σε υψηλά P , κάτι που δείχνει ότι τα επικοινωνιακά/συγχρονιστικά κόστη (halo exchange, latency, wait) εξακολουθούν να θέτουν το τελικό όριο στην κλιμάκωση.

2) Total vs Computation time

Από τα time-plots μπορούμε να εκτιμήσουμε το communication + synchronization overhead ως:

$$\text{Toverhead} = \text{Ttotal} - \text{Tcomp.}$$

(α) Matrix 2048×2048

- $P=8$: $\text{Ttotal}=4.409\text{s}$, $\text{Tcomp}=4.195\text{s} \rightarrow \text{Toverhead}=0.214\text{s}$ ($\sim 4.9\%$ του total)
- $P=16$: $\text{Ttotal}=2.189\text{s}$, $\text{Tcomp}=1.845\text{s} \rightarrow \text{Toverhead}=0.344\text{s}$ ($\sim 15.7\%$)
- $P=32$: $\text{Ttotal}=0.836\text{s}$, $\text{Tcomp}=0.488\text{s} \rightarrow \text{Toverhead}=0.348\text{s}$ ($\sim 41.6\%$)
- $P=64$: $\text{Ttotal}=0.438\text{s}$, $\text{Tcomp}=0.058\text{s} \rightarrow \text{Toverhead}=0.380\text{s}$ ($\sim 86.8\%$)

Εδώ φαίνεται καθαρά η strong-scaling συμπεριφορά: όσο αυξάνει το P , ο υπολογισμός ανά διεργασία μειώνεται έντονα, ενώ το κόστος των halo exchanges και των αναμονών (Waitall) δεν μειώνεται με τον ίδιο ρυθμό. Παράλληλα, στο 2048×2048 εμφανίζεται και το φαινόμενο superlinear βελτίωσης κυρίως στο άλμα $16 \rightarrow 32 \rightarrow 64$, που συμβαδίζει με την παρατήρηση ότι το Tcomp πέφτει δυσανάλογα (ιδίως στο $P=64$: 0.058s). Η πιο πιθανή αιτία είναι τα cache/memory effects: με αρκετά μικρό local block, ο υπολογισμός γίνεται πολύ πιο cache-friendly (λιγότερα cache misses, καλύτερη temporal/spatial locality, πιθανώς καλύτερη NUMA τοπικότητα), οπότε ο ίδιος αριθμός ενημερώσεων εκτελείται ταχύτερα από ότι θα προέβλεπε ο απλός επιμερισμός 1/P.

Ωστόσο, το ίδιο διάγραμμα δείχνει και το τελικό όριο της strong scaling κλιμάκωσης: ενώ το Tcomp σχεδόν μηδενίζεται στο $P=64$, το Toverhead παραμένει ~σταθερό (≈ 0.35 – 0.38s από $P=16$ και μετά) και τελικά κυριαρχεί στο συνολικό χρόνο ($\approx 86.8\%$ στο $P=64$). Άρα, στο μεγάλο P το σύστημα γίνεται καθαρά communication/synchronization-bound: η επίδοση δεν περιορίζεται πλέον από τον υπολογιστικό πυρήνα, αλλά από latency, ανταλλαγές ορίων και τον συγχρονισμό που επιβάλλει το Waitall ανά iteration.

(β) Matrix 4096×4096

- P=8: $17.646 - 17.165 = 0.481\text{s}$ ($\sim 2.7\%$)
- P=16: $9.477 - 8.481 = 0.996\text{s}$ ($\sim 10.5\%$)
- P=32: $5.627 - 4.498 = 1.129\text{s}$ ($\sim 20.1\%$)
- P=64: $4.492 - 3.182 = 1.310\text{s}$ ($\sim 29.2\%$)

Για το 4096×4096, τα time-plots δείχνουν ότι το overhead (Ttotal-Tcomp) αυξάνει τόσο σε απόλυτη τιμή όσο και ως ποσοστό του συνολικού χρόνου, κάτι που εξηγεί άμεσα τον κορεσμό του speedup μετά το P=32. Συγκεκριμένα, το Toverhead ανεβαίνει από $\sim 0.48\text{s}$ (P=8) σε $\sim 1.00\text{s}$ (P=16), $\sim 1.13\text{s}$ (P=32) και $\sim 1.31\text{s}$ (P=64), ενώ παράλληλα το ποσοστό του overhead αυξάνει περίπου από $\sim 2.7\%$ σε $\sim 10.5\%$, $\sim 20.1\%$ και $\sim 29.2\%$. Αυτό σημαίνει ότι, όσο αυξάνουμε τις διεργασίες, οι ανταλλαγές halo και οι αναμονές/synchronization ανά iteration καταναλώνουν όλο και μεγαλύτερο μέρος του συνολικού χρόνου.

Παρότι ο υπολογισμός παραμένει ακόμη το κυρίαρχο κομμάτι (το Tcomp δεν είναι αμελητέο), η μείωσή του από P=32 → P=64 είναι περιορισμένη ($4.498\text{s} \rightarrow 3.182\text{s}$), άρα ο διπλασιασμός διεργασιών δεν μπορεί να φέρει αντίστοιχο κέρδος. Η επιπλέον παραλληλία μεταφράζεται κυρίως σε: (i) υψηλότερο σχετικό κόστος halo exchange (surface/volume effect καθώς μικραίνει το local block), (ii) latency-dominated επικοινωνία σε μικρότερα μηνύματα και (iii) αυξημένες αναμονές στο Waitall/critical path (η πιο αργή διεργασία καθορίζει τον ρυθμό). Έτσι, μετά το P=32 η εκτέλεση μπαίνει σταδιακά σε μια περιοχή όπου το communication/synchronization και η μειωμένη αποδοτικότητα πόρων (π.χ. contention σε interconnect/μνήμη) περιορίζουν το πρόσθετο όφελος και οδηγούν σε κορεσμό του speedup.

(γ) Matrix 6144×6144

- P=8: $39.624 - 38.524 = 1.100\text{s}$ ($\sim 2.8\%$)
- P=16: $21.258 - 19.327 = 1.931\text{s}$ ($\sim 9.1\%$)
- P=32: $12.612 - 10.255 = 2.357\text{s}$ ($\sim 18.7\%$)
- P=64: $10.740 - 8.141 = 2.599\text{s}$ ($\sim 24.2\%$)

Για το 6144×6144 παρατηρούμε την ίδια βασική εικόνα με το 4096×4096, αλλά σε απόλυτα μεγαλύτερους χρόνους: το overhead (Ttotal-Tcomp) αυξάνει συστηματικά με το P, ενώ η μείωση του υπολογιστικού χρόνου από P=32 → P=64

είναι πλέον σχετικά περιορισμένη. Από τα δεδομένα προκύπτει ότι το Toverhead ανεβαίνει από $\sim 1.10s$ ($P=8$) σε $\sim 1.93s$ ($P=16$), $\sim 2.36s$ ($P=32$) και $\sim 2.60s$ ($P=64$), ενώ το πιο συστό του overhead αυξάνει αντίστοιχα ($\sim 2.8\% \rightarrow \sim 9.1\% \rightarrow \sim 18.7\% \rightarrow \sim 24.2\%$). Άρα, όσο αυξάνεται το πλήθος διεργασιών, οι ανταλλαγές halo και οι αναμονές/synchronization γίνονται ολοένα πιο σημαντικό τμήμα του συνολικού χρόνου.

Παρότι το πρόβλημα είναι πιο compute-intensive (άρα αντέχει καλύτερα την κλιμάκωση στα μικρότερα P), σε υψηλά P κυριαρχεί ξανά το surface/volume effect: το τοπικό block μικραίνει, η σχετική περίμετρος αυξάνει, και ο χρόνος επικοινωνίας γίνεται πιο latency-dominated. Επιπλέον, το Waitall ανά iteration επιβάλλει bulk-synchronous ρυθμό, όπου τυχόν καθυστερήσεις (π.χ. λόγω contention στο interconnect ή ανομοιομορφιών στους κόμβους) μετατρέπονται σε πραγματικό κόστος αναμονής για όλους. Έτσι, ο διπλασιασμός από $32 \rightarrow 64$ διεργασίες μειώνει μεν το Tcomp ($10.255s \rightarrow 8.141s$), αλλά το Toverhead παραμένει σημαντικό και αυξάνει, με αποτέλεσμα το συνολικό κέρδος να είναι μικρό ($12.612s \rightarrow 10.740s$) και το speedup να εμφανίζει κορεσμό μετά το $P=32$ — ακριβώς όπως προβλέπει η θεωρία strong scaling όταν τα communication/synchronization κόστη γίνονται συγκρίσιμα με τον υπολογισμό.

3) Περαιτέρω ερμηνεία

- Surface/volume effect: Με 2D decomposition, το compute ανά διεργασία είναι περίπου $O(N^2/P)$, ενώ η επικοινωνία halo είναι περίπου $O(N/\sqrt{P})$ ανά διεργασία (περίμετρος του block). Ο λόγος επικοινωνίας/υπολογισμού αυξάνει όσο αυξάνει το P (και όσο μειώνεται το τοπικό block), άρα το efficiency πέφτει.
- Latency dominance: Σε μεγάλα P , τα μηνύματα γίνονται μικρότερα. Τότε το latency (α) κυριαρχεί έναντι του bandwidth ($\beta \cdot m$), άρα δεν μπορούμε να κερδίσουμε όσο θα περιμέναμε από το $1/P$ του υπολογισμού.
- Συγχρονισμός/critical path: Παρότι χωρίς convergence δεν έχουμε Allreduce, κάθε iteration έχει πρακτικά συγχρονισμό μέσω Waitall (και μέσω της ανάγκης να έχουν φτάσει τα halo πριν τον υπολογισμό). Έτσι η πιο αργή διεργασία καθορίζει την ταχύτητα όλων, ειδικά σε περιπτώσεις μικρής ανισορροπίας ή θορύβου από το σύστημα.
- Περιορισμοί πόρων (μνήμη/NUMA/oversubscription): Σε υψηλά P , μπορεί να εμφανιστεί κορεσμός bandwidth μνήμης, contention σε shared πόρους, ή/και μειωμένο locality. Αυτό εξηγεί γιατί σε $4096/6144$ το compute δεν συνεχίζει να μειώνεται ανάλογα με το P και γιατί το speedup μετά το 32 δεν διπλασιάζεται.

Σε αντίθεση με την 4.2.1, εδώ δεν υπάρχει κόστος global convergence check (τοπικό κριτήριο + Allreduce). Άρα, οποιαδήποτε απόκλιση από ιδανικό speedup οφείλεται κυρίως σε:

- halo exchanges ανά iteration,
- synchronization/αναμονές,
- και γενικότερη strong-scaling αναποτελεσματικότητα (surface/volume, latency, πόροι).

Αυτό επιβεβαιώνεται από το γεγονός ότι το $T_{overhead} = T_{total} - T_{comp}$ αυξάνει συστηματικά με το P και τελικά περιορίζει τον ρυθμό βελτίωσης.

Τελικό συμπέρασμα

Χωρίς έλεγχο σύγκλισης (σταθερό $T=256$), η υλοποίηση αυτή παρουσιάζει strong scaling (αλλά πάντα μακριά από το ιδανικό) μέχρι ~ 32 διεργασίες, αλλά από εκεί και πάνω εμφανίζεται κορεσμός, ιδιαίτερα στα μεγαλύτερα πλέγματα 4096×4096 και 6144×6144 . Τα time-plots δείχνουν καθαρά ότι, όσο αυξάνεται το P , ο υπολογισμός μειώνεται, αλλά το επικοινωνιακό/συγχρονιστικό overhead αυξάνει (σε ποσοστό και συχνά και σε απόλυτη τιμή), οδηγώντας σε χαμηλό efficiency και περιορισμένο επιπλέον κέρδος στα $P=64$.

■ Γενικά Συμπεράσματα

Στην Άσκηση 4 μελετήσαμε την παραλληλοποίηση και βελτιστοποίηση σε αρχιτεκτονικές κατανεμημένης μνήμης με MPI, πάνω σε δύο διαφορετικά μοτίβα υπολογισμού: (i) K-means (κυρίως data-parallel υπολογισμός με συλλογικές επικοινωνίες ανά επανάληψη) και (ii) Jacobi 2D (υπολογισμός με γειτονική επικοινωνία και halo exchange ανά επανάληψη, και προαιρετικό έλεγχο σύγκλισης με global reduction). Τα αποτελέσματα επιβεβαιώνουν τη θεωρία: όσο αυξάνουμε τον αριθμό διεργασιών για σταθερό πρόβλημα, ο καθαρός υπολογισμός ανά διεργασία μειώνεται, αλλά το κόστος επικοινωνίας/συγχρονισμών δεν μειώνεται αντίστοιχα και τελικά γίνεται ο κύριος περιοριστικός παράγοντας (Amdahl + communication overhead).

1) K-means με MPI

Ο K-means παρουσίασε ιδιαίτερα καλή κλιμάκωση στο συγκεκριμένο configuration (Size=256MB, Coords=16, Clusters=32, Loops=10), με χρόνο από 27.573s (sequential) έως 0.464s στα 64 processes και speedup \sim 59.4x. Αυτό είναι αναμενόμενο επειδή:

- Το κυρίαρχο κόστος είναι η ανάθεση σημείων σε κέντρα (assignment step), που είναι πλήρως data-parallel και διαμοιράζεται σχεδόν ιδανικά στα processes.
- Η επικοινωνία ανά iteration είναι κυρίως συλλογική με μικρό μήνυμα (π.χ. αθροίσματα/μετρήσεις για $K \times D$, και counts για K). Άρα η επιβάρυνση είναι περισσότερο latency-dominated αλλά μικρής συνολικής διάρκειας σε σχέση με τον υπολογισμό πάνω σε εκατομμύρια objects.
- Η συγκέντρωση/συγχρονισμός ανά iteration δημιουργεί σημείο συγχρονισμού (bulk synchronous pattern), όμως η αναλογία υπολογισμού/επικοινωνίας παραμένει ευνοϊκή, για αυτό και η απόδοση παραμένει σχεδόν γραμμική μέχρι υψηλά P.

2) Jacobi 2D χωρίς έλεγχο σύγκλισης (σταθερές επαναλήψεις)

Χωρίς έλεγχο σύγκλισης, οι μετρήσεις απομονώνουν καλύτερα το καθαρό κόστος. Παρατηρούμε ότι:

- Για μικρότερο πλέγμα (2048×2048), ο χρόνος επικοινωνίας/αναμονών γίνεται συγκρίσιμος ή και μεγαλύτερος από τον υπολογισμό όταν αυξάνει το P , επειδή το τοπικό υποπλέγμα μικραίνει και ο λόγος επιφάνειας/όγκου (surface/volume) χειροτερεύει. Αυτό φαίνεται χαρακτηριστικά στα 64 processes, όπου ο υπολογισμός πέφτει πολύ (0.058s) αλλά ο συνολικός χρόνος δεν ακολουθεί ανάλογα (0.438s), άρα κυριαρχεί πλέον το κόστος επικοινωνίας και συγχρονισμού.
- Για μεγαλύτερα πλέγματα (4096×4096 και 6144×6144) η κλιμάκωση βελτιώνεται αρχικά, γιατί αυξάνει το computation per process. Ωστόσο, μετά από ένα σημείο (π.χ. από $32 \rightarrow 64$ processes) το όφελος μειώνεται αισθητά: ο συνολικός χρόνος συνεχίζει να μειώνεται, αλλά με ολοένα μικρότερο ρυθμό, επειδή οι ανταλλαγές ορίων και τα Wait/Sync κόστη αυξάνουν με το P και με τον αριθμό γειτόνων ανά iteration.

Θεωρητικά, το halo exchange έχει κόστος που προσεγγίζει $\alpha + \beta \cdot m$ ανά μήνυμα (latency + bandwidth term), όπου m είναι το μέγεθος των γραμμών/στηλών που ανταλλάσσονται. Καθώς m μειώνεται με το partitioning, ο bandwidth όρος μικραίνει, αλλά ο latency όρος (α) και οι συγχρονισμοί (Waitall) παραμένουν, οπότε το ποσοστό μη-χρήσιμου χρόνου μεγαλώνει.

3) Jacobi 2D με έλεγχο σύγκλισης

Με ενεργό έλεγχο σύγκλισης προστίθεται επιπλέον κόστος από global reductions (π.χ. MPI_Allreduce) κάθε C επαναλήψεις. Επειδή το μήνυμα είναι πολύ μικρό (flag/int), το κόστος είναι σχεδόν εξ ολοκλήρου latency-dominated και αυξάνει περίπου με $O(\log P)$ σε βήματα επικοινωνίας, ενώ επιβάλλει και καθολικό συγχρονισμό (όλοι τα ranks πρέπει να φτάσουν στο σημείο ελέγχου). Άρα:

- Η συχνότητα ελέγχου (C) επηρεάζει άμεσα τον χρόνο: μικρό C \rightarrow περισσότερα global syncs \rightarrow χειρότερο strong scaling.
- Παράλληλα, ο έλεγχος μπορεί να μειώσει το συνολικό πλήθος επαναλήψεων όταν υπάρχει έγκαιρη σύγκλιση, άρα υπάρχει trade-off λιγότερες επαναλήψεις έναντι ακριβότερης επανάληψης.

Στο συγκεκριμένο πείραμα (512×512 , 64 processes), ο συνολικός χρόνος είναι πολύ μεγαλύτερος από τον καθαρό υπολογισμό, γεγονός που επιβεβαιώνει ότι σε μικρό πρόβλημα/υψηλό Ρ κυριαρχούν επικοινωνίες/συγχρονισμοί και όχι το υπολογιστικό kernel.

4) Σύγκριση MPI με OpenMP (σε επίπεδο συμπεριφοράς/θεωρίας)

Η σύγκριση με OpenMP αναδεικνύει την ουσιαστική διαφορά κοινής ενάντια σε κατανεμημένη μνήμη:

- Στο OpenMP, η επικοινωνία μεταξύ threads είναι implicit μέσω κοινής μνήμης. Αυτό μειώνει δραστικά το latency σε σχέση με MPI, αλλά εισάγει άλλους περιορισμούς: contention σε shared δεδομένα, memory bandwidth saturation, false sharing και NUMA effects. Επιπλέον, απαιτούνται μηχανισμοί reduction/atomics/critical για αθροίσεις (π.χ. K-means), που μπορεί να περιορίσουν την κλιμάκωση σε πολλά threads.
- Στο MPI, η κλιμάκωση σε πολλούς κόμβους είναι φυσική και το working set ανά process μειώνεται, αλλά πληρώνουμε ρητή επικοινωνία (latency/bandwidth) και συγχρονισμούς (collectives, Waits). Για τέτοια προβλήματα, η απόδοση εξαρτάται έντονα από το surface/volume και την τοπολογία επικοινωνίας (neighbors), ενώ για data-parallel με μικρά collectives (K-means) η κλιμάκωση μπορεί να είναι πολύ καλή.

Συνολικά, σε single-node περιβάλλον το OpenMP συχνά υπερέχει για μικρό/μεσαίο Ρ λόγω χαμηλού communication overhead, ενώ σε multi-node/μεγάλα δεδομένα η MPI προσφέρει την απαραίτητη κατανεμημένη κλιμάκωση. Πρακτικά, η βέλτιστη προσέγγιση σε σύγχρονα συστήματα είναι συχνά υβριδική (MPI μεταξύ κόμβων + OpenMP εντός κόμβου), ώστε να μειωθούν τα MPI ranks και τα halo/collective κόστη.