

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ



Συστήματα Παράλληλης Επεξεργασίας

9ο Εξάμηνο, 2024-2025

Εργαστηριακή Αναφορά

των φοιτητών:

Λάζου Μαρία-Αργυρώ (el20129)

Σπηλιώτης Αθανάσιος (el20175)

Ομάδα: **parlab09**

Conway's GameofLife

Υλοποίηση

Για την παραλληλοποίηση του αλγορίθμου τροποποίησαμε τον κώδικα που δίνεται προσθέτοντας απλώς το #pragma directive στο κύριο loop για τα (i,j) του body:

Game_Of_Life.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
15 #include <stdio.h>
16 #include <stdlib.h>
17 #include <sys/time.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     int N;           //array dimensions
31     int T;           //time steps
32     int ** current, ** previous; //arrays - one for current timestep, one for previous timestep
33     int ** swap;      //array pointer
34     int t, i, j, nbrs; //helper variables
35
36     double time;      //variables for timing
37     struct timeval ts,tf;
38
39     /*Read input arguments*/
40     if ( argc != 3 ) {
41         fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
42         exit(-1);
43     }
44     else {
45         N = atoi(argv[1]);
46         T = atoi(argv[2]);
47     }
48
49     /*Allocate and initialize matrices*/
50     current = allocate_array(N);        //allocate array for current time step
51     previous = allocate_array(N);       //allocate array for previous time step
52
53     init_random(previous, current, N); //initialize previous array with pattern
54
55     #ifdef OUTPUT
56     print_to_pgm(previous, N, 0);
57     #endif
58
59     /*Game of Life*/
60
61     gettimeofday(&ts,NULL);
62     for ( t = 0 ; t < T ; t++ ) {
63         #pragma omp parallel for shared(current, previous) private (nbrs, i, j)
64         for ( i = 1 ; i < N-1 ; i++ ) {
```

```

65     for ( j = 1 ; j < N-1 ; j++ ) {
66         nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
67             + previous[i][j-1] + previous[i][j+1] \
68             + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
69         if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
70             current[i][j]=1;
71         else
72             current[i][j]=0;
73     }
74 }
75
76 #ifdef OUTPUT
77 print_to_pgm(current, N, t+1);
78#endif
79 //Swap current array with previous array
80 swap=current;
81 current=previous;
82 previous=swap;
83 }
84 gettimeofday(&tf,NULL);
85 time=(tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec)*0.000001;
86
87 free_array(current, N);
88 free_array(previous, N);
89 printf("GameOfLife: Size %d Steps %d Time %lf\n", N, T, time);
90 #ifdef OUTPUT
91 system(FINALIZE);
92#endif
93 }
94
95 int ** allocate_array(int N) {
96     int ** array;
97     int i,j;
98     array = malloc(N * sizeof(int*));
99     for ( i = 0; i < N ; i++ )
100         array[i] = malloc( N * sizeof(int));
101     for ( i = 0; i < N ; i++ )
102         for ( j = 0; j < N ; j++ )
103             array[i][j] = 0;
104     return array;
105 }
106
107 void free_array(int ** array, int N) {
108     int i;
109     for ( i = 0 ; i < N ; i++ )
110         free(array[i]);
111     free(array);
112 }
113
114 void init_random(int ** array1, int ** array2, int N) {
115     int i,pos,x,y;
116
117     for ( i = 0 ; i < (N * N)/10 ; i++ ) {
118         pos = rand() % ((N-2)*(N-2));
119         array1[pos%(N-2)+1][pos/(N-2)+1] = 1;
120         array2[pos%(N-2)+1][pos/(N-2)+1] = 1;
121
122     }
123 }
124
125 void print_to_pgm(int ** array, int N, int t) {
126     int i,j;
127     char * s = malloc(30*sizeof(char));
128     sprintf(s,"out%d.pgm",t);
129     FILE * f = fopen(s,"wb");
130     fprintf(f, "P5\n%d %d 1\n", N,N);
131     for ( i = 0; i < N ; i++ )
132         for ( j = 0; j < N ; j++ )
133             if ( array[i][j]==1 )
134                 fputc(1,f);
135             else
136                 fputc(0,f);
137     fclose(f);
138 }
```

```
139     free(s);  
140 }
```

Για την μεταγλώτιση και εκτέλεση στον scirouter χρησιμοποίησαμε το ακόλουθα scripts :

```
#!/bin/bash  
## Give the Job a descriptive name  
#PBS -N make_gameoflife  
## Output and error files  
#PBS -o make_gameoflife.out  
#PBS -e make_gameoflife.err  
## How many machines should we get?  
#PBS -l nodes=1:ppn=1  
## Start  
## Run make in the src folder (modify properly)  
module load openmpi/1.8.3  
cd /home/parallel/parlab09/a1  
make
```

```
#!/bin/bash  
## Give the Job a descriptive name  
#PBS -N run_gameoflife  
## Output and error files  
#PBS -o omp_gameoflife_all.out  
#PBS -e omp_gameoflife_all.err  
## Limit memory, runtime etc.  
#PBS -l walltime=01:00:00  
##Number of nodes aka threads  
#PBS -l nodes=1:ppn=8  
module load openmpi/1.8.3  
cd /home/parallel/parlab09/a1  
for threads in 1 2 4 6 8  
do  
export OMP_NUM_THREADS=$threads  
echo "Running with OMP_NUM_THREADS=$OMP_NUM_THREADS"  
.omp_gameoflife 64 1000  
.omp_gameoflife 1024 1000  
.omp_gameoflife 4096 1000  
echo "Finished run with OMP_NUM_THREADS=$OMP_NUM_THREADS"  
echo "-----"  
done
```

Αποτελέσματα Μετρήσεων:

```
Running with OMP_NUM_THREADS=1  
GameOfLife: Size 64 Steps 1000 Time 0.023112  
GameOfLife: Size 1024 Steps 1000 Time 10.965944  
GameOfLife: Size 4096 Steps 1000 Time 175.900314  
Finished run with OMP_NUM_THREADS=1
```

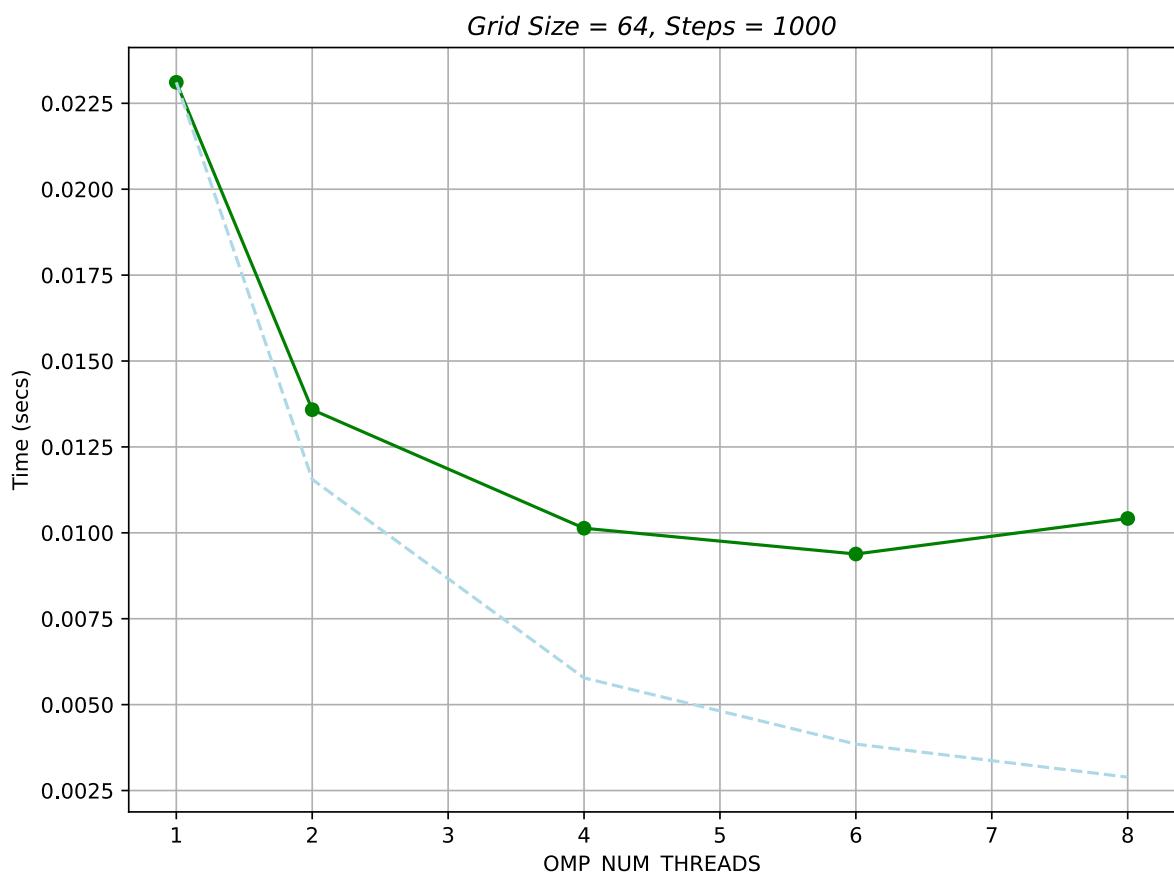
```
-----  
Running with OMP_NUM_THREADS=2  
GameOfLife: Size 64 Steps 1000 Time 0.013583  
GameOfLife: Size 1024 Steps 1000 Time 5.458949  
GameOfLife: Size 4096 Steps 1000 Time 88.263665  
Finished run with OMP_NUM_THREADS=2
```

```
-----  
Running with OMP_NUM_THREADS=4  
GameOfLife: Size 64 Steps 1000 Time 0.010134  
GameOfLife: Size 1024 Steps 1000 Time 2.723798  
GameOfLife: Size 4096 Steps 1000 Time 45.901567  
Finished run with OMP_NUM_THREADS=4  
-----
```

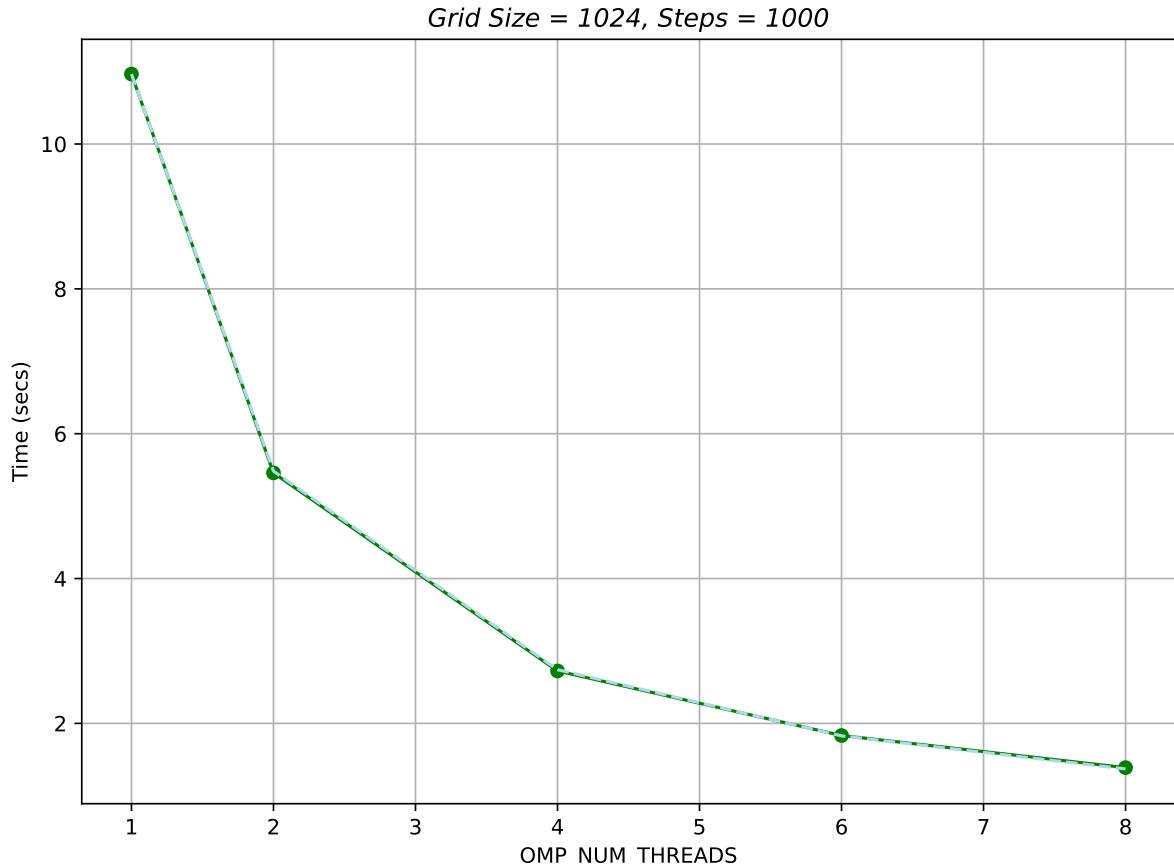
```
Running with OMP_NUM_THREADS=6  
GameOfLife: Size 64 Steps 1000 Time 0.009383  
GameOfLife: Size 1024 Steps 1000 Time 1.832227  
GameOfLife: Size 4096 Steps 1000 Time 43.661123  
Finished run with OMP_NUM_THREADS=6  
-----
```

```
Running with OMP_NUM_THREADS=8  
GameOfLife: Size 64 Steps 1000 Time 0.010417  
GameOfLife: Size 1024 Steps 1000 Time 1.389175  
GameOfLife: Size 4096 Steps 1000 Time 43.186379  
Finished run with OMP_NUM_THREADS=8  
-----
```

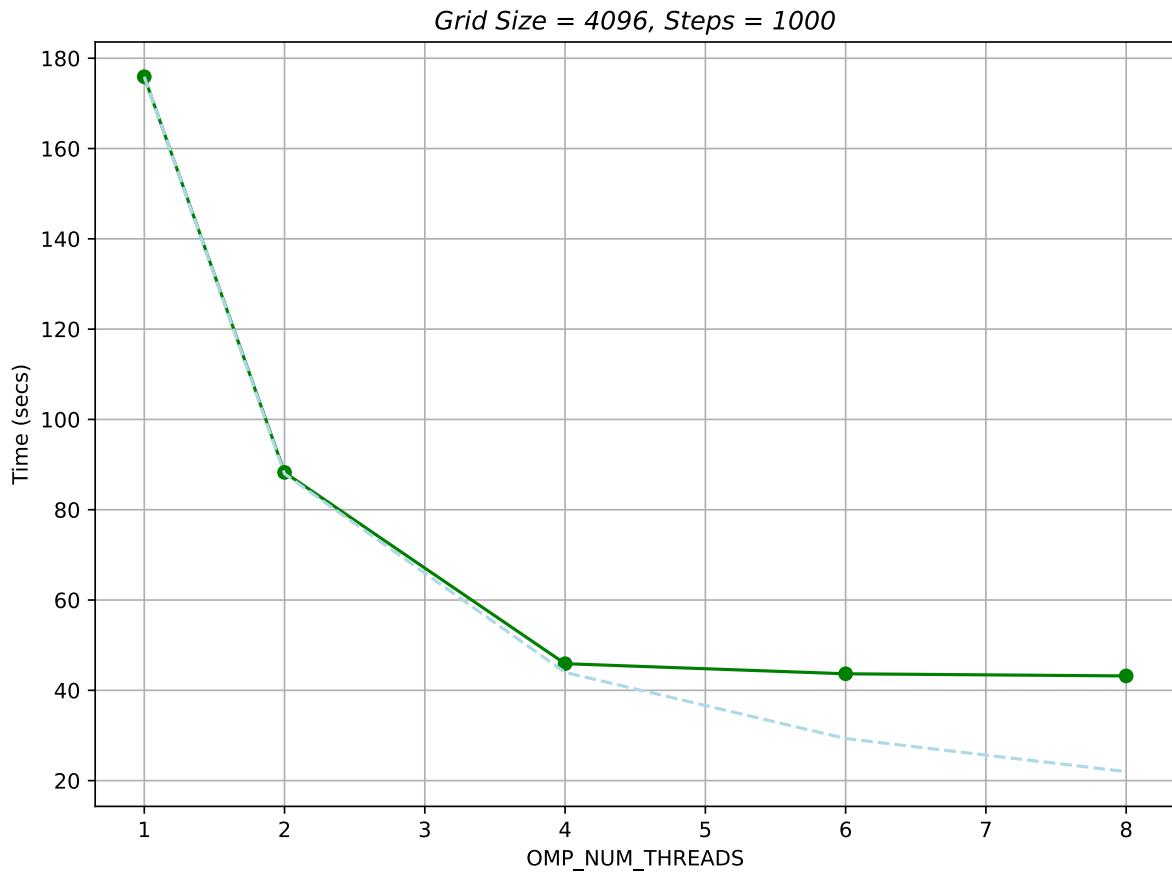
Γραφική Απεικόνιση και Παρατηρήσεις



Παρατηρούμε ότι για μικρό μέγεθος grid (με συνολική απαίτηση μνήμης $4*64*64\text{bytes} = 16\text{KB}$), δεν υπάρχει ομοιόμορφη κλιμάκωση της επίδοσης με αύξηση των νημάτων από 4 και πάνω. Bottleneck κόστους θα θεωρήσουμε την ανάγκη συγχρονισμού των threads και το overhead της δημιουργίας τους συγκρυτικά με τον φόρτο εργασίας που τους ανατίθεται (granularity).



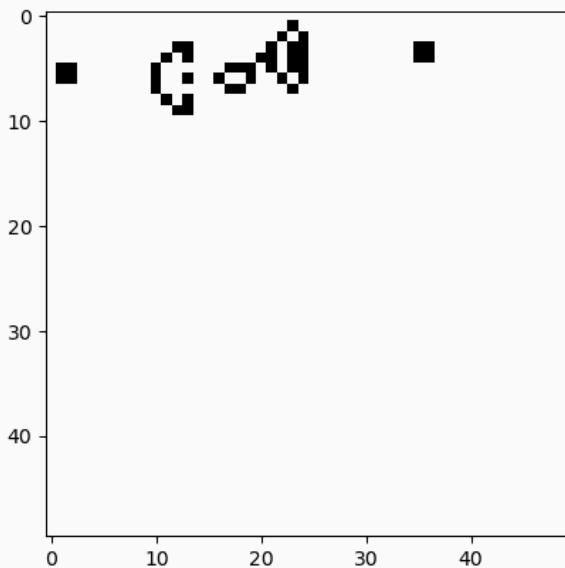
Για μέγεθος grid με συνολική απαίτηση μνήμης $4*1024*1024 \text{ bytes} = 4\text{MB}$, η επίδοση βελτίωνεται ομοιόμορφα και ανάλογα με το μέγεθος των νημάτων . Εικάζουμε, λοιπόν, πως η cache χωράει ολόκληρο το grid ώστε το κάθε νήμα δεν επιβαρύνει την μνήμη με loads των αντίστοιχων rows, ο φόρτος εργασίας είναι ισομοιρασμένος στους workers και το κόστος επικοινωνίας αμελητέο. Συνεπώς, προκύπτει perfect scaling.



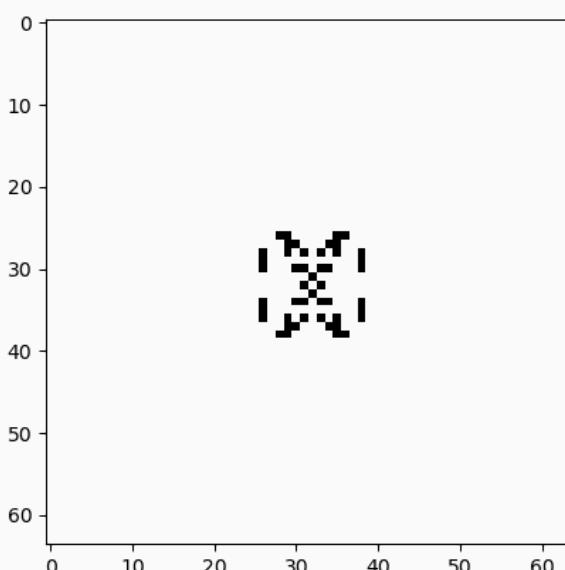
Για μεγάλο grid (με συνολική απαίτηση μνήμης $4 * 4096 * 4096$ bytes = 64MB), η κλιμάκωση παύει να υφίσταται για περισσότερα από 4 νήματα. Bottleneck κόστους εδώ θεωρούμε το memory bandwidth. Επειδή ολόκληρο το grid δεν χωράει στην cache, δημιουργούνται misses όταν ξεχωριστά νήματα προσπαθούν να διαβάσουν ξεχωριστές γραμμές του previous. Σε κάθε memory request αδειάζουν χρήσιμα data για άλλα νήματα, φέρνοντας τις δικές τους γραμμές και στο μεταξύ οι υπολογισμοί stall-άρουν.

Bonus

Δύο ενδιαφέρουσες ειδικές αρχικοποιήσεις του ταμπλό είναι το pulse και το gosper glider gun για τις οποίες η εξέλιξη των γενιών σε μορφή κινούμενης εικόνας φαίνεται με μορφή gif παρακάτω:



glider_gun animation



pulse animation

Πράρτημα

Για την εξαγωγή των γραφικών παραστάσεων χρησιμοποιήθηκε ο κώδικας σε Python που ακολουθεί:

plots.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import re
4 import sys
5
6 outfile = "omp_gameoflife_all.out"
7
8 thread_pattern = r"Running with OMP_NUM_THREADS=(\d+)"
9 time_pattern = r"GameOfLife: Size (\d+) Steps 1000 Time ([\d.]+)"
10
11 with open (outfile, 'r') as fout:
12     data = fout.read()
13
14 thread_vals = re.findall(thread_pattern, data)
15 time_vals = re.findall(time_pattern, data)
16
17 #print(thread_vals, time_vals)
18
19 results_mapping = {}
20
21 for i in range(0, len(thread_vals)):
22     omp_num_thredas = int(thread_vals[i])
23
24     for j in range(0,3):
25         size = int(time_vals[i*3+j][0])
26         time = float(time_vals[i*3+j][1])
27         ## print(f"From {i,j} extracted size: {size} with time: {time}")
28
29         if size not in results_mapping :
30             results_mapping[size] = {}
31
32         results_mapping[size][omp_num_thredas] = time
33
34 for idx, (size, omp_times) in enumerate(results_mapping.items()):
35     print(f"Size: {size}, results: {omp_times}")
36
37     # Create a new figure for each graph
38     plt.figure(figsize=(8, 6))
39
40     # Plot the original times
41     plt.plot(omp_times.keys(), omp_times.values(), color='g', marker='o')
42
43     # Plot the inverse times
44     plt.plot(omp_times.keys(), [omp_times[1] / i for i in omp_times.keys()], color='lightblue',
45     linestyle='--')
46
47     # Add labels and title
48     plt.title(f"Grid Size = {size}, Steps = 1000", fontstyle='oblique', size=12)
49     plt.xlabel("OMP_NUM_THREADS")
50     plt.ylabel("Time (secs)")
51     plt.grid()
52
53     # Show the plot
54     plt.tight_layout()
55     plt.savefig(f"grid{size}.svg", format="svg")
```

KMEANS

1) Shared Clusters

Υλοποίηση

Για την παραλληλοποίηση της συγκεκριμένης έκδοσης χρησιμοπιήσαμε το parallel for directive του omp και για την αποφυγή race conditions τα omp atomic directives. Αυτά εμφανίζονται όταν περισσότερα από 1 νήματα προσπαθούν να ανανεώσουν τιμές στους shared πίνακες newClusters και newClusterSize σε indexes τα οποία δεν είναι μοναδικά για το καθένα καθώς και στην shared μεταβλητή delta. Για αυτήν προσφέρεται η χρήση reduction και εδώ μπορεί να αγνοηθεί εντελώς αφού η σύγκλιση του αλγορίθμου καθορίζεται από των πολύ μικρό αριθμό των επαναλήψεων(10). Ωστόσο, χρησιμοποιούμε atomic για ορθότητα της τιμής του και για παρατήρηση με βάση το μεγαλύτερο δυνατό overhead.

```
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
8 // square of Euclid distance between two multi-dimensional points
9 inline static double euclid_dist_2(int      numdims, /* no. dimensions */
10                                double * coord1,   /* [numdims] */
11                                double * coord2)   /* [numdims] */
12 {
13     int i;
14     double ans = 0.0;
15
16     for(i=0; i<numdims; i++)
17         ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
18
19     return ans;
20 }
21
22 inline static int find_nearest_cluster(int      numClusters, /* no. clusters */
23                                       int      numCoords, /* no. coordinates */
24                                       double * object,    /* [numCoords] */
25                                       double * clusters) /* [numClusters][numCoords] */
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
45 void kmeans(double * objects,           /* in: [numObjs][numCoords] */
46             int      numCoords,        /* no. coordinates */
47             int      numObjs,          /* no. objects */
48             int      numClusters,      /* no. clusters */
49             double   threshold,        /* minimum fraction of objects that change membership */
50             long    loop_threshold,    /* maximum number of iterations */
51             int     * membership,      /* out: [numObjs] */
```

```

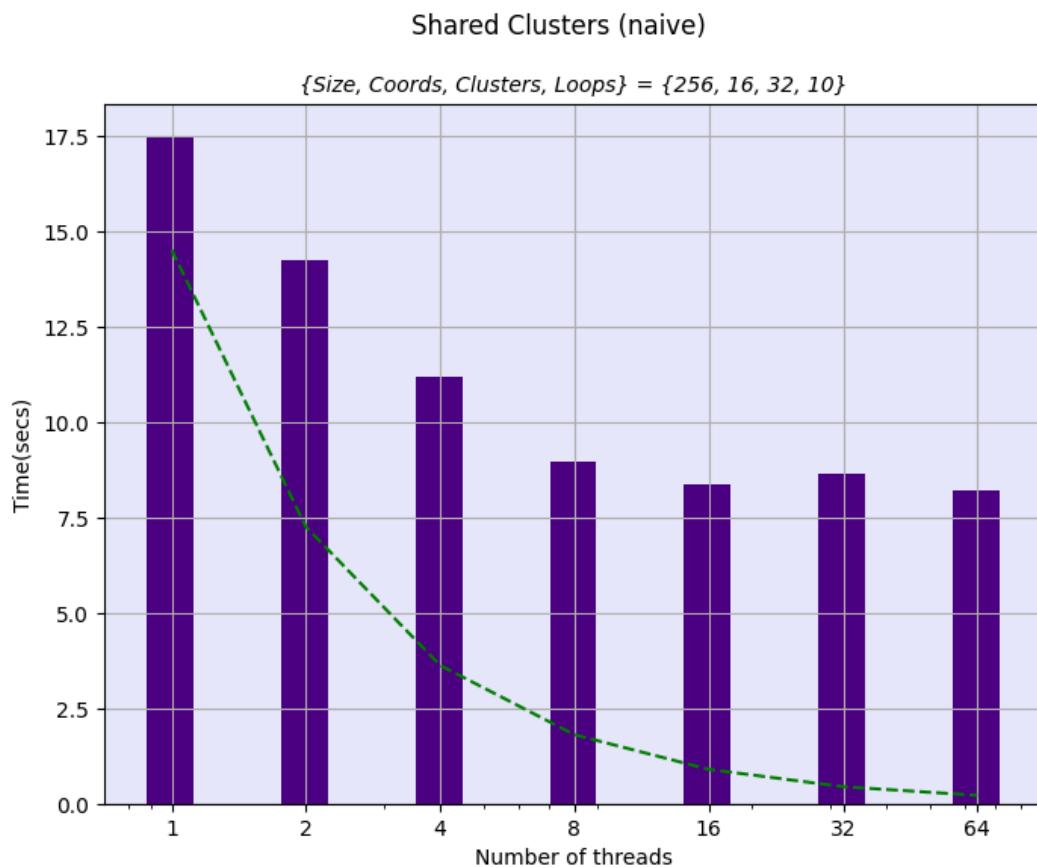
52     double * clusters)          /* out: [numClusters][numCoords] */
53 {
54     int i, j;
55     int index, loop=0;
56     double timing = 0;
57
58     double delta;           // fraction of objects whose clusters change in each loop
59     int * newClusterSize; // [numClusters]: no. objects assigned in each new cluster
60     double * newClusters; // [numClusters][numCoords]
61     int nthreads;         // no. threads
62
63     nthreads = omp_get_max_threads();
64     printf("OpenMP Kmeans - Naive\t(number of threads: %d)\n", nthreads);
65
66     // initialize membership
67     for (i=0; i<numObjs; i++)
68         membership[i] = -1;
69
70     // initialize newClusterSize and newClusters to all 0
71     newClusterSize = (typeof(newClusterSize)) calloc(numClusters, sizeof(*newClusterSize));
72     newClusters = (typeof(newClusters)) calloc(numClusters * numCoords, sizeof(*newClusters));
73
74     timing = wtime();
75
76     do {
77         // before each loop, set cluster data to 0
78         for (i=0; i<numClusters; i++) {
79             for (j=0; j<numCoords; j++)
80                 newClusters[i*numCoords + j] = 0.0;
81             newClusterSize[i] = 0;
82         }
83
84         delta = 0.0;
85
86         /*
87          * TODO: Detect parallelizable region and use appropriate OpenMP pragmas
88         */
89         #pragma omp parallel for private(i, j, index) shared(newClusters, newClusterSize,
90         membership) schedule(static)
91         for (i=0; i<numObjs; i++) {
92             // find the array index of nearest cluster center
93             index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
94
95             // if membership changes, increase delta by 1
96             if (membership[i] != index)
97                 #pragma omp atomic
98                 delta += 1.0;
99
100            // assign the membership to object i
101            membership[i] = index;
102
103            // update new cluster centers : sum of objects located within
104            /*
105             * TODO: protect update on shared "newClusterSize" array
106             */
107            #pragma omp atomic
108            newClusterSize[index]++;
109            for (j=0; j<numCoords; j++)
110                /*
111                 * TODO: protect update on shared "newClusters" array
112                 */
113                #pragma omp atomic
114                newClusters[index*numCoords + j] += objects[i*numCoords + j];
115
116            // average the sum and replace old cluster centers with newClusters
117            // #pragma omp parallel for private(i,j)
118            for (i=0; i<numClusters; i++) {
119                if (newClusterSize[i] > 0) {
120                    for (j=0; j<numCoords; j++)
121                        clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
122                }
123            }
124        }
125

```

```

126 // Get fraction of objects whose membership changed during this loop. This is used as a
127 convergence criterion.
128     delta /= numObjs;
129
130     loop++;
131     printf("\r\tcompleted loop %d", loop);
132     fflush(stdout);
133 } while (delta > threshold && loop < loop_threshold);
134 timing = wtime() - timing;
135 printf("\n          nloops = %3d    (total = %7.4fs)  (per loop = %7.4fs)\n", loop, timing,
136 timing/loop);
137
138 free(newClusters);
139 free(newClusterSize);
140 }
```

Απεικονίζουμε παρακάτω τα αποτελέσματα των δοκιμών στον sandman για τις διάφορες τιμές της environmental variable OMP_NUM_THREADS:



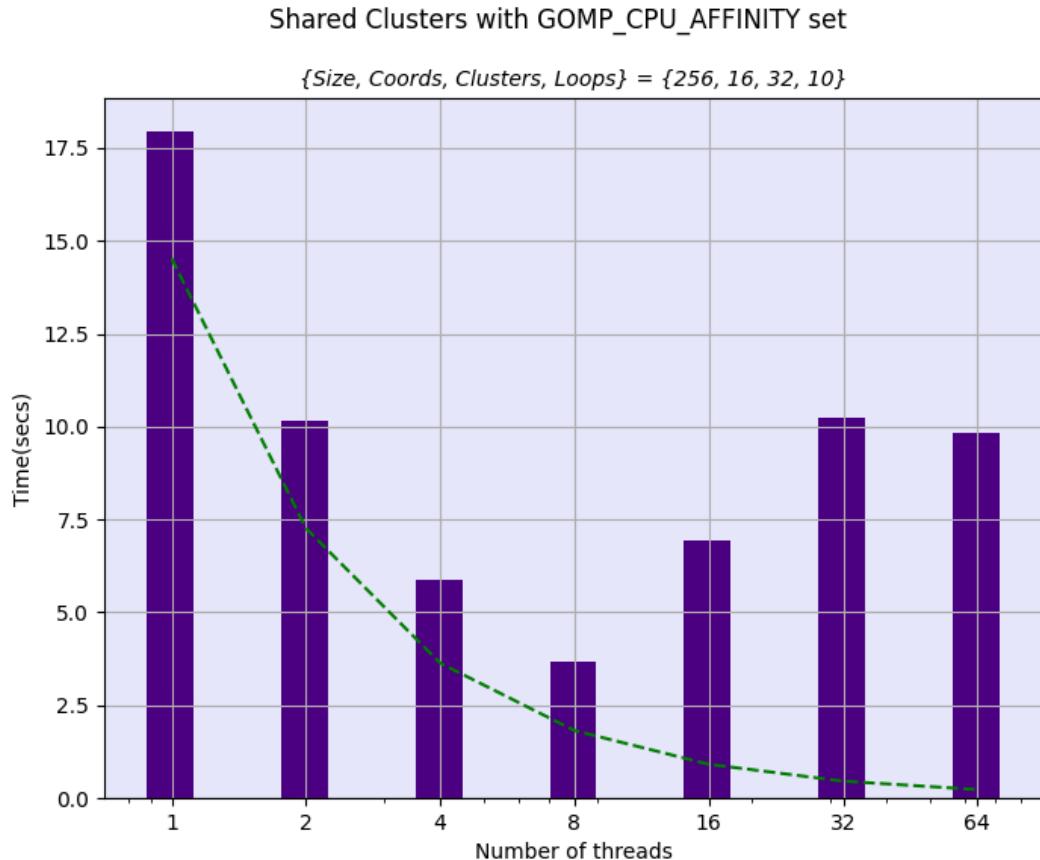
Παρατηρούμε πως ο αλγόριθμος δεν κλιμακώνει καθόλου καλά από 8 και πάνω νήματα εξαιτείας της σειριποίησης των εγγραφών ολοένα και περισσότερων νημάτων που επιβάλλει η omp atomic, και της αυξανόμενης συμφόρησης στο bus κατά την απόκτηση του lock.

Εκμετάλλευση του GOMP_CPU_AFFINITY

Με την χρήση του environmental variable GOMP_CPU_AFFINITY και στατικό shceduling κάνουμε ριν νήματα σε πυρήνες(εφόσον δεν υπάρχει ανάγκη για περίπλοκη δυναμική δρομολόγηση). Έτσι, δεν σπαταλάται καθόλου χρόνος σε flash πυρήνων και αχρείαστη μεταφορά δεδομένων από πυρήνα σε άλλον.

Για την υλοποίηση τροποποίησαμε κατάλληλα το script υποβολής στον sandman και προσθέσαμε την παράμετρο **schedule (static)** στο parallel for.

Αποτελέσματα



Παρατηρούμε σημαντική βελτίωση στην κλιμάκωση μέχρι 8 νήματα όμως μετά σταματάει να κλιμακώνει ο αλγόριθμος λόγω της δομής που έχει ο sandman. Για 16 νήματα και πάνω δεν μπορούμε να τα κάνουμε ριν στο ίδιο cluster οπότε δεν μοιράζονται τα νήματα την ίδια L3 cache και υπάρχει συνεχής μεταφορά δεδομένων των shared πινάκων και bus invalidations λόγω του cache coherence protocol. Ακόμη τα L3 misses κοστίζουν ξεχωριστά για κάθε cluster. Εαν αξιοποιήσουμε το hyperthreading και κάνουμε ριν τα threads 9-16 στους cores 32-40 που πέφτουν μέσα στο cluster 1 μπορούμε να μειώσουμε σημαντικά τον χρόνο για τα 16 νήματα. Από εκεί και πέρα η κλιμάκωση σταματάει. Παραθέτουμε το τελικό script υποβολής ακολούθως:

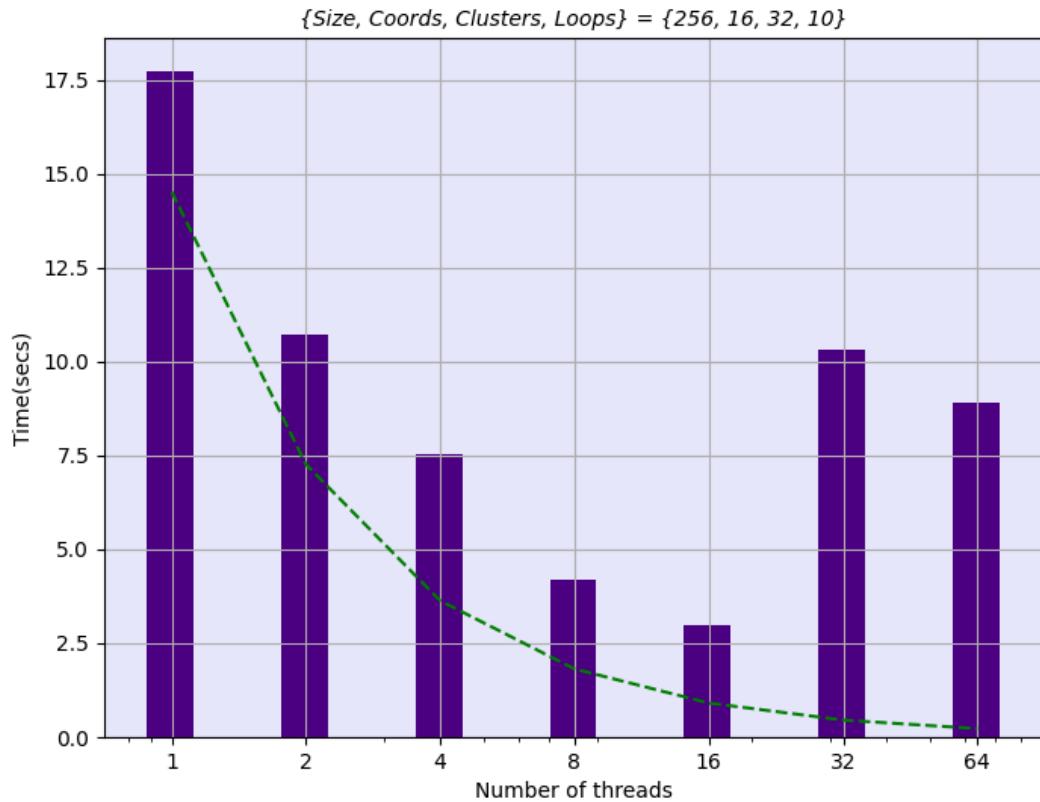
```

#!/bin/bash
## Give the Job a descriptive name
#PBS -N run_kmeans
## Output and error files
#PBS -o gomp_hyper_kmeans.out
#PBS -e gomp_hyper_kmeans.err
## How many machines should we get?
#PBS -l nodes=1:ppn=8
##How long should the job run for?
#PBS -l walltime=00:10:00
## Start
## Run make in the src folder (modify properly)
module load openmp
cd /home/parallel/parlab09/kmeans
Size=256
Coords=16
Clusters=32
Loops=10
for i in 1 2 4 8 16 32 64; do
    export OMP_NUM_THREADS=$i
    if [[ $i -eq 16 ]]; then
        export GOMP_CPU_AFFINITY=$(seq -s, 0 7),$(seq -s, 32 40)"
    else
        export GOMP_CPU_AFFINITY=$(seq -s, 0 $((i - 1)))"
    fi
done
./kmeans_omp_naive -s $Size -n $Coords -c $Clusters -l $Loops
done

```

Αποτελέσματα

Shared Clusters with GOMP_CPU_AFFINITY[0-7][32-40]



2) Copied Clusters & Reduce

Υλοποίηση

Μοιράζουμε σε κάθε νήμα ένα διαφορετικό τμήμα των πινάκων newClusters, newClusterSize οπότε τα δεδομένα γίνονται private, δεν υπάρχουν race conditions αλλά απαιτείται reduction (με πρόσθεση) στο τέλος για το τελικό αποτέλεσμα (η οποία πραγματοποιείται εδώ από 1 νήμα).

```
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
8 // square of Euclid distance between two multi-dimensional points
9 inline static double euclid_dist_2(int      numdims, /* no. dimensions */
10                                double * coord1,   /* [numdims] */
11                                double * coord2)  /* [numdims] */
12{
13     int i;
14     double ans = 0.0;
15
16     for(i=0; i<numdims; i++)
17         ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
18
19     return ans;
20 }
21
22 inline static int find_nearest_cluster(int      numClusters, /* no. clusters */
23                                       int      numCoords,    /* no. coordinates */
24                                       double * object,      /* [numCoords] */
25                                       double * clusters)   /* [numClusters][numCoords] */
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
43     return index;
44 }
45
46 void kmeans(double * objects,           /* in: [numObjs][numCoords] */
47             int      numCoords,        /* no. coordinates */
48             int      numObjs,          /* no. objects */
49             int      numClusters,       /* no. clusters */
50             double   threshold,        /* minimum fraction of objects that change membership */
51             long     loop_threshold,   /* maximum number of iterations */
52             int      * membership,     /* out: [numObjs] */
53             double   * clusters)       /* out: [numClusters][numCoords] */
54{
55     int i, j, k;
56     int index, loop=0;
57     double timing = 0;
58
59     double delta;           // fraction of objects whose clusters change in each loop
60     int * newClusterSize; // [numClusters]: no. objects assigned in each new cluster
61     double * newClusters; // [numClusters][numCoords]
62     int nthreads;           // no. threads
63
64     nthreads = omp_get_max_threads();
```

```

64  printf("OpenMP Kmeans - Reduction\t(number of threads: %d)\n", nthreads);
65
66 // initialize membership
67 for (i=0; i<numObjs; i++)
68     membership[i] = -1;
69
70 // initialize newClusterSize and newClusters to all 0
71 newClusterSize = (typeof(newClusterSize)) malloc(numClusters, sizeof(*newClusterSize));
72 newClusters = (typeof(newClusters)) malloc(numClusters * numCoords, sizeof(*newClusters));
73
74 // Each thread calculates new centers using a private space. After that, thread 0 does an
75 // array reduction on them.
76 int * local_newClusterSize[nthreads]; // [nthreads][numClusters]
77 double * local_newClusters[nthreads]; // [nthreads][numClusters][numCoords]
78
79 /*
80  * Hint for false-sharing
81  * This is noticed when numCoords is low (and neighboring local_newClusters exist close to
82  * each other).
83  * Allocate local cluster data with a "first-touch" policy.
84  */
85 // Initialize local (per-thread) arrays (and later collect result on global arrays)
86 for (k=0; k<nthreads; k++)
87 {
88     local_newClusterSize[k] = (typeof(*local_newClusterSize)) malloc(numClusters,
89     sizeof(**local_newClusterSize));
90     local_newClusters[k] = (typeof(*local_newClusters)) malloc(numClusters * numCoords,
91     sizeof(**local_newClusters));
92 }
93
94 timing = wtime();
95 do {
96     /* before each loop, set cluster data to 0
97     // #pragma omp parallel for private(i,j)
98     for (i=0; i<numClusters; i++) {
99         for (j=0; j<numCoords; j++)
100             newClusters[i*numCoords + j] = 0.0;
101         newClusterSize[i] = 0;
102     }
103
104     delta = 0.0;
105
106     /* TODO: Initiliaze local cluster data to zero (separate for each thread)
107     */
108
109     #pragma omp parallel for private(k, i, j) shared(local_newClusters, local_newClusterSize)
110     schedule(static)
111     for (k=0; k<nthreads; ++k){
112         for (i=0; i<numClusters; i++) {
113             for (j=0; j<numCoords; j++)
114                 local_newClusters[k][i*numCoords + j] = 0.0;
115             local_newClusterSize[k][i] = 0;
116         }
117     }
118
119     int thread_id;
120
121     #pragma omp parallel for private(i, j, thread_id, index) shared(local_newClusters,
122     local_newClusterSize) reduction(+:delta) schedule(static)
123     for (i=0; i<numObjs; i++)
124     {
125         thread_id = omp_get_thread_num();
126
127         // find the array index of nearest cluster center
128         index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
129
130         // if membership changes, increase delta by 1
131         if (membership[i] != index)
132             delta += 1.0;
133
134         // assign the membership to object i
135         membership[i] = index;
136
137     }
138 }
```

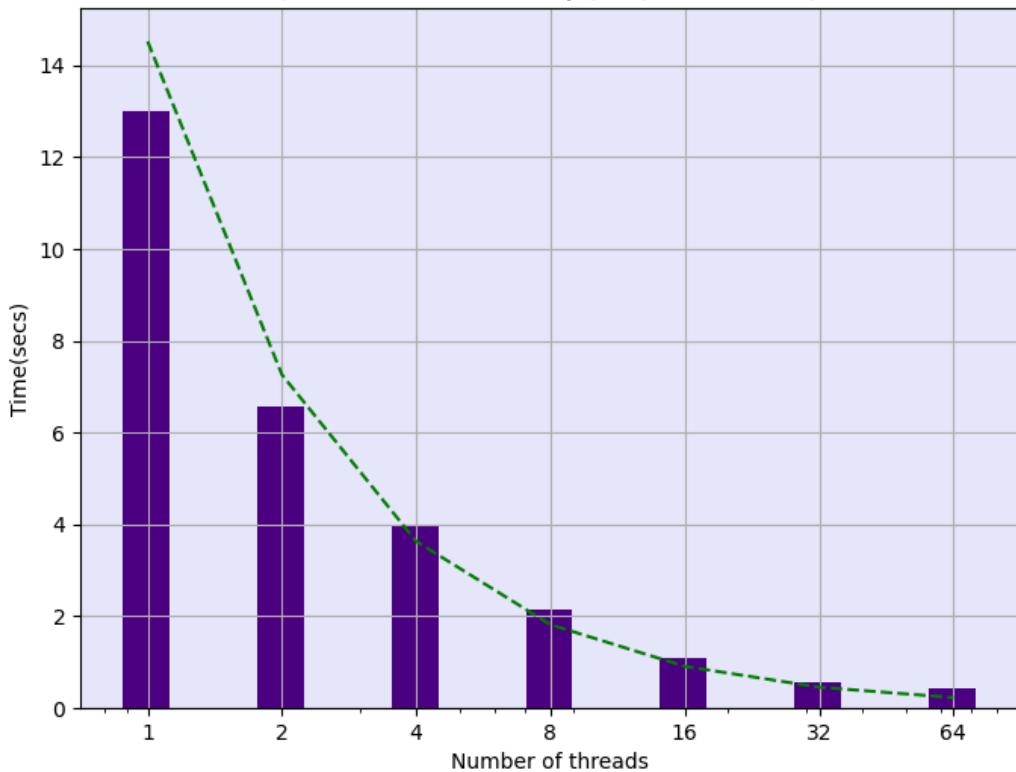
```

131     // update new cluster centers : sum of all objects located within (average will be
132     // performed later)
133     /*
134      * TODO: Collect cluster data in local arrays (local to each thread)
135      * Replace global arrays with local per-thread
136      */
137
138     local_newClusterSize[thread_id][index]++;
139     for (j=0; j<numCoords; j++)
140         local_newClusters[thread_id][index*numCoords + j] += objects[i*numCoords + j];
141 }
142 /*
143  * TODO: Reduction of cluster data from local arrays to shared.
144  * This operation will be performed by one thread
145 */
146
147 for (i=0; i<numClusters; ++i){
148     for (k=0; k<nthreads; ++k){
149         newClusterSize[i] += local_newClusterSize[k][i];
150         for (j=0; j<numCoords; ++j)
151             newClusters[i*numCoords+j] += local_newClusters[k][i*numCoords+j];
152     }
153 }
154
155
156 // average the sum and replace old cluster centers with newClusters
157 // #pragma omp parallel for private(i,j)
158 for (i=0; i<numClusters; i++) {
159     if (newClusterSize[i] > 0) {
160         for (j=0; j<numCoords; j++) {
161             clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
162         }
163     }
164 }
165
166 // Get fraction of objects whose membership changed during this loop. This is used as a
167 // convergence criterion.
168 delta /= numObjs;
169
170 loop++;
171 printf("\r\ntcompleted loop %d", loop);
172 fflush(stdout);
173 } while (delta > threshold && loop < loop_threshold);
174 timing = wtime() - timing;
175 printf("\n          nloops = %3d    (total = %7.4fs)  (per loop = %7.4fs)\n", loop, timing,
176 timing/loop);
177
178 for (k=0; k<nthreads; k++)
179 {
180     free(local_newClusterSize[k]);
181     free(local_newClusters[k]);
182 }
183 free(newClusters);
184 free(newClusterSize);
185 }
```

Αποτελέσματα

Copied Clusters & Reduction

$\{Size, Coords, Clusters, Loops\} = \{256, 16, 32, 10\}$



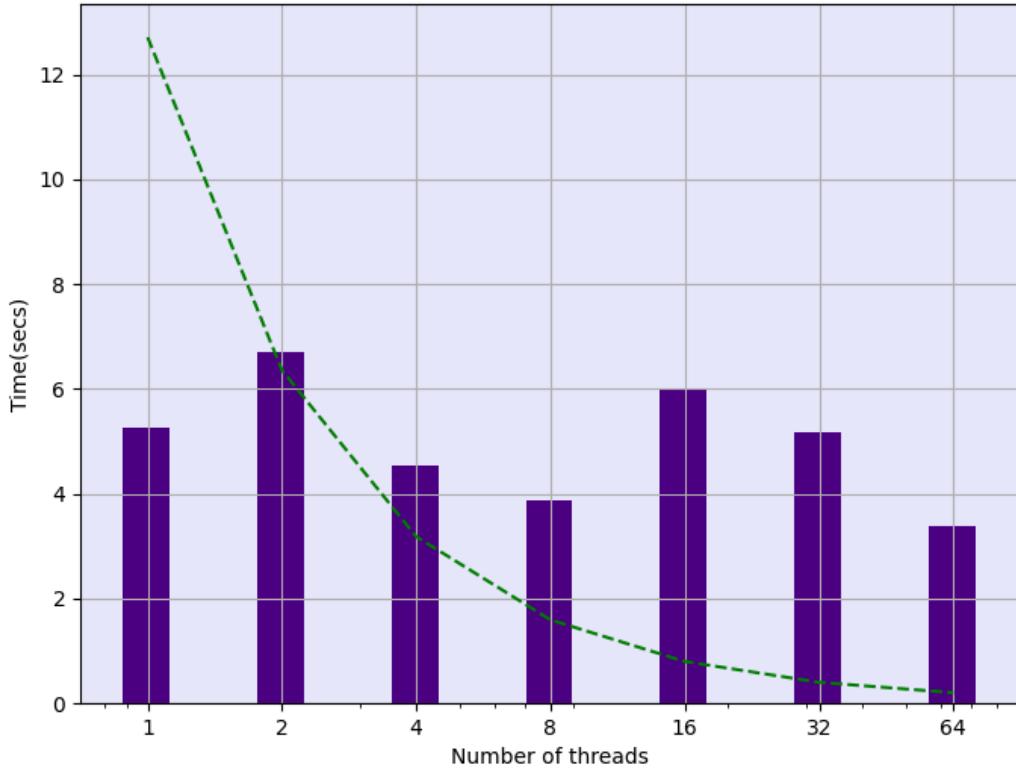
Παρατηρούμε τέλεια κλιμάκωση μέχρι και τα 32 νήματα και αρκετά καλή και στα 64 εφόσον δεν εισάγουμε overheads συγχρονισμού και η σειριακή ενοποίηση (reduction) δεν είναι computational intensive για να καθυστρεί τον αλγόριθμο.

Δοκιμές με μικρότερο dataset

Τα αποτελέσματα δεν είναι ίδια για άλλα μεγέθη πινάκων. Συγκεκριμένα για το επόμενο configuration παρατηρούμε τα εξής:

Copied Clusters & Reduction

$\{Size, Coords, Clusters, Loops\} = \{256, 1, 4, 10\}$



Κυρίαρχο ρόλο για αυτήν την συμπεριφορά αποτελεί το φαινόμενο false sharing, που εμφανίζεται σε μικρά datasets (εδώ κάθε object έχει μόνο 1 συντεταγμένη!) όταν σε ένα cache line καταφέρνουν να χωρέσουν παραπάνω από 1 objects και σε κάθε εγγραφή γίνονται πάρα πολλά περιττά invalidations. Μια λύση είναι το padding όμως έχει memory overhead και δεν προτιμάται.

First-touch Policy

Προς αποφυγή των παραπάνω εκμεταλλευόμαστε την πολιτική των linux κατά το mapping των virtual με physical addresses. Η δέσμευση φυσικής μνήμης πραγματοποιείται κατά την 1η εγγραφή του αντικειμένου (η calloc το εξασφαλίζει γράφοντας 0 ενώ η malloc όχι) οπότε εαν το κάθε νήμα γράψει ξεχωριστά στο κομμάτι του πίνακα που του αντιστοιχεί (ουσιαστικά παραλληλοποιώντας την αντιγραφή των shared πινάκων) θα απεικονιστεί στην μνήμη του αυτό και μόνο.

Υλοποίηση

```
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
8 // square of Euclid distance between two multi-dimensional points
9 inline static double euclid_dist_2(int numdims, /* no. dimensions */
10                                 double * coord1, /* [numdims] */
11                                 double * coord2) /* [numdims] */
12 {
13     int i;
14     double ans = 0.0;
15 }
```

```

16     for(i=0; i<numdims; i++)
17         ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
18
19     return ans;
20 }
21
22 inline static int find_nearest_cluster(int      numClusters, /* no. clusters */
23                                         int      numCoords,   /* no. coordinates */
24                                         double * object,    /* [numCoords] */
25                                         double * clusters) /* [numClusters][numCoords] */
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
45 void kmeans(double * objects,           /* in: [numObjs][numCoords] */
46             int      numCoords,        /* no. coordinates */
47             int      numObjs,          /* no. objects */
48             int      numClusters,       /* no. clusters */
49             double   threshold,        /* minimum fraction of objects that change membership */
50             long    loop_threshold,    /* maximum number of iterations */
51             int    * membership,       /* out: [numObjs] */
52             double  * clusters)        /* out: [numClusters][numCoords] */
53 {
54     int i, j, k;
55     int index, loop=0;
56     double timing = 0;
57
58     double delta;           // fraction of objects whose clusters change in each loop
59     int * newClusterSize; // [numClusters]: no. objects assigned in each new cluster
60     double * newClusters; // [numClusters][numCoords]
61     int nthreads;          // no. threads
62
63     nthreads = omp_get_max_threads();
64     printf("OpenMP Kmeans - Reduction\t(number of threads: %d)\n", nthreads);
65
66     // initialize membership
67     for (i=0; i<numObjs; i++)
68         membership[i] = -1;
69
70     // initialize newClusterSize and newClusters to all 0
71     newClusterSize = (typeof(newClusterSize)) calloc(numClusters, sizeof(*newClusterSize));
72     newClusters = (typeof(newClusters))  calloc(numClusters * numCoords, sizeof(*newClusters));
73
74     // Each thread calculates new centers using a private space. After that, thread 0 does an
75     // array reduction on them.
76     int * local_newClusterSize[nthreads]; // [nthreads][numClusters]
77     double * local_newClusters[nthreads]; // [nthreads][numClusters][numCoords]
78
79     /*
80      * Hint for false-sharing
81      * This is noticed when numCoords is low (and neighboring local_newClusters exist close to
82      * each other).
83      * Allocate local cluster data with a "first-touch" policy.
84      */
85
86     timing = wtime();
87     do {
88         // before each loop, set cluster data to 0
89         for (i=0; i<numClusters; i++) {
90             for (j=0; j<numCoords; j++)

```

```

89         newClusters[i*numCoords + j] = 0.0;
90         newClusterSize[i] = 0;
91     }
92
93     delta = 0.0;
94
95     /*
96      * TODO: Initiliaze local cluster data to zero (separate for each thread)
97      */
98
99     #pragma omp parallel for private(k,i,j) schedule(static)
100    for (k=0; k<nthreads; ++k){
101        local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters,
102        sizeof(**local_newClusterSize));
103        local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords,
104        sizeof(**local_newClusters));
105
106        for (i=0; i<numClusters; i++) {
107            for (j=0; j<numCoords; j++)
108                local_newClusters[k][i*numCoords + j] = 0.0;
109                local_newClusterSize[k][i] = 0;
110        }
111    int thread_id;
112
113    #pragma omp parallel for private(i, j, thread_id, index) shared(local_newClusters,
114    local_newClusterSize) reduction(+:delta) schedule(static)
115    for (i=0; i<numObjs; i++)
116    {
117        thread_id = omp_get_thread_num();
118
119        // find the array index of nearest cluster center
120        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
121
122        // if membership changes, increase delta by 1
123        if (membership[i] != index)
124            delta += 1.0;
125
126        // assign the membership to object i
127        membership[i] = index;
128
129        // update new cluster centers : sum of all objects located within (average will be
130        // performed later)
131
132        local_newClusterSize[thread_id][index]++;
133        for (j=0; j<numCoords; j++)
134            local_newClusters[thread_id][index*numCoords + j] += objects[i*numCoords + j];
135    }
136
137    for (i=0; i<numClusters; ++i){
138        for (k=0; k<nthreads; ++k){
139            newClusterSize[i] += local_newClusterSize[k][i];
140            for (j=0; j<numCoords; ++j)
141                newClusters[i*numCoords+j] += local_newClusters[k][i*numCoords+j];
142        }
143
144        for (i=0; i<numClusters; i++) {
145            if (newClusterSize[i] > 0) {
146                for (j=0; j<numCoords; j++) {
147                    clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
148                }
149            }
150        }
151        delta /= numObjs;
152
153        loop++;
154        printf("\r\tcompleted loop %d", loop);
155        fflush(stdout);
156    } while (delta > threshold && loop < loop_threshold);
157    timing = wtime() - timing;

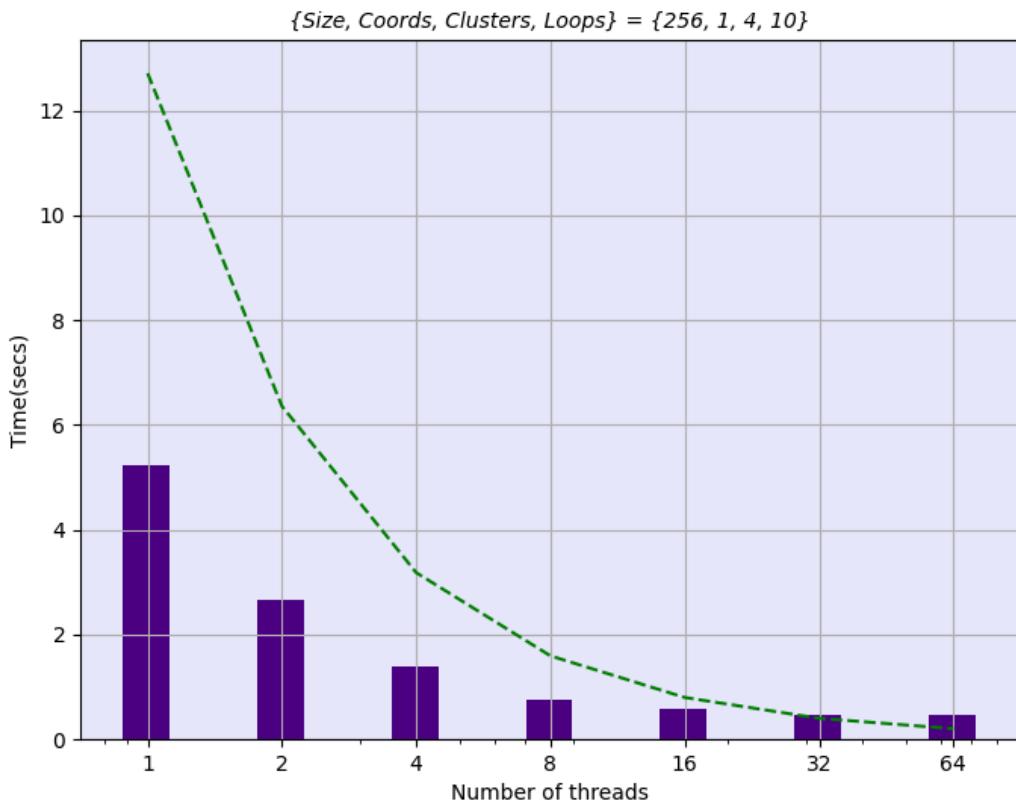
```

```

158     printf("\n          nloops = %3d    (total = %7.4fs)  (per loop = %7.4fs)\n", loop, timing,
159     timing/loop);
160
161     for (k=0; k<nthreads; k++)
162     {
163         free(local_newClusterSize[k]);
164         free(local_newClusters[k]);
165     }
166     free(newClusters);
167     free(newClusterSize);
168 }
```

Αποτελέσματα

Copied Clusters & Reduction with First-Touch Policy



Υπάρχει σαφής βελτίωση και καλή κλιμάκωση μέχρι τα 32 νήματα ακόμα και σε σχέση με την ιδανική εκτέλεση του σειριακού αλγορίθμου. Ο καλύτερος χρόνος σε αυτό το ερώτημα είναι 0.4605s στα 32 νήματα!

Numa-aware initialization

Με βάση όσα αναφέρθηκαν για το pinning σε cores και την πολιτική first-touch η αρχικοποίηση των shared πινάκων μπορεί να γίνει και αυτή ατομικά από κάθε νήμα σε ένα private τμήμα αυτού. Για την υλοποίηση προσθέτουμε το `omp parallel for directive` με στατική δρομολόγηση. Αυτή είναι απαραίτητη ώστε τα νήματα που θα βάλουν τους τυχαίους αριθμούς στα objects να είναι τα ίδια νήματα με αυτά που θα τα επεξεργαστούν στην `main.c` με σκοπό να είναι ήδη στις caches και να μην χρειάζεται να τα μεταφέρουν από την κύρια μνήμη ή από άλλα νήματα.

Υλοποίηση

Τροποποιούμε το `file_io.c` που δίνεται :

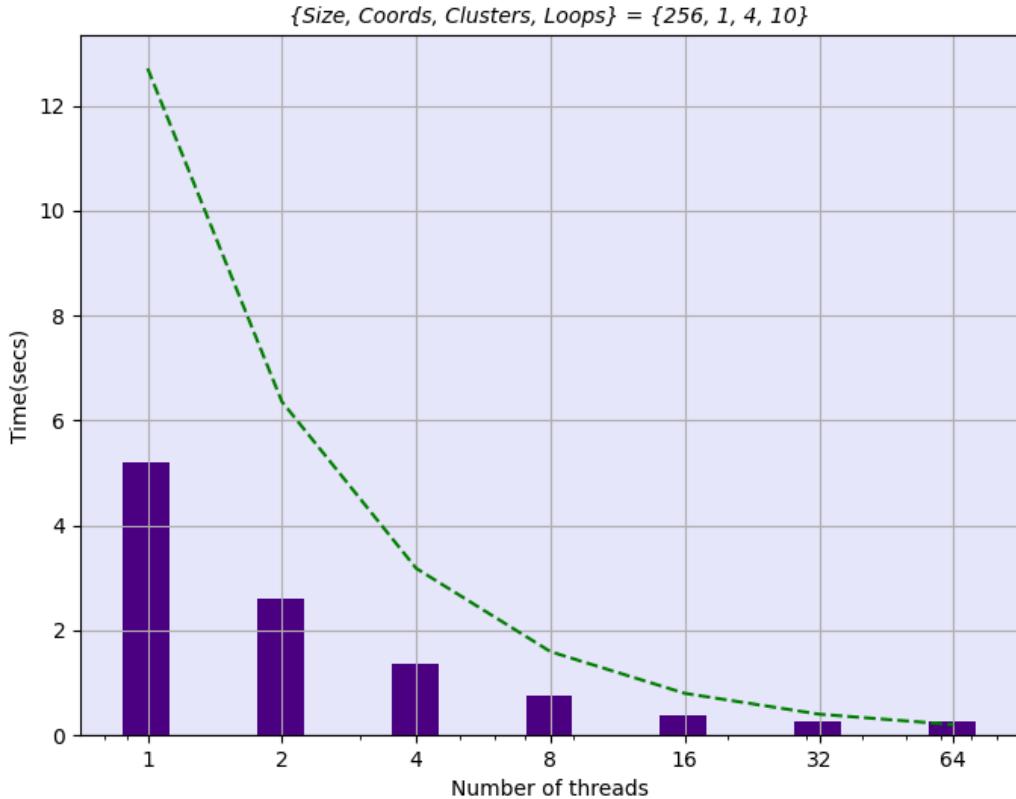
file_io.c

```
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 them.
26      *        reminder : First-touch data placement policy
27     */
28     int nthreads = omp_get_max_threads();
29     int chunk = numObjs / nthreads;
30     int thread_id, start_offs, end_offs;
31
32     #pragma omp parallel private(i, j, thread_id, start_offs, end_offs) shared(nthreads, chunk,
33     objects, numObjs, numCoords, val_range)
34     {
35         //set the binding to cores manually
36
37         thread_id = omp_get_thread_num();
38         start_offs = thread_id * chunk;
39         end_offs = (thread_id == nthreads-1) ? numObjs : start_offs + chunk;
40
41         for (i=start_offs; i<end_offs; i++)
42         {
43             unsigned int seed = i;
44             for (j=0; j <numCoords; j++)
45             {
46                 objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) * val_range;
47                 if (_debug && i == 0)
48                     printf("object[i=%ld][j=%ld]=%f\n", i, j, objects[i*numCoords + j]);
49             }
50         }
51     }
52     return objects;
53 }
```

)

Αποτελέσματα

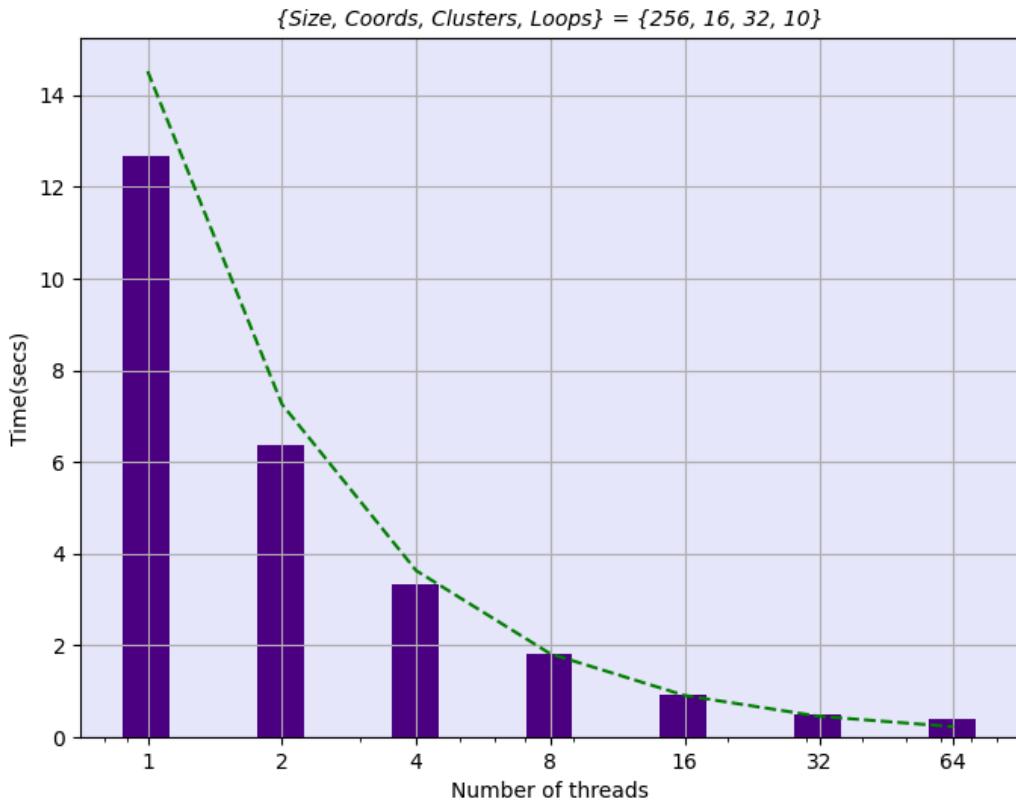
Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization



Παρατηρούμε καλύτερη κλιμάκωση μέχρι τα 32 νήματα με χρόνο 0.2667s! Το κυρίαρχο bottleneck σε αυτήν την περίπτωση είναι το overhead της δημιουργίας των νημάτων.

Τέλος με όλες τις προηγούμενες αλλαγές δοκιμάζουμε ξανά το μεγάλο dataset που είχαμε στην αρχή:

Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization



Παρατηρούμε πως υπάρχει τέλεια κλιμάκωση του αλγορίθμου. Οπότε bottleneck θα μπορούσε να θεωρηθεί το computive intensity για κάθε object.

FLOYD WARSHALL

1) Recursive

Υλοποίηση

Δημιουργούμε ένα παράλληλο section κατά την πρώτη κλήση αφού έχουμε ενεργοποιήσει την επιλογή για nested tasks μέσω της `omp_set_nested(1)`. (**Μπορούμε να το θέσουμε και ως environmental variable (OMP_NESTED=TRUE, OMP_MAX_ACTIVE_LEVELS=64)**) Για την διατήρηση των εξαρτήσεων κατά τον υπολογισμό των blocks (A11) -> (A12 A21) -> A22 και αντιστρόφως τοποθετούμε κατάλληλα barriers έμμεσα με τα taskwait directives.

```
fw_sr.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 "util.h"
13 #include <omp.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;
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     A = (int **) malloc(N*sizeof(int *));
39     for(i=0; i<N; i++) A[i] = (int *) malloc(N*sizeof(int));
40
41     graph_init_random(A,-1,N,128*N);
42     //enable nested task generation
43     omp_set_nested(1);
44     // default is equal to 1
45     omp_set_max_active_levels(64);
46
47     gettimeofday(&t1,0);
48
49     #pragma omp parallel
50     {
51         #pragma omp single
52         {
53             FW_SR(A,0,0, A,0,0,A,0,0,N,B);
54         }
55     }
56     gettimeofday(&t2,0);
```

```

57     time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)/1000000;
58     printf("FW_SR,%d,%d,.4f\n", N, B, time);
59
60     /*
61      for(i=0; i<N; i++)
62        for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
63     */
64
65     return 0;
66 }
67
68
69 inline int min(int a, int b)
70 {
71     if(a<=b) return a;
72     else return b;
73 }
74
75 void FW_SR (int **A, int arow, int acol,
76             int **B, int brow, int bcol,
77             int **C, int crow, int ccol,
78             int myN, int bsize)
79 {
80     int k,i,j;
81
82     /*
83      * The base case (when recursion stops) is not allowed to be edited!
84      * What you can do is try different block sizes.
85     */
86
87     if(myN<=bsize)
88         for(k=0; k<myN; k++)
89             for(i=0; i<myN; i++)
90                 for(j=0; j<myN; j++)
91                     A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
92     else {
93
94         FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize); // A00
95         #pragma omp task
96         FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize); //A01
97         #pragma omp task
98         FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize); //A10
99         #pragma omp taskwait
100        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize); //
101        A11
102        FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2,
103              bsize); //A11
104        #pragma omp task
105        FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); //A10
106        #pragma omp task
107        FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize); //A01
108    }
109    // printf("Nested parallelism enabled: %d\n", omp_get_nested());
110 }

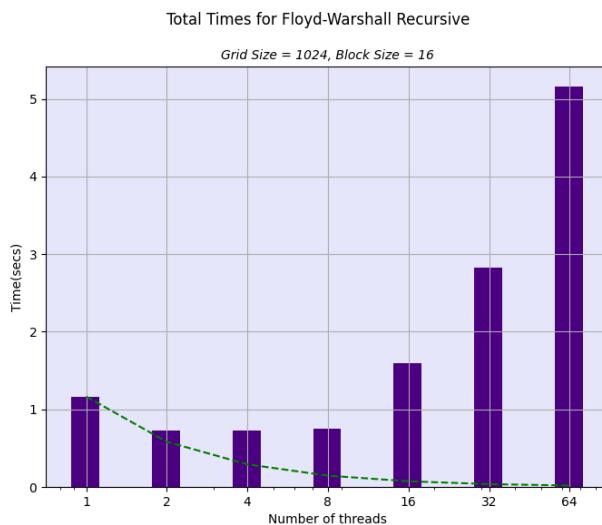
```

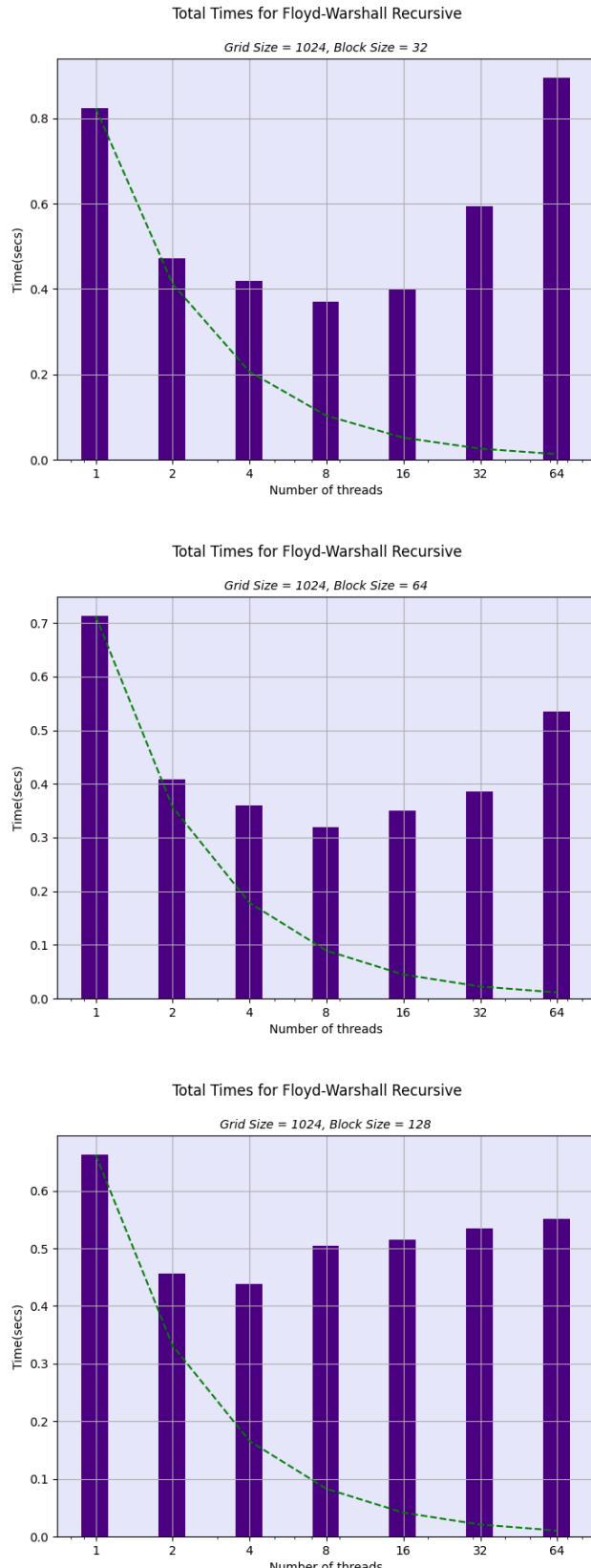
Πειραματιστήκαμε σχετικά με την βέλτιστη τιμή του BSIZE τρέχοντας τις προσομοιώσεις που ακολουθούν. Διαισθητικά η optimal τιμή οφείλει να εκμεταλλεύεται πλήρως το cache size και δεδομένου ότι έχουμε τετράγωνο grid για 1 recursive call που δημιουργεί 4 sub-blocks μεγέθους B θα είναι $B_{opt} = \text{sqrt}(\text{cache size})$. Για τα πειράματα χρησιμοποιήσαμε το ακόλουθο script:

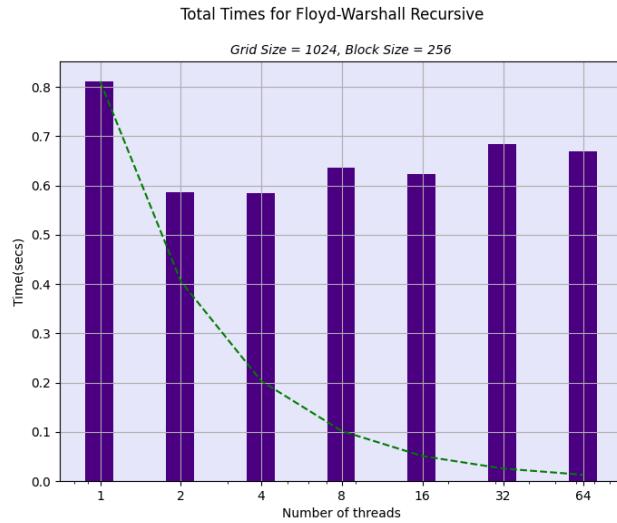
```
#!/bin/bash
## Give the Job a descriptive name #PBS -N run_fw
## Output and error files #PBS -o run_fw_recursive.out #PBS -e run_fw_recursive.err
## How many machines should we get? #PBS -l nodes=1:ppn=8
## How long should the job run for? #PBS -l walltime=00:10:00
## Start
## Run make in the src folder (modify properly)
module load openmp/1.8.3
cd /home/parallel/parlab09/a2/Fw
./fw $SIZE
export OMP_NESTED=TRUE
export OMP_MAX_ACTIVE_LEVELS=64
for SIZE in 1024 2048 4096; do
    for BSIZE in 16 32 64 128 256; do
        echo -e "\nBSIZE=${BSIZE}\n"
        for n in 1 2 4 8 16 32 64; do
            export OMP_NUM_THREADS=${n}
            echo -e "\nNumber of threads: ${n}"
            ./fw_sr $SIZE $BSIZE
        done
    done
done
```

Αποτελέσματα

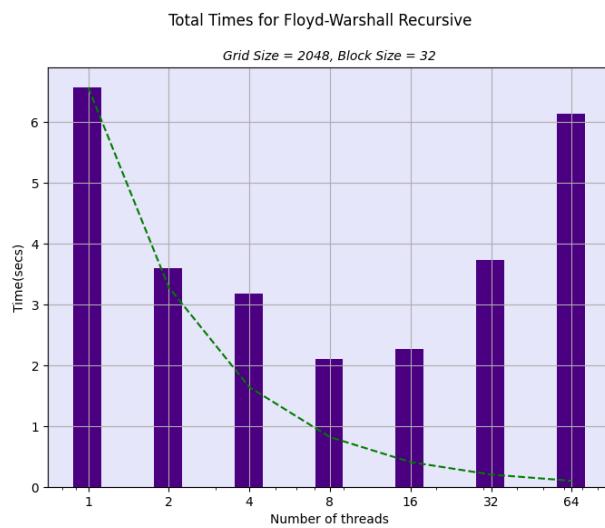
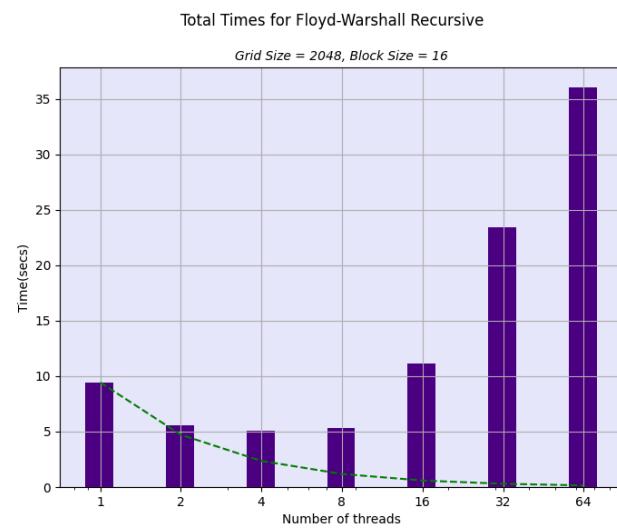
$$\{N = 1024\}$$





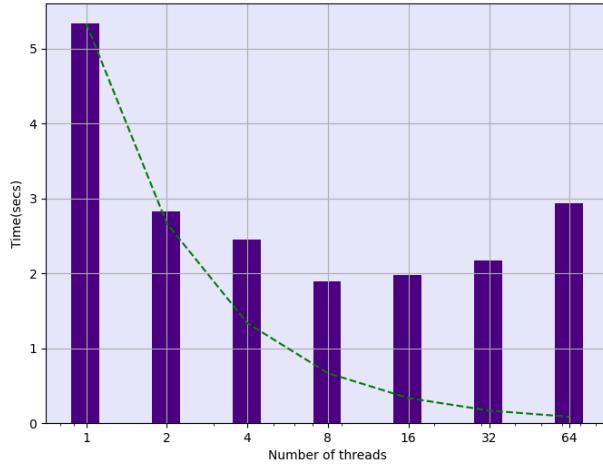


{N = 2048}



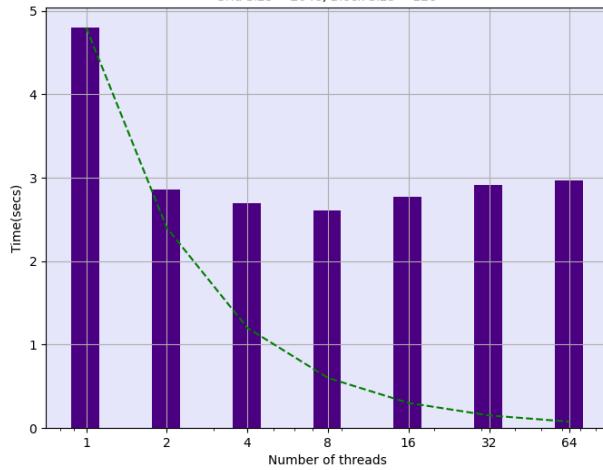
Total Times for Floyd-Warshall Recursive

Grid Size = 2048, Block Size = 64



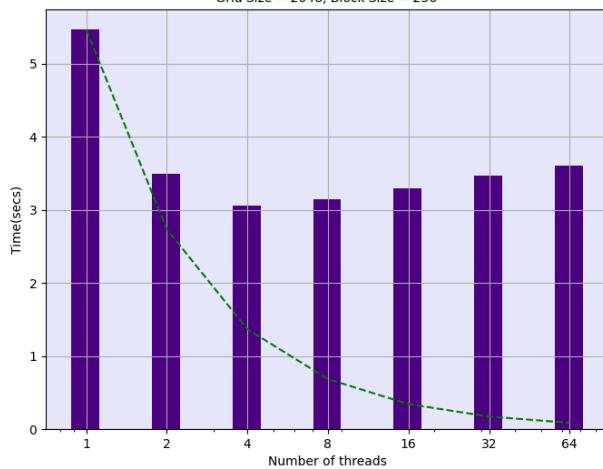
Total Times for Floyd-Warshall Recursive

Grid Size = 2048, Block Size = 128

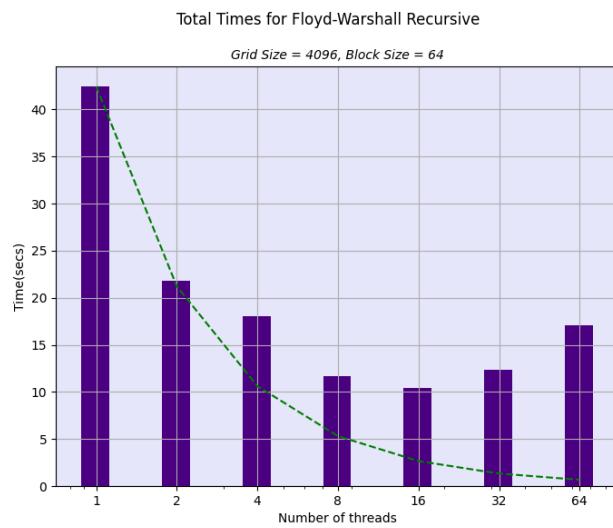
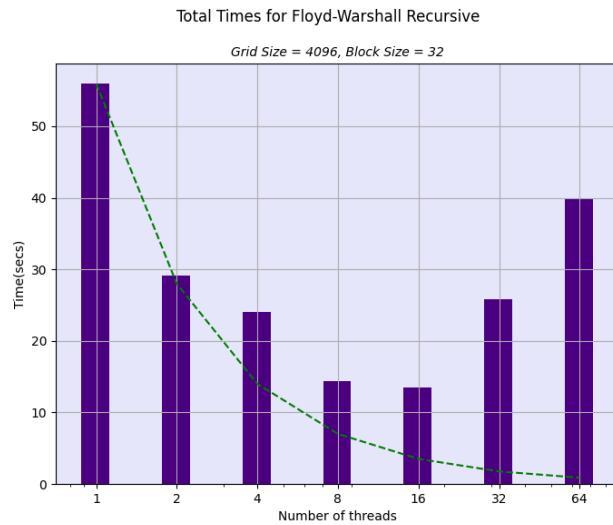
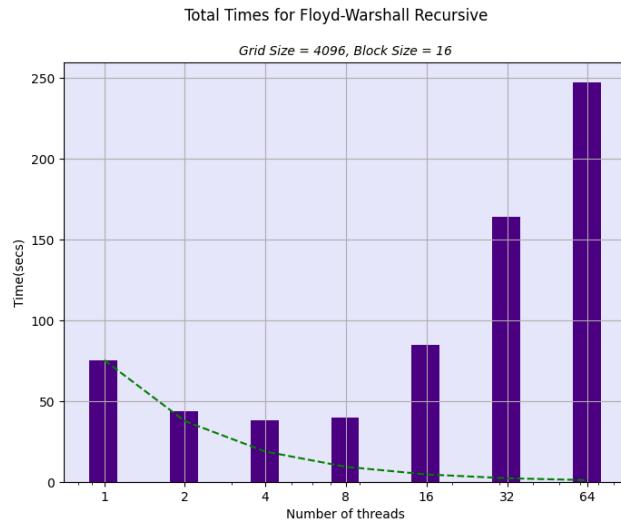


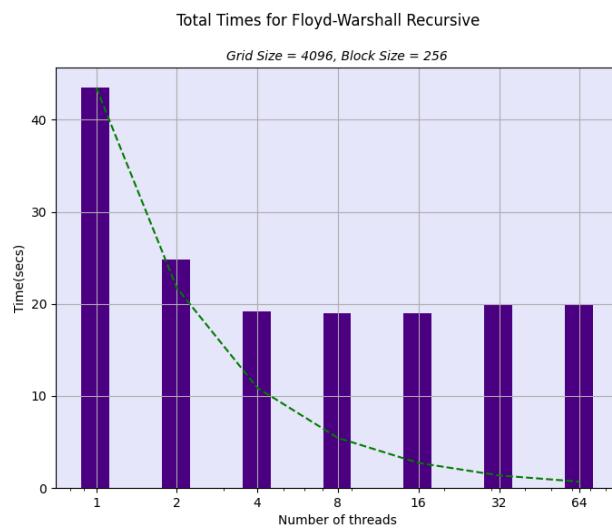
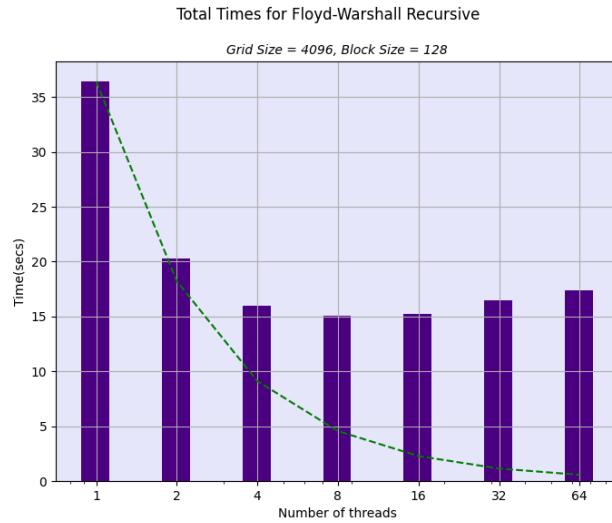
Total Times for Floyd-Warshall Recursive

Grid Size = 2048, Block Size = 256



$$\{ N = 4096 \}$$





Καταλήξαμε πως η ιδανική τιμή είναι $B=64$ και ο καλύτερος χρόνος που πετύχαμε χρησιμοποιώντας αυτήν για 4096 μέγεθος πίνακα ήταν 10.4486 με 16 threads. Από το σημείο αυτό και έπειτα ο αλγόριθμος δεν κλιμακώνει και φανερώνει την αδυναμία του χάρη στην αναδρομή.

2) TILED

Υλοποίηση

Φτιάχνουμε 1 παράλληλο section με κατάλληλα barriers ώστε να υπολογίζεται πρώτα (single) το κοστό στοιχείο στην διαγώνιο, έπειτα όσα βρίσκονται κατά μήκος του “σταυρού” που σχηματίζεται εκατέρωθεν αυτού, και τέλος τα blocks στοιχείων που απομένουν. Καθένα από τα στάδια 2 και 3 έχει 4 for loops που μπορούν να παραλληλοποιηθούν με parallel for και επειδή είναι ανεξάρτητα μεταξύ τους με παράμετρο nowait. Το collapse(2) πραγματοποιεί flattening για καλύτερη λειτουργία του parallel for για nested loops. Με χρήση μόνο των παραπάνω επιτυγχάνουμε χρόνο εκτέλεσης 2.2 secs.

Για περαιτέρω βελτίωση επιχειρήσαμε να χρησιμοποιήσουμε SIMD εντολές αρχικά μέσω του OpenMP με το αντίστοιχο directive και στην συνέχεια γράφοντας χειροκίνητα τις intrinsics εντολές για AVX μοντέλο που υποστηρίζει 4-size vector operations καθώς διαπιστώσαμε ότι vector operations μεγαλύτερου μεγέθους (π.χ με 8 στοιχεία AVX2) δεν υποστηρίζεται στο εν λόγω μηχάνημα και λαμβάνουμε σφάλμα Illegal hardware instruction. Στην πρώτη εκδοχή λάβαμε συνολικό χρόνο εκτέλεσης 1.7secs.

Η χρήση των intrinsics απευθείας μας δίνει την δυνατότητα να εκμεταλλευτούμε πλήρως και την αρχιτεκτονική της κρυφής μνήμης μέσω loop unrolling. Συγκεκριμένα, αναγνωρίσαμε ότι το size του cacheline είναι 64bytes, συνεπώς χωράνε 16 integers, ή 4 vectors 4άδων σε όρους AVX. Άρα επιτυγχάνουμε μέγιστο locality exploitation κάνοντας unroll με παράγοντα 4 και αυξάνοντας το j κατά 16 σε κάθε iteration. Ακόμη, παρατηρούμε ότι τα στοιχεία A[i][k] είναι ανεξάρτητα του j και η φόρτωση αυτών των vectors μπορεί να γίνει στο εξωτερικό loop. Ο καλύτερος χρόνος εκτέλεσης που επιτύχαμε με αυτήν την εκδοχή είναι **1.39 secs!**

```
fw_smd.c

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/time.h>
4 #include <immintrin.h> // For SSE2 intrinsics
5 #include <omp.h>
6
7 inline void FW(int **A, int K, int I, int J, int B);
8
9 int main(int argc, char **argv)
10 {
11     int **A;
12     int i, j, k;
13     struct timeval t1, t2;
14     double time;
15     int B = 64;
16     int N = 1024;
17
18     if (argc != 3) {
19         fprintf(stdout, "Usage %s N B\n", argv[0]);
20         exit(0);
21     }
22
23     N = atoi(argv[1]);
24     B = atoi(argv[2]);
25
26     // Allocate memory for A with 32-byte alignment
27     posix_memalign((void**)&A, 32, N * sizeof(int*));
28     for (i = 0; i < N; ++i) {
29         posix_memalign((void**)&A[i], 32, N * sizeof(int));
30     }
31
32     // Initialize the graph with random values
33     graph_init_random(A, -1, N, 128 * N);
34
35     // Start timer
36     gettimeofday(&t1, 0);
37 }
```

```

38 // Main loop of the Floyd-Warshall algorithm with tiling
39 for (k = 0; k < N; k += B) {
40     #pragma omp parallel
41     {
42         #pragma omp single
43         {
44             FW(A, k, k, k, B);
45         }
46         #pragma omp for nowait
47         for (i = 0; i < k; i += B)
48             FW(A, k, i, k, B);
49
50         #pragma omp for nowait
51         for (i = k + B; i < N; i += B)
52             FW(A, k, i, k, B);
53
54         #pragma omp for nowait
55         for (j = 0; j < k; j += B)
56             FW(A, k, k, j, B);
57
58         #pragma omp for nowait
59         for (j = k + B; j < N; j += B)
60             FW(A, k, k, j, B);
61
62         #pragma omp barrier
63
64         #pragma omp for collapse(2) nowait
65         for (i = 0; i < k; i += B)
66             for (j = 0; j < k; j += B)
67                 FW(A, k, i, j, B);
68
69         #pragma omp for collapse(2) nowait
70         for (i = 0; i < k; i += B)
71             for (j = k + B; j < N; j += B)
72                 FW(A, k, i, j, B);
73
74         #pragma omp for collapse(2) nowait
75         for (i = k + B; i < N; i += B)
76             for (j = 0; j < k; j += B)
77                 FW(A, k, i, j, B);
78
79         #pragma omp for collapse(2) nowait
80         for (i = k + B; i < N; i += B)
81             for (j = k + B; j < N; j += B)
82                 FW(A, k, i, j, B);
83
84         #pragma omp barrier
85     }
86 }
87
88 // Stop timer and calculate execution time
89 gettimeofday(&t2, 0);
90 time = (double)((t2.tv_sec - t1.tv_sec) * 1000000 + t2.tv_usec - t1.tv_usec) / 1000000;
91 fprintf(stdout, "FW_TILED,%d,%d,%f\n", N, B, time);
92
93 // Free the memory
94 for (i = 0; i < N; i++) {
95     _mm_free(A[i]); // Free each row
96 }
97 _mm_free(A); // Free the pointer array
98
99 return 0;
100}
101
102 inline void FW(int **A, int K, int I, int J, int B)
103 {
104     int i, j, k;
105
106     // Iterate over a block of tiles (3D loop over the block)
107     for (k = K; k < K + B; k++) {
108         for (i = I; i < I + B; i++) {
109             // _mm_prefetch((const char*)&A[i][j], _MM_HINT_T0);
110             // _mm_prefetch((const char*)&A[k][j], _MM_HINT_T0);
111             // _mm_prefetch((const char*)&A[i][j + 16], _MM_HINT_T0);
112             // _mm_prefetch((const char*)&A[k][j + 16], _MM_HINT_T0);

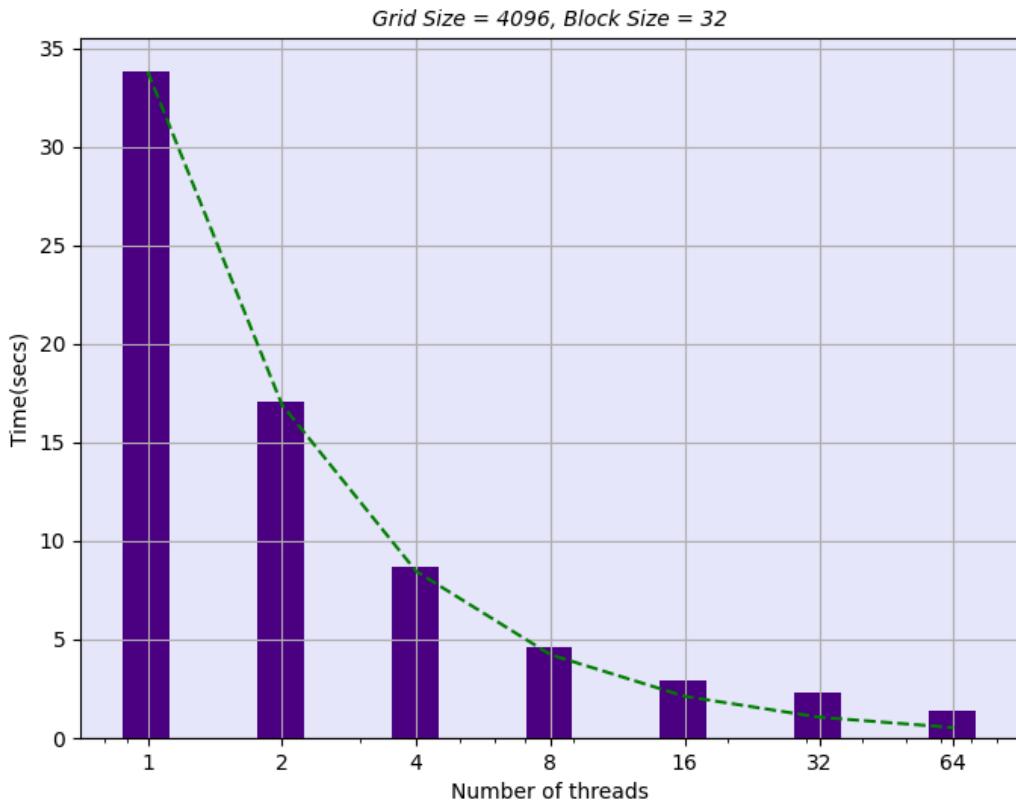
```

```

114     __m128i A_i_k = _mm_load_si128((__m128i*)&A[i][k]);
115
116     for (j = J; j < J + B; j+=16){
117
118         __m128i A_i_j = _mm_load_si128((__m128i*)&A[i][j]);
119         __m128i A_k_j = _mm_load_si128((__m128i*)&A[k][j]);
120
121         __m128i A_plus = _mm_add_epi32(A_i_k, A_k_j);
122         __m128i result = _mm_min_epi32(A_i_j, A_plus);
123
124         _mm_store_si128((__m128i*)&A[i][j], result);
125
126         // next chunk
127         A_i_j = _mm_load_si128((__m128i*)&A[i][j+4]);
128         A_k_j = _mm_load_si128((__m128i*)&A[k][j+4]);
129
130         A_plus = _mm_add_epi32(A_i_k, A_k_j);
131         result = _mm_min_epi32(A_i_j, A_plus);
132
133         _mm_store_si128((__m128i*)&A[i][j+4], result);
134
135         //next chunk
136         A_i_j = _mm_load_si128((__m128i*)&A[i][j+8]);
137         A_k_j = _mm_load_si128((__m128i*)&A[k][j+8]);
138
139         A_plus = _mm_add_epi32(A_i_k, A_k_j);
140         result = _mm_min_epi32(A_i_j, A_plus);
141
142         _mm_store_si128((__m128i*)&A[i][j+8], result);
143
144         //next chunk
145         A_i_j = _mm_load_si128((__m128i*)&A[i][j+12]);
146         A_k_j = _mm_load_si128((__m128i*)&A[k][j+12]);
147
148         A_plus = _mm_add_epi32(A_i_k, A_k_j);
149         result = _mm_min_epi32(A_i_j, A_plus);
150
151         _mm_store_si128((__m128i*)&A[i][j+12], result);
152
153         // if(j == J)
154         //     _mm_prefetch((const char*)&A[i+1][k], _MM_HINT_T0);
155     }
156 }
157 }
158 // if (k + 1 < K + B) {
159 //     _mm_prefetch((const char*)&A[i][k + 1], _MM_HINT_T0);
160 // }
161 }
162 }
```

Αποτελέσματα

Total Times for Floyd-Warshall Tiled



Παραθέτουμε αναλυτικά και τους βέλτιστους χρόνους:

Number of threads: 1

FW_TILED,4096,32,33.8411

Number of threads: 2

FW_TILED,4096,32,17.0405

Number of threads: 4

FW_TILED,4096,32,8.7231

Number of threads: 8

FW_TILED,4096,32,4.5795

Number of threads: 16

FW_TILED,4096,32,2.9022

Number of threads: 32

FW_TILED,4096,32,2.3016

Number of threads: 64

FW_TILED,4096,32,1.3925

Παράρτημα

Για την δημιουργία των γραφικών παραστάσεων χρημιοποιηθηκαν οι εξής κώδικες σε Python :

results.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 #import pandas
4 import re
5
6 regex_total = r"\(total\s*=\s*([\d.]+)\s\)"
7 regex_loop = r"\(per\s+loop\s*=\s*([\d.]+)\s\)"
8
9 total_times = []
10 loop_times = []
11
12 with open("results.txt", 'r') as file:
13     for line in file:
14         match_total = re.search(regex_total, line)
15         match_loop = re.search(regex_loop, line)
16         if (match_total is not None and match_loop is not None):
17             total_times.append(float(match_total.group(1)))
18             loop_times.append(float(match_loop.group(1)))
19
20
21 seq_totals = total_times[0:2]
22 seq_loop = loop_times[0:2]
23
24 total_times = total_times[2::]
25 loop_times = loop_times[2::]
26
27 print("Sequential Times: ", seq_totals, seq_loop)
28 print("Total Times:", total_times)
29 print("Loop Times:", loop_times)
30
31 threads = [1,2,4,8,16,32,64]
32 nthreads = len(threads)
33 titles = ["Shared Clusters (naive)",
34           "Shared Clusters with GOMP_CPU_AFFINITY set",
35           "Copied Clusters & Reduction",
36           "Copied Clusters & Reduction",
37           "Copied Clusters & Reduction with First-Touch Policy",
38           "Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization",
39           "Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization",
40           "Shared Clusters with GOMP_CPU_AFFINITY[0-7][32-40]"]
41
42 subtitles = [{"Size, Coords, Clusters, Loops} = {256, 16, 32, 10}",
43               "{Size, Coords, Clusters, Loops} = {256, 1, 4, 10}",
44               "{Size, Coords, Clusters, Loops} = {256, 16, 32, 10}"]
45
46 for i in range(0,8):
47     x_axis = threads
48     y_axis = total_times[i*nthreads:i*nthreads+nthreads]
49     if (i==3 or i==4 or i==5) : seq_time = seq_totals[1]
50     else: seq_time = seq_totals[0]
51     print(f"Times for version{i}: ", y_axis)
52     plt.figure(figsize=(8,6))
53     plt.gca().set_facecolor("#e6e6fa")
54     plt.xscale('log')
55     widths = 0.6*np.diff(threads, prepend=threads[0] / 2) * 0.8
56     plt.bar(x_axis, y_axis, width=widths, color="#4b0082", align="center")
57     plt.xticks(x_axis, [str(t) for t in threads])
58     plt.plot(x_axis, [seq_time/t for t in threads], color ='g', linestyle='--')
59     plt.suptitle(titles[i], size = 12)
60     plt.title(subtitles[int(i/3)], fontstyle = 'oblique', size = 10)
61     plt.xlabel("Number of threads")
62     plt.ylabel("Time(secs)")
63     plt.grid('y')
64     plt.savefig(f"fig{i}.png")
65     plt.clf()
```

plots.py

```
1 # import seaborn as sns
2 import numpy as np
3 import matplotlib.pyplot as plt
4 #import pandas
5
6
7 total_times = []
8 loop_times = []
9
10 with open("sandman.out", 'r') as file:
11     for line in file:
12         if (line.startswith("omp_num_threads")):
13             t = float(line.split(',')[-1])
14             if (t is not None):
15                 total_times.append(t)
16
17
18 print("Total Times:", total_times)
19 print("Loop Times:", loop_times)
20
21 threads = [1,2,4,8,16,32,64]
22 GridSize = [1024, 2048, 4096]
23 BSizes = [256, 128, 64, 32, 16]
24 nthreads = len(threads)
25
26 for grid in range (0,len(GridSize)):
27     for bsize in range(0,len(BSizes)):
28         start_idx = grid*(nthreads*len(BSizes)) + nthreads*bsize
29         end_idx = start_idx + nthreads
30         x_axis = threads
31         y_axis = total_times[start_idx:end_idx]
32         print(f"Times for S={GridSize[grid]}, BS={BSizes[bsize]}: ", y_axis)
33         plt.figure(figsize=(8,6))
34         plt.gca().set_facecolor("#e6e6fa")
35         plt.xscale('log')
36         widths = 0.6*np.diff(threads, prepend=threads[0] / 2) * 0.8
37         plt.bar(x_axis, y_axis, width=widths, color="#4b0082", align="center")
38         plt.xticks(x_axis, [str(t) for t in threads])
39         plt.plot(x_axis, [y_axis[0]/t for t in threads], color ='g', linestyle='--')
40         plt.suptitle("Total Times for Floyd-Warshall Recursive", size = 12)
41         plt.title(f"Grid Size = {GridSize[grid]}, Block Size = {BSizes[bsize]}", fontstyle =
42 'oblique', size = 10)
43         plt.xlabel("Number of threads")
44         plt.ylabel("Time(secs)")
45         plt.grid('y')
46         plt.savefig(f"fig{GridSize[grid]}_{BSizes[bsize]}.png")
47         plt.clf()
```

Αμοιβαίος Αποκλεισμός-Κλειδώματα

Στο συγκεκριμένο ερώτημα καλούμαστε να αξιολογήσουμε τους διαφορετικούς τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό.

Μας δίνονται έτοιμες όλες οι υλοποίησεις των κλειδωμάτων. Για την εκτέλεση του συγκεκριμένου data set (Size = 32, Coords = 16, Clusters = 32, Loops = 10) στον scirouter χρησιμοποιήσαμε το ακόλουθο script :

```
#!/bin/bash
## Give the Job a descriptive name
#PBS -N run_kmeans
## Output and error files
#PBS -o run_kmeans.out
#PBS -e run_kmeans.err
## How many machines should we get?
#PBS -l nodes=1:ppn=8
##How long should the job run for?
#PBS -l walltime=00:10:00
## Start
## Run make in the src folder (modify properly)
module load openmp
cd /home/parallel/parlab09/a2_new/a2/kmeans
export SIZE=32
export COORDS=16
export CLUSTERS=32
export LOOPS=10
for n in 1 2 4 8 16 32 64; do
    export OMP_NUM_THREADS=$n
    echo "Setting OMP_NUM_THREADS=$n" >&2
    export GOMP_CPU_AFFINITY="0-$((($n - 1)))"
    echo "Running ./kmeans_omp_array_lock" >&2
    ./kmeans_omp_array_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_clh_lock" >&2
    ./kmeans_omp_clh_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_critical" >&2
    ./kmeans_omp_critical -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_naive" >&2
    ./kmeans_omp_naive -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_nosync_lock" >&2
    ./kmeans_omp_nosync_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_pthread_mutex_lock" >&2
    ./kmeans_omp_pthread_mutex_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_pthread_spin_lock" >&2
    ./kmeans_omp_pthread_spin_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_tas_lock" >&2
    ./kmeans_omp_tas_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_ttas_lock" >&2
    ./kmeans_omp_ttas_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
done
```

Τεχνικές συγχρονισμού

1) pthread_mutex_lock

Σε αυτήν την τεχνική χρησιμοποιείται ένα κλείδωμα αμοιβαίου αποκλεισμού και μεσολαβεί το λειτουργικό σύστημα σε περίπτωση αποτυχίας (context switch). Έτσι, επιτρέπει σε άλλες διεργασίες να τρέχουν μέχρι να ξυπνήσει από κάποιο κατάλληλο signal. Τότε, επιχειρεί εκ νέου να μπει στο κρίσιμο τμήμα.

2) pthread_spin_lock

Σε αυτήν την τεχνική, χρησιμοποιείται και πάλι ένα κλείδωμα όμως το κάθε νήμα εκτελεί busy waiting loop για την απόκτηση του. Έτσι, δεν επιτρέπει σε άλλο νήμα να τρέξει στην θέση του (εκτός

φυσικά εαν περάσει χρόνος ίσος με το runtime quantum και το αποσύρει ο scheduler) σπαταλώντας ωφέλιμο CPU time. Αυτό το μοντέλο προσφέρεται για operations που μπλοκάρουν μόνο για λίγους κύκλους, δηλαδή ο χρόνος αναμονής είναι μικρότερος του context switching overhead.

3) tas_lock

Αυτή η τεχνική βασίζεται στην υποστήριξη από το υλικό και χρησιμοποιεί την ατομική εντολή test_and_set που προσφέρει το ISA. Θέτει **ταυτόχρονα** το κλείδωμα (μεταβλητή state) σε 1 και επιστρέφει την προηγούμενη τιμή της (μεταβλητή test). Αν η προηγούμενη τιμή είναι 0 τότε το νήμα δέσμευσε επιτυχώς το κλείδωμα. Κάθε νήμα που προσπαθεί να μπει στο κρίσιμο εκτελεί ένα dummy while loop με την tas. Σε κάθε iteration γράφει στην θέση μνήμης της state, που είναι **μοιραζόμενη**, και στέλνει cache line invalidation στα υπόλοιπα με βάση το πρωτόκολλο MESI για συνάφεια κρυφών μνημών. Συνεπώς, δημιουργείται υπερβολικά μεγάλη και περιττή συμφόρηση στο δίαυλο.

4) ttas_lock

Μοιάζει με την tas, οστόσω το νήμα δεν γράφει απευθείας την state αλλά επιχειρεί πρώτα να την διαβάσει (test). Εαν η τιμή της δεν είναι 0, παραμένει μέσα στο busy wit loop και μόλις διαβάσει τιμή 0, προσπαθεί να γράψει σε αυτήν (test_and_set). Τα διαδοχικά reads δεν κοστίζουν σε bandwidth αφού δεν στέλνουν κάποια ενημέρωση μέσω bus. Ο αριθμός των writes μειώνεται σημαντικά όρα και τα συνολικά invalidations. Περεταίρω βελτίωση γίνεται με εκθετική οπισθοχώρηση (κατά το read και έτσι μειώνονται dummy CPU cycles και αποφεύγονται περισσότερα αποτυχημένα writes) αλλά δεν το εξετάζουμε σε αυτήν την άσκηση.

5) array_lock

Σε αυτήν την τεχνική κάθε νήμα έχει μια δική του μεταβλητή slot, ένα global πίνακα flag και που είναι τώρα το τέλος της ουράς. Κάθε φορά που προσπαθεί ένα νήμα να πάρει το κλείδωμα, παίρνει το τέλος της ουράς και κάνει ατομική αύξηση κατά 1, θέτει αυτό ως δικό του slot και περιμένει πότε θα γίνει true. Κάθε φορά που ένα νήμα αποδεσμεύει το κλείδωμα, ξαναθέτει το slot του ως false και κάνει το επόμενο true ώστε να πάρει το κλείδωμα αυτός που έχει το επόμενο slot. Αυτή η τεχνική έχει λιγότερη συμφόρηση στο δίαυλο γιατί κάθε νήμα κάνει πάντα 3 αλλαγές για να δεσμεύσει και να αποδεσμεύσει. Επίσης είναι δίκαιη, δηλαδή τα νήματα εκτελούν το κρίσιμο τμήμα με την ίδια σειρά που προσπάθησαν να το δεσμεύσουν. Ωστόσω, ένα νήμα πρέπει να περιμένει στην χειρότερη περίπτωση μια πλήρη περιστροφή του δακτυλίου ώστε να μπει στο κρίσιμο τμήμα ακόμη και αν είναι το μοναδικό που το επιδιώκει.

6) clh_lock

Σε αυτήν την τεχνική κάθε νήμα έχει ένα κόμβο με ένα κλείδωμα. Κάθε φορά που προσπαθεί να δεσμεύσει το κλείδωμα βάζει το δικό του κλείδωμα να είναι 1, αλλάζει ατομικά τον δείκτη στο κόμβο που αναπαριστά το τέλος της ουράς στον εαυτό του και μετά περιμένει πότε το κλείδωμα του προηγούμενου θα γίνει 0. Αντίστοιχα όταν αποδεσμεύει το κλείδωμα απλά θέτει το δικό του κλείδωμα σε 0. Το μεγάλο πλεονέκτημα αυτής της τεχνικής είναι πως είναι πολύ κλιμακώσιμη για αρκετά threads καθώς υπάρχει μόνο 1 κοινή μεταβλητή για τα threads και όχι ολόκληρος πίνακας.

7) pragma omp critical

Θα αξιολογήσουμε και την επίδοση της βιβλιοθήκης του OpenMP για το κρίσιμο τμήμα.

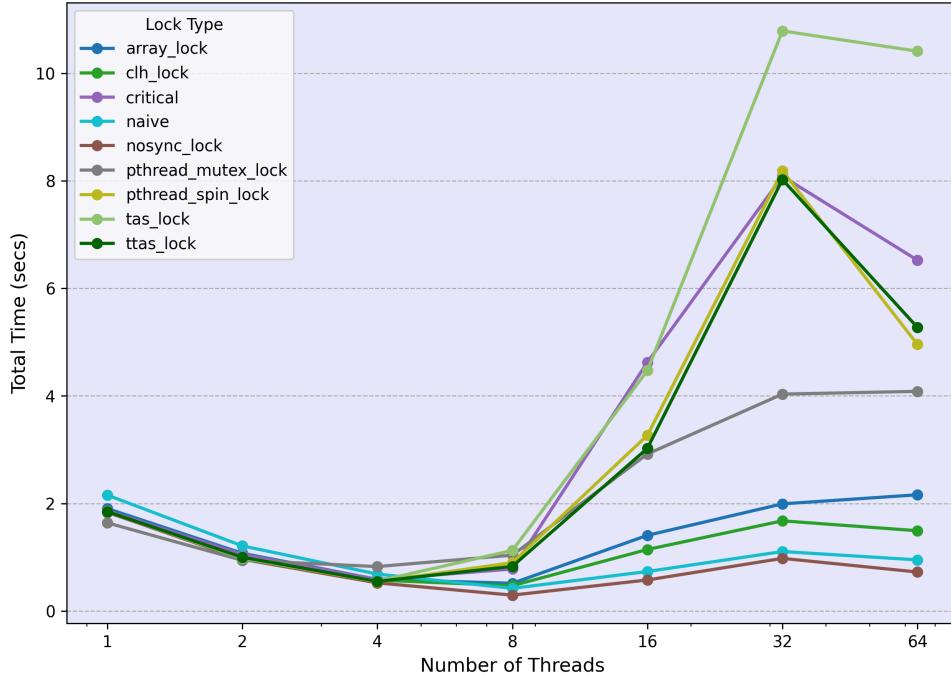
8) pragma omp atomic

Θα αξιολογήσουμε επίσης και την επίδοση μέχρι μόνο 2 ατομικών εντολών και όχι κρίσιμου τμήματος, καθώς μπορεί να μην αλλάζουν τις ίδιες μεταβλητές 2 νήματα.

Αποτελέσματα

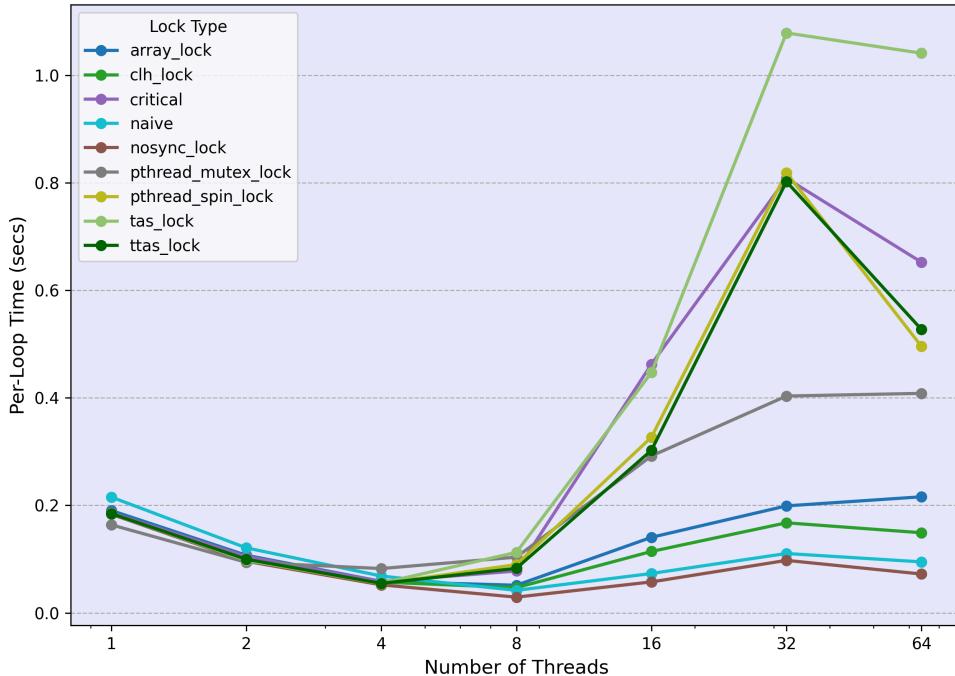
Locking Mechanism vs Time

Configuration = {32, 16, 32, 10}



Locking Mechanism vs Time

Configuration = {32, 16, 32, 10}



Παρατηρήσεις

Γενικά με την αύξηση των threads τα race conditions είναι συχνότερα οπότε οι συνολικοί χρόνοι εκτέλεσης ανεξαρτήτως μηχανισμού έχουν bottleneck το κόστος συγχρονισμού όταν πια η λύση μας πάψει να είναι scalable (από τα 8 threads και πάνω όπως φαίνεται στο σχήμα).

naive

Η υλοποίηση naive χρησιμοποιεί atomic add για την εγγραφή στα arrays newClusterSize και newClusters. Συγκεκριμένα, για παράμετρο coords = 16, πραγματοποιεί 16 + 1 (για το neeClusterSize) ατομικές εγγραφές. Από 8 threads και πάνω, προσεγγίζει καλύτερα από τις υπόλοιπες την no_sync η οποία δεν χρησιμοποιεί κανένα μηχανισμό συγχρονισμού οπότε παράγει λάθος αποτελέσματα) και χρησιμοποιείται μόνο ως σημείο αναφοράς βέλτιστου χρόνου.

spinlocks

Από 8 threads και πάνω οι μηχανισμοί *tas* και *ttas* δεν κλιμακώνουν εξαιτείας της συμφόρησης που προκαλούν στον δίαυλο με τα αλλεπάλληλα cache line invalidations ειδικότερα όταν τα δεδομένα χρειάζεται να μεταφέρονται πλέον εκτός του NUMA cluster οπότε χρειάζεται να διατηρείται η συνάφεια και μεταξύ των L3 Caches. Ακόμη, όσο περισσότερα γίνονται τα νήματα τόσο αυξάνονται και οι αποτυχημένες προσπάθεις της *test_and_set* και στις δύο περιπτώσεις εξαιτείας της μεγάλης ζήτησης του ίδιου *locl*. Η *ttas* έχει οστόσω καλύτερες επιδόσεις αφού γλιτώνει κάποια περιττά writes, και μάλιστα για < 8 νήματα έχει την καλύτερη επίδοση. Παρόμοια είναι και η συμπεριφορά του *pthread_spinlock* αφού όλα βασίζονται στην λογική των busy-wait loops σπαταλώντας CPU time. Με βάση το implementation στην glibc, η *pthread_spin_lock* χρησιμοποιεί ένα υβρίδιο των προηγούμενων 2. Στην 1η προσπάθεια, χρησιμοποιεί την ατομική εντολή *atomic_exchange* που θεωρεί ταχύτερη εαν παρέχεται από το υλικό και δεν καλεί την CAS (*compare_and_swap*). Σε περίπτωση αποτυχίας απλά διαβάζει την μεταβλητή όπως η *ttas*. Παραθέτουμε τον αντίστοιχο κώδικα:

pthread_spin_lock.c

```
1  /* pthread_spin_lock -- lock a spin lock. Generic version.
2   Copyright (C) 2012-2024 Free Software Foundation, Inc.
3   This file is part of the GNU C Library.
4
5   The GNU C Library is free software; you can redistribute it and/or
6   modify it under the terms of the GNU Lesser General Public
7   License as published by the Free Software Foundation; either
8   version 2.1 of the License, or (at your option) any later version.
9
10  The GNU C Library is distributed in the hope that it will be useful,
11  but WITHOUT ANY WARRANTY; without even the implied warranty of
12  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
13  Lesser General Public License for more details.
14
15  You should have received a copy of the GNU Lesser General Public
16  License along with the GNU C Library; if not, see
17  <https://www.gnu.org/licenses/>. */
18
19 #include <atomic.h>
20 #include "pthreadP.h"
21 #include <shlib-compat.h>
22
23 int
24 __pthread_spin_lock (pthread_spinlock_t *lock)
25 {
26     int val = 0;
27
28     /* We assume that the first try mostly will be successful, thus we use
29      atomic_exchange if it is not implemented by a CAS loop (we also assume
30      that atomic_exchange can be faster if it succeeds, see
31      ATOMIC_EXCHANGE_USES_CAS). Otherwise, we use a weak CAS and not an
32      exchange so we bail out after the first failed attempt to change the
33      state. For the subsequent attempts we use atomic_compare_and_exchange
34      after we observe that the lock is not acquired.
35      See also comment in pthread_spin_trylock.
36      We use acquire M0 to synchronize-with the release M0 store in
37      pthread_spin_unlock, and thus ensure that prior critical sections
38      happen-before this critical section. */
39 #if ! ATOMIC_EXCHANGE_USES_CAS
```

```

40  /* Try to acquire the lock with an exchange instruction as this architecture
41  has such an instruction and we assume it is faster than a CAS.
42  The acquisition succeeds if the lock is not in an acquired state. */
43  if (__glibc_likely (atomic_exchange_acquire (lock, 1) == 0))
44  return 0;
45 #else
46  /* Try to acquire the lock with a CAS instruction as this architecture
47  has no exchange instruction. The acquisition succeeds if the lock is not
48  acquired. */
49  if (__glibc_likely (atomic_compare_exchange_weak_acquire (lock, &val, 1)))
50  return 0;
51 #endif
52
53 do
54 {
55  /* The lock is contended and we need to wait. Going straight back
56  to cmpxchg is not a good idea on many targets as that will force
57  expensive memory synchronizations among processors and penalize other
58  running threads.
59  There is no technical reason for throwing in a CAS every now and then,
60  and so far we have no evidence that it can improve performance.
61  If that would be the case, we have to adjust other spin-waiting loops
62  elsewhere, too!
63  Thus we use relaxed MO reads until we observe the lock to not be
64  acquired anymore. */
65  do
66  {
67  /* TODO Back-off. */
68
69  atomic_spin_nop ();
70
71  val = atomic_load_relaxed (lock);
72 }
73 while (val != 0);
74
75 /* We need acquire memory order here for the same reason as mentioned
76 for the first try to lock the spinlock. */
77 }
78 while (!atomic_compare_exchange_weak_acquire (lock, &val, 1));
79
80 return 0;
81 }
82 versioned_symbol (libc, __pthread_spin_lock, pthread_spin_lock, GLIBC_2_34);
83
84 #if OTHER_SHLIB_COMPAT (libpthread, GLIBC_2_2, GLIBC_2_34)
85 compat_symbol (libpthread, __pthread_spin_lock, pthread_spin_lock, GLIBC_2_2);
86#endif

```

array locks

To array_lock και clh_lock είναι οι πιο scaleable μηχανισμοί αφού δεν χαρακτηρίζονται το overhead της atomic σε < 8 νήματα και από το overhead του πρωτοκόλλου MESI για >8 νήματα. Επειδή τα νήματα εισέρχονται στο κρίσιμο τμήμα με ένα καθορισμένο μοτίβο και όλα εν τέλει θα μπουν σε αυτό, κανένα slot στον δακτύλιο δεν μένει αδρανές, ενώ όσο τα νήματα περιμένουν την σειρά τους δεν πραγματοποιούν ανούσιες προβάσεις την μνήμη. Αντιθέτως, εαν το concurrency rate ήταν μικρότερο, ο μηχανισμός θα έπασχε από το overhead διάσχισης ολόκληρου του δακτυλίου προκειμένου να ικανοποιήσει λ.χ. ένα μόνο αίτημα. Αντίστοιχα, για το chl_lock μειώνεται το contention για ένα κοινό global lock και κάθε νήμα εκτελεί spinlock στην μεταβλητή του προηγούμενου node. Συγκρούσεις εξακολουθούμε να έχουμε για την είσοδο στο τέλος της ουράς, παρόλα αυτά η διατήρηση μιας λίστας είναι πιο φθηνή από έναν circular buffer και φαίνεται πως γι' αυτό πετυχείνει καλύτερους χρόνους.

mutex

Για < 8 νήματα το context switch είναι αρκετά ακριβό (τουλάχιστον για το συκεκριμένο configuration όπου απαιτεί 17 μόνο πράξεις στο κρίσιμο τμήμα που δεν είναι τόσο computational intense), όμως φαίνεται να υπερτερεί έναντι των spinlocks για 16 και πάνω νήματα όπου το bus traffic γίνεται

αφόρητο. Η υλοποίηση του omp critical φαίνεται ότι συδέεται με την χρήση ενός mutex οστόσω έχει χειρότερη επίδοση από το να το καλέσουμε explicitly, που διακιολογούμε εφόσον παρέχει ένα higher level abstraction άρα και επιπλέον overheads. Συγκεκριμένα, διατηρεί μέσω του global context manager ένα mapping για named critical regions το οποίο εισάγει μια πολυπλοκότητα, ενώ τα omp directives απαιτούν κάθε φορά και την κλήση της libomp. Παραθέτουμε τον αντίστοιχο κώδικα:

gomp_critical.c

```

1  /* Copyright (C) 2005-2024 Free Software Foundation, Inc.
2   Contributed by Richard Henderson <rth@redhat.com>.
3
4   This file is part of the GNU Offloading and Multi Processing Library
5   (libgomp).
6
7   Libgomp is free software; you can redistribute it and/or modify it
8   under the terms of the GNU General Public License as published by
9   the Free Software Foundation; either version 3, or (at your option)
10  any later version.
11
12  Libgomp is distributed in the hope that it will be useful, but WITHOUT ANY
13  WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
14  FOR A PARTICULAR PURPOSE. See the GNU General Public License for
15  more details.
16
17  Under Section 7 of GPL version 3, you are granted additional
18  permissions described in the GCC Runtime Library Exception, version
19  3.1, as published by the Free Software Foundation.
20
21  You should have received a copy of the GNU General Public License and
22  a copy of the GCC Runtime Library Exception along with this program;
23  see the files COPYING3 and COPYING.RUNTIME respectively. If not, see
24  <http://www.gnu.org/licenses/>. */
25
26 /* This file handles the CRITICAL construct. */
27
28 #include "libgomp.h"
29 #include <stdlib.h>
30
31
32 static gomp_mutex_t default_lock;
33
34 void
35 GOMP_critical_start (void)
36 {
37   /* There is an implicit flush on entry to a critical region. */
38   __atomic_thread_fence (MEMMODEL_RELEASE);
39   gomp_mutex_lock (&default_lock);
40 }
41
42 void
43 GOMP_critical_end (void)
44 {
45   gomp_mutex_unlock (&default_lock);
46 }
47
48 #ifndef HAVE_SYNC_BUILTINS
49 static gomp_mutex_t create_lock_lock;
50#endif
51
52 void
53 GOMP_critical_name_start (void **pptr)
54 {
55   gomp_mutex_t *plock;
56
57   /* If a mutex fits within the space for a pointer, and is zero initialized,
58    then use the pointer space directly. */
59   if (GOMP_MUTEX_INIT_0
60     && sizeof (gomp_mutex_t) <= sizeof (void *)
61     && __alignof (gomp_mutex_t) <= sizeof (void *))
62     plock = (gomp_mutex_t *) pptr;
63
64   /* Otherwise we have to be prepared to malloc storage. */
65   else
66   {

```

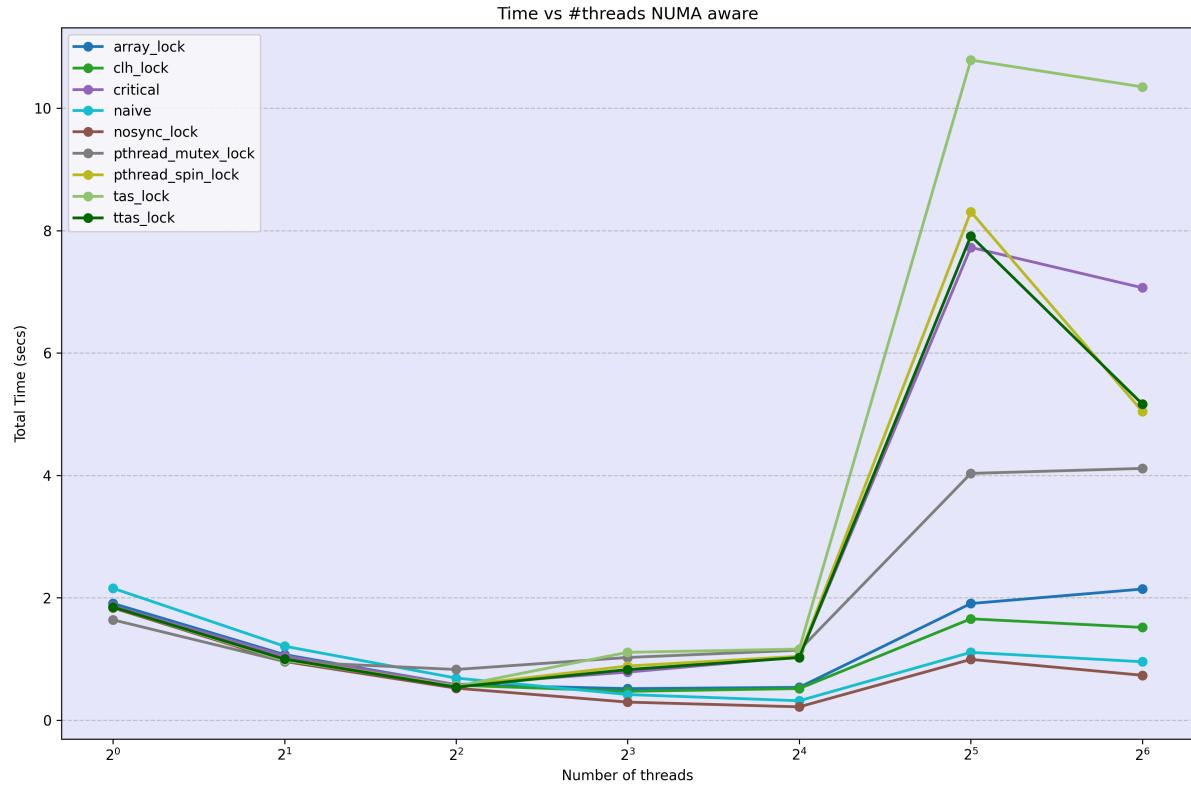
```

67     plock = *pptr;
68
69     if (plock == NULL)
70     {
71 #ifdef HAVE_SYNC_BUILTINS
72     gomp_mutex_t *nlock = gomp_malloc (sizeof (gomp_mutex_t));
73     gomp_mutex_init (nlock);
74
75     plock = __sync_val_compare_and_swap (pptr, NULL, nlock);
76     if (plock != NULL)
77     {
78         gomp_mutex_destroy (nlock);
79         free (nlock);
80     }
81     else
82     {
83         plock = nlock;
84     }
85 #else
86     gomp_mutex_lock (&create_lock_lock);
87     plock = *pptr;
88     if (plock == NULL)
89     {
90         plock = gomp_malloc (sizeof (gomp_mutex_t));
91         gomp_mutex_init (plock);
92         __sync_synchronize ();
93         *pptr = plock;
94     }
95     gomp_mutex_unlock (&create_lock_lock);
96 #endif
97     }
98
99     gomp_mutex_lock (plock);
100 }
101
102 void
103 GOMP_critical_name_end (void **pptr)
104 {
105     gomp_mutex_t *plock;
106
107     /* If a mutex fits within the space for a pointer, and is zero initialized,
108      then use the pointer space directly. */
109     if (GOMP_MUTEX_INIT_0
110         && sizeof (gomp_mutex_t) <= sizeof (void *)
111         && __alignof (gomp_mutex_t) <= sizeof (void *))
112         plock = (gomp_mutex_t *) pptr;
113     else
114         plock = *pptr;
115
116     gomp_mutex_unlock (plock);
117 }
118
119 #if !GOMP_MUTEX_INIT_0
120 static void __attribute__((constructor))
121 initialize_critical (void)
122 {
123     gomp_mutex_init (&default_lock);
124 #ifndef HAVE_SYNC_BUILTINS
125     gomp_mutex_init (&create_lock_lock);
126 #endif
127 }
128#endif

```

Τέλος, όταν έχουμε 1 μόνο νήμα, δεν μπλοκάρει σε καμία απόπειρα απόκτησης του lock γι' αυτό όλες οι υλοποιήσεις είναι καλύτερες από την naive, (tas,ttas,array,clh εκτελούν ακριβώς τις ίδεις εντολές - ανάγονται σε 1 ανάγνωση και 1 εγγραφή) και η mutex υπερτερεί γιατί δεν πραγματοποιεί κανένα context switch.

Σημείωση : Αν χρησιμοποιούσαμε hyperthreading για τα 16 νήματα, δηλαδή βάζαμε τα 8 τελευταία νήματα εκτός των 64 λογικών, θα δούμε κλιμάκωση και για 16 νήματα όπως φαίνεται παρακάτω :



Ταυτόχρονες Δομές δεδομένων

Σε αυτό το ερώτημα εξετάζουμε πως κλιμακώνουν διάφορες ταυτόχρονες υλοποιήσεις για μια απλά συνδεδεμένη λίστα.

Οι ταυτόχρονες υλοποιήσεις που θα εξετάσουμε είναι οι εξής :

1) Coarse-grain locking

Σε αυτήν την υλοποίηση υπάρχει ένα γενικό κλείδωμα για όλη την δομή. Για κάθε προσθήκη ή αφαίρεση στοιχείου στη λίστα, το νήμα προσπαθεί να δεσμεύσει το κλείδωμα και να κάνει την κατάλληλη αλλαγή. Είναι πολύ απλό στην υλοποίηση όμως δεν θα κλιμακώσει καθόλου καθώς όλοι περιμένουν το ίδιο κλείδωμα και δεν εκμεταλλευόμαστε καθόλου παραλληλία σε ανεξάρτητα τμήματα της λίστας.

2) Fine-grain locking

Σε αυτήν την υλοποίηση υπάρχει ένα κλείδωμα για κάθε στοιχείο της λίστας. Ο τρόπος διάσχισης είναι hand-over-hand locking δηλαδή ένα νήμα προσπαθεί να δεσμεύσει τον επόμενο, όταν τα καταφέρει, αφήνει τον προηγούμενο. Αυτό είναι αναγκαστικό προς αποφυγή deadlock, εφόσον για ένα operation απαιτούνται κλειδώματα σε 2 στοιχεία (pred, curr) και θα δημιουργούνταν πρόβλημα αν 2 νήματα επιχειρούσαν να αλλάξουν 2 γειτονικά nodes και προσπαθούσαν να πάρουν τα κλειδώματα με αντίθετη σειρά. Μπορεί να δουλέψει καλύτερα από την coarse grain σε συγκεκριμένες περιπτώσεις αλλά το σημαντικότερο πρόβλημα είναι πως αν ένα νήμα θέλει να αλλάξει κάτι που βρίσκεται νωρίς στη λίστα, μπλοκάρει όλα τα άλλα νήματα που θέλουν να ψάξουν ή αλλάξουν κάτι που είναι πιο μετά στη λίστα.

3) Optimistic synchronization

Σε αυτήν την υλοποίηση ένα νήμα για κάθε αλλαγή, βρίσκει τον προηγούμενο και τον επόμενο προς αλλαγή, προσπαθεί να τους δεσμεύσει, ελέγχει αν η δομή είναι ακόμη συνεπής (δηλαδή είναι προσβάσιμοι και διαδοχικοί) και κάνει την αλλαγή. Η contains στη συγκεκριμένη υλοποίηση χρησιμοποιεί επίσης κλειδώματα αν και δεν χρειάζεται. Το κύριο πρόβλημα αυτής της υλοποίησης είναι πως η validate διατρέχει όλη την λίστα για να επιβεβαιώσει την συνέπεια και αυτό είναι πάρα πολύ χρονοβόρο.

4) Lazy synchronization

Σε αυτήν την υλοποίηση προσθέτουμε στη δομή μια boolean μεταβλητή που δείχνει αν ο κόμβος βρίσκεται στη λίστα ή έχει διαγραφεί. Η contains διατρέχει τη λίστα χωρίς να κλειδώνει και ελέγχει αυτήν την boolean μεταβλητή οπότε είναι wait-free. Η validate δεν διατρέχει την λίστα αλλά κάνει τοπικούς ελέγχους στον προηγούμενο και επόμενο κόμβο, δηλαδή ελέγχει αν ανήκουν στη δομή και οι 2 με την επιπλέον μεταβλητή και ο next του προηγούμενο είναι ο τωρινός. Η add/remove κάνουν πρώτα λογική και μετά φυσική αλλαγή των κόμβων.

5) Non-blocking

Σε αυτήν την υλοποίηση προσπαθούμε να αφαίρουμε τελείως την ανάγκη για κλειδώματα και να χρησιμοποιήσουμε τις ατομικές εντολές που μας δίνει το instruction set του εκάστοτε επεξεργαστή. Η κεντρική ιδέα είναι να χειριστούμε την boolean μεταβλητή marked και το πεδίο next σαν μία μεταβλητή. Κάνει ατομικό σύνθετο έλεγχο και αλλαγή με 1 εντολή compare and set. Έτσι η διαγραφή κάνει με 1 εντολή validate και λογική διαγραφή και 1 μόνο προσπάθεια φυσικής διαγραφής. Η find/contains είναι αυτή που εξετάζει αν υπάρχει στοιχείο που έχει διαγραφεί λογικά και όχι φυσικά και το αναλαμβάνει εκείνη. Η προσθήκη αναγκαστικά ξαναπροσπαθεί μέχρι να τα καταφέρει για να είναι συνεπής η δομή.

Μας δίνονται έτοιμες όλες οι παραπάνω ταυτόχρονες υλοποιήσεις. Για την ζητούμενη εκτέλεση, το σειριακό πρόγραμμα εκτελέστηκε μόνο με 1 thread αλλιώς θα υπάρχει πρόβλημα, για 128 νήματα χρησιμοποιήθηκε oversubscription. Δηλαδή η μεταβλητή MT_CONF τέθηκε σε 0,1,...63,0,1,...63 ώστε να δημιουργηθούν και να πινάρουν 128 νήματα σε συγκεκριμένους πυρήνες. Επειδή οι λογικοί πυρήνες του sandman είναι 64, το scheduling των νημάτων πλέον το αναλαμβάνει το λειτουργικό και το software και όχι το ίδιο το υλικό όπως όταν χρησιμοποιούμε hyperthreading.

```

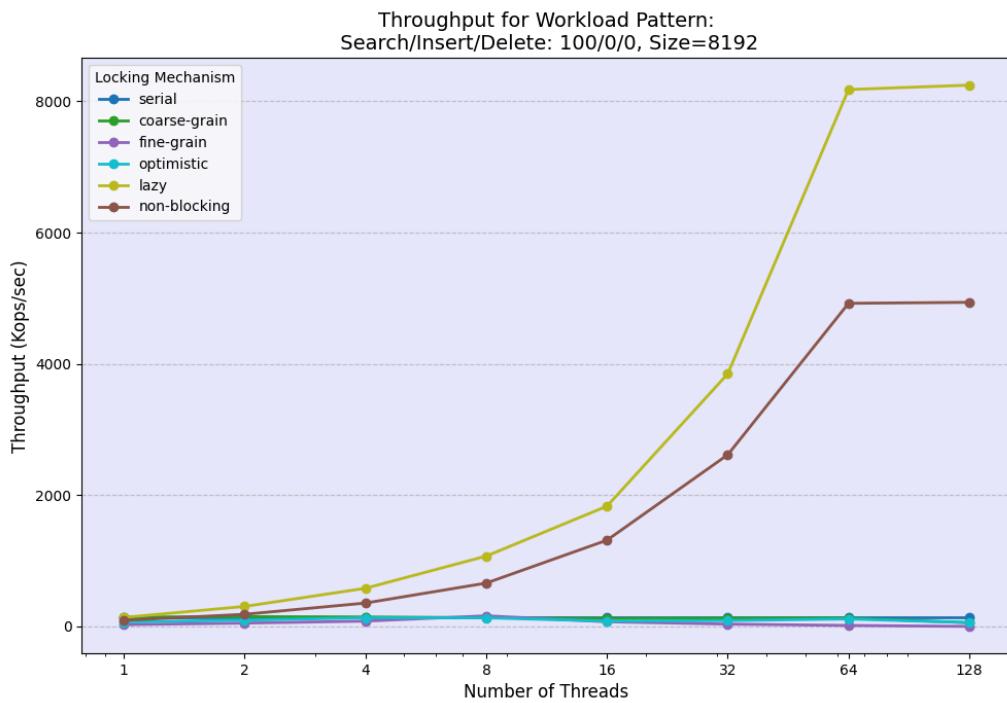
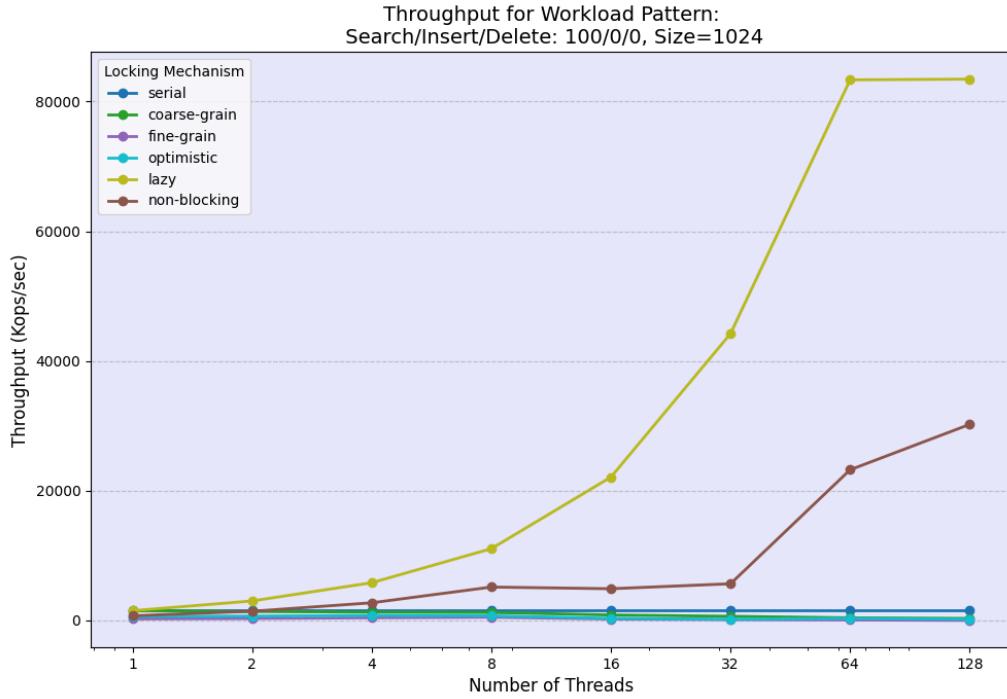
#!/bin/bash
## Give the Job a descriptive name
#PBS -N run_conc_ll
## Output and error files
#PBS -o run_conc_ll.out
#PBS -e run_conc_ll.err
## How many machines should we get?
#PBS -l nodes=1:ppn=8
##How long should the job run for?
#PBS -l walltime=01:00:00
## Start
## Run make in the src folder (modify properly)
module load openmp
cd /home/parallel/parlab09/a2_new/a2/conc_ll
choices=(
    "100 0 0"
    "80 10 10"
    "20 40 40"
    "0 50 50"
)
for lsize in 1024 8192; do
    export LSIZE=$lsize
    echo "LSIZE=$LSIZE"
    for choice in "${choices[@]}"; do
        read -r CONTAINS_PCT ADD_PCT REMOVE_PCT <<< "$choice"
        export MT_CONF="0"
        echo "serial"
        ./x.serial $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
        for n in 1 2 4 8 16 32 64 128; do
            if [ "$n" -eq 128 ]; then
                # Special case for n=128 for over subscription
                export MT_CONF="$((seq -s, 0 63),$(seq -s, 0 63))"
            else
                # Default case for n=1, 2, 4, ..., 64
                export MT_CONF=$((seq -s, 0 $((n-1))) )
            fi
            echo "coarse-grain"
            ./x.cgl $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
            echo "fine-grain"
            ./x.fgl $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
            echo "optimistic"
            ./x.opt $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
            echo "lazy"
            ./x.lazy $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
            echo "non-blocking"
            ./x.nb $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
        done
    done
done

```

Αποτελέσματα

Παρουσιάζονται τα Kops/sec με την μορφή line plots ανά κάθε διαφορετικό workload configuration

Η serial εκδοχή μας δείχνει την δυναμική του κάθε core, δεν χρησιμοποιεί παραλληλισμό ούτε κλειδώματα και έχει σταθερό throughput ανεξάρτητα από το πλήθος των νημάτων γι'αυτό και την θεωρούμε ως σημείο αναφοράς. Τα queries add / delete είναι υπολογιστικά ισοδύναμα οπότε μπορούμε να εξάγουμε το συνολικό ποσοστό τους έναντι των search για την ανάλυσή μας.

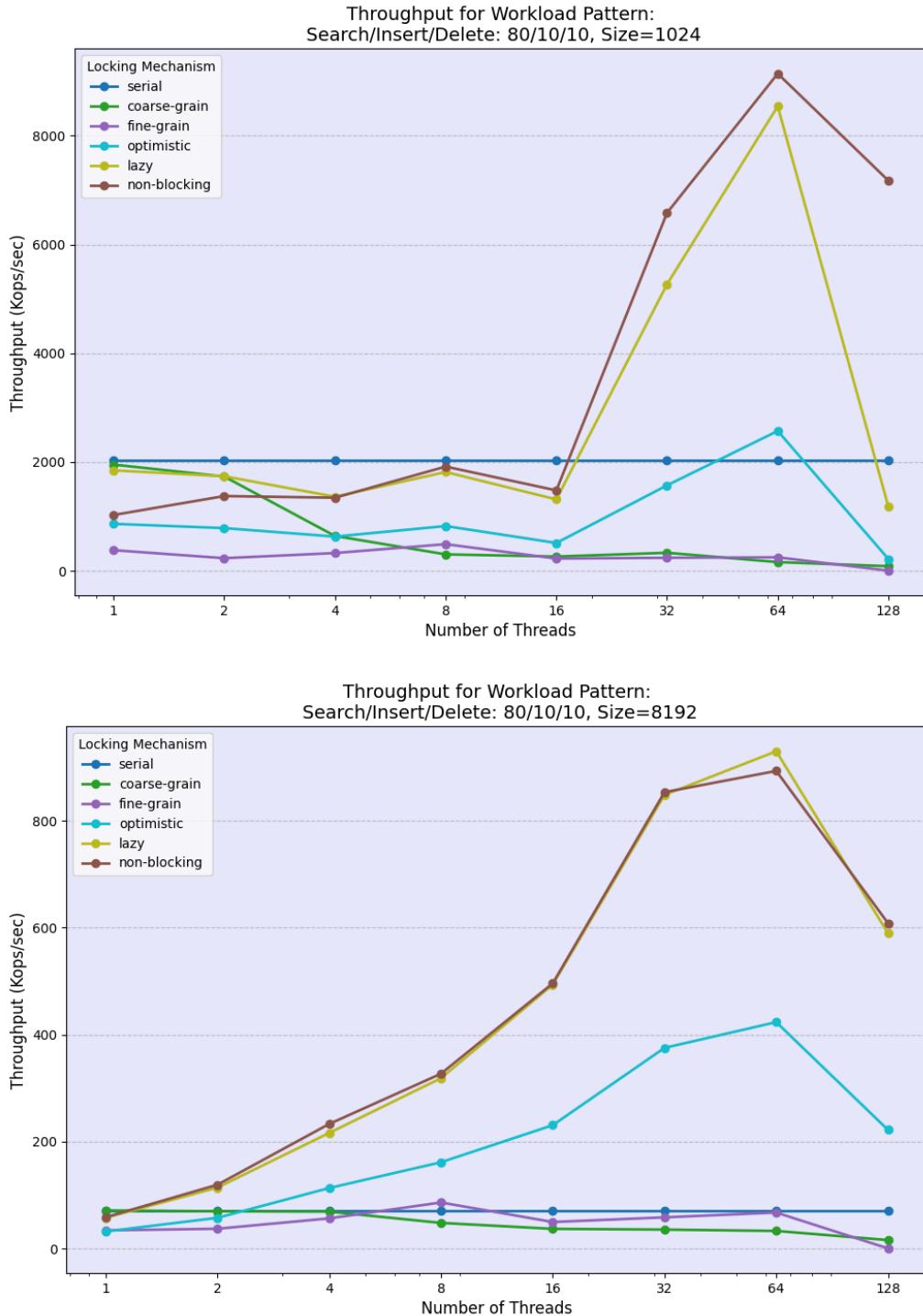


Workload 100/0/0

Στην περίπτωση που το workload αποτελείται εξ'ολοκλήρου από search queries, οι **coarse-grain**, **fine-grain** & και **optimistic** εκδοχές δεν κλιμακώνουν καθόλου για κανένα μέγεθος λίστας αφού μπλοκάρονται όλες σε κάποιο κλείδωμα (είτε είναι global για όλη την δομή στην πρώτη περίπτωση, είτε στην αρχή της λίστας και περνιέται μέσω hand-over-hand για τις άλλες). Ουσιαστικά όλες οι προσβάσεις σειριοποιούνται, οπότε δεν κερδίζουμε τίποτα από τον παραλληλισμό, μονάχα το κόστος δημιουργίας των threads και υλοποίησης των κλειδωμάτων.

Αντίθετα, η **lazy** είναι wait-free στην αναζήτηση επομένως κερδίζουμε throughput χαριν στον παραλληλισμό χωρίς bottlenecks. Στα 64 - 128 νήματα η επίδοση μένει σταθερή γιατί το μηχάνημα

έχει ήδη **100% utilization** (64 logical cores).



Workload 80/10/10

Προσθέτοντας μικρό ποσοστό add / delete βλέπουμε ότι η επιδόσεις όλων είναι καλύτερες αφού μέρος αυτών των λειτουργιών είναι η συνάρτηση contains που πραγματαποιεί αναζήτηση αλλά με wait free τρόπο. Χειρότερη είναι η **fine-grain** ειδικότερα αν queries στο τέλος της λίστας έπονται από queries στην αρχή της και αναγκάζονται να μπλοκάρουν μέχρι να αποκτήσουν το lock hand-over-hand τρόπο. Ακόμη, η διαδικασία αυτή εισάγει έξτρα πολυπλοκότητα μέσα από αλλεπάληλες κλήσεις lock / unlock μέχρι να φτάσει στους ζητούμενους κόμβους, οπότε καταλήγει να έχει χειρότερη

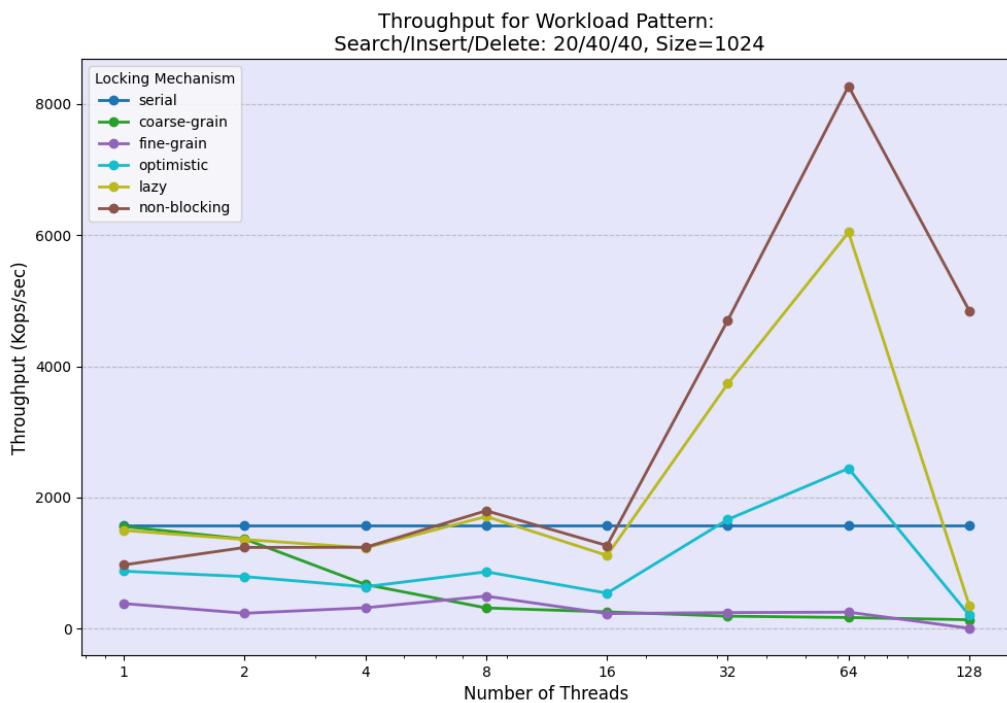
επίδοση από την **coarse-grain**. Η κατάσταση θα μπορούσε να αλλάξει εάν αντιστρέφαμε την σειρά των queries, όμως αυτά παράγονται τυχαία.

Η **optimistic** ακολουθεί την λογική readers-writer lock και διατρέχει την λίστα wait free κάνοντας μετά έναν έλεγχο συνέπειας της δομής (validate). Δεν κλιμακώνει όπως θα περίμεναμε επειδή η validate έχει γραμμική πολυπλοκότητα και κυριαρχεί το κόστος να διατρέξει την λίστα από την αρχή.

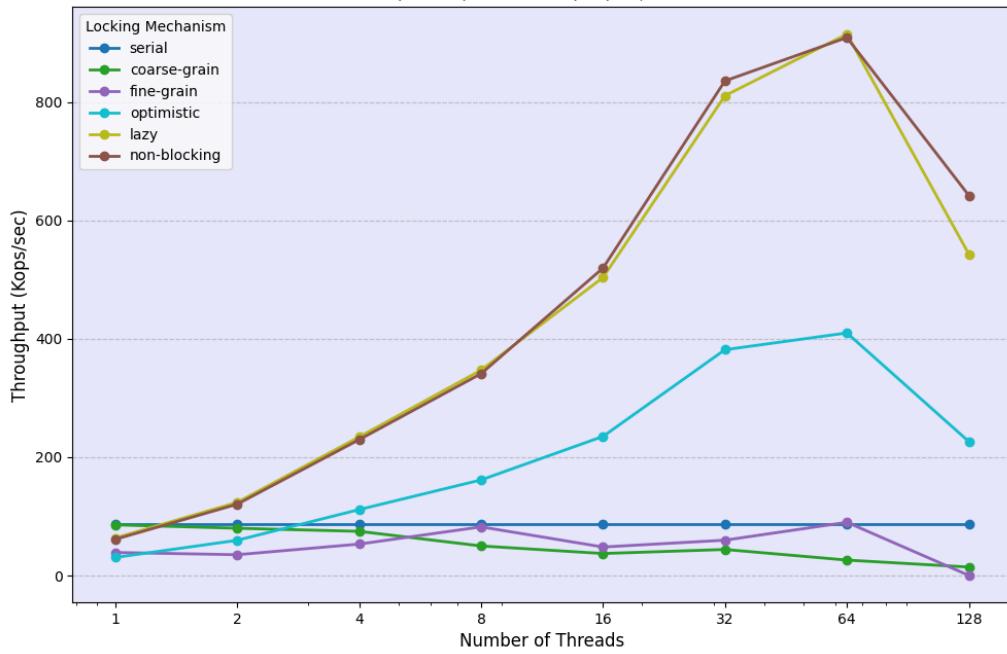
Αντίθετα, η **lazy** υλοποιεί την validate με σταθερό χρόνο, κοιτάζοντας απλά το valid bit για τους κόμβους pred, curr και από 16 threads και πάνω παρουσιάζει τεράστια κλιμάκωση.

Τέλος, η **non-block** είναι εγγενώς wait free και αξιοποιεί packed εντολές που παρέχει το ISA και είναι optimal.

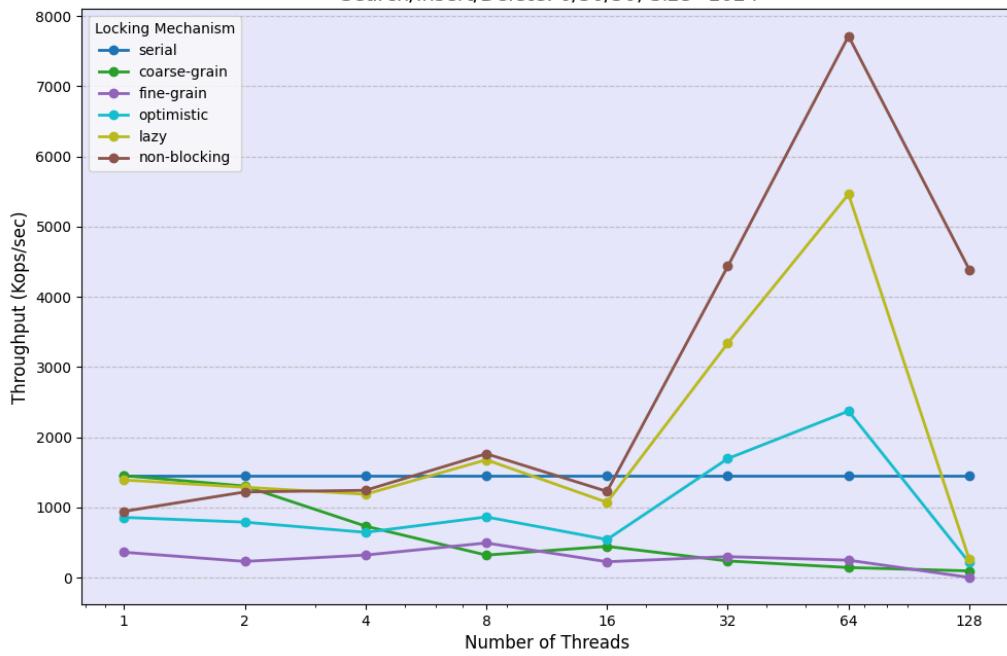
Η μείωση του throughput από τα 8 στα 16 οφείλεται στο ότι τόσο τα locks όσο και τα list data είναι shared στα threads και βγαίνουμε εκτός NUMA cluster. Δεν ισχύει το ίδιο και για μέγεθος 8192 όπου έτσι και αλλιώς η λίστα δεν χωράει ολόκληρη στην cache.

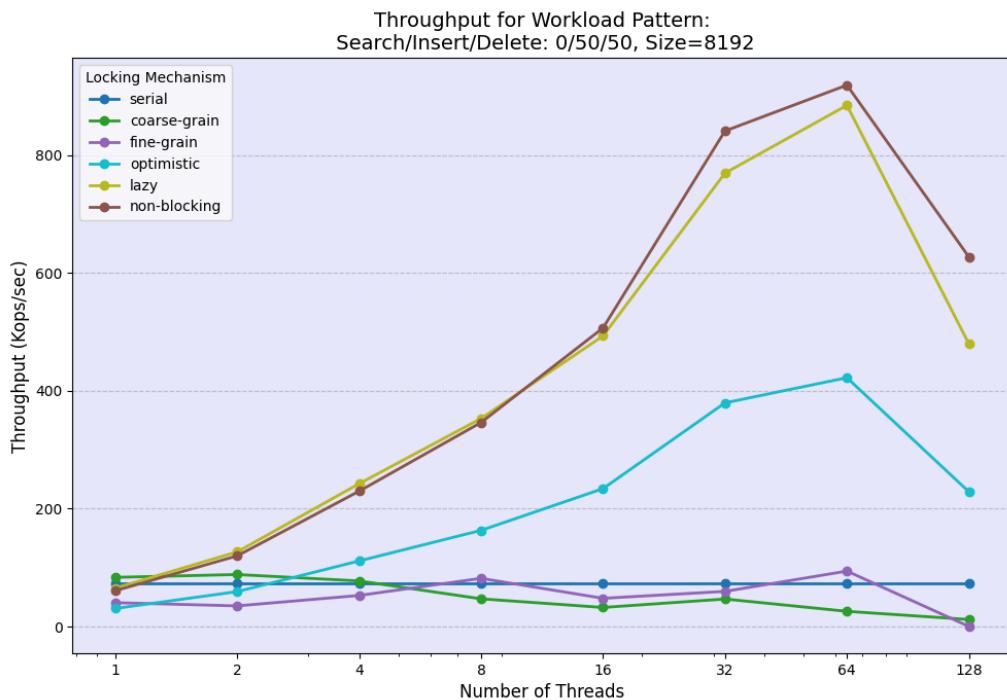


Throughput for Workload Pattern:
Search/Insert/Delete: 20/40/40, Size=8192



Throughput for Workload Pattern:
Search/Insert/Delete: 0/50/50, Size=1024





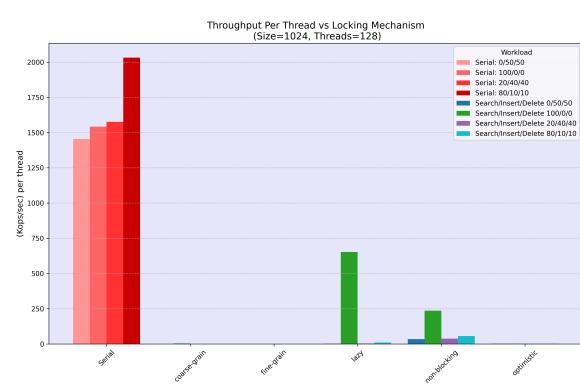
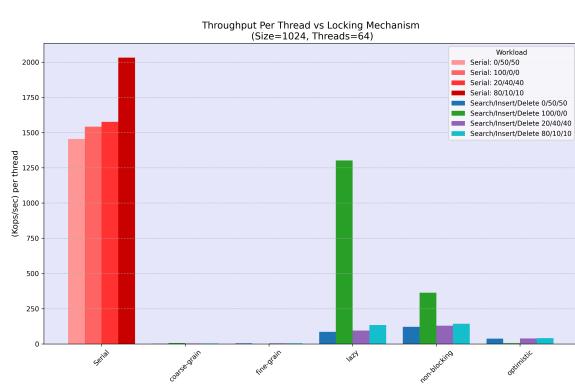
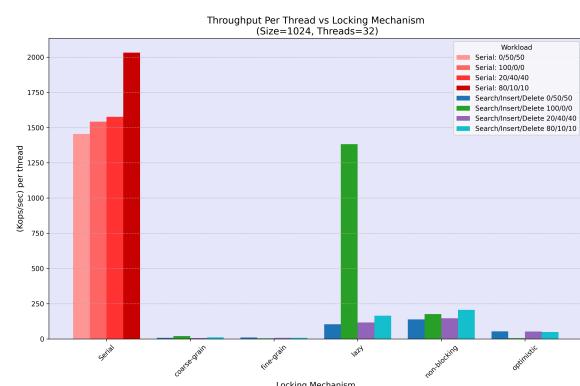
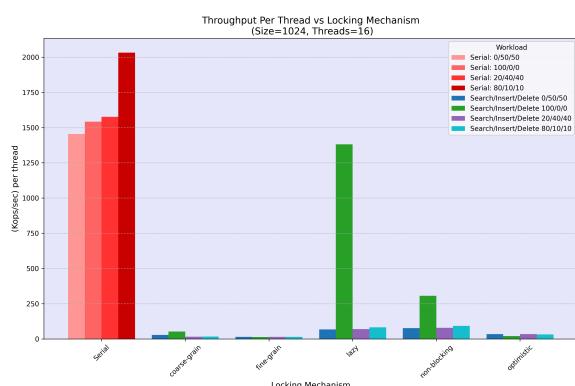
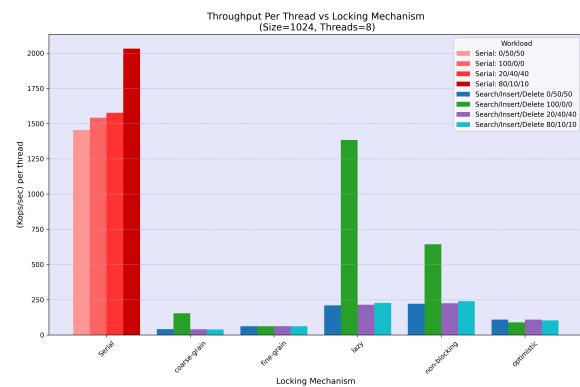
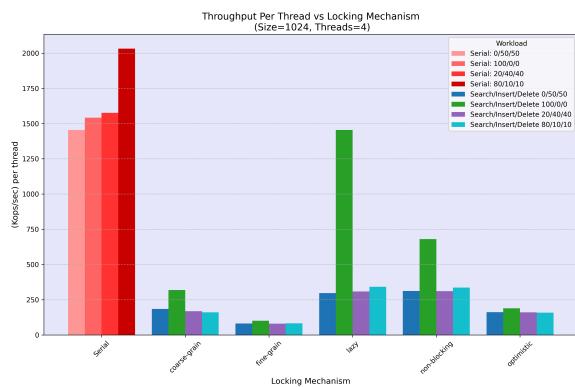
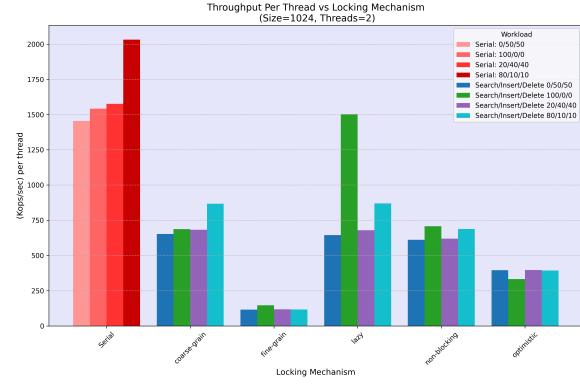
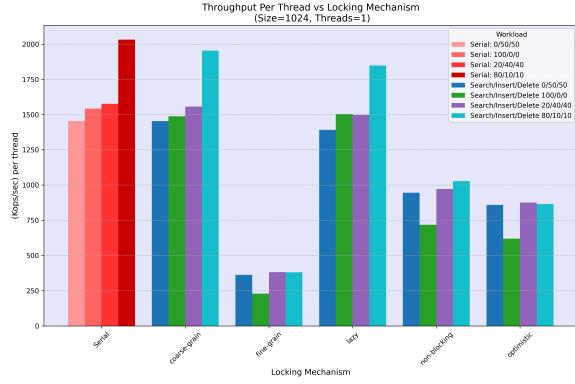
Workload 20/40/40 & Workload 0/50/50

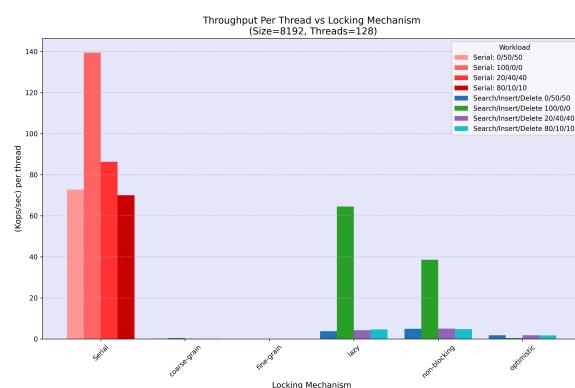
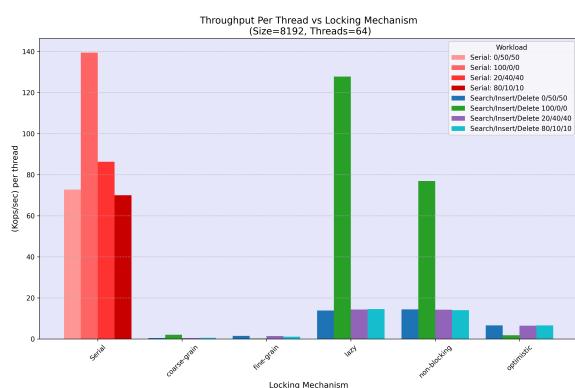
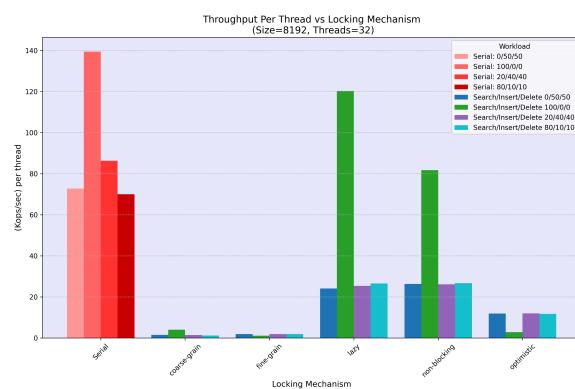
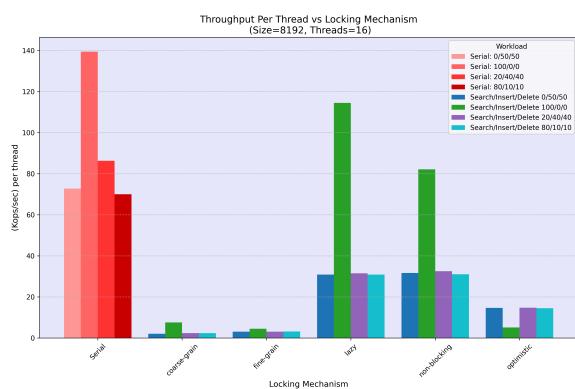
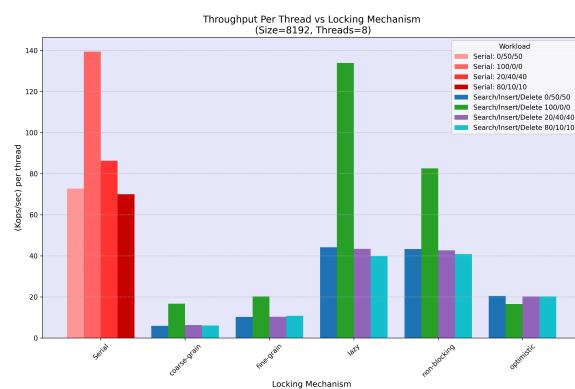
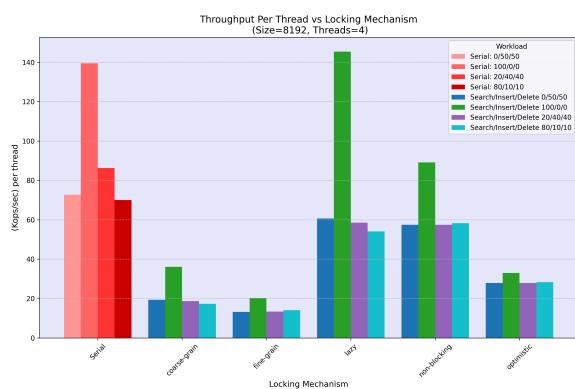
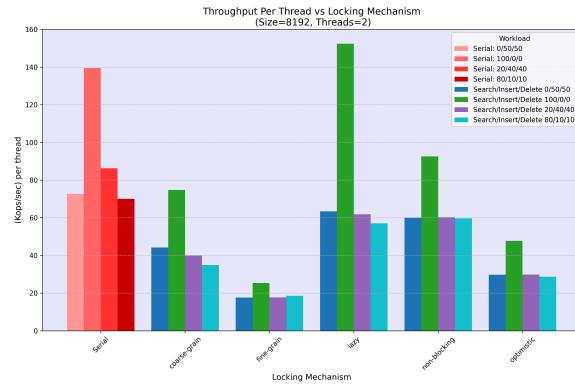
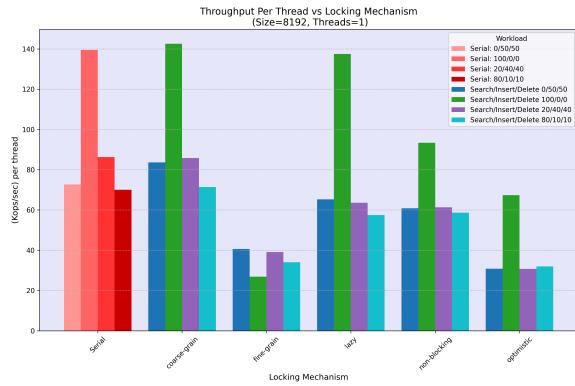
Αυξάνοντας παραπάνω το ποσοστό των add / delete η **lazy** χάνει σε throughput ενώ η **optimistic** κερδίζει στην περίπτωση του μικρού size=1024, και δεν παρουσιάζει αισθητές διαφορές για μεγάλο size=8192. Αυτό συμβαίνει επειδή τα conflicts για μεταβολή κοινών δεδομένων είναι σπανιότερα σε μεγάλο μέγεθος λίστας και εφόσον ο αριθμός των queries μένει σταθερός. Έστι το overhead των locking mechanisms μένει σχετικά σταθερό χάριν στο μικρό contention.

Σε όλα τα Workloads,

το throughput στην πολύ μεγάλη λίστα είναι σημαντικά μικρότερο αφού όλες οι λειτουργίες είναι γραμμικές ως προς το μέγεθος της λίστας και δεν μπορεί να αξιοποιηθεί πλήρως το data locality. Επιπλέον, η **lazy** τα πάει χειρότερα στο μικρό μήκος από την **non blocking**. Η ειδοποιός διαφορά των δύο αυτών υλοποιήσεων είναι πως η **lazy** στις διαγραφές είναι επίμονη και προσπαθεί να αποκτήσει τα locks των κόμβων που απαιτούνται μέχρι να τα καταφέρει, ενώ η **non blocking** κάνει την λογική διαγραφή και 1 μόνο προσπάθεια φυσικής διαγραφής. Σε workloads με λίγα contains(που κάνει την φυσική αφαίρεση στην **non blocking**) και αρκετά add/remove η **lazy** κάνει συνέχεια και τις φυσικές διαγραφές ενώ η **non blocking** κάνει λίγες και γλυτώνει χρόνο.

Παρακάτω φαίνονται τα κανονικοποιημένα throughputs / thread_count σε σύγκριση με την serial ενδοχή και πως αυτά επηρεάζονται από το workload:





Παράρτημα

Για την δημιουργία των γραφικών παραστάσεων χρημιοποίηθηκαν οι εξής κώδικες σε Python :

scraping_kmeans.py

```
1 import os
2 import matplotlib.pyplot as plt
3
4 #replace with the actual path
5 folder_path = "../"
6 file_name = "run_kmeans_hyper.err"
7
8 total_path = os.path.join(folder_path, file_name)
9
10 with open(total_path, "r") as file:
11     data = file.read()
12
13 omp_num_threads = 1
14 current_running = 'array_lock'
15 #we need just one the two, but may we need both
16 results_dict= {}
17 results_array = []
18 for line in data.splitlines():
19     words = line.split()
20     if len(words) == 0:
21         continue
22     #set the nthreads, set what is running or the exec time
23     if words[0] == 'Setting':
24         omp_num_threads = int(words[1].split('=')[-1])
25     if words[0] == 'Running':
26         temp = words[1].split('_')[2:]
27         current_running = '_'.join(temp)
28     if words[0] == 'nloops':
29         temp_res = float(words[5][0:-2])
30         if current_running not in results_dict:
31             results_dict[current_running] = {}
32             results_array[current_running] = []
33         results_dict[current_running][omp_num_threads] = temp_res
34         results_array[current_running].append(temp_res)
35
36
37 nthreads = [1, 2, 4, 8, 16, 32, 64]
38 colours = [
39     "#1f77b4", # Blue
40     "#2ca02c", # Green
41     "#9467bd", # Purple
42     "#17becf", # Cyan
43     "#8c564b", # Muted Purple
44     "#7f7f7f", # Grayish Blue
45     "#bcbd22", # Olive Green
46     "#93c572", # Pistachio Green
47     "#006400" # Deep Green
48 ]
49
50 plt.figure(figsize=(12, 8))
51 plt.gca().set_facecolor("#e6e6fa")
52 counter = 0
53 for technique in results_array.keys():
54     plt.plot(nthreads, results_array[technique], linewidth=2, marker = 'o',
55               color=colours[counter], label=technique)
56     counter += 1
57 plt.xlabel('Number of threads')
58 plt.xscale('log', base=2)
59 plt.ylabel('Total Time (secs)')
60 plt.title("Time vs #threads NUMA aware")
61 plt.grid(axis="y", linestyle="--", alpha=0.7)
62 plt.legend(loc='best')
63 plt.tight_layout()
64 plt.savefig("kmeans_results_hyper.png", dpi=300)
65 plt.close()
```

conc_plt.py

```
1 import re
2 from collections import defaultdict
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 outfile1 = "./conc_ll_1024.out"
7 outfile2 = "./conc_ll_8196.out"
8 size1 = 1024
9 size2 = 8192
10
11 throughputs = defaultdict(dict)
12
13 outfile=outfile2
14 size=size2
15
16 mech_regex = r"^(serial|coarse-grain|fine-grain|optimistic|lazy|non-blocking)$"
17 threads_regex = r"Nthreads:\s+(\d+)"
18 workload_regex = r"Workload:\s+(\d+/\d+/\d+)"
19 throughput_regex = r"Throughput\((Kops/sec\):\s+([\d.]+)\)"
20
21 locking_mech = None
22 workload = None
23
24 with open(outfile, 'r') as f:
25     lines = f.readlines()
26     for line in lines:
27         mech_match = re.match(mech_regex, line)
28         if mech_match:
29             locking_mech = mech_match.group(1)
30
31         workload_match = re.search(workload_regex, line)
32         if workload_match:
33             workload = workload_match.group(1)
34
35         threads_match = re.search(threads_regex, line)
36         throughput_match = re.search(throughput_regex, line)
37         if threads_match and throughput_match:
38             threads = int(threads_match.group(1))
39             throughput = float(throughput_match.group(1))
40
41             if locking_mech and workload:
42                 throughputs[(locking_mech, workload)][threads] = throughput
43
44 for key, value in throughputs.items():
45     mech, workload = key
46     print(f"Mechanism: {mech}, Workload: {workload}")
47     for threads, throughput in sorted(value.items()):
48         print(f" Threads: {threads}, Throughput: {throughput}")
49
50
51 mechanisms = sorted(set(pair[0] for pair in throughputs.keys()))
52 workloads = sorted(set(pair[1] for pair in throughputs.keys()))
53 threads = [1, 2, 4, 8, 16, 32, 64, 128]
54
55 #Line Plot settings
56 color_palette = ["#1f77b4", "#2ca02c", "#9467bd", "#17becf", "#bcbd22", "#8c564b"]
57
58 for workload in workloads:
59     plt.figure(figsize=(10, 7))
60
61     for idx, mech in enumerate(mechanisms):
62         throughput_values = [throughputs.get((mech, workload), {}).get(n, 0) for n in threads]
63         plt.plot(
64             threads, throughput_values,
65             marker="o", # Add markers for clarity
66             color=color_palette[idx % len(color_palette)], # Cycle colors
67             label=mech
68         )
69
70     plt.gca().set_facecolor("#e6e6fa")
71     plt.xscale("log") # Log scale for thread counts
72     plt.xticks(threads, [str(t) for t in threads])
73     plt.xlabel("Number of Threads", fontsize=12)
```

```

74     plt.ylabel("Throughput (Kops/sec)", fontsize=12)
75     plt.title(f"Throughput vs Threads for Workload Pattern:\n Search/Insert/Delete: {workload},
76 Size={size}", fontsize=14)
76     plt.grid(which="major", axis="y", linestyle="--", alpha=0.7)
77     plt.legend(title="Locking Mechanism", fontsize=10)
78
79     plt.tight_layout()
80     plt.savefig(f"line_{workload.replace('/', '_')}{size}.png", dpi=300)
81
82
83 # Bar Plot settings
84 bar_width = 0.2
85 x_indices = np.arange(len(mechanisms))
86
87 serial = [mech for mech in mechanisms if mech == "serial"]
88 non_serial_mechanisms = [mech for mech in mechanisms if mech != "serial"]
89
90
91 color_palette = ["#1f77b4", "#2ca02c", "#9467bd", "#17becf"] # Blue, Green, Purple, Cyan
92 serial_palette = ["#ff9999", "#ff6666", "#ff3333", "#cc0000"] # Shades of red for serial
93
94 for n in threads:
95     plt.figure(figsize=(12, 8))
96
97     # Plot serial bars first
98     if serial:
99         for idx, workload in enumerate(workloads):
100             serial_throughput = throughputs.get((serial[0], workload), {}).get(1, 0) # Serial
101             only at n=1
102             plt.bar(
103                 x_indices[0] + idx * bar_width,
104                 serial_throughput,
105                 width=bar_width,
106                 color=serial_palette[idx % len(serial_palette)],
107                 label=f"Serial: {workload}"
108             )
109
110     throughput_data = {
111         mech: [throughputs.get((mech, workload), {}).get(n, 0) for workload in workloads]
112         for mech in non_serial_mechanisms
113     }
114
115     for idx, workload in enumerate(workloads):
116         workload_throughput = [throughput_data[mech][idx] for mech in non_serial_mechanisms]
117         workload_norm = np.divide(workload_throughput, n)
118         plt.bar(
119             x_indices[1:] + idx * bar_width, # Offset bars for non-serial mechanisms
120             workload_norm,
121             width=bar_width,
122             color=color_palette[idx % len(color_palette)],
123             label=f"Search/Insert/Delete {workload}"
124         )
125
126     # Formatting the plot
127     plt.gca().set_facecolor("#e6e6fa")
128     plt.xticks(
129         x_indices + bar_width * (len(workloads) - 1) / 2,
130         ["Serial"] + non_serial_mechanisms, # Add "Serial" as the first x-axis label
131         rotation=45
132     )
133     plt.xlabel("Locking Mechanism", fontsize=12)
134     plt.ylabel("(Kops/sec) per thread", fontsize=12)
135     plt.title(f"Throughput Per Thread vs Locking Mechanism\n (Size={size}, Threads={n})",
136               fontsize=14)
137     plt.legend(title="Workload", fontsize=10)
138     plt.grid(axis="y", linestyle="--", alpha=0.7)
139
140     # Save the figure
141     plt.tight_layout()
142     plt.savefig(f"conc_{size}_{n}.png", dpi=300)

```

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Σκοπός αυτής της άσκησης είναι η υλοποίηση και η βελτιστοποίηση του αλγορίθμου Kmeans σε μια κάρτα γραφικών της Nvidia με την χρήση της Cuda. Αρχικά, υλοποιούμε μια naïve προσέγγιση και έπειτα αξιολογούμε και συγκρίνουμε άλλες υλοποιήσεις που αφορούν τεχνικές βελτιώσεις σε GPUs.

Naive

Αρχικά υπολογίζουμε το global id στην συνάρτηση `get_id()` ως εξής : $\text{thread_block_size} \times \text{block_id} + \text{local_thread_id}$.

Έπειτα, στη συνάρτηση `euclidean_distance()` υπολογίζεται η ευκλείδια απόσταση μεταξύ δύο σημείων στον n-διάστατο χώρο. Ο πίνακας συντεταγμένων των αντικειμένων καθώς και των clusters είναι μονοδιάστατος. Οπότε, η συντεταγμένη j του cluster i είναι η $\text{cluster}[i \times \text{numCoords} + j]$ καθώς ο δισδιάστατος πίνακας θα είχε διαστάσεις $[\text{numClusters}][\text{numCoords}]$, αντίστοιχα για τα objects. Με ένα for loop υπολογίζεται το άθροισμα όλων των διαφορών στο τετράγωνο. Δεν χρειάζεται να υπολογιστεί η ρίζα καθώς οι αποστάσεις είναι μη αρνητικές και η ύψωση στο τετράγωνο δεν αλλάζει την μεταξύ τους διάταξη.

Η συνάρτηση `find_nearest_cluster` χρειάζεται να υπολογιστεί μόνο για τα threads που έχουν global id μικρότερο από τον αριθμό των objects, ώστε να μην βρεθεί εκτός ορίων πίνακα και κάθε thread να κάνει έναν υπολογισμό για ένα αντικείμενο. Για να βρεθεί το κοντυνότερο cluster, υπολογίζεται αρχικά η απόσταση από το πρώτο cluster και έπειτα υπολογίζεται σειριακά η απόσταση για τα υπόλοιπα. Αν για κάποιο cluster, είναι μικρότερη από την ήδη υπάρχουσα, κρατάμε αυτό στη θέση του προηγούμενου. Ακόμη, η ανανέωση του delta κατά 1 αν διαφέρεια από το προηγούμενο κοντυνότερο cluster χρειάζεται να γίνει με atomicAdd ώστε να έχουμε σωστό συνολικό αποτέλεσμα καθώς είναι κοινό δεδομένο για όλα τα thread blocks και χρειάζεται κατάλληλος συγχρονισμός.

Ο αριθμός των thread blocks υπολογίζεται ως $\frac{\text{numObjects} + \text{blocksize} - 1}{\text{blocksize}}$. Αυτό υπολογίζει την πράξη ceil του λόγου του αριθμού των αντικειμένων προς το blocksize ώστε ακόμη και 1 παραπάνω thread να χρειάζεται να δημιουργηθεί καινούριο thread block.

Τέλος αντιγράφουμε τα clusters, memberships, delta χρησιμοποιώντας την εντολή `cudaMemcpy` με κατάλληλο μέγεθος και κατεύθυνση της αντιγραφής.

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 blockDim.x*blockIdx.x + threadIdx.x;
25     //return 0; /* TODO: Calculate 1-Dim global ID of a thread */
26 }
27
28 /* square of Euclid distance between two multi-dimensional points */
29 __host__ __device__ inline static
30 double euclid_dist_2(int numCoords,
31             int numObjs,
32             int numClusters,
33             double *objects,      // [numObjs][numCoords]
34             double *clusters,    // [numClusters][numCoords]
35             int objectId,
36             int clusterId) {
37     int i;
38     double ans = 0.0;
39
40     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
41     clusters*/
42     for(i=0; i<numCoords; i++)
43         ans += (objects[objectId*numCoords + i] - clusters[clusterId*numCoords+i]) *
44     (objects[objectId*numCoords + i] - clusters[clusterId*numCoords + i];
45     return (ans);
46 }
47
48 __global__ static
49 void find_nearest_cluster(int numCoords,
50             int numObjs,
51             int numClusters,
52             double *objects,      // [numObjs][numCoords]
53             double *deviceClusters, // [numClusters][numCoords]
54             int *deviceMembership, // [numObjs]
55             double *devdelta) {
56
56     /* Get the global ID of the thread. */
57     int tid = get_tid();
58
59     /* TODO: Maybe something is missing here... should all threads run this? */
60     if (tid < numObjs) {
61         int index, i;
62         double dist, min_dist;
63
64         /* find the cluster id that has min distance to object */
65         index = 0;
66         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
67         min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters, tid, 0);
68
69         for (i = 1; i < numClusters; i++) {
70             /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
71             dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters, tid, i);
72             /* no need square root */
73             if (dist < min_dist) { /* find the min and its array index */
74                 min_dist = dist;
75                 index = i;
76             }
77
78         if (deviceMembership[tid] != index) {
79             /* TODO: Maybe something is missing here... is this write safe? */
80             atomicAdd(devdelta, 1.0);
81         }
82
83         /* assign the deviceMembership to object objectId */
84         deviceMembership[tid] = index;
85     }
86 }
87
88 /**
89 // -----
90 // DATA LAYOUT
91 //
92 // objects      [numObjs][numCoords]
93 // clusters     [numClusters][numCoords]
```

```

94 // newClusters      [numClusters][numCoords]
95 // deviceObjects    [numObjs][numCoords]
96 // deviceClusters   [numClusters][numCoords]
97 // -----
98 //
99 /* return an array of cluster centers of size [numClusters][numCoords] */ 
100 void kmeans_gpu(double *objects,          /* in: [numObjs][numCoords] */
101                 int numCoords,        /* no. features */
102                 int numObjs,         /* no. objects */
103                 int numClusters,     /* no. clusters */
104                 double threshold,   /* % objects change membership */
105                 long loop_threshold,/* maximum number of iterations */
106                 int *membership,    /* out: [numObjs] */
107                 double *clusters,    /* out: [numClusters][numCoords] */
108                 int blockSize) {
109     double timing = wtime(), timing_internal, timer_min = le42, timer_max = 0;
110     double timing_gpu, timing_cpu, timing_transfers, transfers_time = 0.0, cpu_time = 0.0,
111     gpu_time = 0.0;
112     int loop_iterations = 0;
113     int i, j, index, loop = 0;
114     int *newClusterSize; /* [numClusters]: no. objects assigned in each
115                           new cluster */
116     double delta = 0, *dev_delta_ptr;           /* % of objects change their clusters */
117     double **newClusters = (double **) calloc_2d(numClusters, numCoords, sizeof(double));
118
119     double *deviceObjects;
120     double *deviceClusters;
121     int *deviceMembership;
122
123     printf("\n|-----Naive GPU Kmeans-----|\n\n");
124
125     /* initialize membership[] */
126     for (i = 0; i < numObjs; i++) membership[i] = -1;
127
128     /* need to initialize newClusterSize and newClusters[0] to all 0 */
129     newClusterSize = (int *) calloc(numClusters, sizeof(int));
130     assert(newClusterSize != NULL);
131
132     timing = wtime() - timing;
133     printf("t_alloc: %lf ms\n\n", 1000 * timing);
134     timing = wtime();
135
136     int minGridSize, bestblockSize;
137     cudaOccupancyMaxPotentialBlockSize(&minGridSize, &bestblockSize, find_nearest_cluster, 0,
138     0);
139     printf("Naive kmeans, min_grid_size = %d, best_block_size = %d\n\n", minGridSize,
140     bestblockSize);
141
142     //for the first exercise
143     //const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize) ? blockSize : numObjs;
144     //for the upgraded version to find best block size
145     //uncomment properly
146     const unsigned int numThreadsPerClusterBlock = bestblockSize;
147     const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
148     numThreadsPerClusterBlock; /* TODO: Calculate Grid size, e.g. number of blocks. */
149
150     const unsigned int clusterBlockSharedDataSize = 0;
151
152     checkCuda(cudaMalloc(&deviceObjects, numObjs * numCoords * sizeof(double)));
153     checkCuda(cudaMalloc(&deviceClusters, numClusters * numCoords * sizeof(double)));
154     checkCuda(cudaMalloc(&deviceMembership, numObjs * sizeof(int)));
155     checkCuda(cudaMalloc(&dev_delta_ptr, sizeof(double)));
156
157     timing = wtime() - timing;
158     printf("t_alloc_gpu: %lf ms\n\n", 1000 * timing);
159     timing = wtime();
160
161     checkCuda(cudaMemcpy(deviceObjects, objects,
162                         numObjs * numCoords * sizeof(double), cudaMemcpyHostToDevice));
163     checkCuda(cudaMemcpy(deviceMembership, membership,
164                         numObjs * sizeof(int), cudaMemcpyHostToDevice));
165     timing = wtime() - timing;
166     printf("t_get_gpu: %lf ms\n\n", 1000 * timing);
167     timing = wtime();

```

```

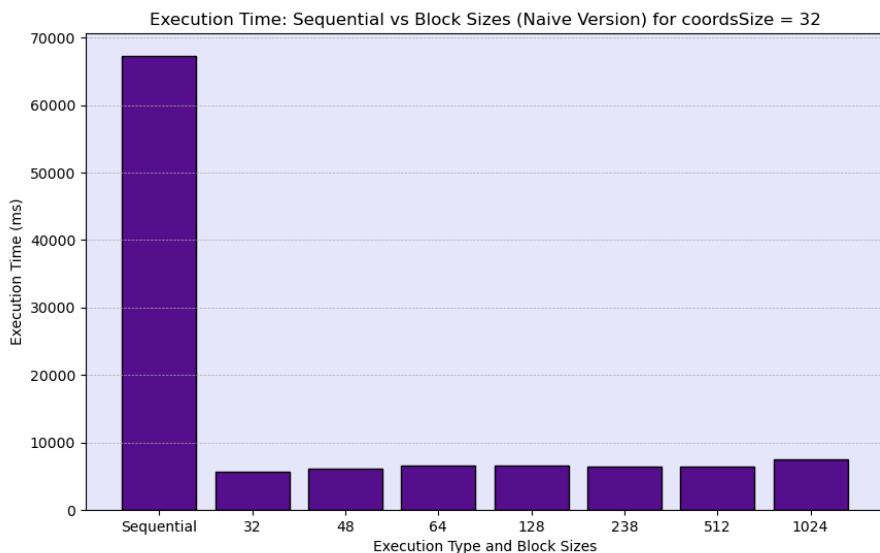
164
165     do {
166         timing_internal = wtime();
167
168         /* GPU part: calculate new memberships */
169
170         timing_transfers = wtime();
171         /* TODO: Copy clusters to deviceClusters*/
172         checkCuda(cudaMemcpy(deviceClusters, clusters, numClusters * numCoords * sizeof(double),
173             cudaMemcpyHostToDevice));
174         transfers_time += wtime() - timing_transfers;
175
176         checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
177
178         //printf("Launching find_nearest_cluster Kernel with grid_size = %d,
179         block_size = %d, shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock,
180         clusterBlockSharedDataSize/1000);
181         timing_gpu = wtime();
182         find_nearest_cluster
183         <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
184         (numCoords, numObjs, numClusters,
185          deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
186
187         cudaDeviceSynchronize();
188         checkLastCudaError();
189         gpu_time += wtime() - timing_gpu;
190         //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
191
192         timing_transfers = wtime();
193         /* TODO: Copy deviceMembership to membership */
194         checkCuda(cudaMemcpy(membership, deviceMembership, numObjs * sizeof(int),
195             cudaMemcpyDeviceToHost));
196
197         /* TODO: Copy dev_delta_ptr to &delta */
198         checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
199         transfers_time += wtime() - timing_transfers;
200
201         /* CPU part: Update cluster centers*/
202         timing_cpu = wtime();
203         for (i = 0; i < numObjs; i++) {
204             /* find the array index of nestest cluster center */
205             index = membership[i];
206
207             /* update new cluster centers : sum of objects located within */
208             newClusterSize[index]++;
209             for (j = 0; j < numCoords; j++) {
210                 newClusters[index][j] += objects[i * numCoords + j];
211             }
212
213             /* average the sum and replace old cluster centers with newClusters */
214             for (i = 0; i < numClusters; i++) {
215                 for (j = 0; j < numCoords; j++) {
216                     if (newClusterSize[i] > 0)
217                         clusters[i * numCoords + j] = newClusters[i][j] / newClusterSize[i];
218                     newClusters[i][j] = 0.0; /* set back to 0 */
219                 }
220                 newClusterSize[i] = 0; /* set back to 0 */
221             }
222
223             delta /= numObjs;
224             //printf("delta is %f - ", delta);
225             loop++;
226             //printf("completed loop %d\n", loop);
227             cpu_time += wtime() - timing_cpu;
228
229             timing_internal = wtime() - timing_internal;
230             if (timing_internal < timer_min) timer_min = timing_internal;
231             if (timing_internal > timer_max) timer_max = timing_internal;
232         } while (delta > threshold && loop < loop_threshold);
233
234         timing = wtime() - timing;
235         printf("nloops = %d : total = %lf ms\n\t-> t_loop_avg = %lf ms\n\t-> t_loop_min = %lf
236         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
237         ms\n\n-----|\n",
238

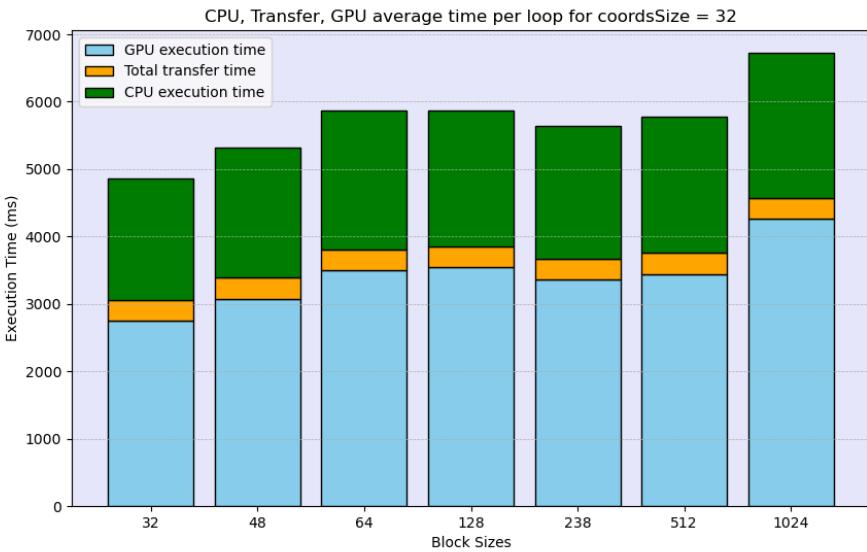
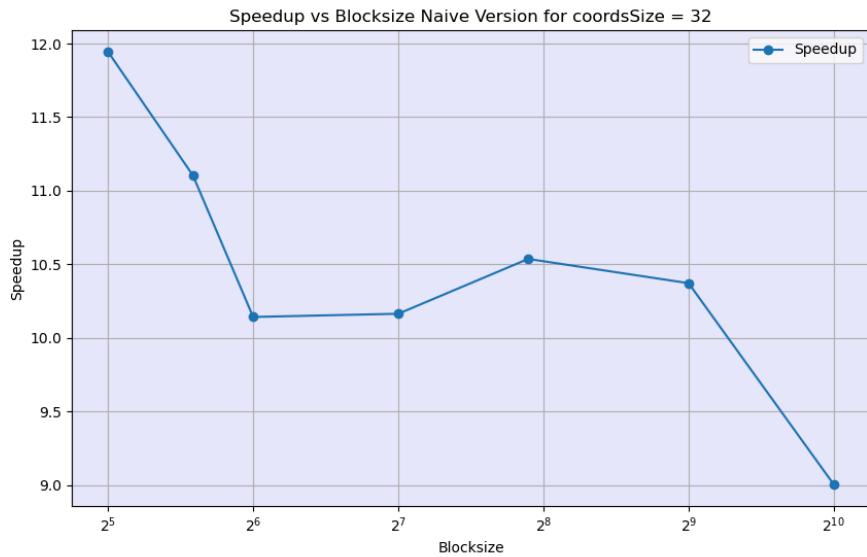
```

```

233     loop, 1000 * timing, 1000 * timing / loop, 1000 * timer_min, 1000 * timer_max,
234     1000 * cpu_time / loop, 1000 * gpu_time / loop, 1000 * transfers_time / loop);
235
236     char outfile_name[1024] = {0};
237     sprintf(outfile_name, "Execution_logs/silver1-V100_Sz-%lu_Coo-%d_Cl-%d.csv",
238             numObjs * numCoords * sizeof(double) / (1024 * 1024), numCoords, numClusters);
239     FILE *fp = fopen(outfile_name, "a+");
240     if (!fp) error("Filename %s did not open successfully, no logging performed\n", outfile_name);
241     fprintf(fp, "%s,%d,%lf,%lf\n", "Naive", blockSize, timing / loop, timer_min, timer_max);
242     fclose(fp);
243     checkCuda(cudaFree(deviceObjects));
244     checkCuda(cudaFree(deviceClusters));
245     checkCuda(cudaFree(deviceMembership));
246
247     free(newClusters[0]);
248     free(newClusters);
249     free(newClusterSize);
250
251     return;
252 }
```

Μετρήθηκαν οι επιδόσεις της σειριακής και της naive έκδοσης για τα διάφορα blocksizes για {Size, Coords, Clusters, Loops} = {1024, 32, 64, 10}. Με βάση αυτές, προκύπτουν τα παρακάτω διαγράμματα χρόνου εκτέλεσης, speedup, χρόνου gpu-cpu-transfer αντίστοιχα.





To speedup του kmeans για την naive έκδοση κυμαίνεται από 9 έως 12 ανάλογα το blocksize. Αυτή η επίδοση είναι αρκετά καλή για την πιο απλή υλοποίηση σε GPU. Ο αλγόριθμος kmeans όμως δεν είναι ιδανικός για GPU καθώς κάνει αρκετά transfers και allocations. Επίσης το operational intensity δεν είναι το μέγιστο που μπορεί να υποστηρίξει η συγκεκριμένη GPU καθώς δεν κάνει 6000 πράξεις ανά δεδομένο ώστε να αξιζει πλήρως η μεταφορά του.

Οι χρόνοι μεταφοράς και της CPU είναι προφανώς ίδιοι, οπότε οι αλλαγές του χρόνου της GPU μεταφράζονται άμεσα σε speedup. Το blocksize 32 έχει το καλύτερο, διότι έχει την μεγαλύτερη ευελιξία για το scheduling και κάθε thread block είναι ένα warp. Αφού τα objects είναι τελείως ανεξάρτητα μεταξύ τους για τον υπολογισμό του κοντινότερου cluster, χρησιμοποιούνται στο μέγιστο οι πόροι της gpu. Το blocksize 1024 έχει την μικρότερη ευελιξία και κάποιοι πόροι δεν αξιοποιούνται στο μέγιστο. Τέλος, το blocksize 238 παρόλο που δεν είναι πολλαπλάσιο του 32 και δεν αξιοποιεί όλα τα warps στο μέγιστο, αφήνονται αναξιόπιστα μόνο 18 threads οπότε η διαφορά στο σύνολο δεν είναι τόσο μεγάλη και γι' αυτό παρατηρείται μια καλή επίδοση.

Transpose

Οι μονοδιάστατοι πίνακες της GPU είναι πλέον column-based και όχι row-based όπως πριν. Οπότε, ο πίνακας clusters έχει διαστάσεις [numCoords][numClusters]. Η συντεταγμένη j του cluster i είναι η cluster[j × numClusters + i], καθώς πλέον ο πίνακας έχει για κάθε συντεταγμένη μία γραμμή από την τιμή της για κάθε cluster.

```
transpose_euclid_dist.cu

1  __host__ __device__ inline static
2  double euclid_dist_2 transpose(int numCoords,
3      int numObjs,
4      int numClusters,
5      double *objects,           // [numCoords][numObjs]
6      double *clusters,         // [numCoords][numClusters]
7      int objectId,
8      int clusterId) {
9
10     int i;
11     double ans = 0.0, diff;
12
13     /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
14      clusters, but for column-base format!!! */
15
16     for(i = 0; i < numCoords; i++) {
17         diff = objects[i*numObjs+ objectId] - clusters[i*numClusters + clusterId];
18         ans += diff * diff;
19     }
20
21     return (ans);
22 }
```

Η `find_nearest_cluster` μένει απαράλλακτη καθώς αλλάζει μόνο η δομή των δεδομένων. Οι πίνακες `dimObjects`, `dimClusters`, `newClusters` έχουν `numCoords` γραμμές ώστε να είναι column-based και γίνεται κατάλληλο allocation με την `calloc_2d` που μας παρέχεται.

Πριν την εκτέλεση του αλγορίθμου χρειάζεται αντιγραφή των αντικειμένων σε μορφή column-based, οπότε το $\text{dimObjects}[j][i] = \text{objects}[i][j]$. Ο πίνακας objects όμως είναι μονοδιάστατος, ακολουθώντας την ίδια λογική μετατροπής $\text{objects}[i][j] = \text{objects}[i \times \text{numCoords} + j]$ καθώς numCoords γραμμές θα είχε ο αντίστοιχος δισδιάστατος πίνακας.

```
transpose_allocation.cu

1  /* TODO: Transpose dims */
2  double **dimObjects = (double**) calloc_2d(numCoords, numObjs, sizeof(double)); // 
3  calloc_2d(...) -> [numCoords][numObjs]
4  double **dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double)); //
5  calloc_2d(...) -> [numCoords][numClusters]
6  double **newClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double)); //
7  calloc_2d(...) -> [numCoords][numClusters]

8  double *deviceObjects;
9  double *deviceClusters;
10 int *deviceMembership;

11 printf("\n|-----Transpose GPU Kmeans-----|\n\n");
12
13 // TODO: Copy objects given in [numObjs][numCoords] layout to new
14 // [numCoords][numObjs] layout
15 for (i = 0; i < numObjs; i++) {
16     for (j = 0; j < numCoords; j++) {
17         dimObjects[j][i] = objects[i*numCoords + j];
18     }
19 }
```

Ο αριθμός των thread blocks δεν αλλάζει καθώς και οι αντιγραφές πίσω στην CPU μετά τους υπολο-

γισμούς της GPU. Αλλάζει όμως η αντιγραφή των clusters προς την GPU σε dimClusters καθώς αυτά είναι τα column-based δεδομένα. Ακόμη, αφού ολοκληρωθεί η εκτέλεση του αλγορίθμου χρειάζεται οι συντεταγμένες των τελικών clusters να ξαναγίνουν row-based και κάνουμε την ανάποδη διαδικασία $clusters[i][j] = clusters[i \times numCoords + j] = dimClusters[j][i]$, το οποίο είναι ίδιο με την έκφραση στην αρχή του transpose με κατάλληλη αλλαγή δεικτών. Αναλυτικότερα, για πίνακα m γραμμών και n στηλών το στοιχείο i,j είναι το $i \times n + j$ ή αλλιώς το $j \times m + i$.

transpose_do_while.cu

```

1  do {
2      timing_internal = wtime();
3
4      /* GPU part: calculate new memberships */
5
6      timing_transfers = wtime();
7      /* TODO: Copy clusters to deviceClusters
8      checkCuda(cudaMemcpy(...));
9      checkCuda(cudaMemcpy(deviceClusters, dimClusters[0], numClusters * numCoords *
10         sizeof(double), cudaMemcpyHostToDevice));
11     transfers_time += wtime() - timing_transfers;
12
13     checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
14
15     //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size
16     = %d, shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock,
17     clusterBlockSharedDataSize/1000);
18     timing_gpu = wtime();
19     find_nearest_cluster
20     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
21     (numCoords, numObjs, numClusters,
22     deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
23
24     cudaDeviceSynchronize();
25     checkLastCudaError();
26     gpu_time += wtime() - timing_gpu;
27     //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
28
29     timing_transfers = wtime();
30     /* TODO: Copy deviceMembership to membership
31     checkCuda(cudaMemcpy(...));
32     checkCuda(cudaMemcpy(membership, deviceMembership, numObjs * sizeof(int),
33     cudaMemcpyDeviceToHost));
34     transfers_time += wtime() - timing_transfers;
35
36     /* CPU part: Update cluster centers*/
37
38     timing_cpu = wtime();
39     for (i = 0; i < numObjs; i++) {
40         /* find the array index of nestest cluster center */
41         index = membership[i];
42
43         /* update new cluster centers : sum of objects located within */
44         newClusterSize[index]++;
45         for (j = 0; j < numCoords; j++)
46             newClusters[j][index] += objects[i * numCoords + j];
47     }
48
49     /* average the sum and replace old cluster centers with newClusters */
50     for (i = 0; i < numClusters; i++) {
51         for (j = 0; j < numCoords; j++) {
52             if (newClusterSize[i] > 0)
53                 dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
54             newClusters[j][i] = 0.0; /* set back to 0 */
55         }
56         newClusterSize[i] = 0; /* set back to 0 */
57     }
58
59     delta /= numObjs;

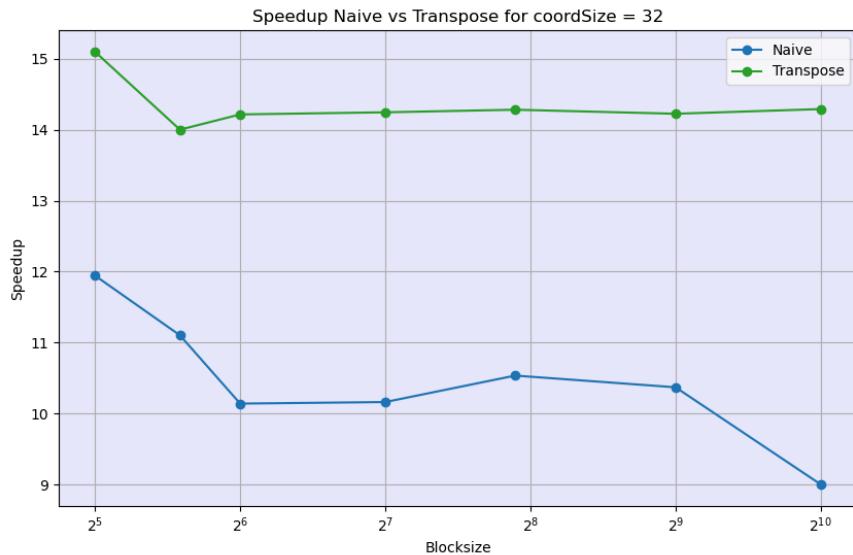
```

```

60     //printf("delta is %f - ", delta);
61     loop++;
62     //printf("completed loop %d\n", loop);
63     cpu_time += wtime() - timing_cpu;
64
65     timing_internal = wtime() - timing_internal;
66     if (timing_internal < timer_min) timer_min = timing_internal;
67     if (timing_internal > timer_max) timer_max = timing_internal;
68 } while (delta > threshold && loop < loop_threshold);
69
70 /*TODO: Update clusters using dimClusters. Be carefull of layout!!! clusters[numClusters]
71 [numCoords] vs dimClusters[numCoords][numClusters] */
72 for (i = 0; i < numClusters; i++) {
73     for (j = 0; j < numCoords; j++) {
74         clusters[i*numCoords + j] = dimClusters[j][i];
75     }

```

Επαναλάβαμε τις μετρήσεις για την transpose εκδοχή και παρακάτω παρουσιάζεται το διάγραμμα speedup σε σύγκριση με την naive εκδοχή.



Παρατηρούμε πως το blocksize παίζει τελείως διαφορετικό ρόλο σε σχέση με την naive περίπτωση. Σε κάθε γραμμή βρίσκεται η τιμή της ίδιας συντεταγμένης για όλα τα objects, clusters. Οπότε, όταν ένα thread κάνει access μια συγκεκριμένη συντεταγμένη θα έρθουν στην cache οι τιμές της αντίστοιχης συντεταγμένης για επόμενα objects, clusters. Καθώς τα warps προχωράνε στις συντεταγμένες στην GPU θα υπάρχουν στην L1 cache στοιχεία που θα χρησιμοποιήσουν τα επόμενα warps ή επόμενα thread του ίδιου warp. Έτσι, καλύπτεται το global memory latency αφού περιμένουν πολύ λιγότερα warps στοιχεία για να κάνουν τους υπολογισμούς τους. Όταν έρχεται 1 cache line, θα καλύψει σύγουρα όσα threads υπολογίζουν συντεταγμένες που αντιστοιχούν σε αυτήν την cache line. Αυτό οδηγεί σε πολύ καλύτερη καλύψη των κενών χρόνων μεταξύ των warps και thread blocks, οπότε τα μεγάλα blocksizes έχουν σχεδόν όλα την ίδια επίδοση. Το blocksize 32 συνεχίζει να έχει αισθητά καλύτερα επίδοση λόγω ευελιξίας, όπως και στην naive εκδοχή.

Shared

Χρειάζεται να επεκτείνουμε την `find_nearest_cluster` κατάλληλα. Η κοινή μνήμη έχει εμβέλεια στο thread block και είναι πολύ πιο γρήγορη από την global μνήμη. Γι' αυτό μπορούν να αποθηκευτούν σε αυτήν οι συντεταγμένες των clusters που τις χρειάζεται όλο το thread block ώστε να έχει γρήγορη πρόσβαση σε αυτές.

Το πρώτο βήμα είναι να αντιγραφούν οι συντεταγμένες των clusters στην shared memory για κάθε thread block. Τα clusters είναι 64 και το ελάχιστο blocksize είναι 32 οπότε σε αυτήν την περίπτωση θα πρέπει κάθε thread μέσα στο thread block να αντιγράψει 2 ολόκληρα clusters. Γενικότερα όμως είναι καλή πρακτική για να τρέχει για οποιδήποτε μέγεθος blocksize, αριθμό cluster να θεωρούμε πως κάθε thread έχει υπό την υπόλοιψή του παραπάνω από 1 cluster.

Για να διαχωριστούν σωστά και ισάξια τα clusters κάθε thread ξεκινάει από το local id του και προχωράει με βήμα όσο το thread block μέχρι να τελειώσει ο αριθμός των clusters. Υπενθυμίζεται πως τα clusters είναι column-based όπως και στην transpose εκδοχή. Οι υπολογισμοί των κοντινότερων clusters πρέπει να ξεκινήσουν αφού αντιγραφούν όλα τα clusters μέσα στο thread block. Οπότε, χρειάζεται ένας συγχρονισμός των νημάτων για να είναι σίγουρο πως θα έχει γίνει αυτό.

Τέλος, για να αξιοποιηθούν σωστά, στην θέση των `deviceClusters` μπαίνει ο πίνακας της διαμοιραζόμενης μνήμης που μόλις γεμίσαμε σαν παράμετρος στην κλήση υπολογισμού της ευκλείδιας απόστασης.

`shared_find_nearest_cluster.cu`

```
1  __global__ static
2  void find_nearest_cluster(int numCoords,
3                           int numObjs,
4                           int numClusters,
5                           double *objects,
6                           double *deviceClusters,    // [numCoords] [numObjs]
7                           int *deviceMembership,     // [numClusters] [numObjs]
8                           double *devdelta) {
9      extern __shared__ double shmemClusters[];
10
11     /* TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
12        BEWARE: Make sure operations is complete before any thread continues... */
13     int no_cluster, i;
14
15     //use local_id because shared memory is per thread block
16     for (no_cluster = threadIdx.x; no_cluster < numClusters; no_cluster+=blockDim.x) {
17         for (i = 0; i < numCoords; i++) {
18             shmemClusters[i * numClusters + no_cluster] = deviceClusters[i * numClusters + no_cluster];
19         }
20     }
21     __syncthreads();
22
23     /* Get the global ID of the thread. */
24     int tid = get_tid();
25
26     /* TODO: Maybe something is missing here... should all threads run this? */
27     if (tid < numObjs) {
28         int index;
29         double dist, min_dist;
30
31         /* find the cluster id that has min distance to object */
32         index = 0;
33         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using clusters
34         in shmem*/
35         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters,
36         tid, 0);
37         for (i = 1; i < numClusters; i++) {
38             /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId using clusters
39             in shmem*/
40             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters,
41             tid, i);
```

```

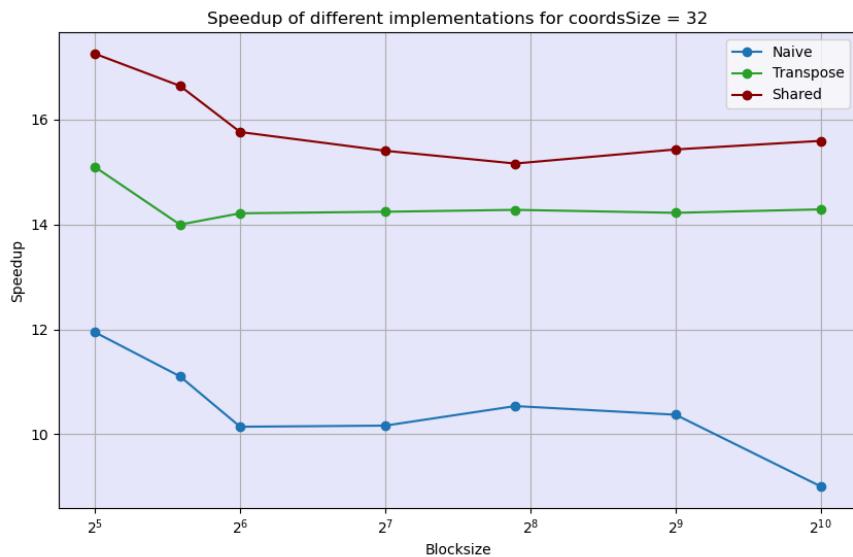
38
39     /* no need square root */
40     if (dist < min_dist) { /* find the min and its array index */
41         min_dist = dist;
42         index = i;
43     }
44 }
45
46 if (deviceMembership[tid] != index) {
47     /* TODO: Maybe something is missing here... is this write safe? */
48     atomicAdd(devdelta, 1.0);
49 }
50
51 /* assign the deviceMembership to object objectId */
52 deviceMembership[tid] = index;
53
54 }
55 }
```

Το μέγεθος της διαμοιραζόμενης μνήμης χρειάζεται να δηλωθεί στην κλήση του gpu kernel. Η διαμοιραζόμενη μνήμη θα έχει $\text{numClusters} \times \text{numCoords}$ πραγματικούς αριθμούς. Οπότε:

`const unsigned int clusterBlockSharedDataSize = numClusters * numCoords * sizeof(double);`

Οι υπόλοιπες διαδικασίες αντιγραφής και μετατροπής των clusters παραμένουν ίδιες με την transpose εκδοχή.

Επαναλάβαμε τις μετρήσεις για την shared εκδοχή και παρακάτω παρουσιάζεται το διάγραμμα speedup σε σύγκριση με τις υπόλοιπες εκδοχές.

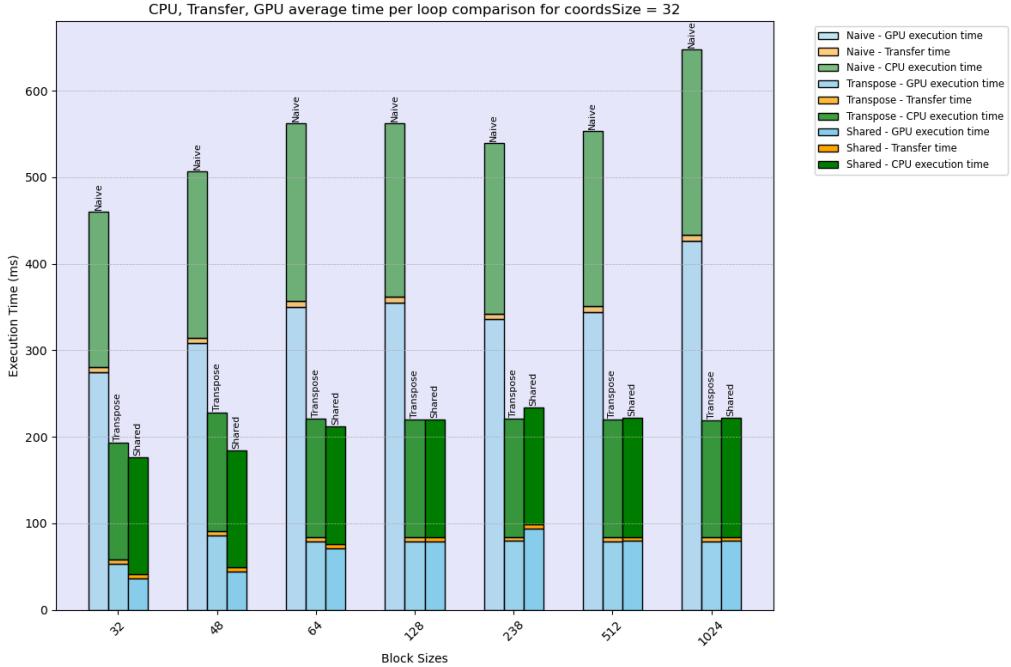


Παρατηρείται μια πτώση επίδοσης για blocksize > 64 και προφανώς επειδή δεν είναι πολλαπλάσιο του 32 και σπαταλάει πόρους το blocksize 238 έχει την χειρότερη επίδοση. Υπάρχουν 3 πιθανές εξηγήσεις για αυτό:

- 1)Καθυστέρηση λόγω δυσκολίας συγχρονισμού πολλών threads μέσα στο thread block
- 2)Η διαμοιραζόμενη μνήμη δέχεται υπερβολικά κοινά αιτήματα από τα threads
- 3)Με μεγάλα blocksizes υπάρχουν λίγα thread blocks ανά SM, δεν είναι τόσο ισομερώς κατανημεμένα, οπότε δεν γίνεται η κολύτερη αξιοποίηση των πόρων.

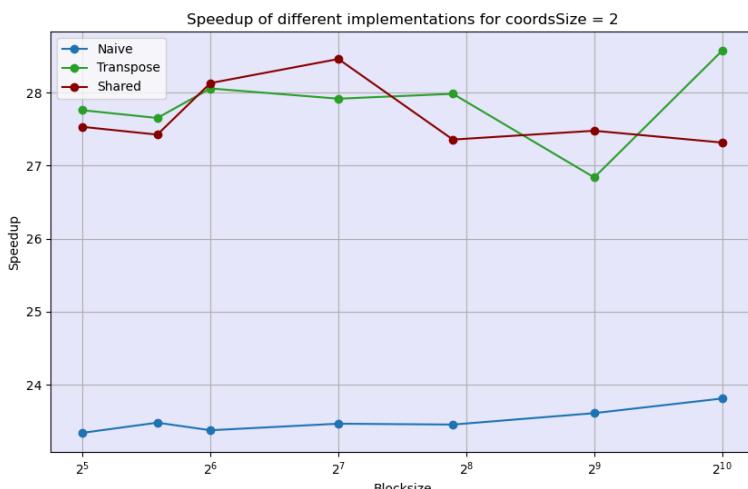
Σύγκριση υλοποιήσεων / bottleneck Analysis

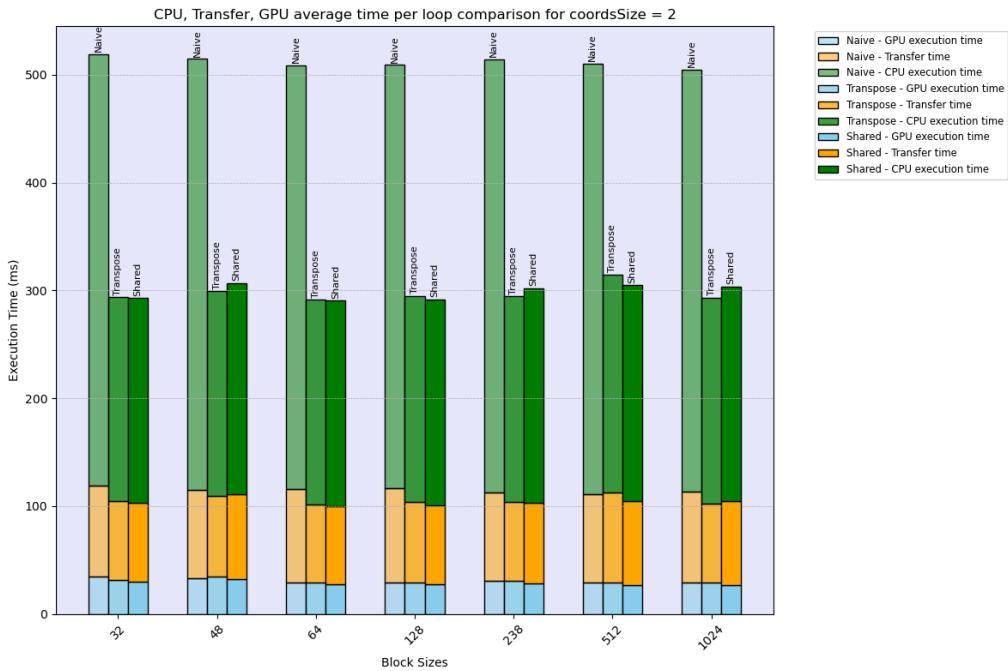
Για καλύτερη ανάλυση των επιμέρους υλοποιήσεων ακολουθεί ένα διάγραμμα ανάλυσης των επιμέρους χρόνων CPU, GPU, transfers ανά επανάληψη.



Όπως είναι λογικό οι χρόνοι μεταφορών και της CPU είναι σχεδόν ίδιοι μεταξύ των υλοποιήσεων. Παρατηρείται πως στα blocksizes με τις καλύτερες επιδόσεις (32, 64), ο χρόνος εκτέλεσης στην GPU είναι σημαντικά μικρότερος από το χρόνο εκτέλεσης στην CPU για τον υπολογισμό των καινούριων clusters. Οπότε, μια ιδέα θα ήταν να μεταφερθεί όλος ο φόρτος των επαναλήψεων στην GPU ακόμη και να μην είναι ιδανικό για GPU, όπως θα δούμε και παρακάτω.

Επαναλάβαμε τις μετρήσεις για όλες τις εκδοχές για $\{Size, Coords, Clusters, Loops\} = \{1024, 2, 64, 10\}$ για τα πιθανά blocksizes. Το μέγεθος του προβλήματος παραμένει το ίδιο αλλά μειώθηκε ο αριθμός των συντεταγμένων από 32 σε 2 οπότε αυξήθηκε αντίστοιχα ο αριθμός των αντικειμένων, άρα και των thread blocks. Ακολουθούν τα διαγράμματα για speedup και ανάλυση χρόνου επανάληψης για το καινούριο configuration.





Καθώς οι συντεταγμένες είναι μόνο 2, ένα cache line περιέχει παραπάνω από 1 συντεταγμένη για όλα τα cluster. Σε αυτήν την περίπτωση υπάρχει παρόμοιο locality και χωρίς διαμοιραζόμενη μνήμη, οπότε οι transpose, shared διαφέρουν ελάχιστα μεταξύ τους στις επιδόσεις. Η shared έχει χειρότερη επίδοση σε μεγάλα blocksizes καθώς η ζήτηση για τα cache lines που πλέον είναι λιγότερα αυξάνεται σημαντικά με αποτέλεσμα η μνήμη να μην μπορεί να καλύψει την ζήτηση με την ίδια ταχύτητα. Η παρούσα shared υλοποίηση δεν είναι κατάλληλη για την επίλυση του kmeans για arbitrary configurations. Για να λειτουργήσει σωστά η ιδέα της διαμοιραζόμενης μνήμης δεν πρέπει τα cache lines να περιέχουν υπερβολικά μικρά δεδομένα όπως είδαμε και στο copied clusters με first-touch policy και την βελτίωση με το num aware.

Bonus1: σε όλες τις περιπτώσεις (και στις επόμενες υλοποιήσεις) η cudaOccupancyMaxPotentialBlockSize επιστρέφει 1024 που είναι το μέγιστο blocksize. Για το configuration με τις πολλές συντεταγμένες (32) αυτή είναι ίσως η χειρότερη επιλογή. Ενώ για τις λίγες συντεταγμένες (2) το 1024 είναι το κατάλληλο blocksize μόνο για την transpose εκδοχή και για τις υπόλοιπες ή έχει παρόμοια επίδοση με άλλα blocksizes ή χειρότερη. Επειδή η συνάρτηση δεν λαμβάνει υπόψιν το μέγεθος του προβλήματος, δεν δύναται να δώσει κατάλληλο blocksize για όλες τις λύσεις. Δυστυχώς όμως, δεν δίνει κατάλληλο blocksize για κανένα από τα 2 configurations.

Full-offload (All-GPU)

Υπάρχουν πολλοί τρόποι να γίνει η υλοποίηση της update_centroid. Επιλέξαμε να κάνουμε την πρόσθεση των συντεταγμένων για τα καινούρια clusters στην find_nearest_cluster με atomicAdds ώστε να αξιοποιηθεί ο ισομερισμός της συνολικής δουλειάς. Επειδή ο αριθμός των clusters × τον αριθμό των συντεταγμένων είναι επαρκώς μεγάλος για τις περισσότερες περιπτώσεις, τα collisions που θα χρειαστούν όντως συγχρονισμό είναι πολύ λιγότερα απ' όσα φαίνονται αρχικά. Ακόμη, ενημερώνεται με atomicAdd και το μέγεθος του cluster που είναι το πιο κοντινό σε κάθε σημείο.

Στην update centroids κάθε thread αναλαμβάνει μία μόνο συντεταγμένη ενός νέου cluster και την διαιρεί με το μέγεθός του. Έπειτα, μηδενίζει την αντίστοιχη συντεταγμένη των υπολογισμένων από την find_nearest_cluster clusters ώστε να μπορεί να ξαναξεκινήσει επανάληψη το do-while από την αρχή σωστά. Χρειάζεται όμως να μηδενιστούν και τα μεγέθη αυτών των clusters. Αυτό μπορεί να γίνει μόνο αφού διαιρεθούν όλες οι συντεταγμένες, οπότε χρειάζεται συγχρονισμός των νημάτων και στο τέλος να μηδενιστούν τα μεγέθη. Σημειώνεται πως επειδή η δομή είναι do-while και όχι απλό while χρειάζεται με την cudaMemcpy να μηδενιστούν αρχικά τα devicenewClusters ώστε οι προσθέσεις να είναι valid στην πρώτη κλήση του kernel find_nearest_cluster.

all_gpu_calculations.cu

```
1  __global__ static
2  void find_nearest_cluster(int numCoords,
3                               int numObjs,
4                               int numClusters,
5                               double *deviceobjects,           // [numCoords][numObjs]
6  /*
7               TODO: If you choose to do (some of) the new centroid calculation
8  here, you will need some extra parameters here (from "update_centroids").
9  */
10                         int *devicenewClusterSize,          // [numClusters]
11                         double *devicenewClusters,        // [numCoords][numClusters]
12                         double *deviceClusters,          // [numCoords][numClusters]
13                         int *deviceMembership,          // [numObjs]
14                         double *devdelta) {
15     extern __shared__ double shmemClusters[];
16
17     /* TODO: copy me from shared version... */
18     int no_cluster, i;
19
20     //use local_id because shared memory is per thread block
21     for (no_cluster = threadIdx.x; no_cluster < numClusters; no_cluster+=blockDim.x) {
22         for (i = 0; i < numCoords; i++) {
23             shmemClusters[i * numClusters + no_cluster] = deviceClusters[i * numClusters + no_cluster];
24         }
25     }
26     __syncthreads();
27
28     /* Get the global ID of the thread. */
29     int tid = get_tid();
30
31     /* TODO: copy me from shared version... */
32     if (tid < numObjs) {
33         int index;
34         double dist, min_dist;
35
36         /* find the cluster id that has min distance to object */
37         index = 0;
38         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using clusters
39         in shmem*/
40         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
41         shmemClusters, tid, 0);
42         for (i = 1; i < numClusters; i++) {
43             /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId using clusters
44             in shmem*/
45             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
46             shmemClusters, tid, i);
```

```

42     /* no need square root */
43     if (dist < min_dist) { /* find the min and its array index */
44         min_dist = dist;
45         index = i;
46     }
47 }
48
49 if (deviceMembership[tid] != index) {
50     /* TODO: Maybe something is missing here... is this write safe? */
51     atomicAdd(devdelta, 1.0);
52 }
53
54 /* assign the deviceMembership to object objectId */
55 deviceMembership[tid] = index;
56
57 /* TODO: additional steps for calculating new centroids in GPU? */
58 //we chose to update the size and do the add here
59 //the division and the actual new Coords will be in update centroids
60 atomicAdd(&devicenewClusterSize[index], 1);
61 for (i = 0; i < numCoords; i++)
62     atomicAdd(&devicenewClusters[i * numClusters + index], deviceobjects[i * numObjs + tid]);
63
64 }
65 }
66
67 __global__ static
68 void update_centroids(int numCoords,
69                     int numClusters,
70                     int *devicenewClusterSize,           // [numClusters]
71                     double *devicenewClusters,        // [numCoords][numClusters]
72                     double *deviceClusters)         // [numCoords][numClusters])
73 {
74
75     /* TODO: additional steps for calculating new centroids in GPU? */
76     int tid = get_tid();
77
78     if (tid < numCoords * numClusters) {
79         /*run through all the elements, just divide by the size of the clusters
80         indexing of the 1d colummn based devicenewClusters is i*numClusters + j
81         so the index of the current cluster is the j, and i the Coords
82         so the index of the current clusters is (i*numClusters + j) % numClusters
83         here the tid runs all the array increasingly so it is i*numClusters + j
84         */
85         deviceClusters[tid] = devicenewClusters[tid] / devicenewClusterSize[tid % numClusters];
86         //reset devicenewClusters after updating deviceClusters
87         devicenewClusters[tid] = 0.0;
88     }
89     __syncthreads();
90     //reset devicenewClusterSize as well
91     if (tid < numClusters) {
92         devicenewClusterSize[tid] = 0;
93     }
94 }
95 }
```

To kernel `find_nearest_cluster` χρειάζεται να κληθεί με περισσότερες παραμέτρους και ίδιο μέγεθος shared memory, ενώ η κλήση της `update_centroid` δεν χρειάζεται καθόλου shared memory.

all_gpu_do_while.cu

```

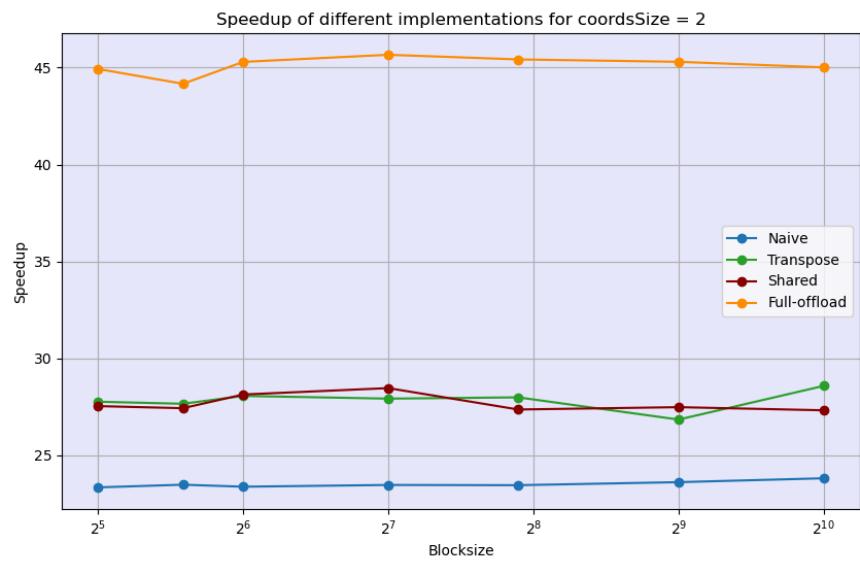
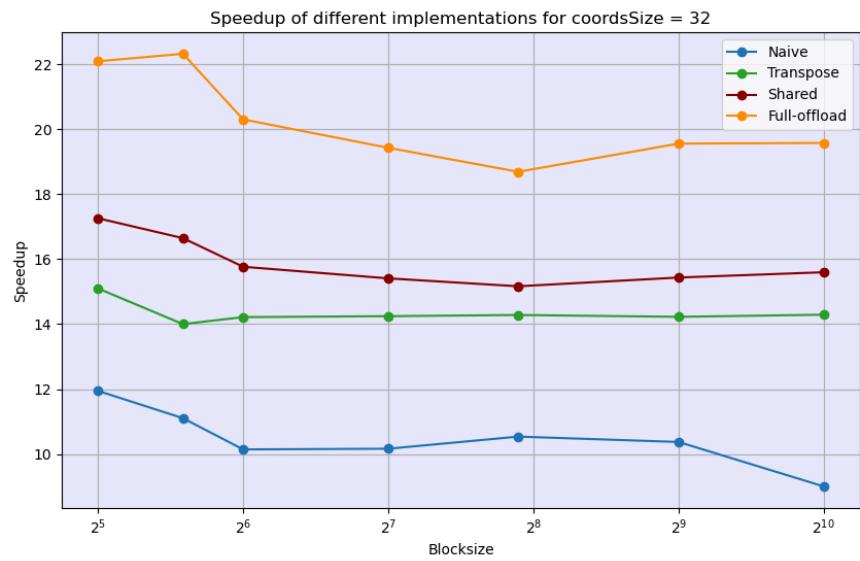
1 do {
2     timing_internal = wtime();
3     checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
4     timing_gpu = wtime();
5     //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size
6     = %d, shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock,
7     clusterBlockSharedDataSize/1000);
8     /* TODO: change invocation if extra parameters needed
9     find_nearest_cluster
10     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
11     (numCoords, numObjs, numClusters,
```

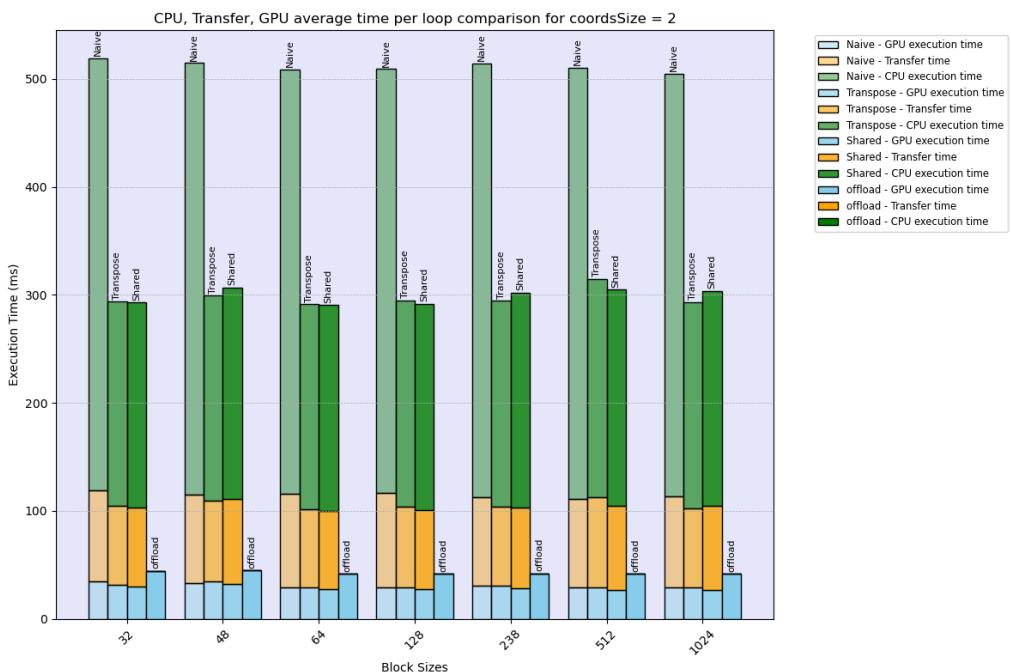
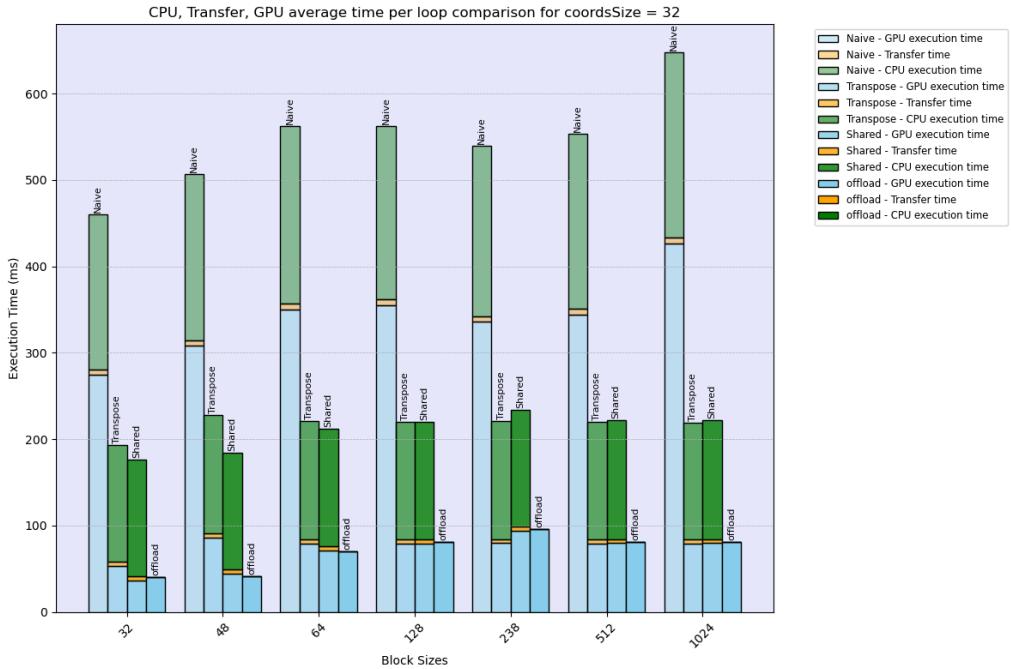
```

10     deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters,
11     deviceMembership, dev_delta_ptr);
12     */
13     find_nearest_cluster<<< numClusterBlocks, numThreadsPerClusterBlock,
14     clusterBlockSharedDataSize >>>
15     (numCoords, numObjs, numClusters,
16     deviceObjects, devicenewClusterSize, devicenewClusters, deviceClusters,
17     deviceMembership, dev_delta_ptr);
18
19     cudaDeviceSynchronize();
20     checkLastCudaError();
21
22     gpu_time += wtime() - timing_gpu;
23
24     //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
25
26     timing_transfers = wtime();
27     /* TODO: Copy dev_delta_ptr to &delta
28      checkCuda(cudaMemcpy(...));
29      checkCuda(cudaMemcpy(&delta, dev_delta_ptr, sizeof(double), cudaMemcpyDeviceToHost));
30      transfers_time += wtime() - timing_transfers;
31
32      const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ?
33      blockSize : numCoords *
34
35      * TODO: can use different blocksize here if deemed better */
36      const unsigned int update_centroids_dim_sz = (numCoords * numClusters +
37      update_centroids_block_sz - 1) / update_centroids_block_sz; /* TODO: calculate dim for
38      "update_centroids" */
39      timing_gpu = wtime();
40      /* TODO: use dim for "update_centroids" and fire it
41      update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
42      (numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters); */
43      update_centroids<<< update_centroids_dim_sz, update_centroids_block_sz, 0 >>>
44      (numCoords, numClusters, devicenewClusterSize, devicenewClusters, deviceClusters);
45
46     cudaDeviceSynchronize();
47     checkLastCudaError();
48     gpu_time += wtime() - timing_gpu;
49
50     timing_cpu = wtime();
51     delta /= numObjs;
52     //printf("delta is %f - ", delta);
53     loop++;
54     //printf("completed loop %d\n", loop);
55     cpu_time += wtime() - timing_cpu;
56
57     timing_internal = wtime() - timing_internal;
58     if (timing_internal < timer_min) timer_min = timing_internal;
59     if (timing_internal > timer_max) timer_max = timing_internal;
60   } while (delta > threshold && loop < loop_threshold);

```

Επαναλάβαμε τις μετρήσεις για όλες τις εκδοχές για τα 2 διαφορετικά configurations και παρακάτω παρουσιάζονται τα διαγράμματα για το speedup και την ανάλυση χρόνου εκτέλεσης ανά επανάληψη ώστε να φαίνονται καλύτερα οι επιδόσεις των διαφόρων εκδοχών.





1) Από τα διαγράμματα speedup παρατηρείται πως η Full-offload εκδοχή έχει καλύτερες επιδόσεις στο configuration με τις πολλές συντεταγμένες (32) ενώ έχει πολύ καλύτερες επιδόσεις στο configuration με τις λίγες συντεταγμένες (2).

2) To blocksize για τις πολλές συντεταγμένες (32) έχει παρόμοια επιρροή με τις υπόλοιπες εκδοχές και έχει βέλτιστη επίδοση για τα 2 μικρότερα block sizes. Όπως έχει προαναφερθεί, τα μικρά blocksizes έχουν την μέγιστη ευελιξία για το scheduling και είναι λογικό, αφού τα δεδομένα είναι πλήρως ανεξάρτητα, να έχουν καλύτερες επιδόσεις. To blocksize για τις λίγες συντεταγμένες (2) δεν επηρεάζει εμφανώς την επίδοση με εξαίρεση το 48 που λόγω half warps και ότι δεν είναι bottleneck η μνήμη, δεν αξιοποιεί πλήρως τους πόρους της GPU.

3) Το κομμάτι update_centroids έχει πολύ χαμηλό computational intensity, οπότε δεν είναι ιδανικό για GPUs. Όμως, είναι πλήρως παραλληλοποιήσιμο, γι' αυτό έχουμε και σαφές speedup σε σχέση

με την χρήση CPU. Μέσω της ανάλυσης χρόνου εκτέλεσης ανά επανάληψη, μπορεί να φανεί πως ο υπολογισμός των καινούριων clusters στην GPU αύξησε ελάχιστα τον χρόνο εκτέλεσης της GPU ενώ προφανώς μηδενίστηκε ο χρόνος εκτέλεσης της CPU. Σε αυτό οφείλεται η διαφορά επίδοσης, απλά στο speedup συνυπολογίζεται και ο χρόνος allocation και αρχικής μεταφοράς και δεν υπάρχει ανάλογη βελτίωση συγκριτικά.

4) Στο configuration με τις λίγες συντεταγμένες (2) ο χρόνος μεταφοράς έχει σημαντικό ποσοστό του συνολικού χρόνου μιας επανάληψης. Καθώς η εκδοχή Full-offload αποφεύγει και τις μεταφορές μέσα στην επανάληψη, το speedup είναι ακόμη μεγαλύτερο.

Bonus 2: Delta Reduction (All-GPU)

Για την υλοποίηση του δενδρικού delta reduction αρχικά χρειάζεται περισσότερη διαμοιραζόμενη μνήμη. Κάθε thread στο thread block πρέπει να έχει τον δικό του delta και μετά να γίνει το reduction. Γι' αυτό χρειάζεται ένας πίνακας από delta και για κάθε thread αντιστοιχεί μια θέση στον πίνακα με όρισμα το local id του.

Κάθε thread αφού βρει το καινούριο κοντινότερο cluster ελέγχει αν είναι το ίδιο με πριν. Αν είναι τότε βάζει το δικό του delta να είναι 0.0, αλλιώς 1.0. Έπειτα, χρειάζεται συγχρονισμός των threads, πριν εκτελεστεί το reduction ώστε να έχουν υπολογιστεί όλα τα επιμέρους delta.

Το δενδρικό reduction ακολουθεί την λογική ότι σε κάθε επανάληψη οι μισοί προσθέτουν στο δικό τους delta, το delta των υπολοίπων. Οπότε οι πρώτοι μισοί εκτελούν

`delta[local_id] += delta[local_id + size]` και σε κάθε επανάληψη μειώνεται το μέγεθος κατά 2, εξού και δέντρο. Μεταξύ των επαναλήψεων χρειάζεται συγχρονισμός των νημάτων για να έχουν υπολογιστεί τα αποτελέσματα όλων των προσθέσεων. Στο τέλος, το συνολικό αποτέλεσμα θα είναι στο πρώτο thread του thread block, το οποίο χρειάζεται να κάνει μόνο 1 atomicAdd στο global delta.

delta_reduction_find_nearest_cluster.cu

```
1  /*-----< find_nearest_cluster() >-----*/
2  __global__ static
3  void find_nearest_cluster(int numCoords,
4      int numObjs,
5      int numClusters,
6      double *deviceobjects,           // [numCoords][numObjs]
7      int *devicenewClusterSize,      // [numClusters]
8      double *devicenewClusters,      // [numCoords][numClusters]
9      double *deviceClusters,        // [numCoords][numClusters]
10     int *deviceMembership,         // [numObjs]
11     double *devdelta) {
12     extern __shared__ double shmem_total[];
13     double *shmemClusters = shmem_total;
14     double *delta_reduce_buff = shmem_total + numClusters * numCoords;
15     /* TODO: copy me from shared version... */
16     int no_cluster, i;
17
18     //use local_id because shared memory is per thread block
19     for (no_cluster = threadIdx.x; no_cluster < numClusters; no_cluster+=blockDim.x) {
20         for (i = 0; i < numCoords; i++) {
21             shmemClusters[i * numClusters + no_cluster] = deviceClusters[i * numClusters + no_cluster];
22         }
23     }
24     __syncthreads();
25
26     /* Get the global ID of the thread. */
27     int tid = get_tid();
28
29     /* TODO: copy me from shared version... */
30     if (tid < numObjs) {
31
32         /* TODO: copy me from shared version... */
33         int index;
34         double dist, min_dist;
35
36         /* find the cluster id that has min distance to object */
37         index = 0;
38         /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId using clusters
39         in shmem*/
40         min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
41         shmemClusters, tid, 0);
42         for (i = 1; i < numClusters; i++) {
43             /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId using clusters
44             in shmem*/
45             dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, deviceobjects,
46             shmemClusters, tid, i);
47
48             /* no need square root */
```

```

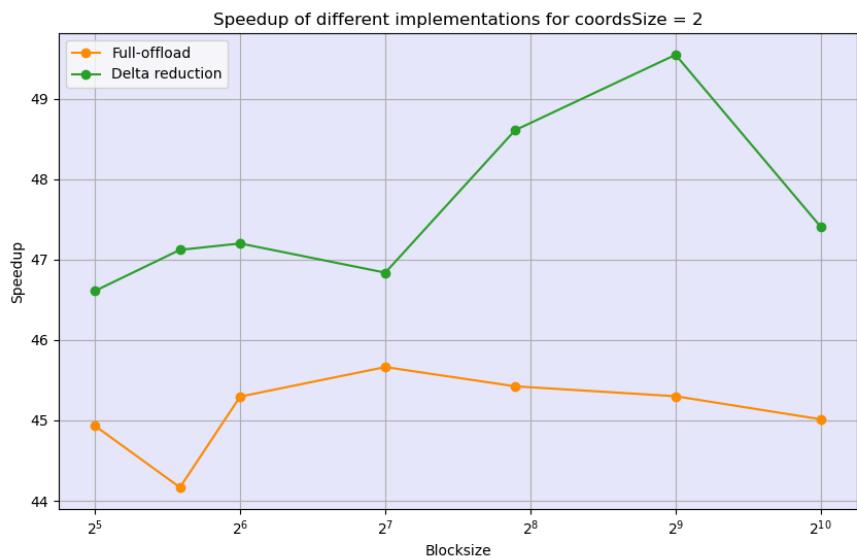
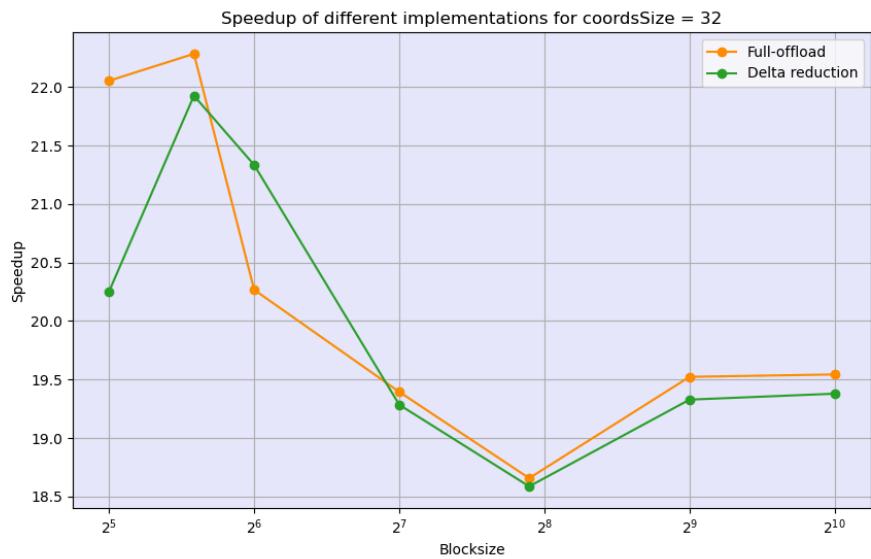
45     if (dist < min_dist) { /* find the min and its array index */
46         min_dist = dist;
47         index = i;
48     }
49 }
50
51 if (deviceMembership[tid] != index) {
52     delta_reduce_buff[threadIdx.x] = 1.0;
53 }
54 else {
55     delta_reduce_buff[threadIdx.x] = 0.0;
56 }
57
58 /* assign the deviceMembership to object objectId */
59 deviceMembership[tid] = index;
60
61 /* TODO: Replacing (*devdelta)+= 1.0; with reduction:
62    - each thread updates the single element of delta_reduce_buff
63    corresponding to its local id (threadIdx.x) -> 1.0 if membership changes, otherwise 0.
64    - Then, ensuring delta_reduce_buff is fully updated, its contents must be summed
65    in delta_reduce_buff[0]
66    either by one thread (lower perf) or with a tree-based reduction (similar to dot reduction
example in slides)
67    - Finally, delta_reduce_buff[0] (local value in block) must be added to devdelta (global
delta value), ensuring write dependencies!
68 */
69
70 /* TODO: additional steps for calculating new centroids in GPU? */
71 atomicAdd(&devicenewClusterSize[index], 1);
72 for (i = 0; i < numCoords; i++) {
73     atomicAdd(&devicenewClusters[i * numClusters + index], deviceobjects[i * numObjs + tid]);
74 }
75
76 __syncthreads();
77 //after everyone in the block is finished do the tree update of delta
78 i = blockDim.x / 2;
79 while (i != 0) {
80     if (threadIdx.x < i) delta_reduce_buff[threadIdx.x] += delta_reduce_buff[threadIdx.x
+ i];
81     __syncthreads();
82     i /= 2;
83 }
84 if (threadIdx.x == 0) atomicAdd(devdelta, delta_reduce_buff[0]);
85 }

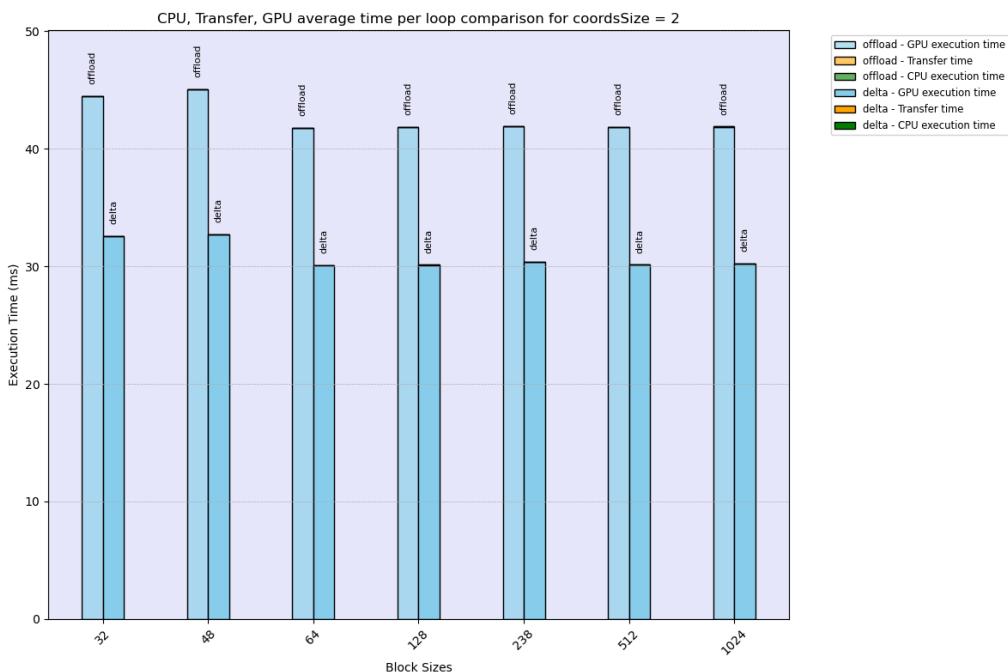
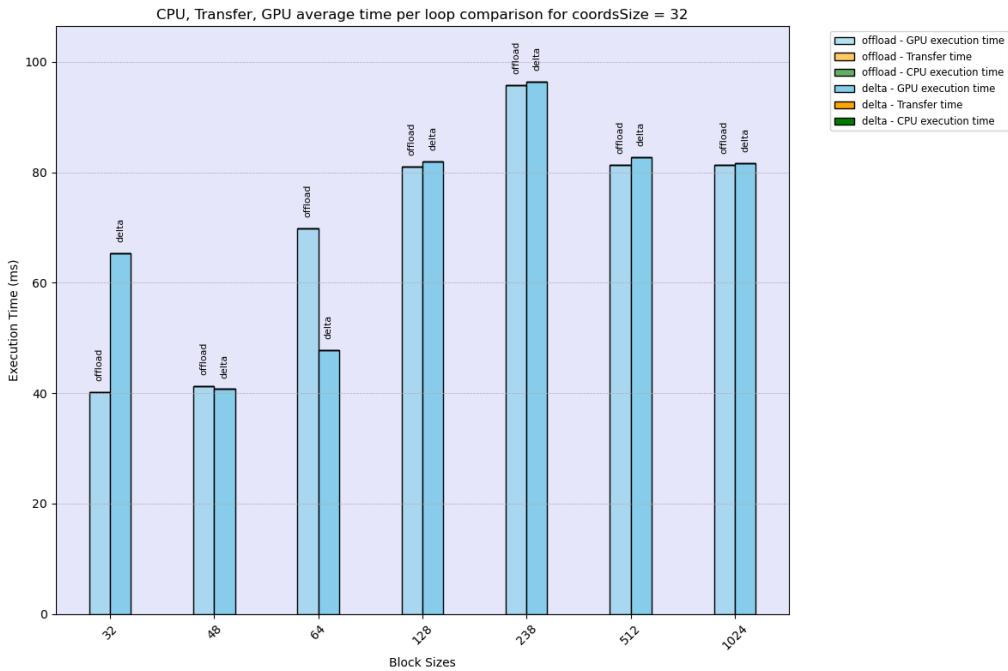
```

Τέλος, η μόνη αλλαγή που χρειάζεται στο υπόλοιπο πρόγραμμα είναι η αύξηση του μεγέθους της διαμοιραζόμενης μνήμης κατά blocksize πραγματικούς. Οπότε:

```
const unsigned int clusterBlockSharedDataSize = numClusters * numCoords * sizeof(double) +
numThreadsPerClusterBlock * sizeof(double);
```

Επαναλάβαμε τις μετρήσεις για την delta reduction εκδοχή και παρακάτω παρουσιάζονται τα διαγράμματα speedup και ανάλυσης χρόνου εκτέλεσης ανά επανάληψη σε σύγκριση με την Full-offload εκδοχή για τα 2 διαφορετικά configurations.





Η επίδοση της delta reduction εκδοχής είναι χειρότερη από την απλή all-gpu στο configuration με τις πολλές συντεταγμένες (32). Αυτό θα μπορούσε να εξηγηθεί ως προς τον παραπάνω συγχρονισμό που απαιτεί η delta reduction εκδοχή. Επειδή θα γίνουν ούτως ή άλλως 32 atomicAdds για τις συντεταγμένες, δεν θα κάνουν όλα τα threads, που έχουν διαφορετικό καινούριο cluster, ταυτόχρονα atomicAdd στο delta. Δεν θα είναι τόσο ταυτόχρονα στο χρόνο τα atomicAdds καθώς θα υπάρχουν διαφορετικές μικρές καθυστερήσεις λόγω των 32 προηγούμενων atomicAdds. Έτσι, όχι μόνο δεν υπάρχει βελτίωση, αλλά υπάρχει και μια μικρή επιπρόσθετη καθυστέρηση.

Σε αντίθεση, στο configuration με τις λίγες συντεταγμένες (2) καθώς τα atomicAdds έχουν όντως πολλές κοντινές χρονικά εκτελέσεις και απαιτείται όντως συγχρονισμός, η εκδοχή του delta reduction έχει όντως βελτίωση επίδοσης. Συγκριτικά, θέλει περίπου 25% λιγότερο χρόνο ανά loop για το βέλτιστο blocksize σε σχέση με την απλή Full-offload έκδοση.

To blocksize έχει διαφορετικό ρόλο μόνο στο configuration με τις λίγες συντεταγμένες (2), καθώς χρειάζεται το blocksize να είναι επαρκώς μεγάλο ώστε η λογαριθμική του πολυπλοκότητα να είναι καλύτερη από την γραμμική που έχει η απλή all-gpu. Σε μικρά blocksizes, η βελτίωση στην επίδοση που υπάρχει είναι μικρότερη καθώς η σταθερά αυτής της πολυπλοκότητας είναι αρκετά μεγάλη.