



# Συστήματα Παράλληλης Επεξεργασίας

9ο Εξάμηνο, 2024-2025

## Εργαστηριακή Αναφορά

των φοιτητών:

Λάζου Μαρία-Αργυρώ (el20129)

Σπηλιώτης Αθανάσιος (el20175)

Ομάδα: **parlab09**

# Conway's Game of Life

## 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 -l 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 }
85 gettimeofday(&tf,NULL);
86 time=(tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec)*0.000001;
87
88 free_array(current, N);
89 free_array(previous, N);
90 printf("GameOfLife: Size %d Steps %d Time %lf\n", N, T, time);
91 #ifdef OUTPUT
92 system(FINALIZE);
93 #endif
94 }
95
96 int ** allocate_array(int N) {
97     int ** array;
98     int i,j;
99     array = malloc(N * sizeof(int*));
100     for ( i = 0; i < N ; i++ )
101         array[i] = malloc( N * sizeof(int));
102     for ( i = 0; i < N ; i++ )
103         for ( j = 0; j < N ; j++ )
104             array[i][j] = 0;
105     return array;
106 }
107
108 void free_array(int ** array, int N) {
109     int i;
110     for ( i = 0 ; i < N ; i++ )
111         free(array[i]);
112     free(array);
113 }
114
115 void init_random(int ** array1, int ** array2, int N) {
116     int i,pos,x,y;
117
118     for ( i = 0 ; i < (N * N)/10 ; i++ ) {
119         pos = rand() % ((N-2)*(N-2));
120         array1[pos%(N-2)+1][pos/(N-2)+1] = 1;
121         array2[pos%(N-2)+1][pos/(N-2)+1] = 1;
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 μεταγλώττιση 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
```

```

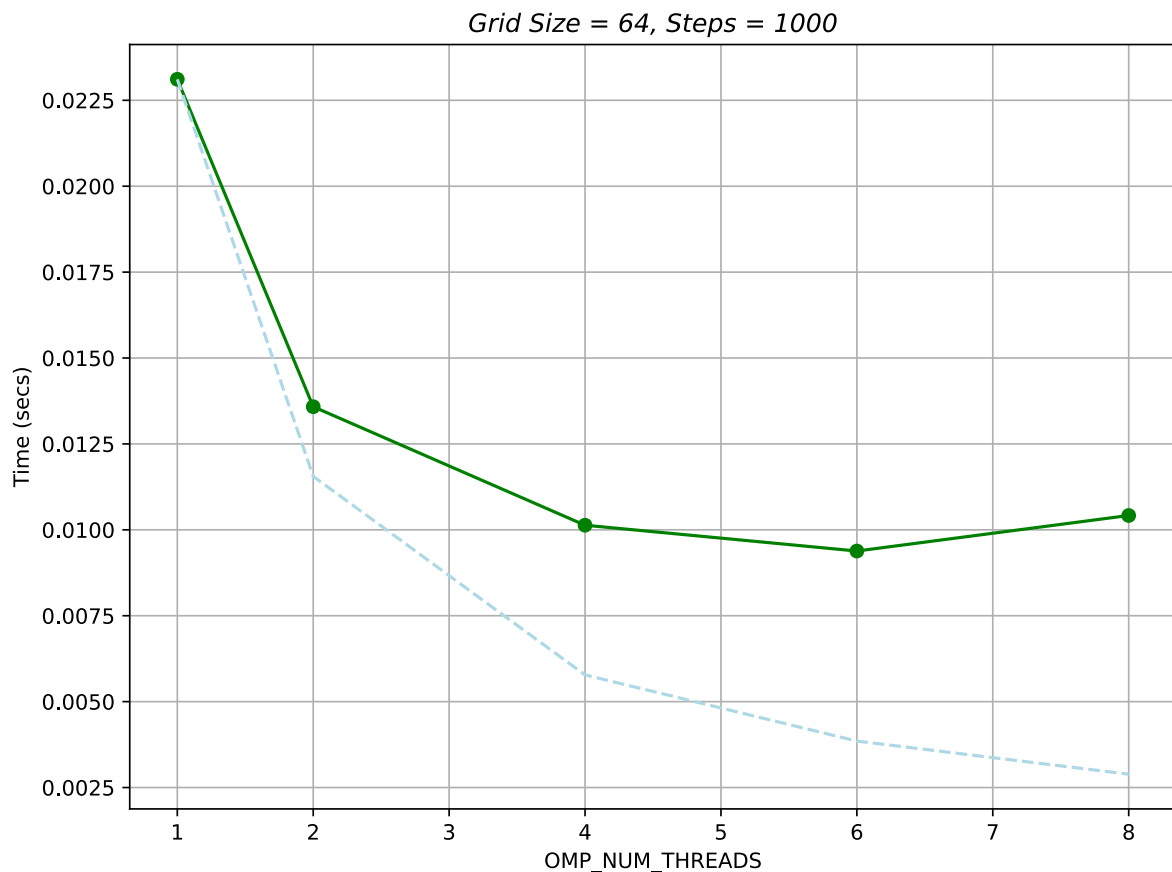
-----
Running with OMP_NUM_THREADS=4
GameOfLife: Size 64 Steps 1000 Time 0.010134
GameOfLife: Size 1024 Steps 1000 Time 2.723798
GameOfLife: Size 4096 Steps 1000 Time 45.901567
Finished run with OMP_NUM_THREADS=4
-----

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
-----

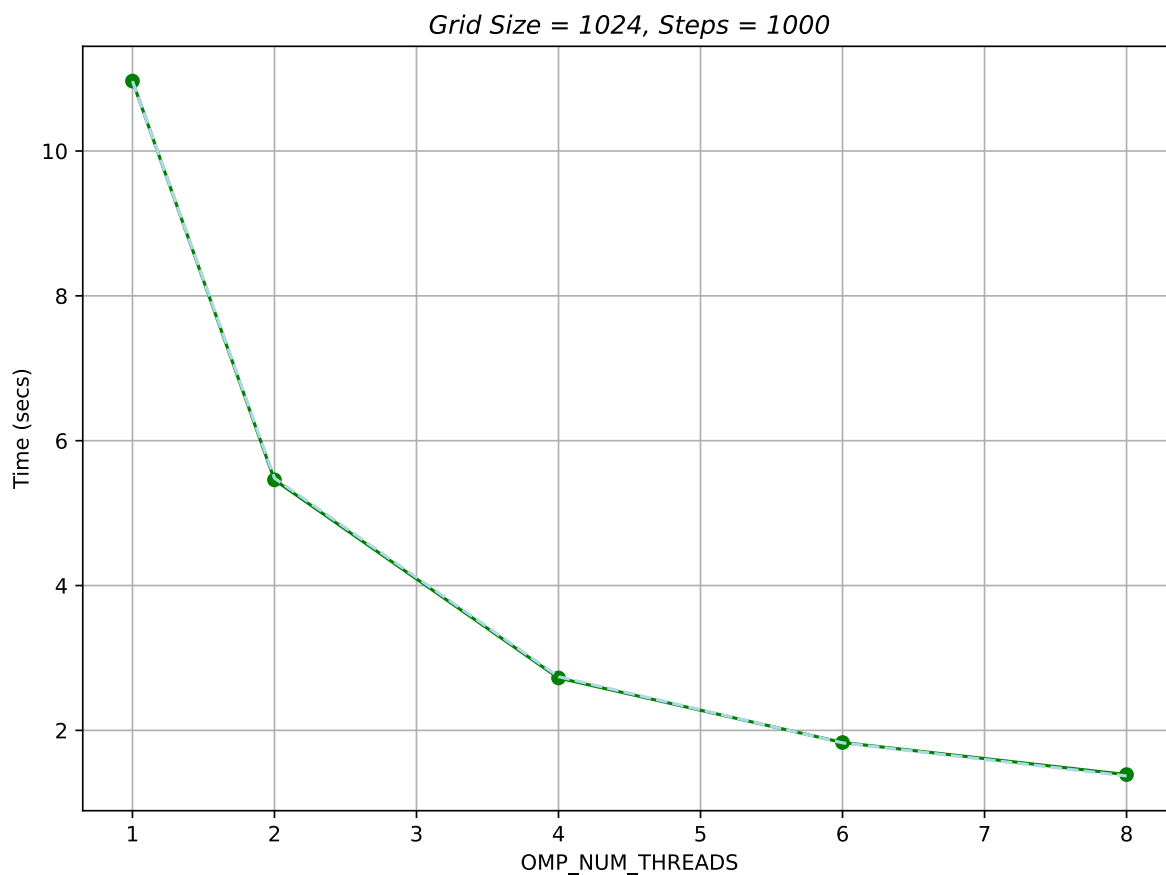
Running with OMP_NUM_THREADS=8
GameOfLife: Size 64 Steps 1000 Time 0.010417
GameOfLife: Size 1024 Steps 1000 Time 1.389175
GameOfLife: Size 4096 Steps 1000 Time 43.186379
Finished run with OMP_NUM_THREADS=8
-----

```

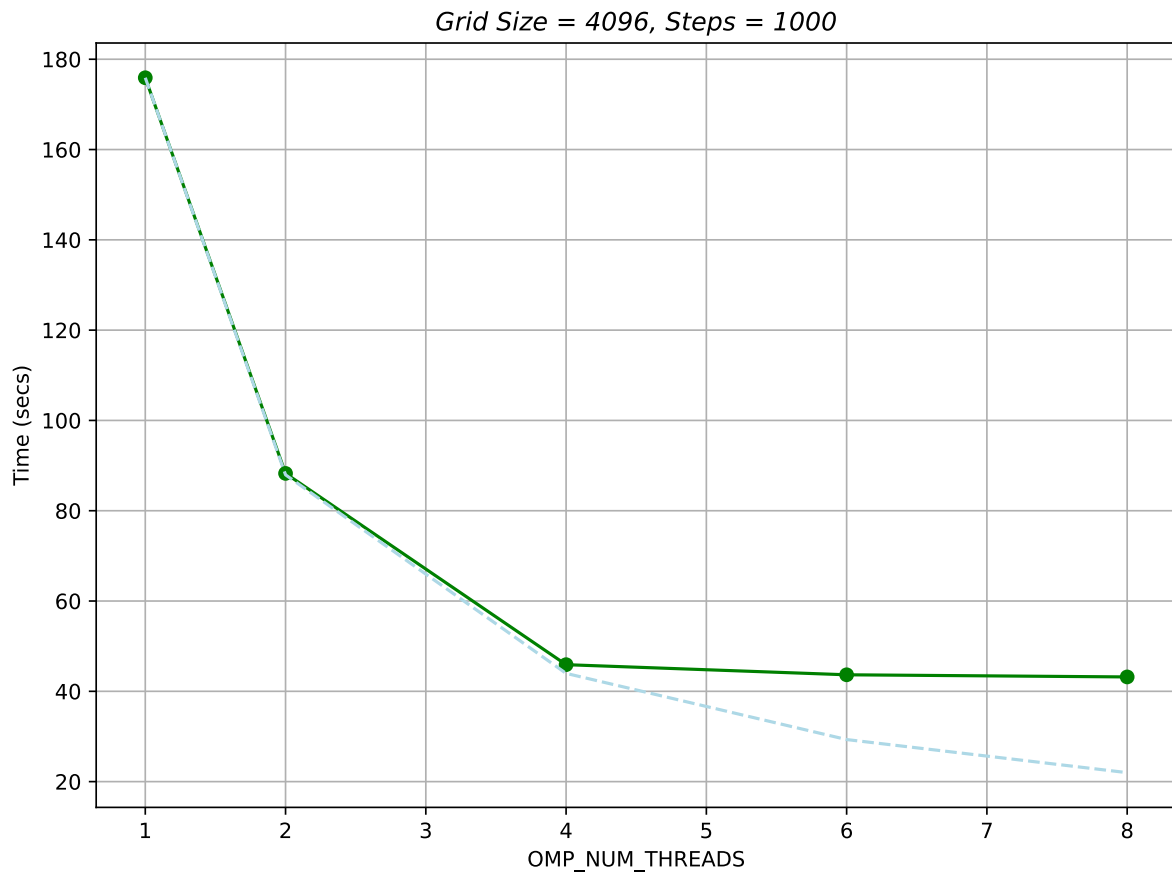
## Γραφική Απεικόνιση και Παρατηρήσεις



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 συγκριτικά με τον φόρτο εργασίας που τους ανατίθεται (granularity).



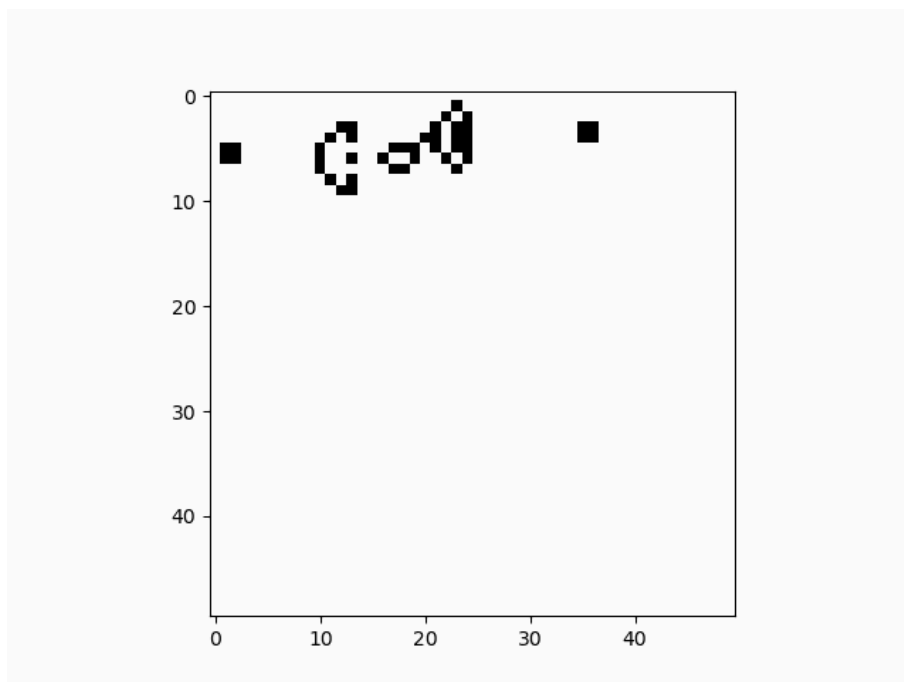
Για μέγεθος grid με συνολική απαίτηση μνήμης  $4*1024*1024\text{ bytes} = 4\text{MB}$ , η επίδοση βελτιώνεται ομοιόμορφα και ανάλογα με το μέγεθος των νημάτων. Εικάζουμε, λοιπόν, πως η cache χωράει ολόκληρο το grid ώστε το κάθε νήμα δεν επιβαρύνει την μνήμη με loads των αντίστοιχων rows, ο φόρτος εργασίας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



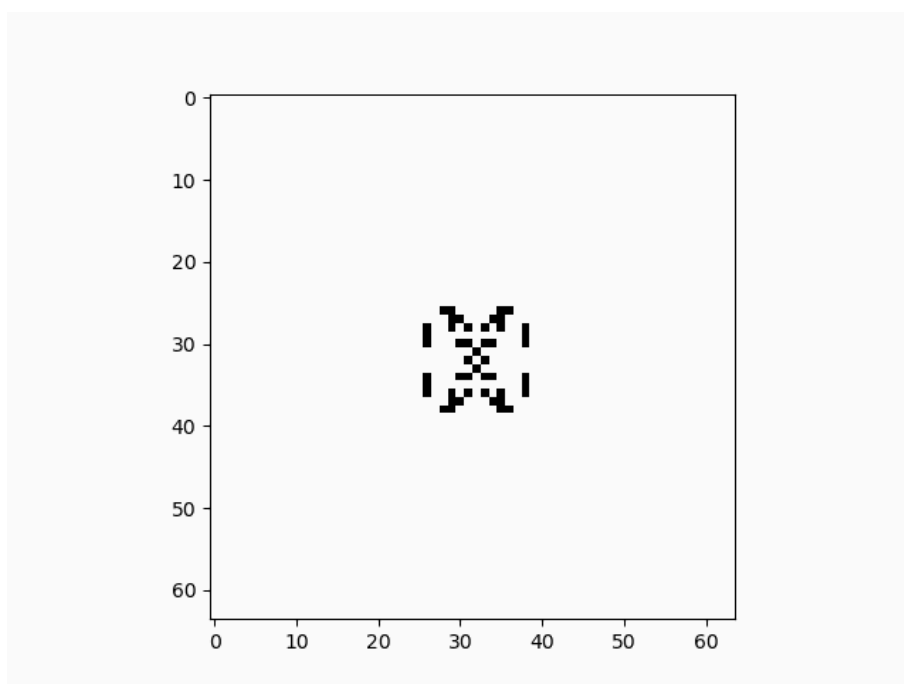
Για μεγάλο grid (με συνολική απαίτηση μνήμης  $4 \cdot 4096 \cdot 4096$  bytes = 64MB), η κλιμάκωση παύει να υφίσταται για περισσότερα από 4 νήματα. Bottleneck κόστους εδώ θεωρούμε το memory bandwidth. Επειδή ολόκληρο το grid δεν χωράει στην cache, δημιουργούνται misses όταν ξεχωριστά νήματα προσπαθούν 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*



## Πρόσκληση

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 χρησιμοποιήσαμε 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
90         #pragma omp parallel for private(i, j, index) shared(newClusters, newClusterSize,
91         membership) schedule(static)
92         for (i=0; i<numObjs; i++) {
93             // find the array index of nearest cluster center
94             index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
95
96             // if membership changes, increase delta by 1
97             if (membership[i] != index)
98                 #pragma omp atomic
99                 delta += 1.0;
100
101             // assign the membership to object i
102             membership[i] = index;
103
104             // update new cluster centers : sum of objects located within
105             /*
106              * TODO: protect update on shared "newClusterSize" array
107              */
108             #pragma omp atomic
109             newClusterSize[index]++;
110             for (j=0; j<numCoords; j++)
111                 /*
112                  * TODO: protect update on shared "newClusters" array
113                  */
114                 #pragma omp atomic
115                 newClusters[index*numCoords + j] += objects[i*numCoords + j];
116         }
117
118         // average the sum and replace old cluster centers with newClusters
119         // #pragma omp parallel for private(i,j)
120         for (i=0; i<numClusters; i++) {
121             if (newClusterSize[i] > 0) {
122                 for (j=0; j<numCoords; j++) {
123                     clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
124                 }
125             }
126         }
127     }
128 }

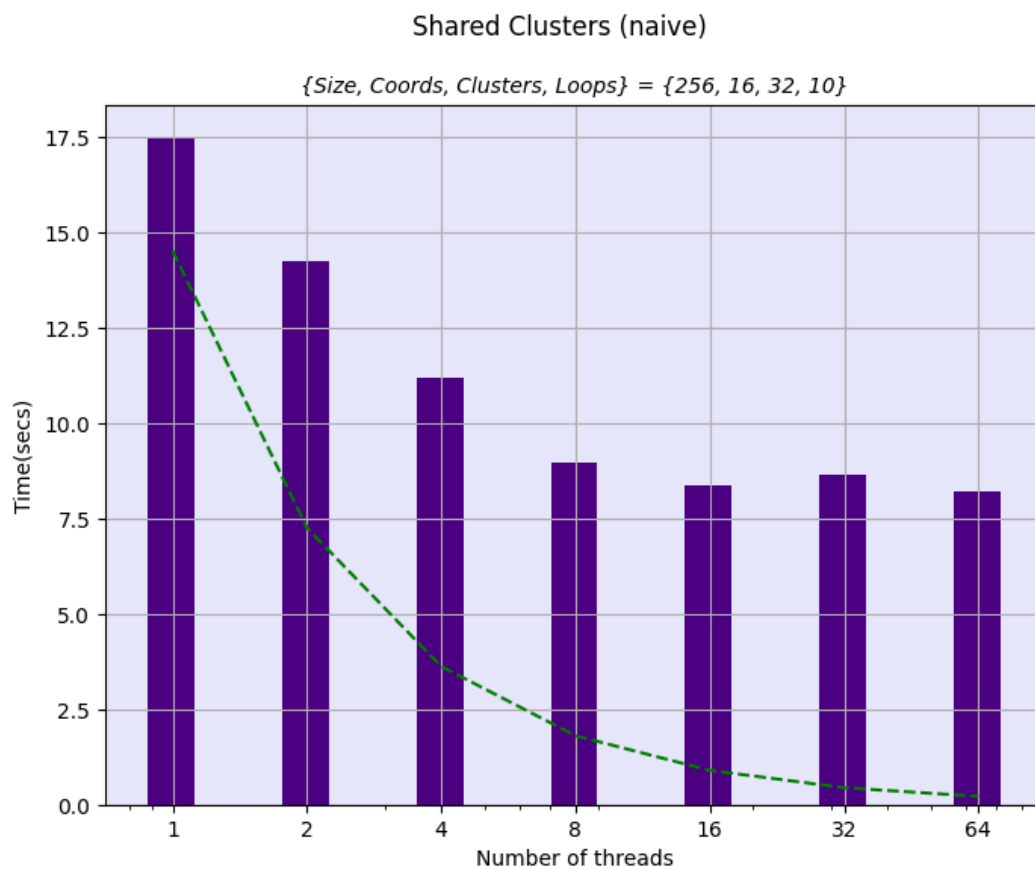
```

```

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

```

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 εξαιτίας της σειριοποίησης των εγγραφών ολόενα 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 pin 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.

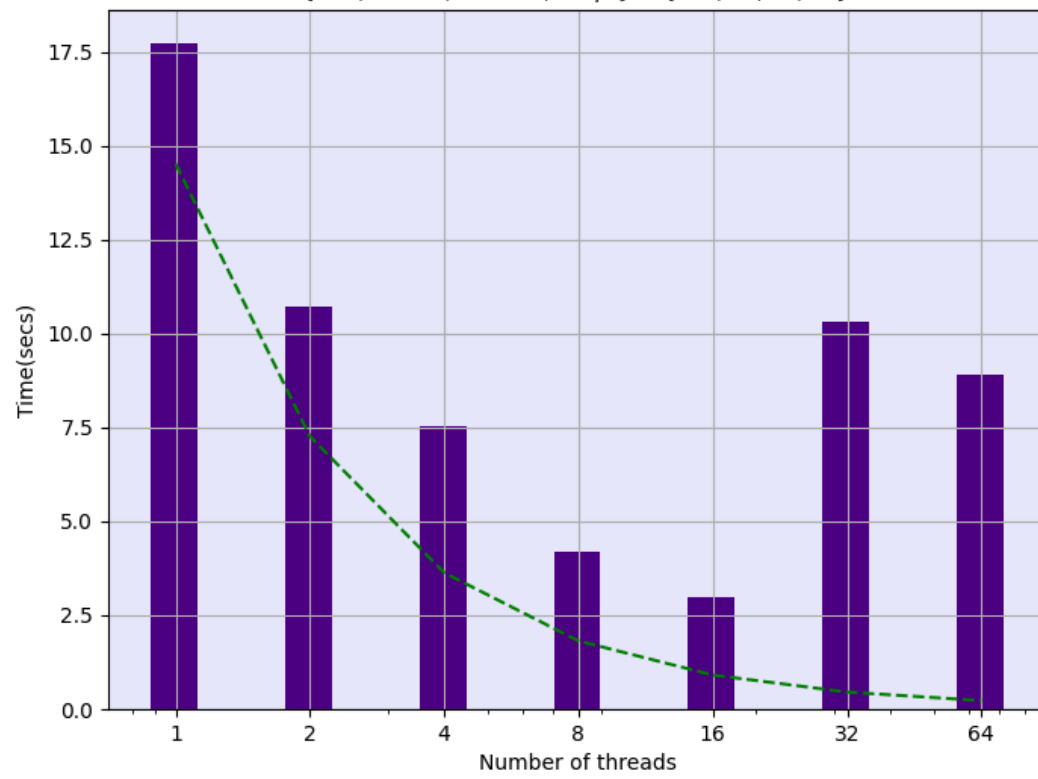
Για την υλοποίηση τροποποιήσαμε κατάλληλα το script υποβολής στον sandman και προσθέσαμε την παράμετρο schedule static στο parallel for. Τα αποτελέσματα παρουσιάζονται παρακάτω:



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 pin 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. Εάν αξιοποιήσουμε το hyperthreading και κάνουμε pin 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 την βελτιωμένη εκδοχή των 16 νημάτων ακολούθως:

### Shared Clusters with GOMP\_CPU\_AFFINITY[0-7][32-40]

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

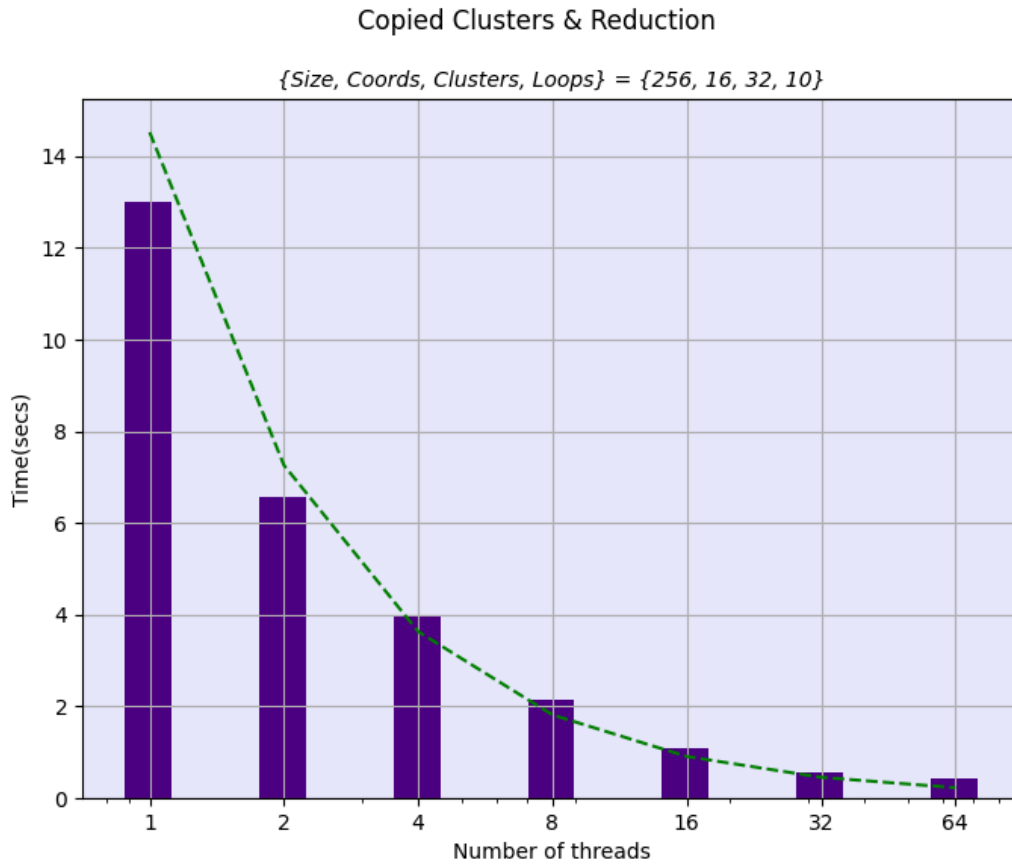


## 2) Copied Clusters & Reduce

### 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).

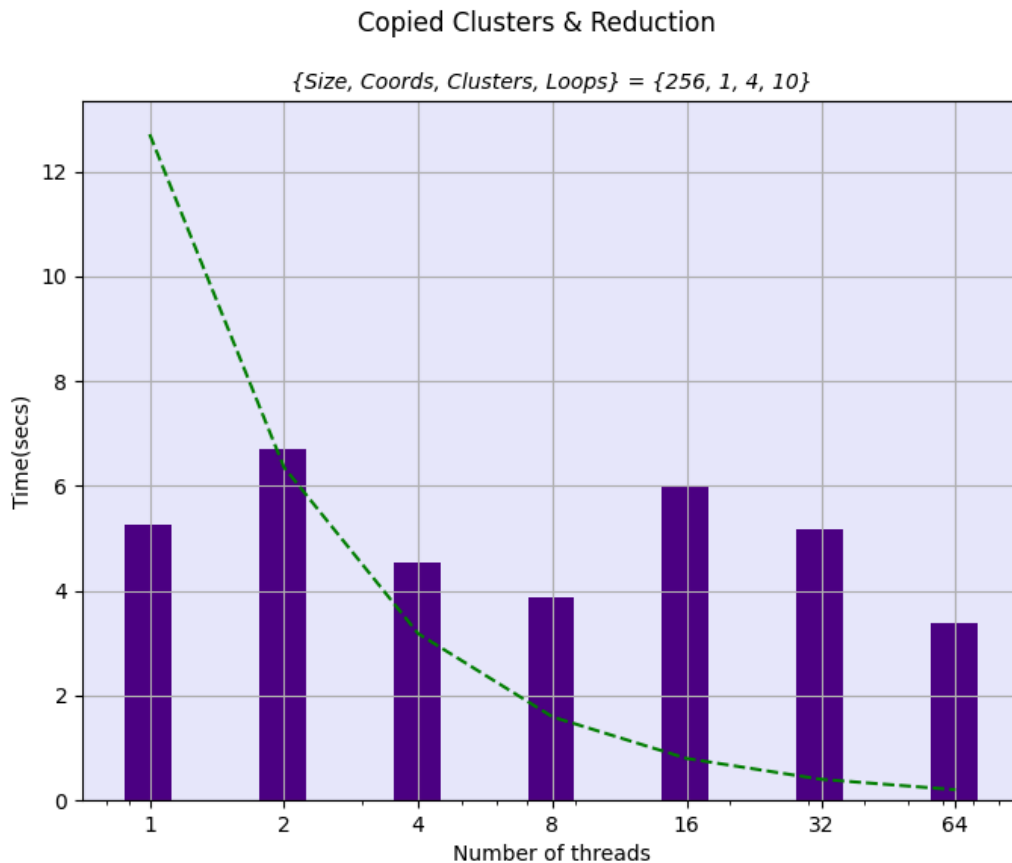
### Αποτελέσματα



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 καθυσטרει τον αλγόριθμο.

### Δοκιμές με μικρότερο dataset

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



Κυριαρχο 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     // 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)) calloc(numClusters,
89         // sizeof(*local_newClusterSize));

```

```

87 // local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords,
sizeof(**local_newClusters));
88 // }
89
90 timing = wtime();
91 do {
92 // before each loop, set cluster data to 0
93 // #pragma omp parallel for private(i,j)
94 for (i=0; i<numClusters; i++) {
95     for (j=0; j<numCoords; j++)
96         newClusters[i*numCoords + j] = 0.0;
97     newClusterSize[i] = 0;
98 }
99
100 delta = 0.0;
101
102 /*
103  * TODO: Initiliaze local cluster data to zero (separate for each thread)
104  */
105 // #pragma omp parallel for private(k, i, j) shared(local_newClusters,
local_newClusterSize) schedule(static)
106 // for (k=0; k<nthreads; ++k){
107 //     for (i=0; i<numClusters; i++) {
108 //         for (j=0; j<numCoords; j++)
109 //             local_newClusters[k][i*numCoords + j] = 0.0;
110 //         local_newClusterSize[k][i] = 0;
111 //     }
112 // }
113 #pragma omp parallel for private(k,i,j) schedule(static)
114 for (k=0; k<nthreads; ++k){
115     local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters,
sizeof(**local_newClusterSize));
116     local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords,
sizeof(**local_newClusters));
117
118     for (i=0; i<numClusters; i++) {
119         for (j=0; j<numCoords; j++)
120             local_newClusters[k][i*numCoords + j] = 0.0;
121         local_newClusterSize[k][i] = 0;
122     }
123 }
124 int thread_id;
125
126 #pragma omp parallel for private(i, j, thread_id, index) shared(local_newClusters,
local_newClusterSize) reduction(+:delta) schedule(static)
127 for (i=0; i<numObjs; i++)
128 {
129     thread_id = omp_get_thread_num();
130
131     // find the array index of nearest cluster center
132     index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
133
134     // if membership changes, increase delta by 1
135     if (membership[i] != index)
136         delta += 1.0;
137
138     // assign the membership to object i
139     membership[i] = index;
140
141     // update new cluster centers : sum of all objects located within (average will be
performed later)
142     /*
143      * TODO: Collect cluster data in local arrays (local to each thread)
144      *         Replace global arrays with local per-thread
145      */
146
147     local_newClusterSize[thread_id][index]++;
148     for (j=0; j<numCoords; j++)
149         local_newClusters[thread_id][index*numCoords + j] += objects[i*numCoords + j];
150 }
151 }
152
153 /*
154  * TODO: Reduction of cluster data from local arrays to shared.

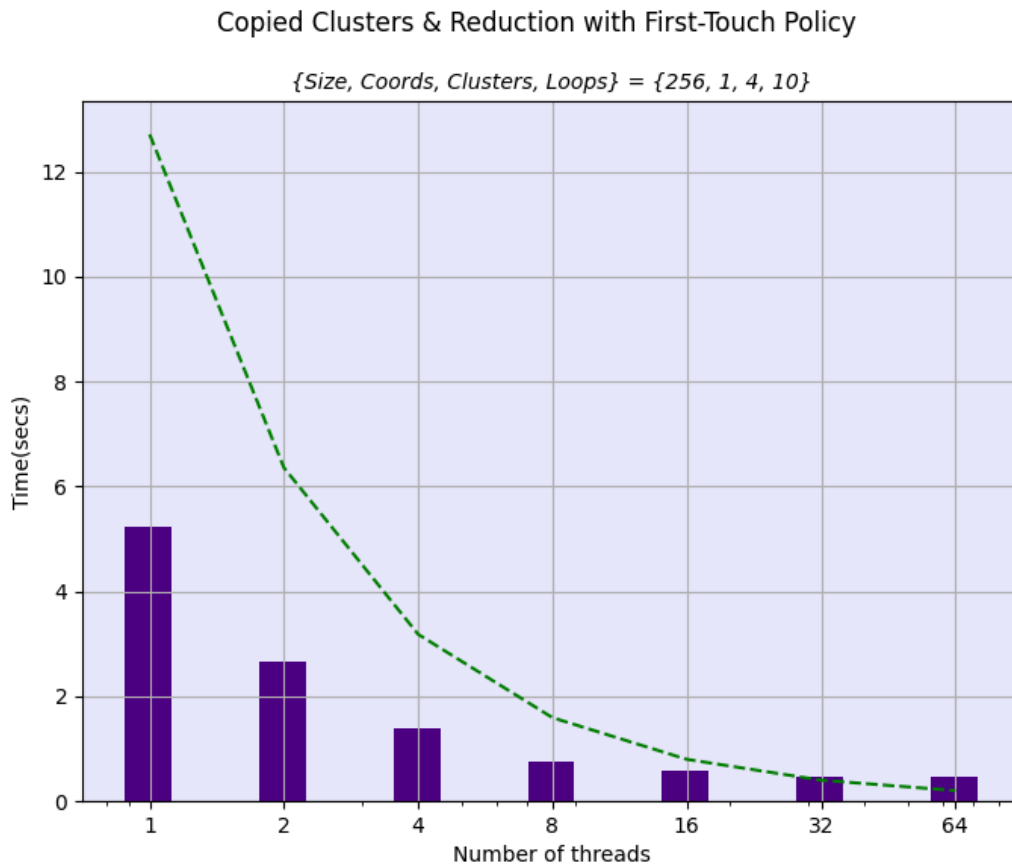
```

```

155     *      This operation will be performed by one thread
156     */
157
158     for (i=0; i<numClusters; ++i){
159         for (k=0; k<nthreads; ++k){
160             newClusterSize[i] += local_newClusterSize[k][i];
161             for (j=0; j<numCoords; ++j)
162                 newClusters[i*numCoords+j] += local_newClusters[k][i*numCoords+j];
163         }
164     }
165
166     // average the sum and replace old cluster centers with newClusters
167     // #pragma omp parallel for private(i,j)
168     for (i=0; i<numClusters; i++) {
169         if (newClusterSize[i] > 0) {
170             for (j=0; j<numCoords; j++) {
171                 clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
172             }
173         }
174     }
175 }
176
177 // Get fraction of objects whose membership changed during this loop. This is used as a
178 // convergence criterion.
179 delta /= numObjs;
180
181 loop++;
182 printf("\r\tcompleted loop %d", loop);
183 fflush(stdout);
184 } while (delta > threshold && loop < loop_threshold);
185 timing = wtime() - timing;
186 printf("\n          nloops = %3d    (total = %7.4fs)    (per loop = %7.4fs)\n", loop, timing,
187 timing/loop);
188
189 for (k=0; k<nthreads; k++)
190 {
191     free(local_newClusterSize[k]);
192     free(local_newClusters[k]);
193 }
194 free(newClusters);
195 free(newClusterSize);
196 }

```

## Αποτελέσματα



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
```

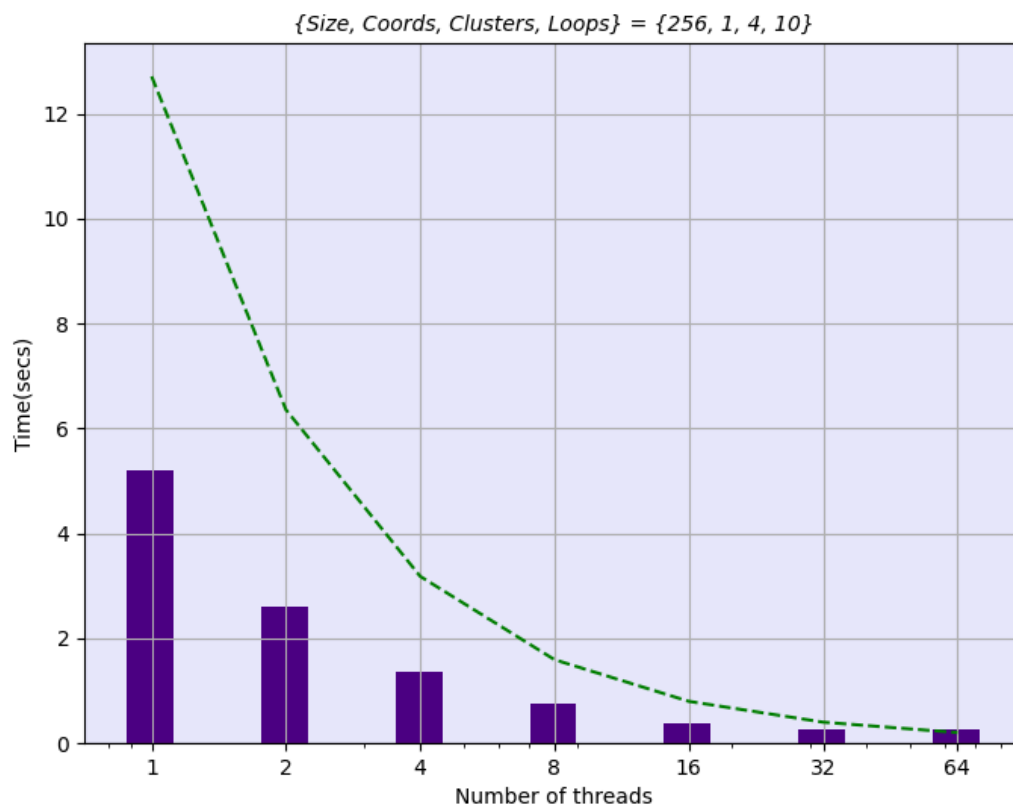
```

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

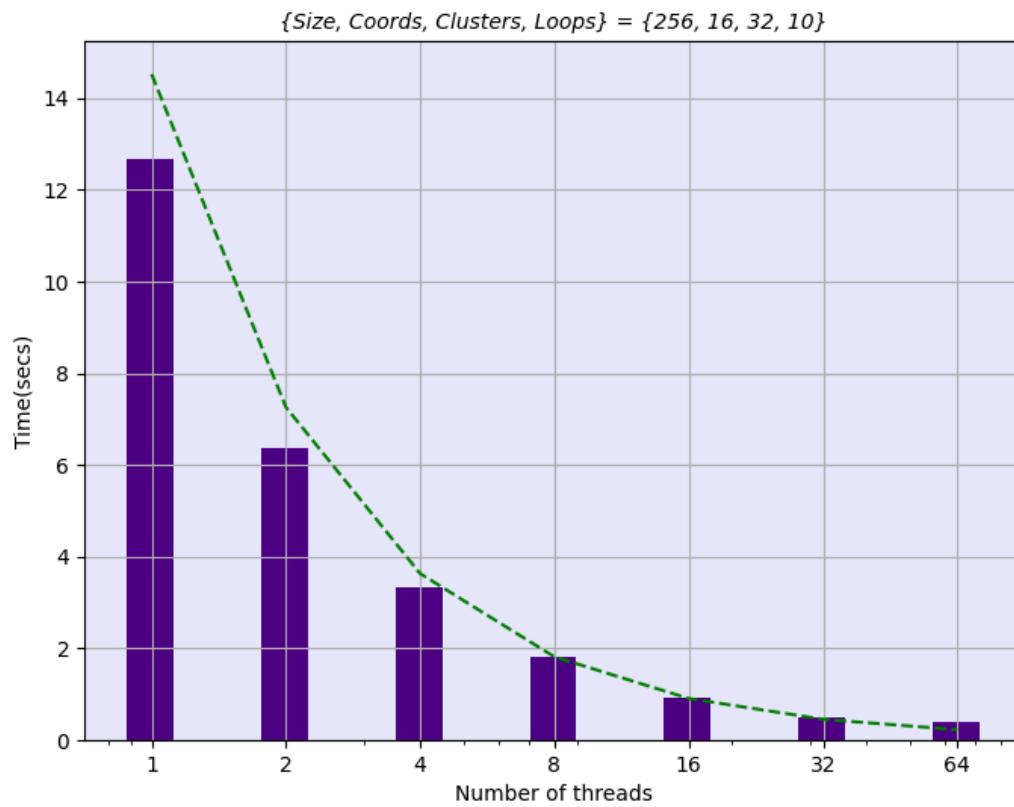
### Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization



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 compute 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`)) Για την διατήρηση των εξαρτήσεων κατά τον υπολογισμό των blocks ( $A_{11} \rightarrow (A_{12} A_{21}) \rightarrow A_{22}$  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
45     gettimeofday(&t1,0);
46
47     #pragma omp parallel
48     {
49         #pragma omp single
50         {
51             FW_SR(A,0,0, A,0,0,A,0,0,N,B);
52         }
53     }
54     gettimeofday(&t2,0);
55
56     time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)/1000000;
```



```

57     printf("FW_SR,%d,%d,%.4f\n", N, B, time);
58
59     /*
60     for(i=0; i<N; i++)
61         for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
62     */
63
64     return 0;
65 }
66
67 inline int min(int a, int b)
68 {
69     if(a<=b) return a;
70     else return b;
71 }
72
73 void FW_SR (int **A, int arow, int acol,
74             int **B, int brow, int bcol,
75             int **C, int crow, int ccol,
76             int myN, int bsize)
77 {
78     int k,i,j;
79
80     /*
81     * The base case (when recursion stops) is not allowed to be edited!
82     * What you can do is try different block sizes.
83     */
84
85     if(myN<=bsize)
86         for(k=0; k<myN; k++)
87             for(i=0; i<myN; i++)
88                 for(j=0; j<myN; j++)
89                     A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
90     else {
91
92         FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize); // A00
93         #pragma omp task
94         FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize); //A01
95         #pragma omp task
96         FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize); //A10
97         #pragma omp taskwait
98         FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C,crow, ccol+myN/2, myN/2, bsize); //
99
100        A11
101        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,
102        bsize); //A11
103        #pragma omp task
104        FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); //
105        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); //
108        A01
109        #pragma omp taskwait
110        FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); //A00
111    }
112    // printf("Nested parallelism enabled: %d\n", omp_get_nested());
113 }

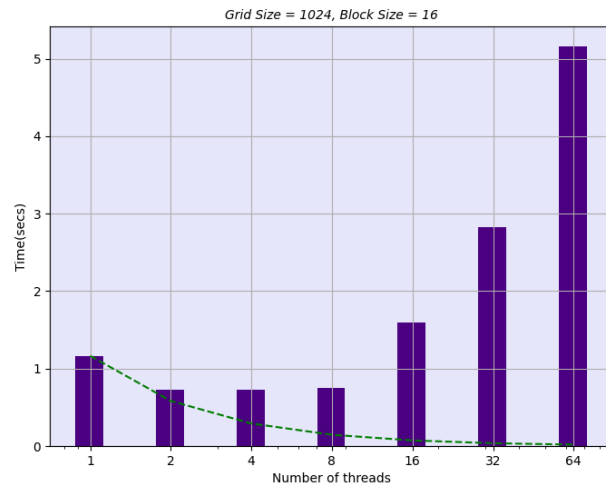
```

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  $B_{opt} = \sqrt{\text{cache size}}$ .

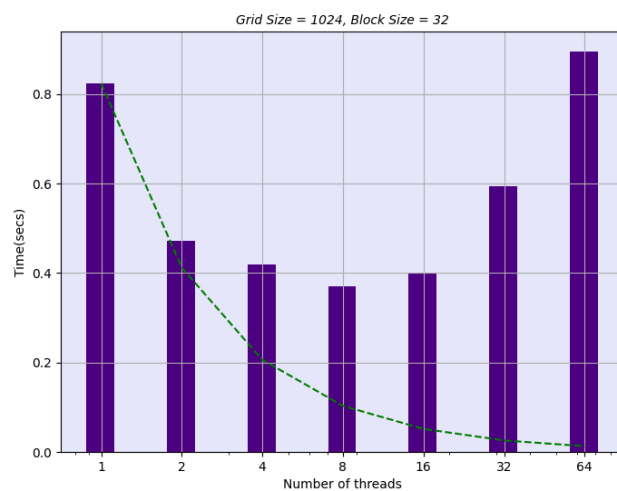
## Αποτελέσματα

$\{N = 1024\}$

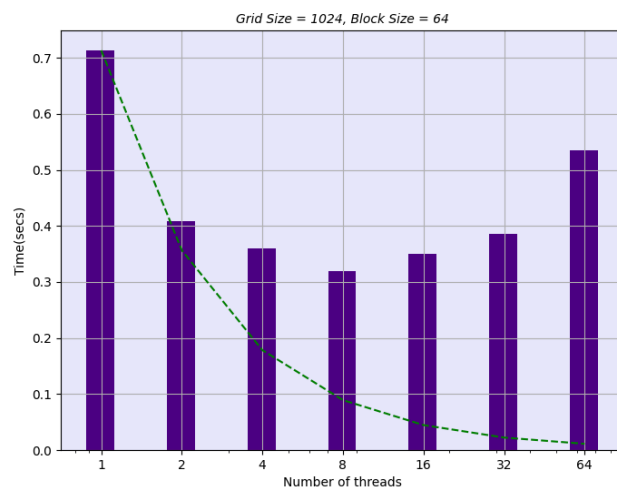
Total Times for Floyd-Warshall Recursive

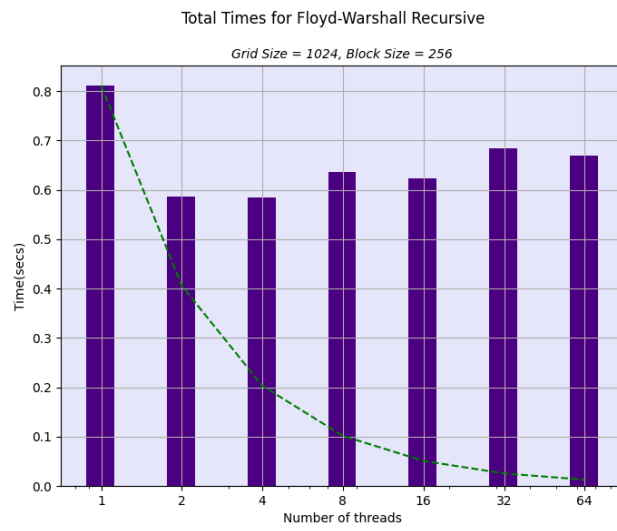
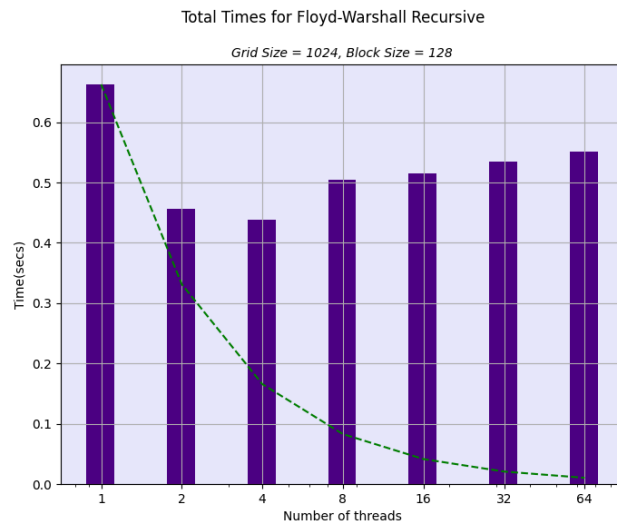


Total Times for Floyd-Warshall Recursive

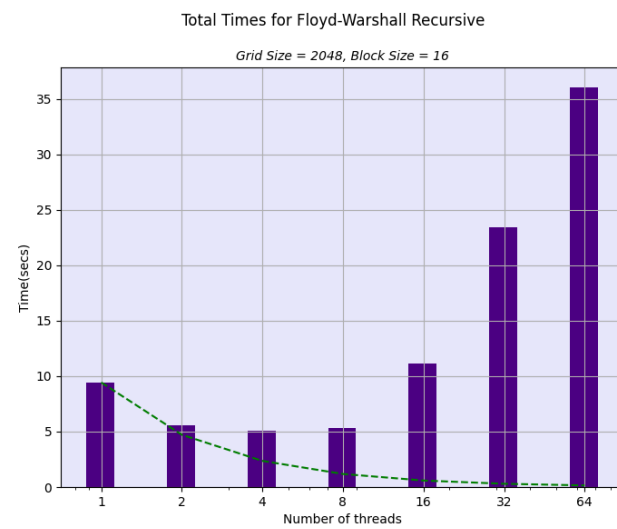


Total Times for Floyd-Warshall Recursive



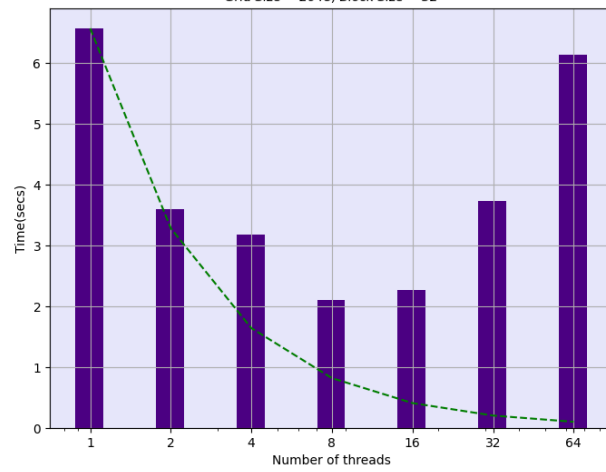


**{N = 2048}**



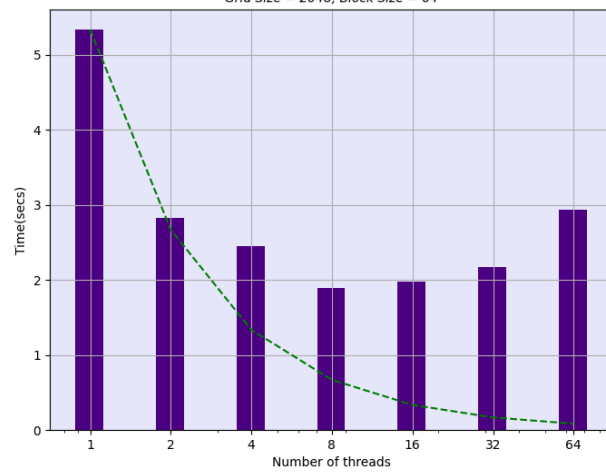
Total Times for Floyd-Warshall Recursive

Grid Size = 2048, Block Size = 32



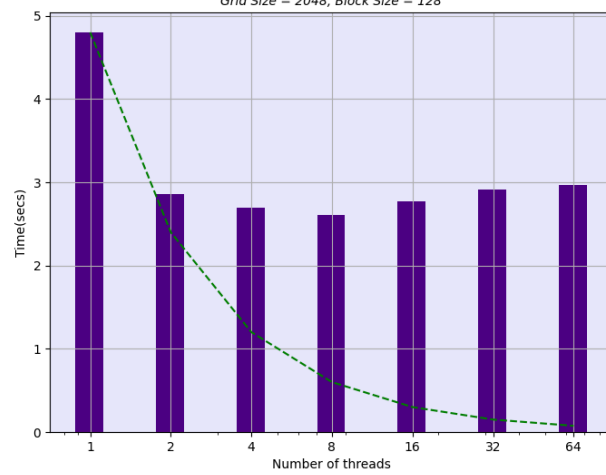
Total Times for Floyd-Warshall Recursive

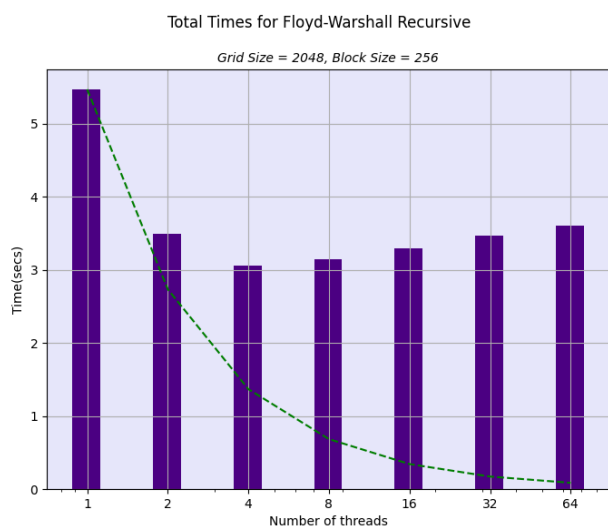
Grid Size = 2048, Block Size = 64



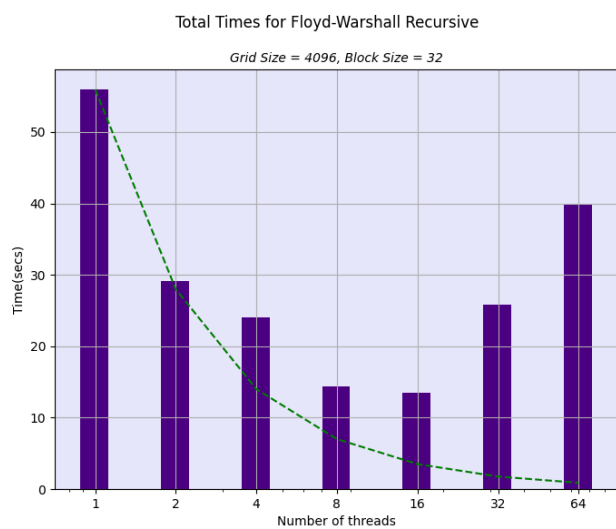
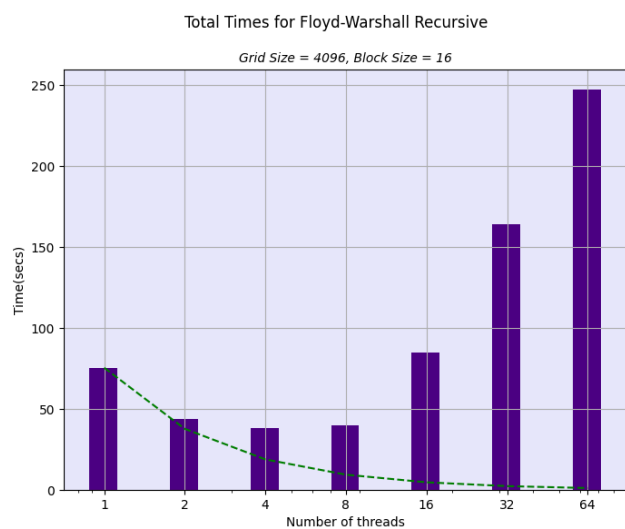
Total Times for Floyd-Warshall Recursive

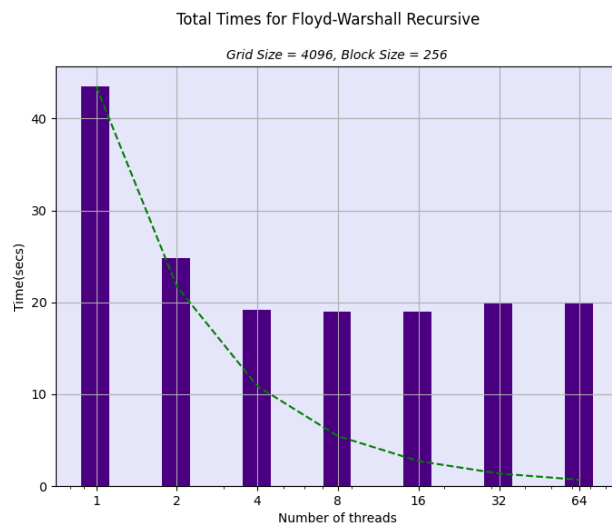
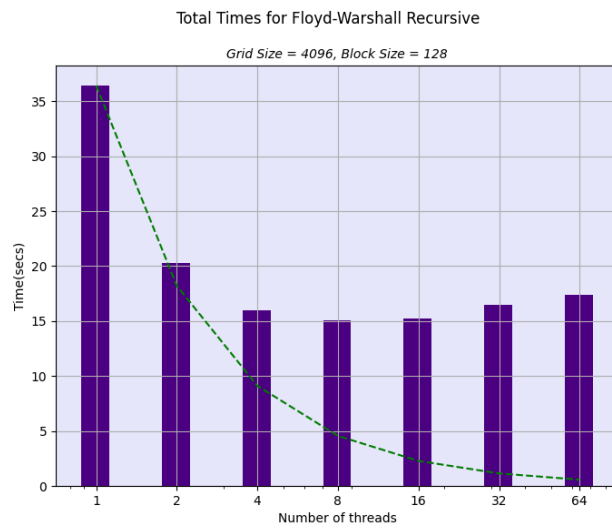
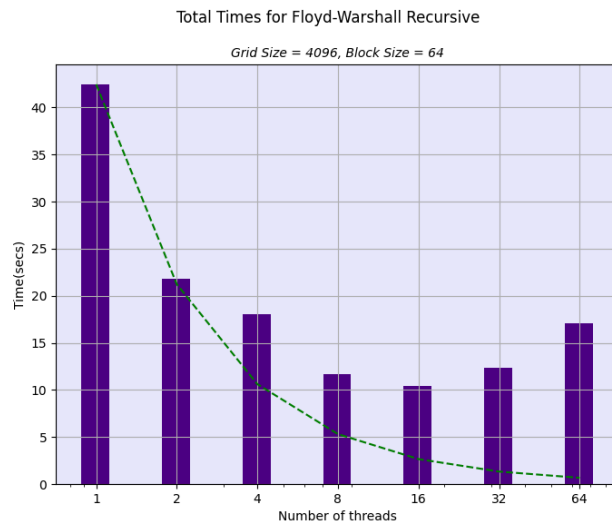
Grid Size = 2048, Block Size = 128





**{ 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 k-οστό 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,%.4f\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);
113

```

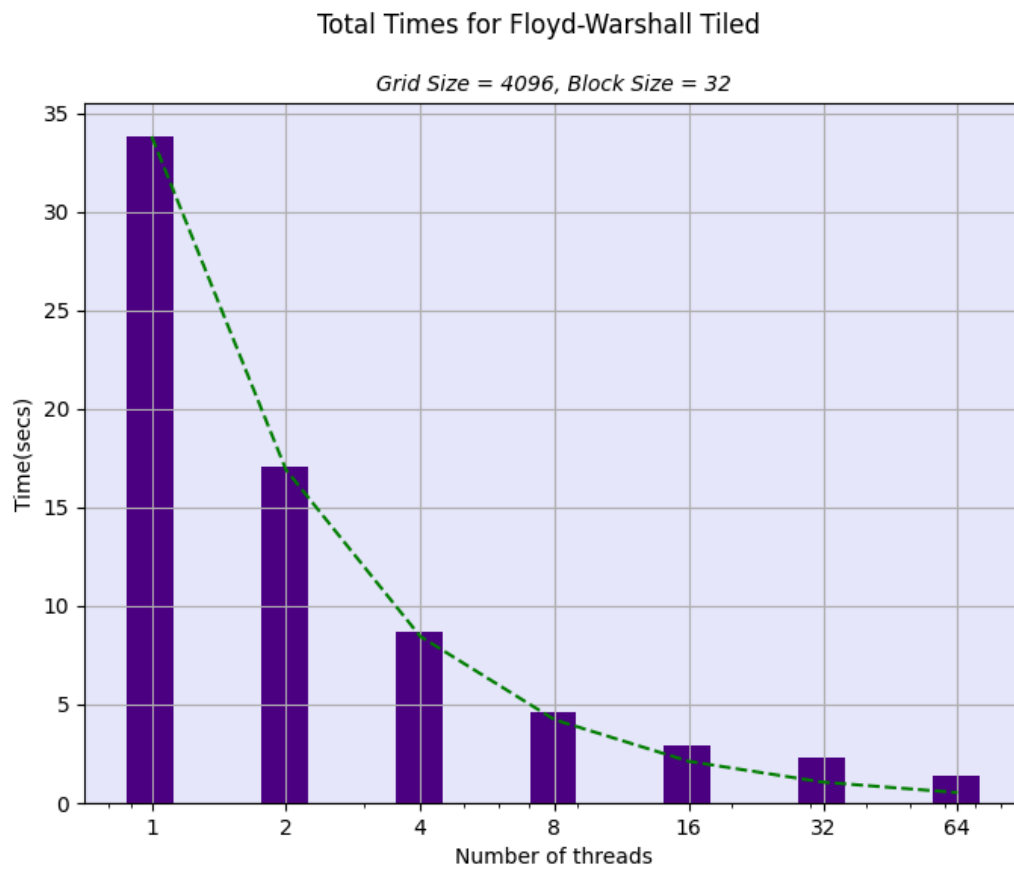


```

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

```

## Αποτελέσματα



Βλέπουμε πως υπάρχει τέλεια κλιμάκωση με αύξηση των threads.

## Παράρτημα

Για την δημιουργία των γραφικών παραστάσεων χρησιμοποιήθηκε 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 =
'oblique', size = 10)
42         plt.xlabel("Number of threads")
43         plt.ylabel("Time(secs)")
44         plt.grid('y')
45         plt.savefig(f"fig{GridSize[grid]}_{BSizes[bsize]}.png")
46         plt.clf()

```