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

```c
/*****************************************************
 ************* Conway's game of life ******************
 *****************************************************

 Usage: ./exec ArraySize TimeSteps

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


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

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

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

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

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

  /*Read input arguments*/
  if ( argc != 3 ) {
    fprintf(stderr, "Usage: ./exec ArraySize TimeSteps\n");
    exit(-1);
  }
  else {
    N = atoi(argv[1]);
    T = atoi(argv[2]);
  }

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

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

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

  /*Game of Life*/

  gettimeofday(&ts,NULL);
  for ( t = 0 ; t < T ; t++ ) {
        #pragma omp parallel for  shared(current, previous) private (nbrs, i, j)
    for ( i = 1 ; i < N-1 ; i++ ) {
```

```
      for ( j = 1 ; j < N-1 ; j++ ) {
        nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
               + previous[i][j-1] + previous[i][j+1] \
               + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
        if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
          current[i][j]=1;
        else
          current[i][j]=0;
      }
    }

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

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

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

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

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

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

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

  }
}

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

```
    free(s);
}
```

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
```

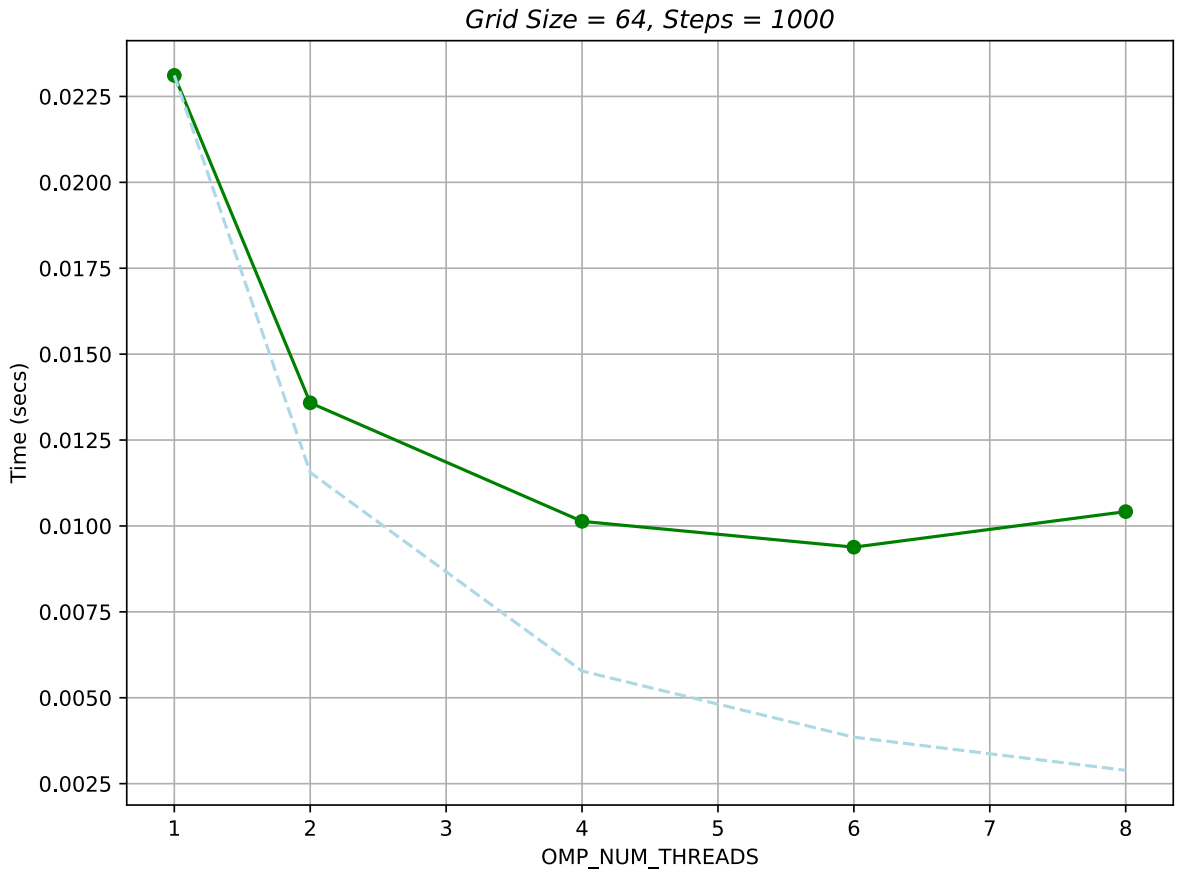
```
#!/bin/bash

## 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:

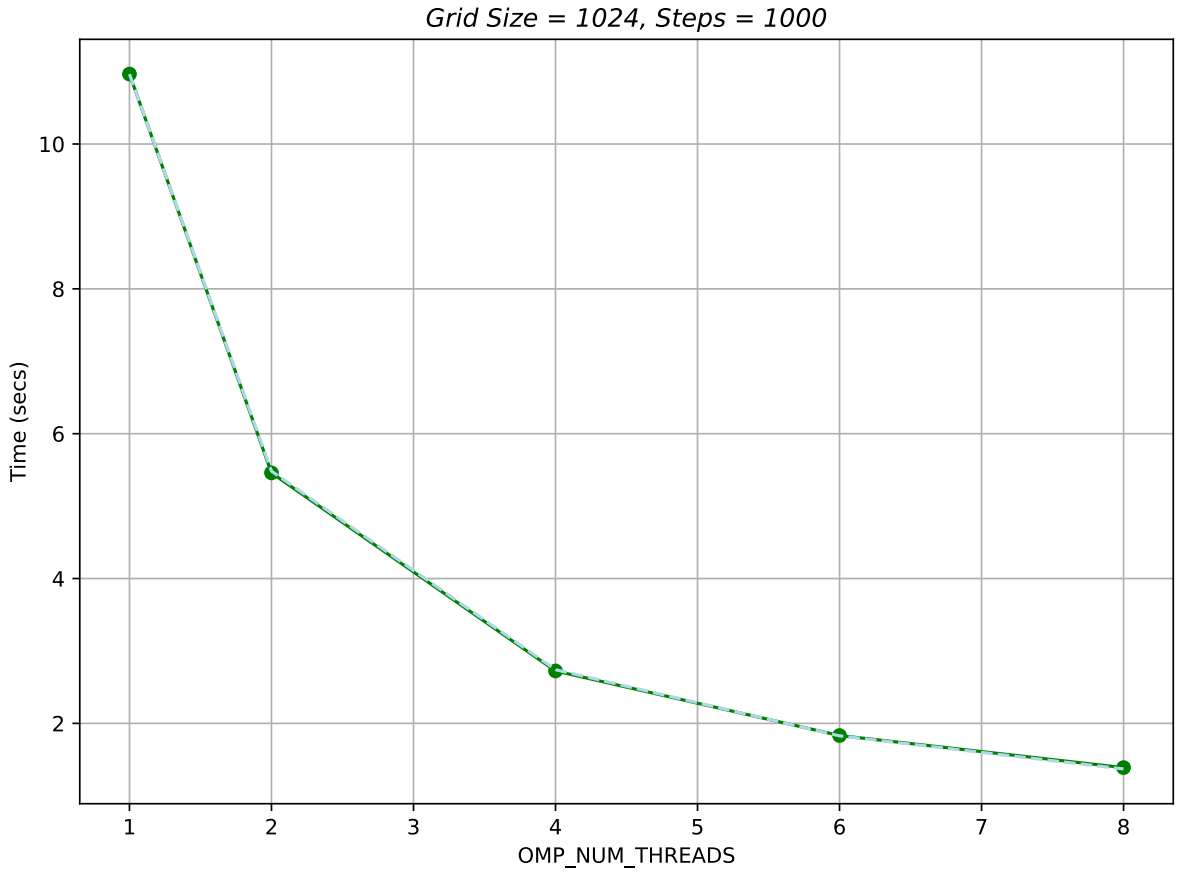
```
Running with OMP_NUM_THREADS=1
GameOfLife: Size 64 Steps 1000 Time 0.023112
GameOfLife: Size 1024 Steps 1000 Time 10.965944
GameOfLife: Size 4096 Steps 1000 Time 175.900314
Finished run with OMP_NUM_THREADS=1
-------------------------------------------------
Running with OMP_NUM_THREADS=2
GameOfLife: 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
-------------------------------------------------
```

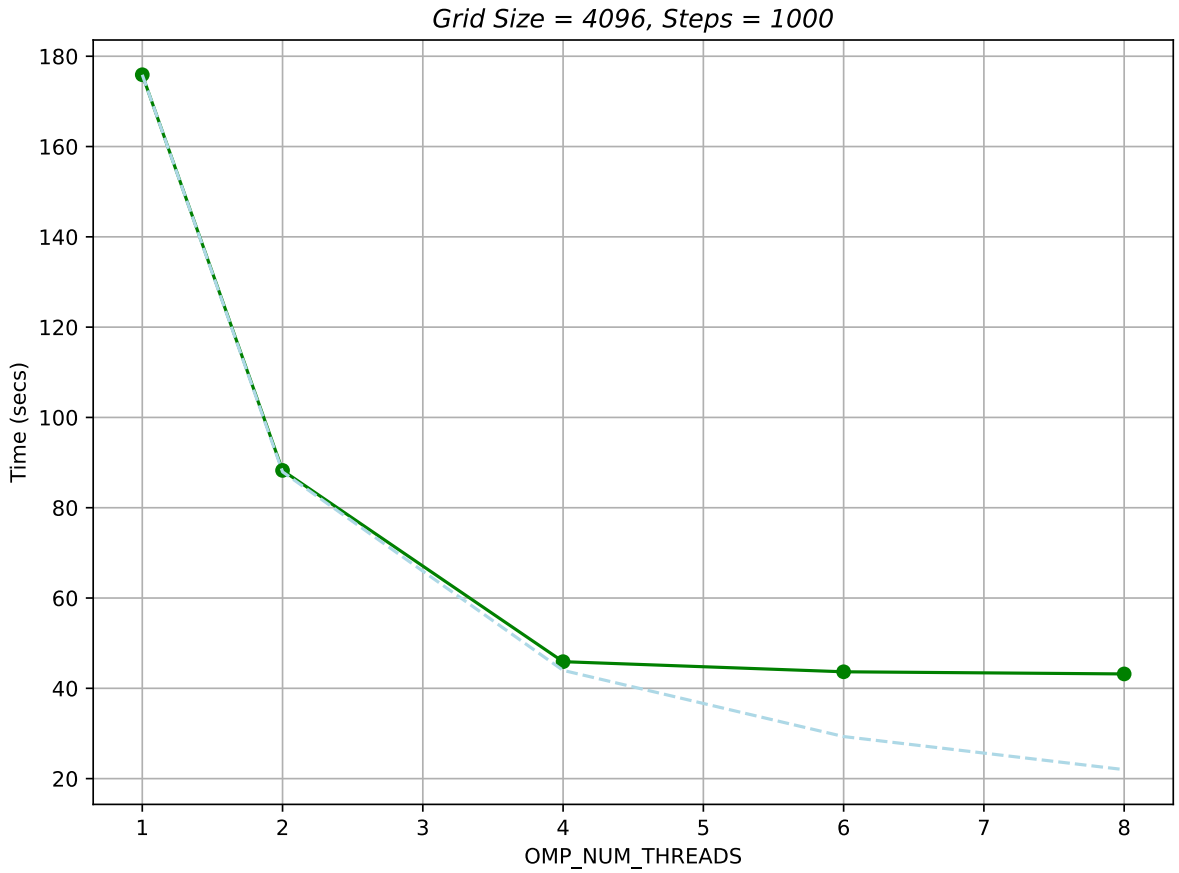
Γραφική Απεικόνιση και Παρατηρήσεις



Παρατηρούμε ότι για μικρό μέγεθος grid (με συνολική απαίτηση μνήμης 4*64*64bytes = 16KB), 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).



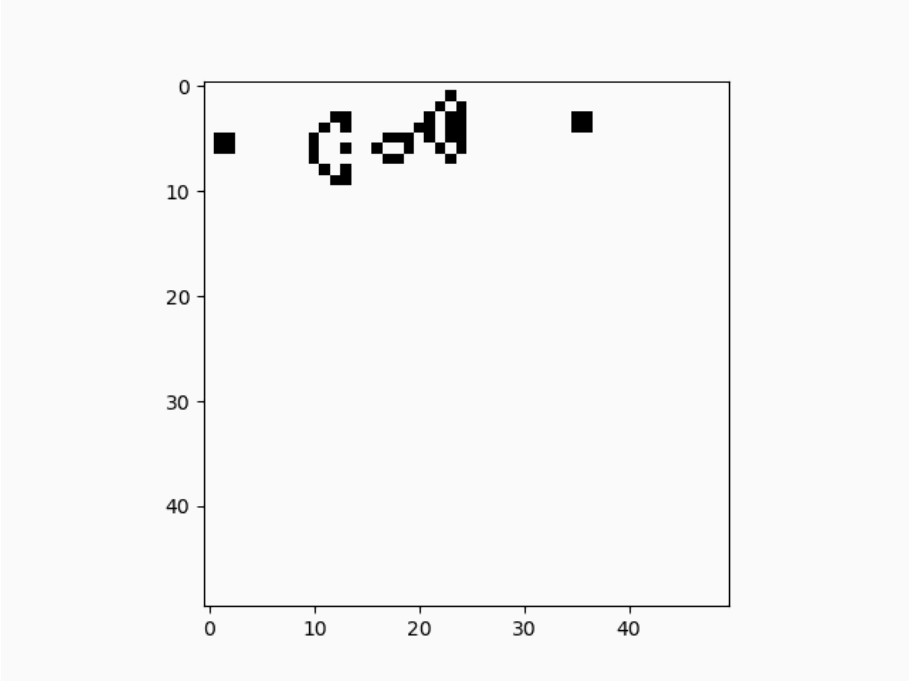
Για μέγεθος grid με συνολική απαίτηση μνήμης 4*1024*1024 bytes = 4MB, η επίδοση βελτίωνεται ομοιόμορφα και ανάλογα με το μέγεθος των νημάτων . Εικάζουμε, λοιπόν, πως η cache χωράει ολόκληρο το grid ώστε το κάθε νήμα δεν επιβαρύνει την μνήμη με loads των αντίστοιχων rows, ο φόρτος εργασίας είναι ισομοιρασμένος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



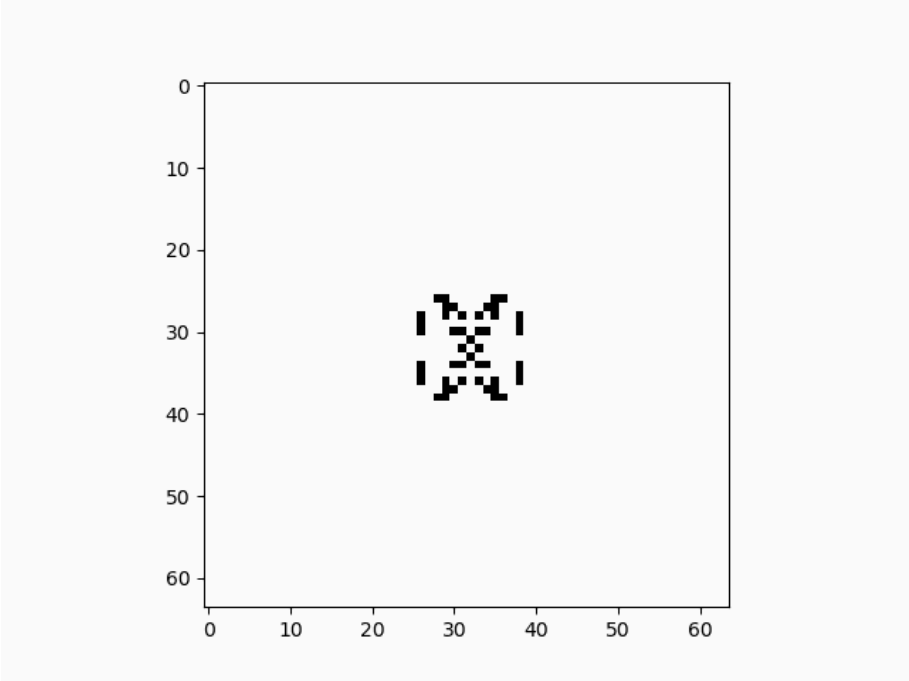
Για μεγάλο grid (με συνολική απαίτηση μνήμης 4*4096*4096 bytes = 64MB), η κλιμάκωση παύει να υφίσταται για περισσότερα από 4 νήματα. Bottleneck κόστους εδώ θεωρούμε το memory bandwidth. Επειδή ολόκληρο το grid δεν χωράει στην cache, δημιουργούνται misses όταν ξεχωριστά νήματα προσπαθούν να διαβάσουν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

```python
import numpy as np
import matplotlib.pyplot as plt
import re
import sys

outfile = "omp_gameoflife_all.out"

thread_pattern = r"Running with OMP_NUM_THREADS=(\d+)"
time_pattern = r"GameOfLife: Size (\d+) Steps 1000 Time ([\d.]+)"

with open (outfile, 'r') as fout:
    data = fout.read()

thread_vals = re.findall(thread_pattern, data)
time_vals = re.findall(time_pattern, data)

#print(thread_vals, time_vals)

results_mapping = {}

for i in range(0, len(thread_vals)):
    omp_num_thredas = int(thread_vals[i])

    for j in range(0,3):
        size = int(time_vals[i*3+j][0])
        time = float(time_vals[i*3+j][1])
        ## print(f"From {i,j} extracted size: {size} with time: {time}")

        if size not in results_mapping :
            results_mapping[size] = {}

        results_mapping[size][omp_num_thredas] = time

for idx, (size, omp_times) in enumerate(results_mapping.items()):
    print(f"Size: {size}, results: {omp_times}")

    # Create a new figure for each graph
    plt.figure(figsize=(8, 6))

    # Plot the original times
    plt.plot(omp_times.keys(), omp_times.values(), color='g', marker='o')

    # Plot the inverse times
    plt.plot(omp_times.keys(), [omp_times[1] / i for i in omp_times.keys()], color='lightblue', linestyle='--')

    # Add labels and title
    plt.title(f"Grid Size = {size}, Steps = 1000", fontstyle='oblique', size=12)
    plt.xlabel("OMP_NUM_THREADS")
    plt.ylabel("Time (secs)")
    plt.grid()

    # Show the plot
    plt.tight_layout()
    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). Ωστόσω, χρησιμοποιούμε atomic για ορθότητα της τιμής του και για παρατήρηση με βάση το μεγαλύτερο δυνατό overhead.

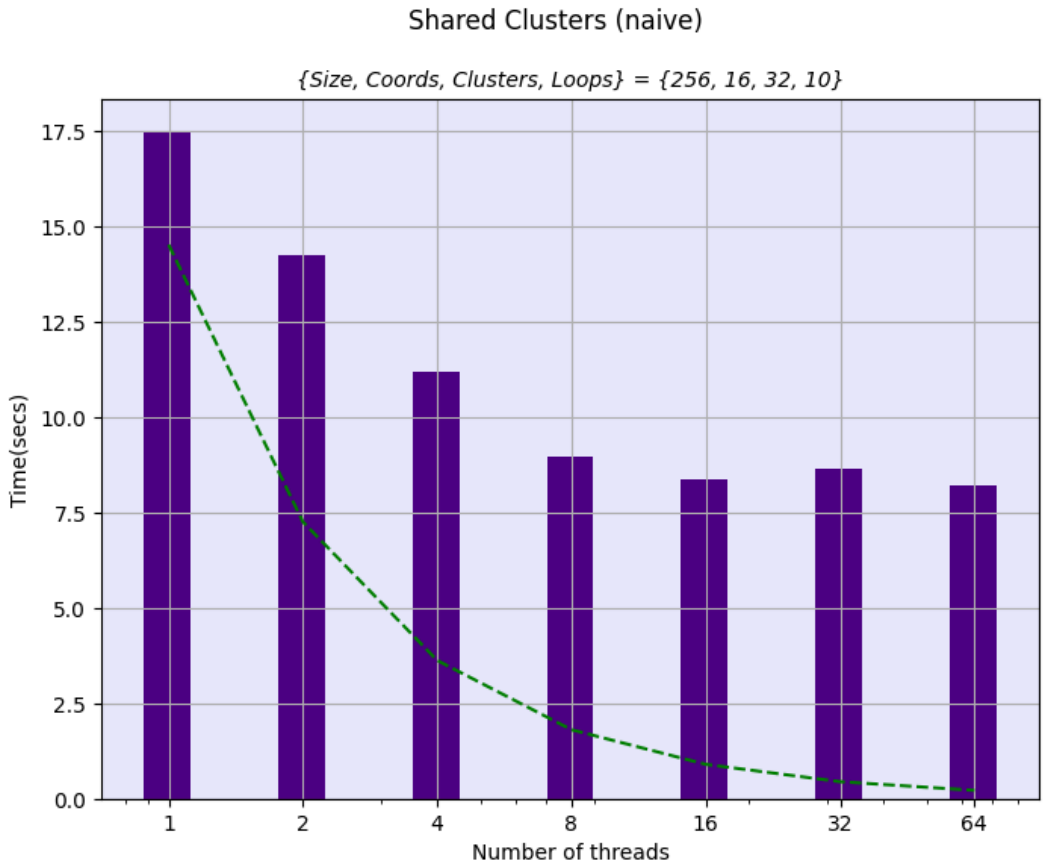
omp_naive_kmeans.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "kmeans.h"
/*
 * TODO: include openmp header file
 */

// square of Euclid distance between two multi-dimensional points
inline static double euclid_dist_2(int    numdims,  /* no. dimensions */
                  double * coord1,   /* [numdims] */
                  double * coord2)   /* [numdims] */
{
  int i;
  double ans = 0.0;

  for(i=0; i<numdims; i++)
    ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);

  return ans;
}

inline static int find_nearest_cluster(int      numClusters, /* no. clusters */
                      int      numCoords,   /* no. coordinates */
                      double * object,      /* [numCoords] */
                      double * clusters)    /* [numClusters][numCoords] */
{
  int index, i;
  double dist, min_dist;

  // find the cluster id that has min distance to object
  index = 0;
  min_dist = euclid_dist_2(numCoords, object, clusters);

  for(i=1; i<numClusters; i++) {
    dist = euclid_dist_2(numCoords, object, &clusters[i*numCoords]);
    // no need square root
    if (dist < min_dist) { // find the min and its array index
      min_dist = dist;
      index    = i;
    }
  }
  return index;
}

void kmeans(double * objects,          /* in: [numObjs][numCoords] */
      int      numCoords,        /* no. coordinates */
      int      numObjs,          /* no. objects */
      int      numClusters,      /* no. clusters */
      double   threshold,        /* minimum fraction of objects that change membership */
      long     loop_threshold,   /* maximum number of iterations */
      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,
    membership) schedule(static)
90      for (i=0; i<numObjs; i++) {
91        // find the array index of nearest cluster center
92        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
93
94        // if membership changes, increase delta by 1
95        if (membership[i] != index)
96          #pragma omp atomic
97          delta += 1.0;
98
99        // assign the membership to object i
100       membership[i] = index;
101
102       // update new cluster centers : sum of objects located within
103       /*
104        * TODO: protect update on shared "newClusterSize" array
105        */
106            #pragma omp atomic
107       newClusterSize[index]++;
108       for (j=0; j<numCoords; j++)
109         /*
110          * TODO: protect update on shared "newClusters" array
111          */
112                 #pragma omp atomic
113         newClusters[index*numCoords + j] += objects[i*numCoords + j];
114     }
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
    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   }
```

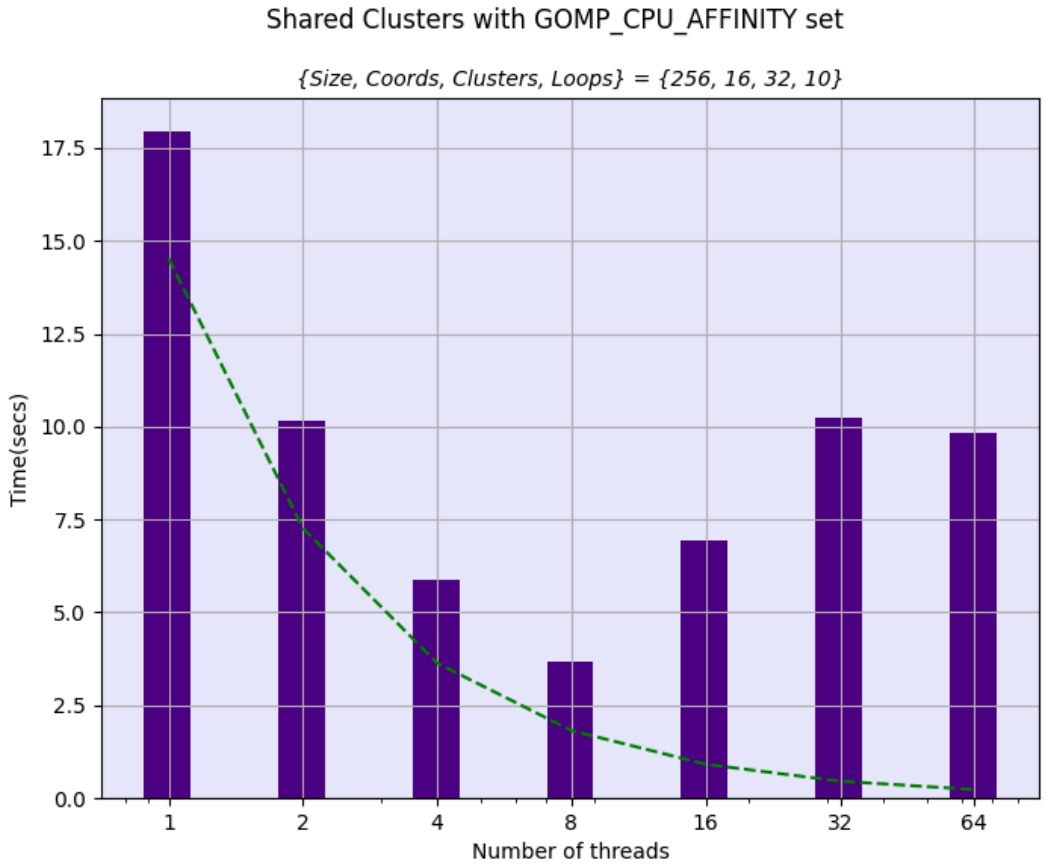
Απεικονίζουμε παρακάτω τα αποτελέσματα των δοκιμών στον sandman για τις διάφορες τιμές της environmental variable OMP_NUM_THREADS:



Παρατηρούμε πως ο αλγόριθμος δεν κλιμακώνει καθόλου καλά από 8 και πάνω νήματα 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 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 τροποποίησαμε κατάλληλα το script υποβολής στον sandman και προσθέσαμε την παράμετρο schedule static στο parallel for. Τα αποτελέσματα παρουσιάζονται παρακάτω:

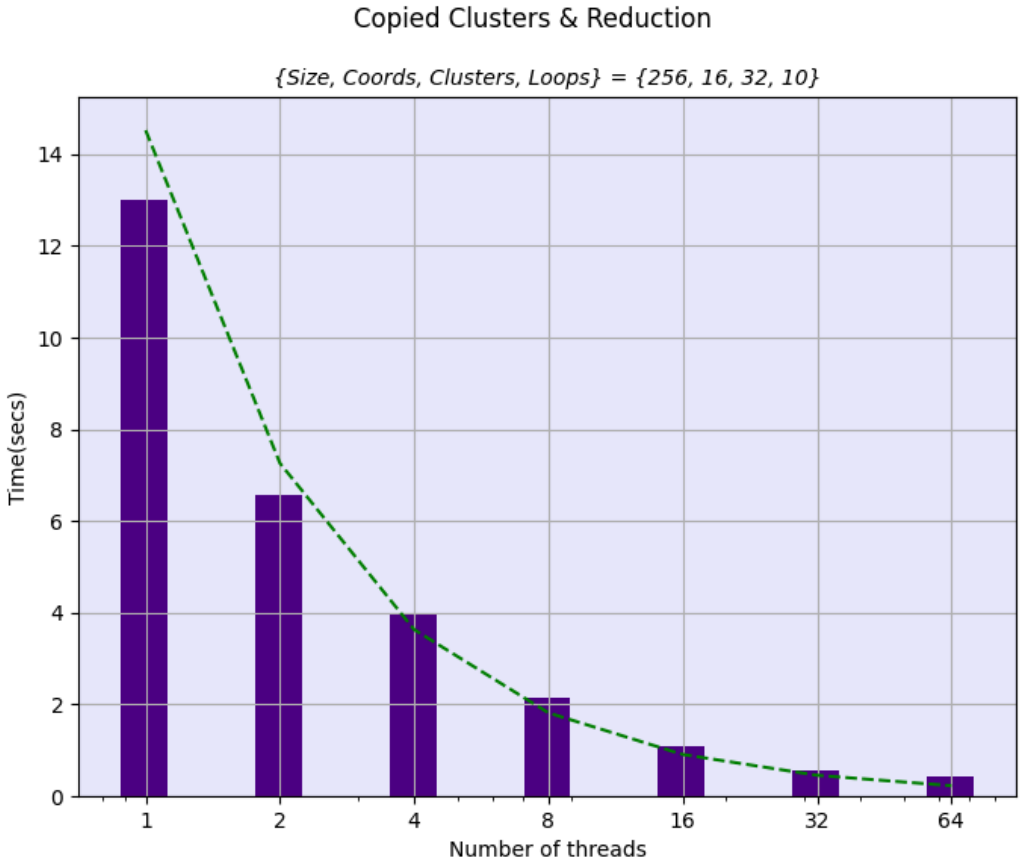


Παρατηρούμε σημαντική βελτίωση στην κλιμάκωση μέχρι 8 νήματα όμως μετά σταματάει να κλιμακώνει ο αλγόριθμος λόγω της δομής που έχει ο sandman. Για 16 νήματα και πάνω δεν μπορούμε να τα κάνουμε pin στο ίδιο cluster οπότε δεν μοιράζονται τα νήματα την ίδια L3 cache και υπάρχει συνεχής μεταφορά δεδομένων των shared πινάκων και bus invalidations λόγω του cache coherence protocol. Ακόμη τα L3 misses κοστίζουν ξεχωριστά για κάθε cluster. Εαν 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 κλιμάκβση σταματάει. Παραθέτουμε την βελτιωμένη εκδοχή των 16 νημάτων ακολούθως:

2) Copied Clusters & Reduce

Υλοποίηση

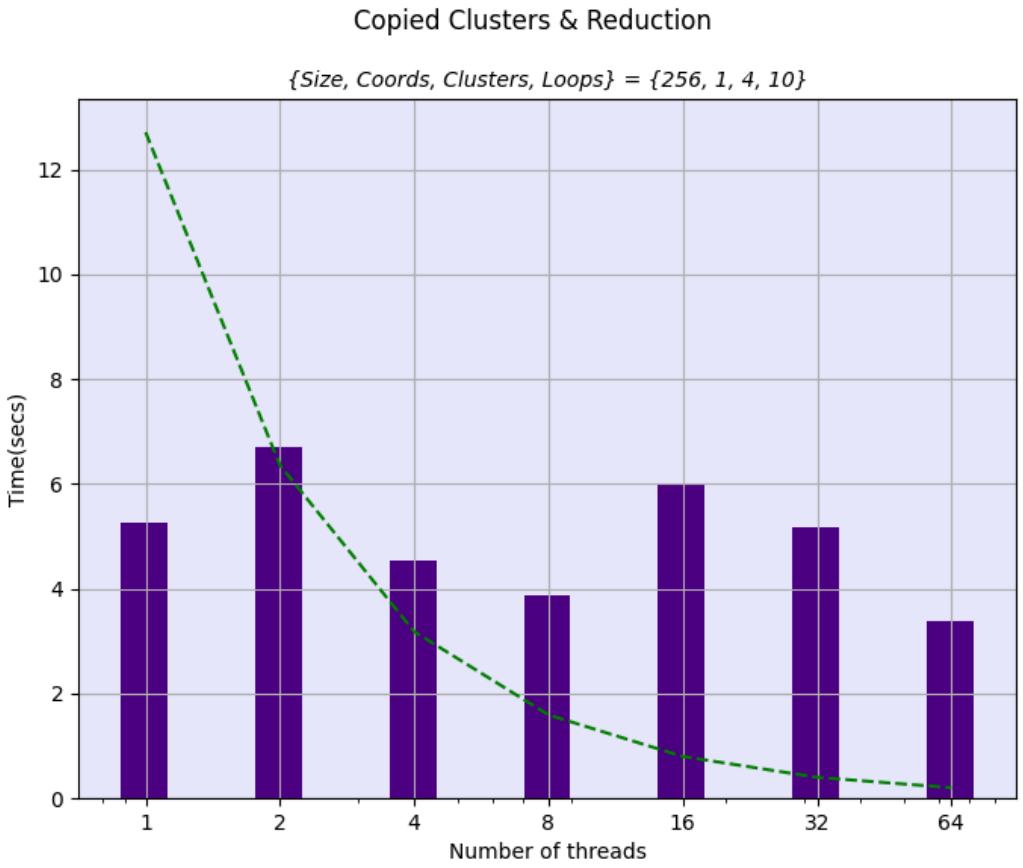
Μοιράζουμε σε κάθε νήμα ένα διαφορετικό τμήμα των πινάκων newClusters, newClusterSize οπότε τα δεδομένα γίνονται private, δεν υπάρχουν race conditions αλλά απαιτείται reduction (με πρόσθεση) στο τέλος για το τελικό αποτέλεσμα (η οποία πραγματοποιείται εδώ από 1 νήμα). Τα αποτελέσματα φαίνονται εδω :



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:



Κυριαρχο ρόλο για αυτην 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.

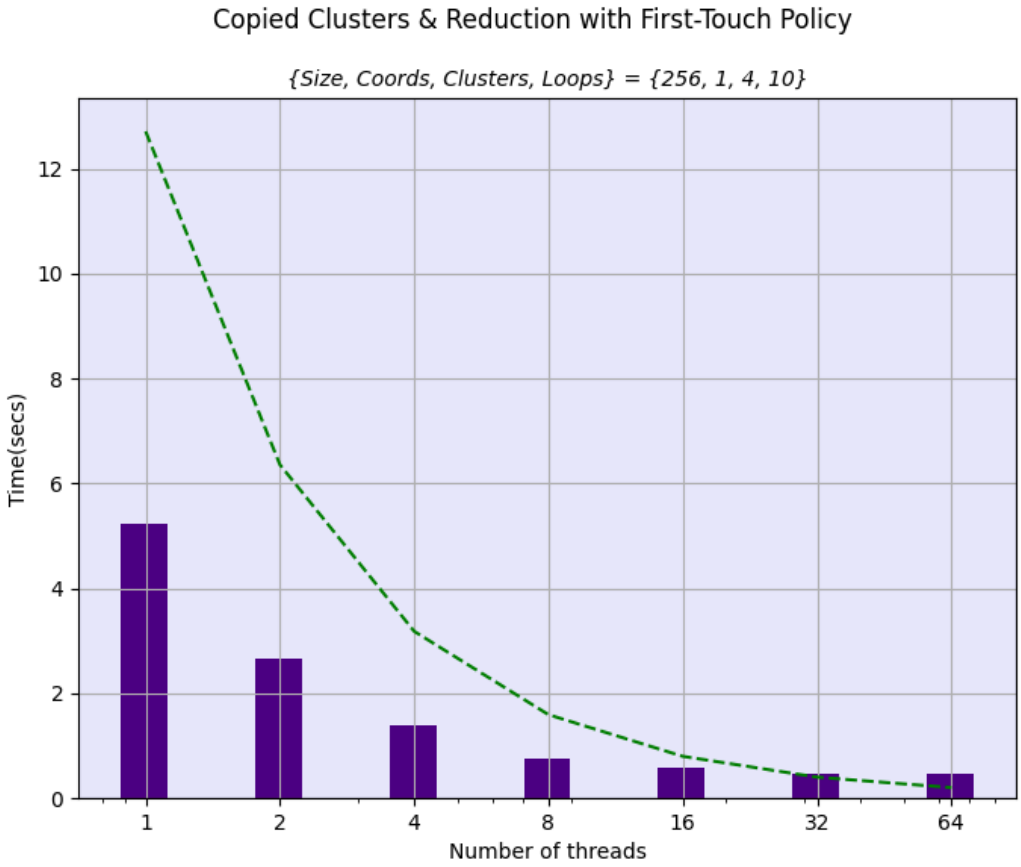
First-touch Policy

Προς αποφυγή των παραπάνω εκμεταλλευόμαστε την πολιτική των linux κατά το mapping των virtual με physical addresses. Η δέσευση 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

...

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!

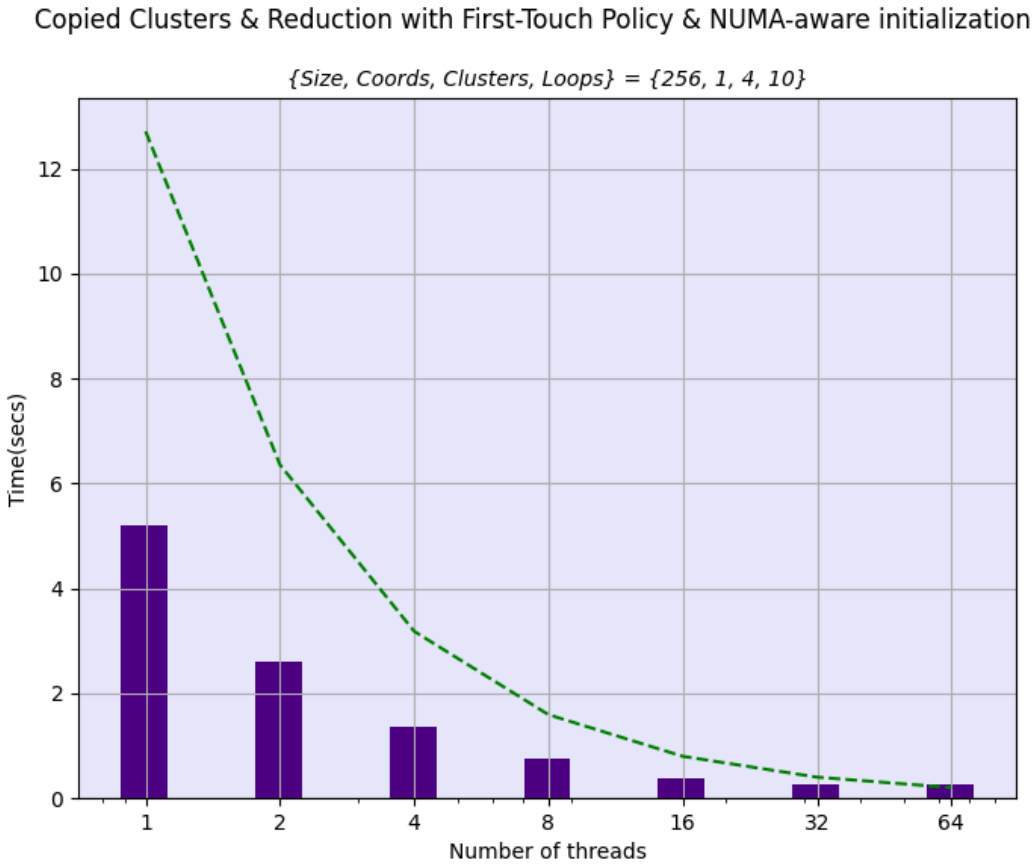
Numa-aware initialization

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 μεταφέρνουν από την κύρια μνήμη ή από άλλα νήματα.

Υλοποίηση

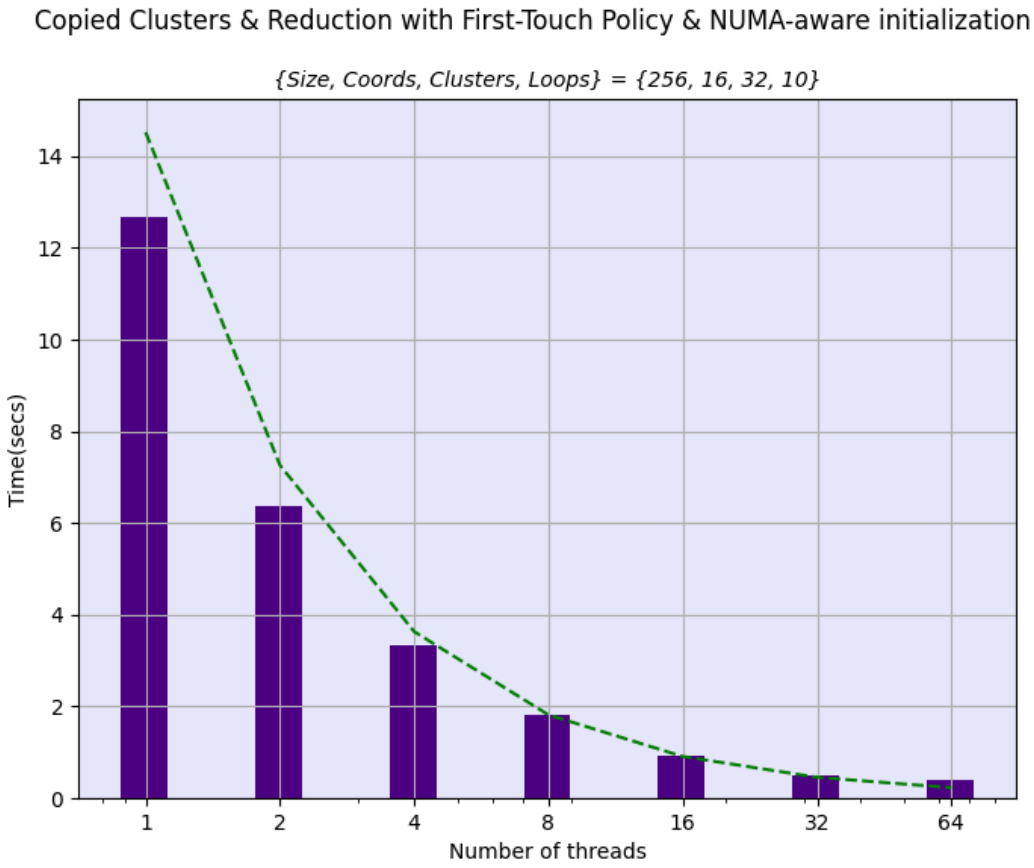
...

Αποτελέσματα



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:



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