



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 :

```

1  #!/bin/bash
2
3  ## Give the Job a descriptive name
4  #PBS -N make_gameoflife
5
6  ## Output and error files
7  #PBS -o make_gameoflife.out
8  #PBS -e make_gameoflife.err
9
10 ## How many machines should we get?
11 #PBS -l nodes=1:ppn=1
12
13 ## Start
14 ## Run make in the src folder (modify properly)
15
16 module load openmpi/1.8.3
17 cd /home/parallel/parlab09/a1
18 make

```

```

1  #!/bin/bash
2
3  ## Give the Job a descriptive name
4  #PBS -N run_gameoflife
5
6  ## Output and error files
7  #PBS -o omp_gameoflife_all.out
8  #PBS -e omp_gameoflife_all.err
9
10 ## Limit memory, runtime etc.
11 #PBS -l walltime=01:00:00
12
13 ##Number of nodes aka threads
14 #PBS -l nodes=1:ppn=8
15
16 module load openmpi/1.8.3
17 cd /home/parallel/parlab09/a1
18
19 for threads in 1 2 4 6 8
20 do
21     export OMP_NUM_THREADS=$threads
22     echo "Running with OMP_NUM_THREADS=$OMP_NUM_THREADS"
23     ./omp_gameoflife 64 1000
24     ./omp_gameoflife 1024 1000
25     ./omp_gameoflife 4096 1000
26     echo "Finished run with OMP_NUM_THREADS=$OMP_NUM_THREADS"
27     echo "-----"
28 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

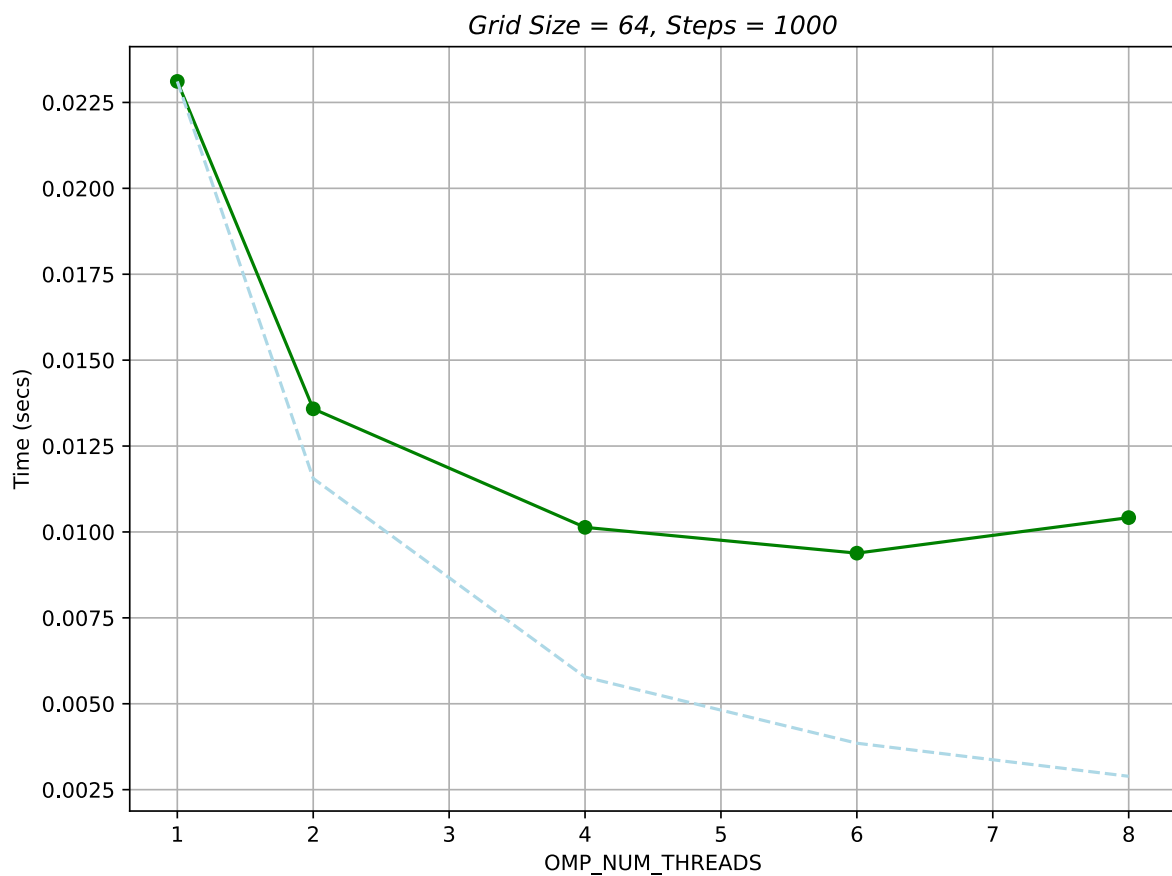
```

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

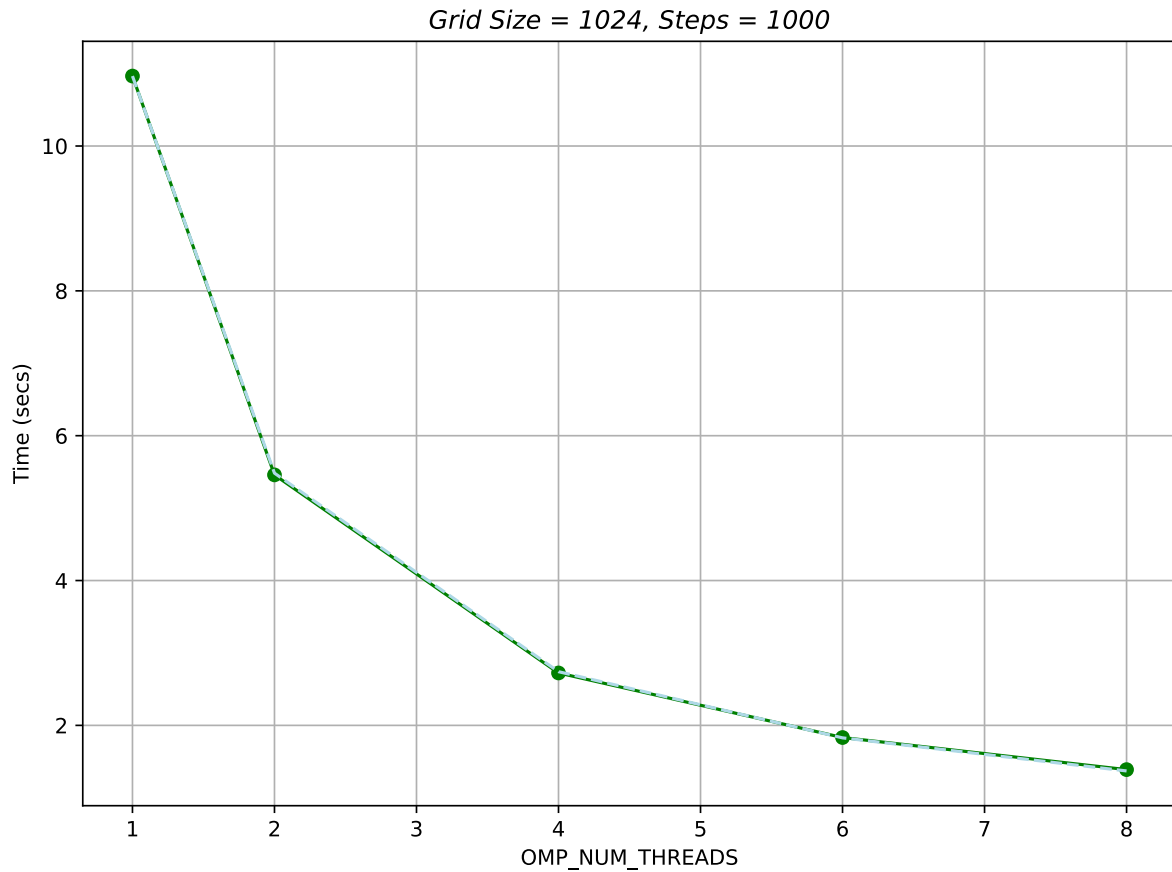
Running with OMP_NUM_THREADS=6
GameOfLife: Size 64 Steps 1000 Time 0.009383
GameOfLife: Size 1024 Steps 1000 Time 1.832227
GameOfLife: Size 4096 Steps 1000 Time 43.661123
Finished run with OMP_NUM_THREADS=6

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

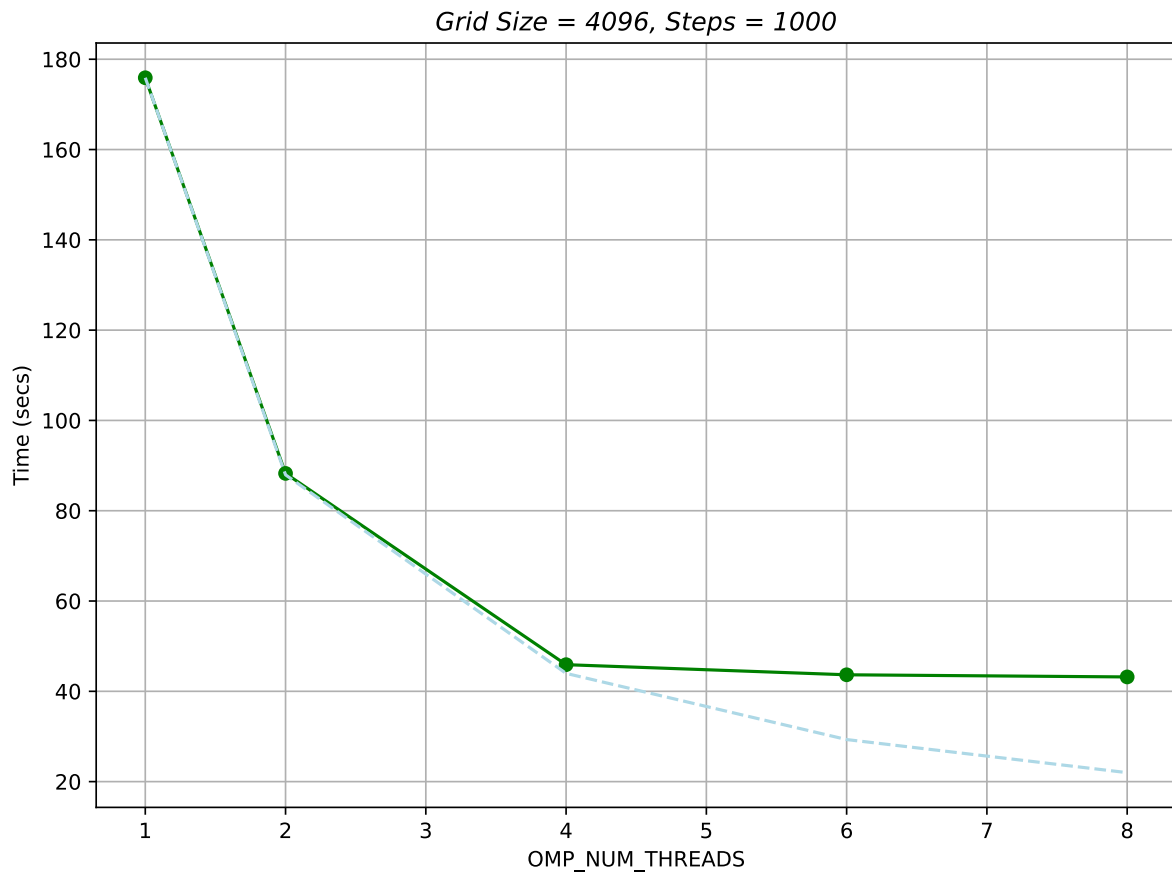
Γραφική Απεικόνιση και Παρατηρήσεις



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).



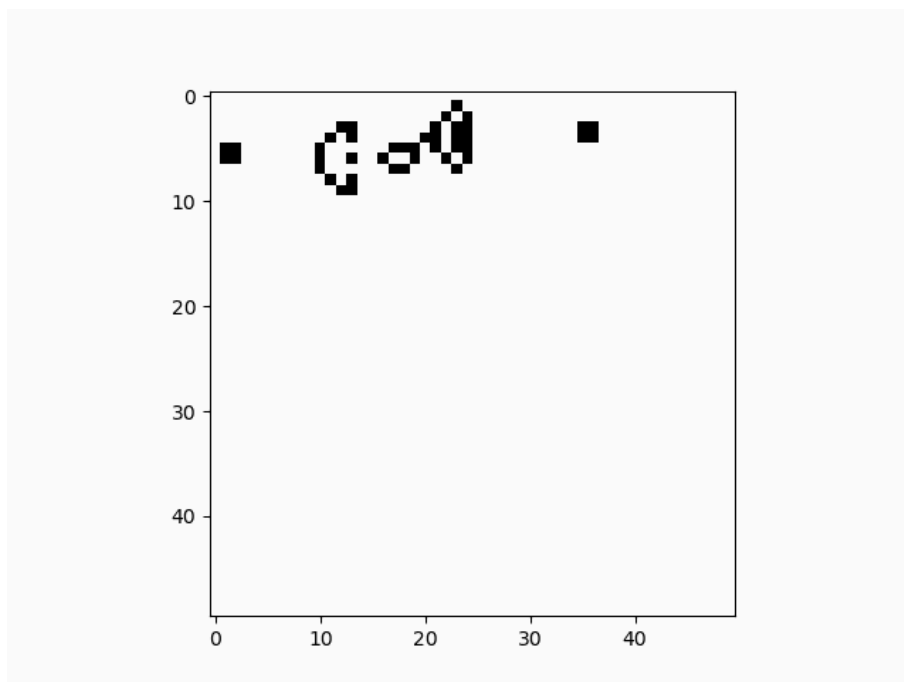
Για μέγεθος grid με συνολική απαίτηση μνήμης $4*1024*1024\text{ bytes} = 4\text{MB}$, η επίδοση βελτιώνεται ομοιόμορφα και ανάλογα με το μέγεθος των νημάτων. Εικάζουμε, λοιπόν, πως η cache χωράει ολόκληρο το grid ώστε το κάθε νήμα δεν επιβαρύνει την μνήμη με loads των αντίστοιχων rows, ο φόρτος εργασίας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



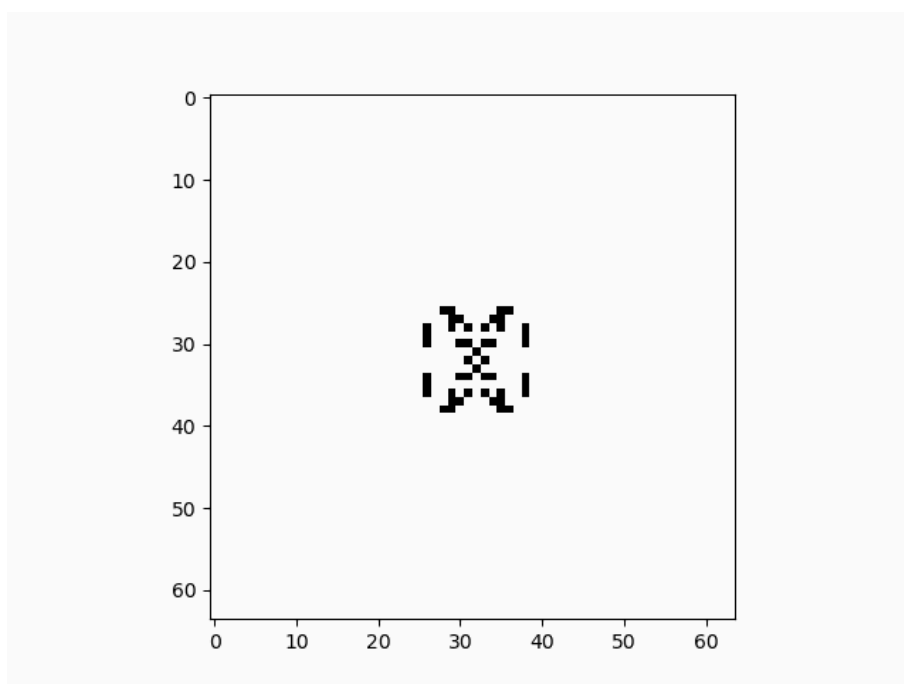
Για μεγάλο grid (με συνολική απαίτηση μνήμης $4 \cdot 4096 \cdot 4096$ bytes = 64MB), η κλιμάκωση παύει να υφίσταται για περισσότερα από 4 νήματα. Bottleneck κόστους εδώ θεωρούμε το memory bandwidth. Επειδή ολόκληρο το grid δεν χωράει στην cache, δημιουργούνται misses όταν ξεχωριστά νήματα προσπαθούν 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 τον 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          * TODO0: 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              * TODO0: protect update on shared "newClusterSize" array
107              */
108             #pragma omp atomic
109             newClusterSize[index]++;
110             for (j=0; j<numCoords; j++)
111                 /*
112                  * TODO0: 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 }

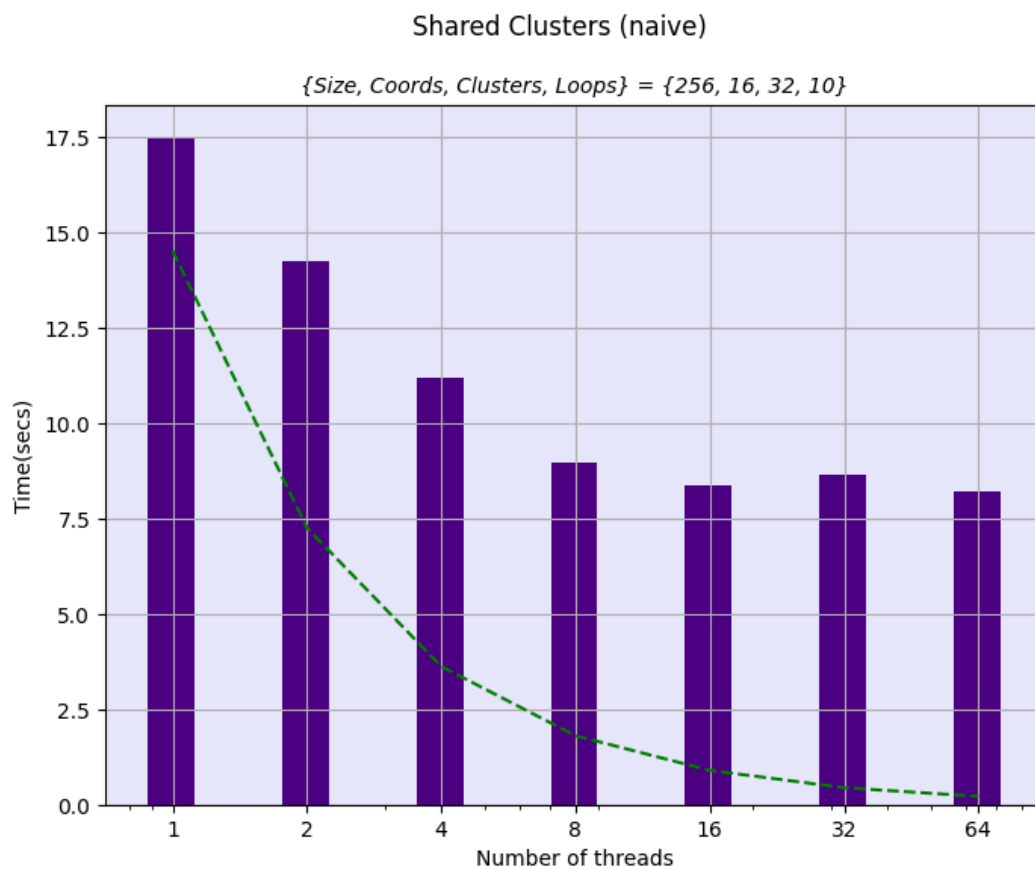
```

```

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 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.

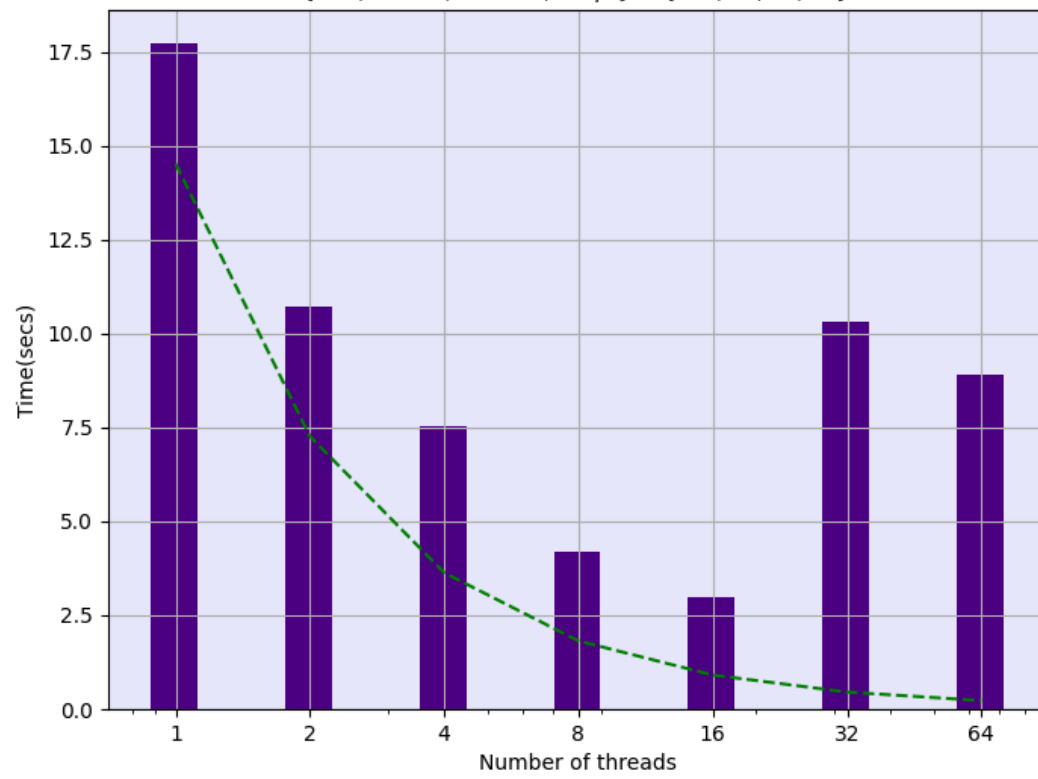
Για την υλοποίηση τροποποίησαμε κατάλληλα το script υποβολής στον sandman και προσθέσαμε την παράμετρο schedule static στο parallel for. Τα αποτελέσματα παρουσιάζονται παρακάτω:



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. 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]

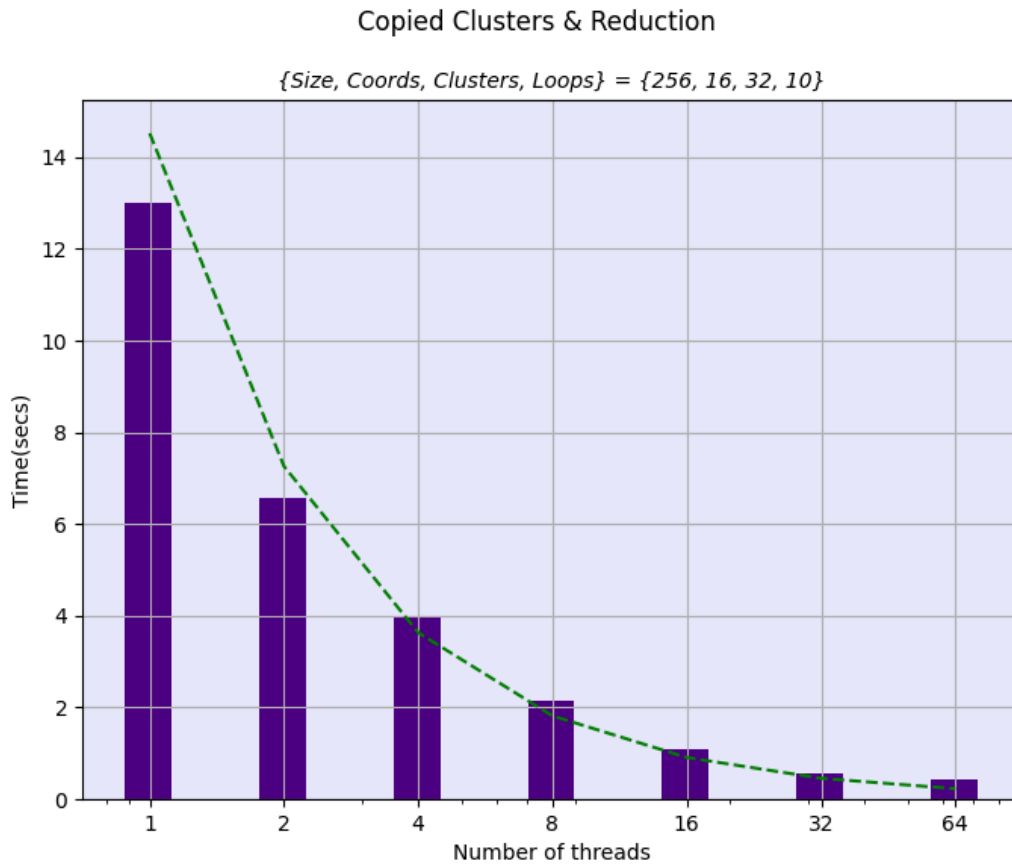
{Size, Coords, Clusters, Loops} = {256, 16, 32, 10}



2) Copied Clusters & Reduce

Υλοποίηση

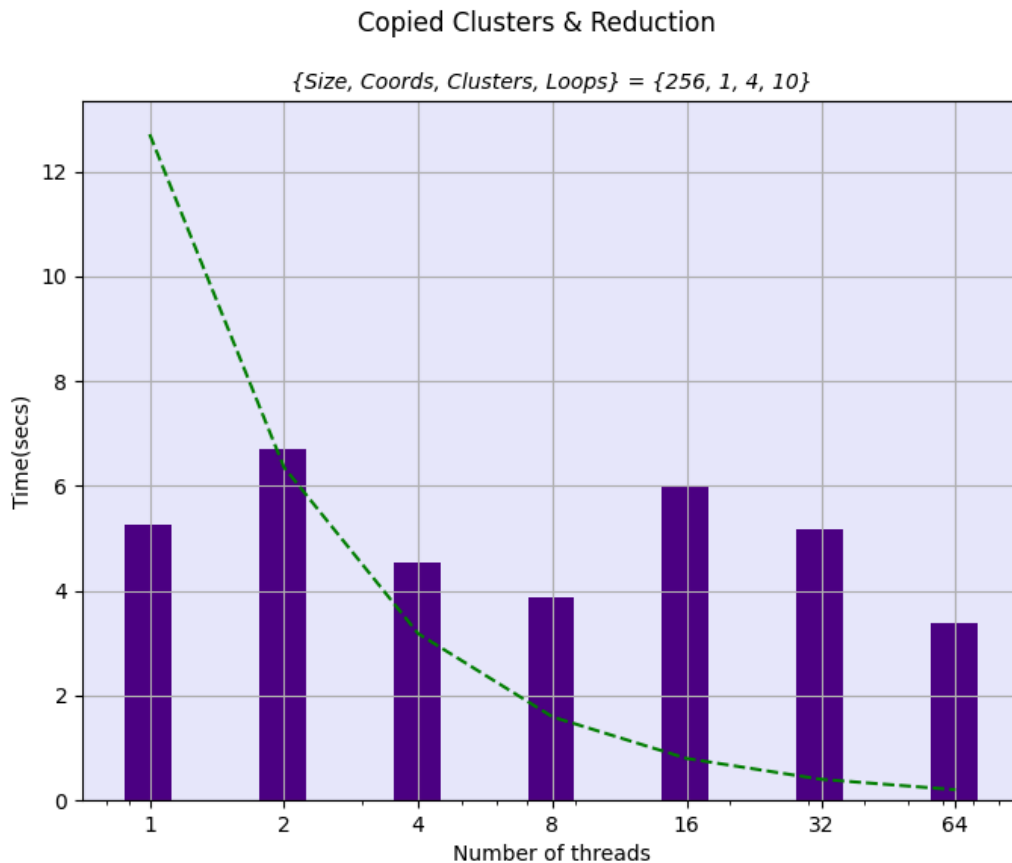
Μοιράζουμε σε κάθε νήμα ένα διαφορετικό τμήμα των πινάκων `newClusters`, `newClusterSize` οπότε τα δεδομένα γίνονται `private`, δεν υπάρχουν `race conditions` αλλά απαιτείται `reduction` (με πρόσθεση) στο τέλος για το τελικό αποτέλεσμα (η οποία πραγματοποιείται εδώ από 1 νήμα). Τα αποτελέσματα φαίνονται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



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.

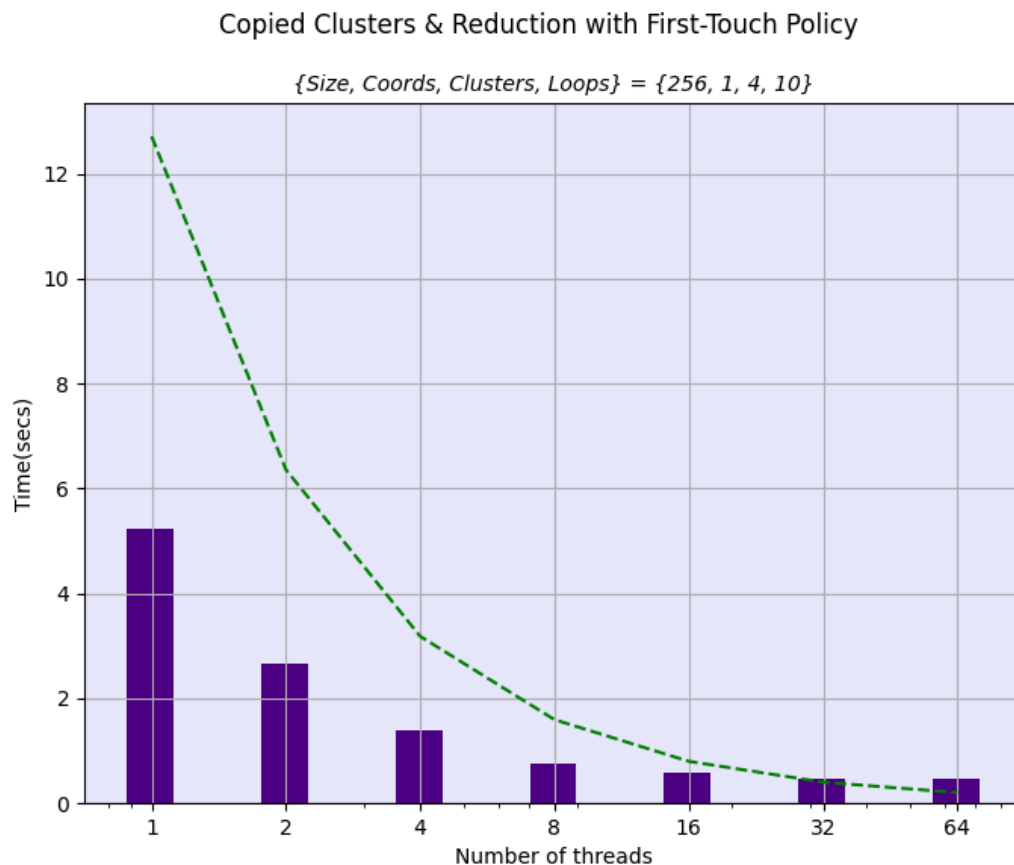
First-touch Policy

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

...

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

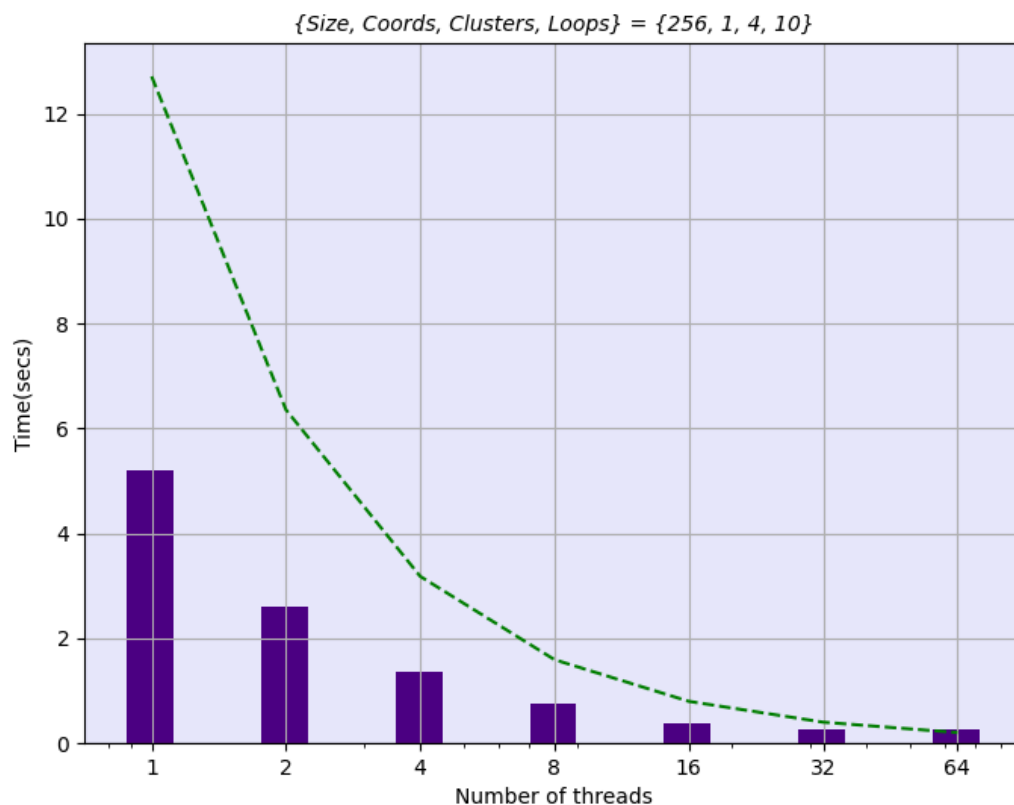
Με βάση όσα αναφέρθηκαν για το pinning σε cores και την πολιτική first-touch η αρχικοποίηση των shared πινάκων μπορεί να γίνει και αυτή ατομικά από κάθε νήμα σε ένα private τμήμα αυτού. Για την υλοποίηση προσθέτουμε το `omp parallel for` directive με στατική δρομολόγηση. Αυτή είναι απαραίτητη ώστε τα 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

...

Αποτελέσματα

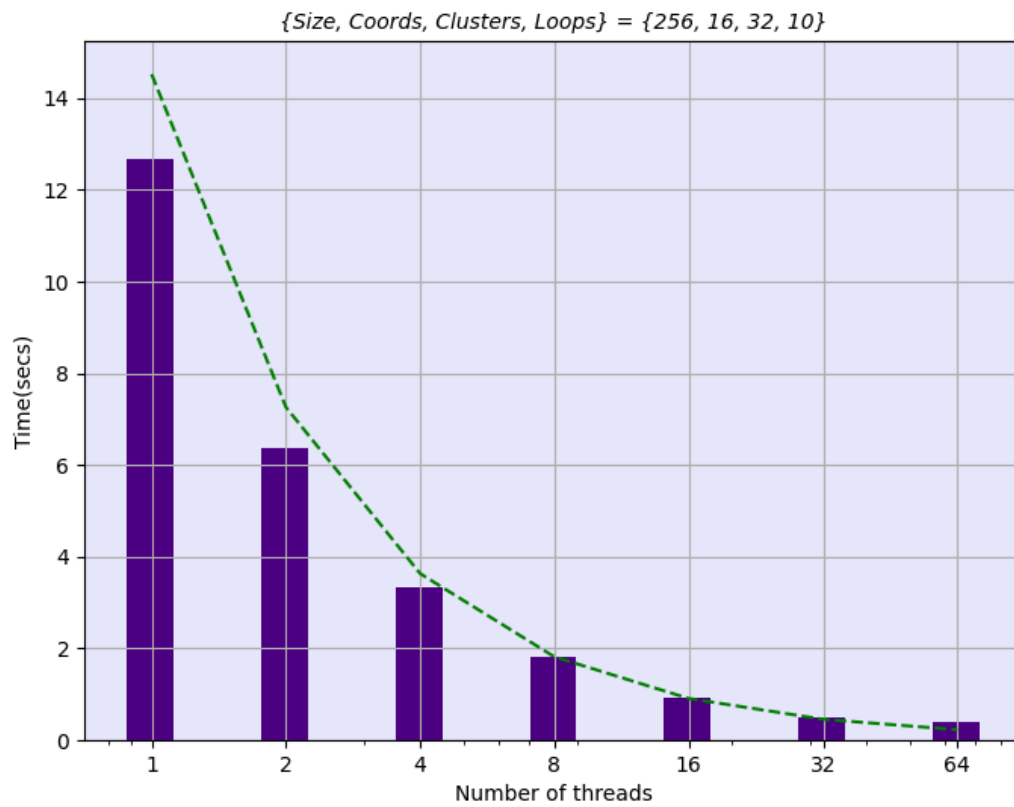
Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization



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.