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);
```

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

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

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_gameoflife

## Output and error files
#PBS -o make_gameoflife.out
#PBS -e make_gameoflife.err

## How many machines should we get?
#PBS -l nodes=1:ppn=1

## Start
## Run make in the src folder (modify properly)

module load openmpi/1.8.3
cd /home/parallel/parlab09/a1
make
```

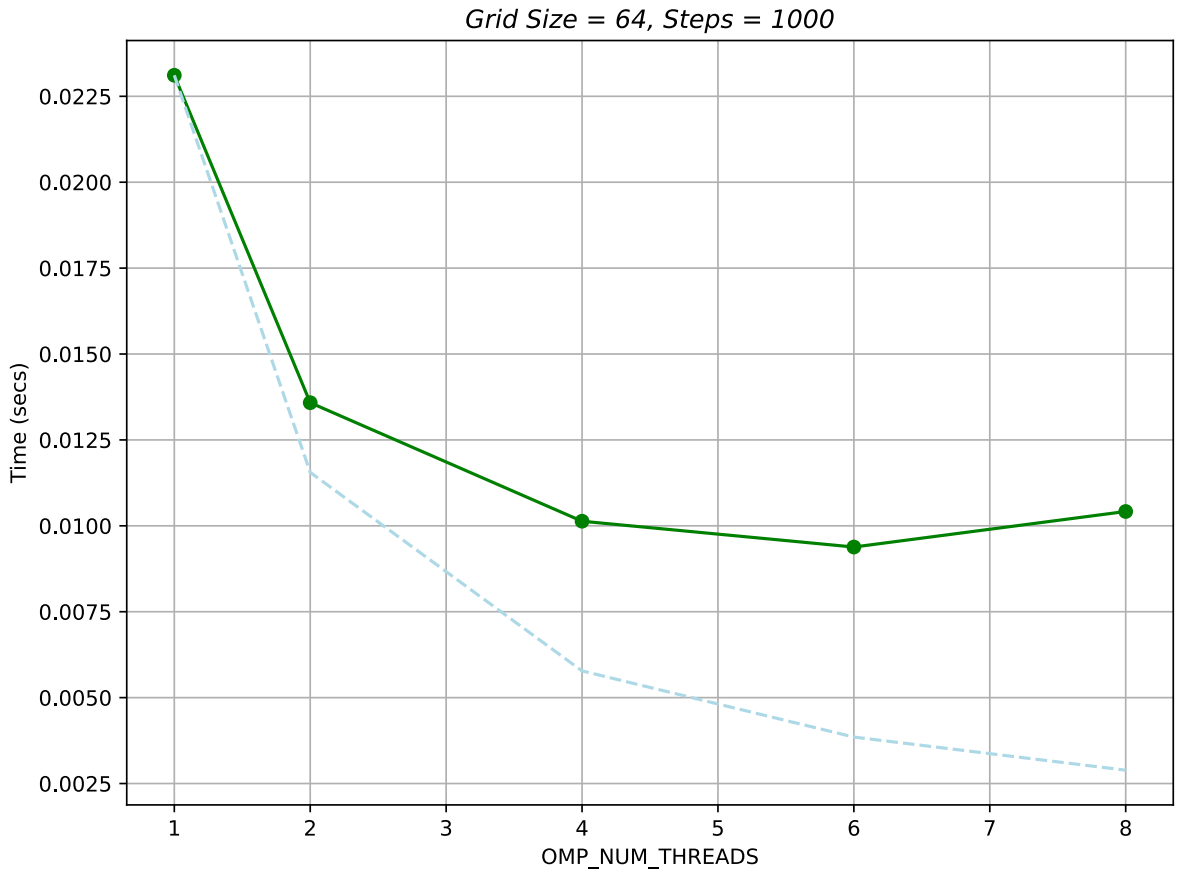
```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_gameoflife

## Output and error files
#PBS -o omp_gameoflife_all.out
#PBS -e omp_gameoflife_all.err

## Limit memory, runtime etc.
#PBS -l walltime=01:00:00

##Number of nodes aka threads
#PBS -l nodes=1:ppn=8

module load openmpi/1.8.3
cd /home/parallel/parlab09/a1

for threads in 1 2 4 6 8
do
export OMP_NUM_THREADS=$threads
echo "Running with OMP_NUM_THREADS=$OMP_NUM_THREADS"
./omp_gameoflife 64 1000
./omp_gameoflife 1024 1000
./omp_gameoflife 4096 1000
echo "Finished run with OMP_NUM_THREADS=$OMP_NUM_THREADS"
echo "-------------------------------------------------"
done
```

Αποτελέσματα Μετρήσεων:

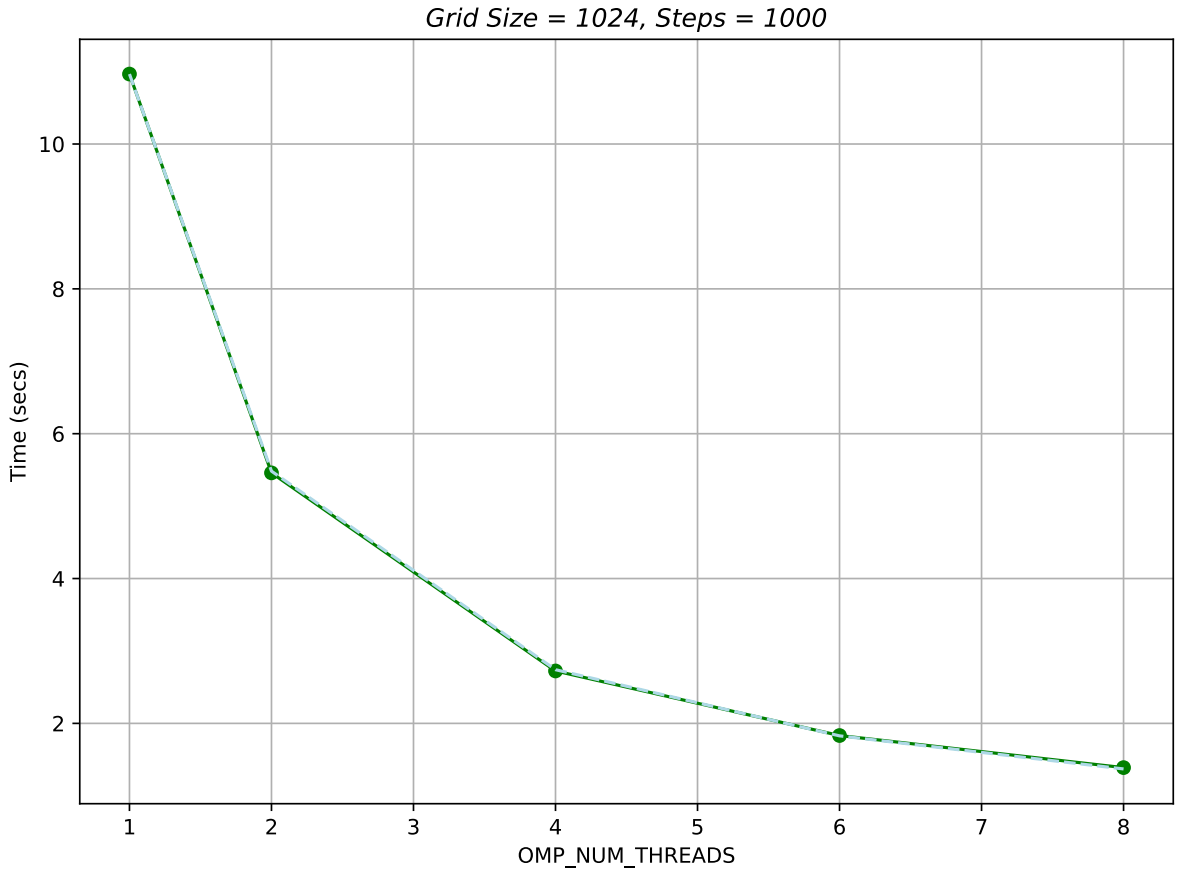
```
Running with OMP_NUM_THREADS=1
GameOfLife: Size 64 Steps 1000 Time 0.023112
GameOfLife: Size 1024 Steps 1000 Time 10.965944
GameOfLife: Size 4096 Steps 1000 Time 175.900314
Finished run with OMP_NUM_THREADS=1
-------------------------------------------------
Running with OMP_NUM_THREADS=2
GameOfLife: Size 64 Steps 1000 Time 0.013583
GameOfLife: Size 1024 Steps 1000 Time 5.458949
GameOfLife: Size 4096 Steps 1000 Time 88.263665
Finished run with OMP_NUM_THREADS=2
```

```
-------------------------------------------------
Running with OMP_NUM_THREADS=4
GameOfLife: Size 64 Steps 1000 Time 0.010134
GameOfLife: Size 1024 Steps 1000 Time 2.723798
GameOfLife: Size 4096 Steps 1000 Time 45.901567
Finished run with OMP_NUM_THREADS=4
-------------------------------------------------
Running with OMP_NUM_THREADS=6
GameOfLife: Size 64 Steps 1000 Time 0.009383
GameOfLife: Size 1024 Steps 1000 Time 1.832227
GameOfLife: Size 4096 Steps 1000 Time 43.661123
Finished run with OMP_NUM_THREADS=6
-------------------------------------------------
Running with OMP_NUM_THREADS=8
GameOfLife: Size 64 Steps 1000 Time 0.010417
GameOfLife: Size 1024 Steps 1000 Time 1.389175
GameOfLife: Size 4096 Steps 1000 Time 43.186379
Finished run with OMP_NUM_THREADS=8
-------------------------------------------------
```

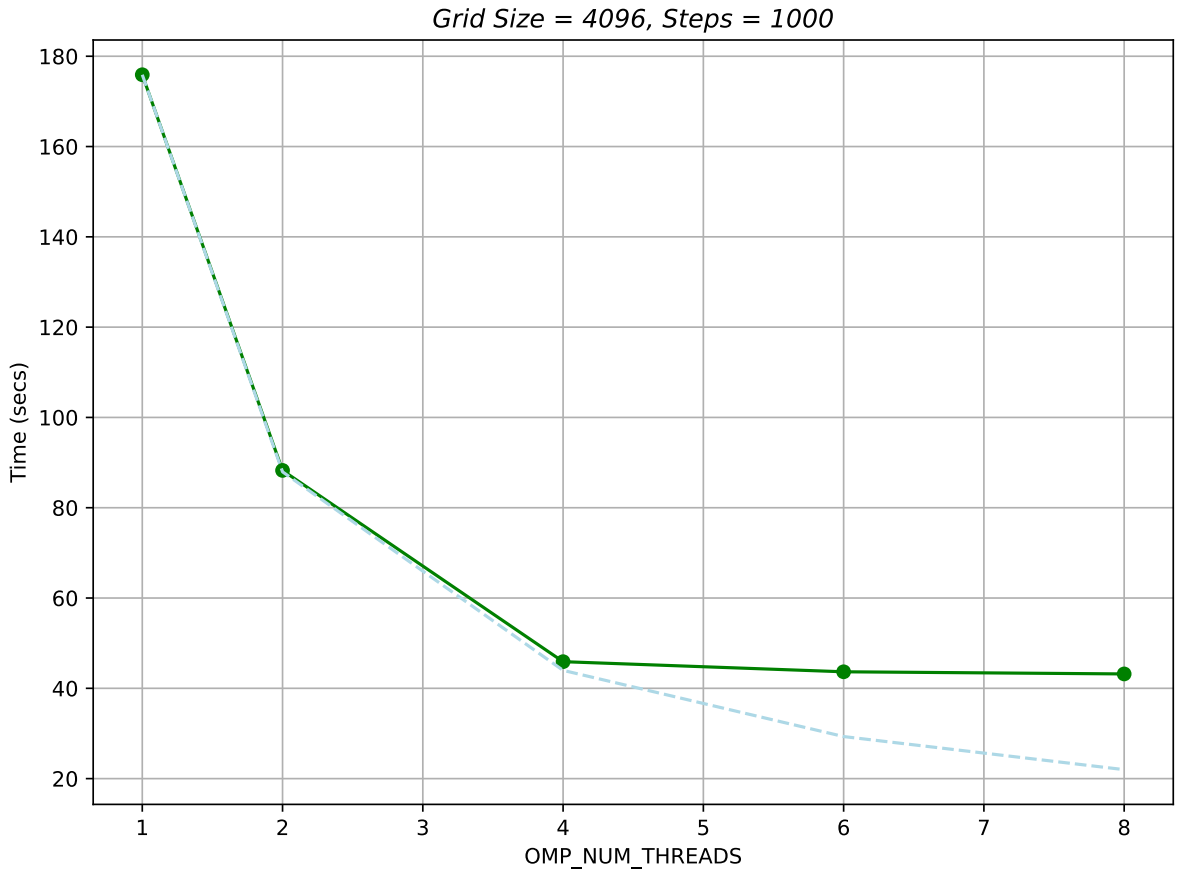
Γραφική Απεικόνιση και Παρατηρήσεις



Παρατηρούμε ότι για μικρό μέγεθος grid (με συνολική απαίτηση μνήμης 4*64*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).



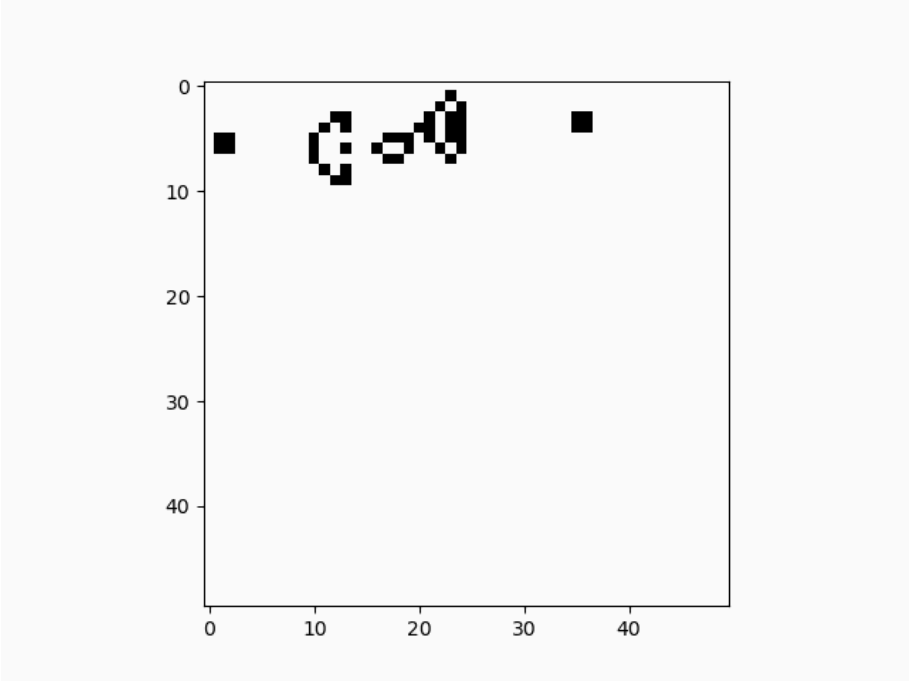
Για μέγεθος grid με συνολική απαίτηση μνήμης 4*1024*1024 bytes = 4MB, η επίδοση βελτίωνεται ομοιόμορφα και ανάλογα με το μέγεθος των νημάτων . Εικάζουμε, λοιπόν, πως η cache χωράει ολόκληρο το grid ώστε το κάθε νήμα δεν επιβαρύνει την μνήμη με loads των αντίστοιχων rows, ο φόρτος εργασίας είναι ισομοιρασμένος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



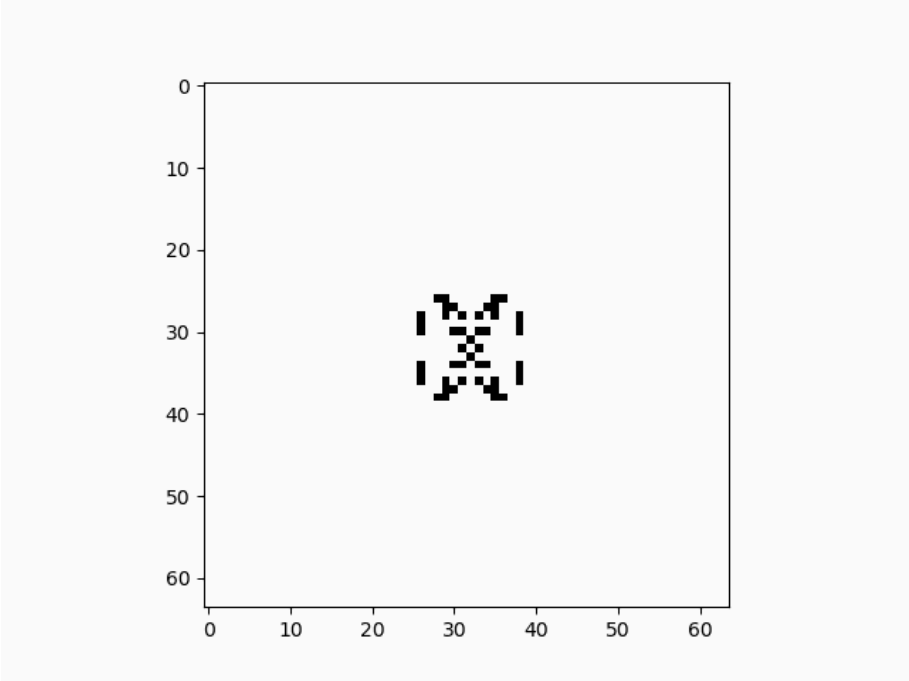
Για μεγάλο grid (με συνολική απαίτηση μνήμης 4*4096*4096 bytes = 64MB), η κλιμάκωση παύει να υφίσταται για περισσότερα από 4 νήματα. Bottleneck κόστους εδώ θεωρούμε το memory bandwidth. Επειδή ολόκληρο το grid δεν χωράει στην cache, δημιουργούνται misses όταν ξεχωριστά νήματα προσπαθούν να διαβάσουν 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). Ωστόσο, χρησιμοποιούμε atomic για ορθότητα της τιμής του και για παρατήρηση με βάση το μεγαλύτερο δυνατό overhead.

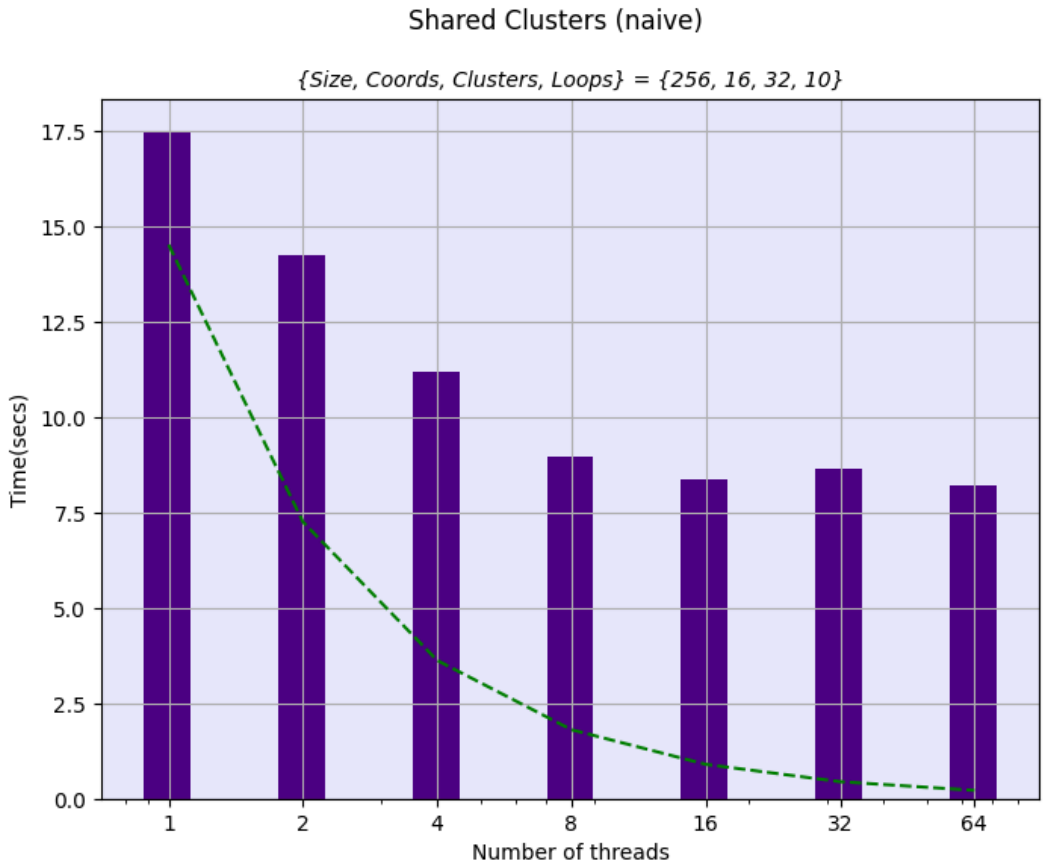
omp_naive_kmeans.c

```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:

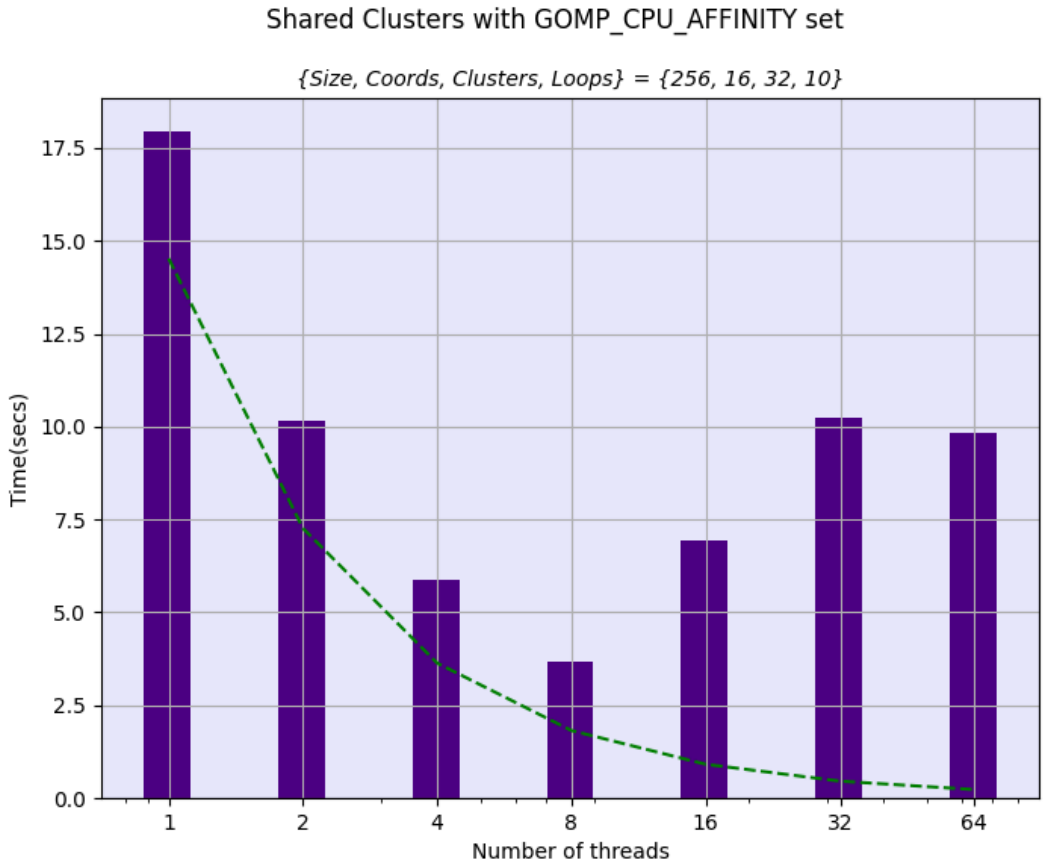


Παρατηρούμε πως ο αλγόριθμος δεν κλιμακώνει καθόλου καλά από 8 και πάνω νήματα εξαιτείας της σειριποίησης των εγγραφών ολοένα και περισσότερων νημάτων που επιβάλλει η omp atomic, και της αυξανόμενης συμφόρησης στο bus κατά την απόκτηση του lock.

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.

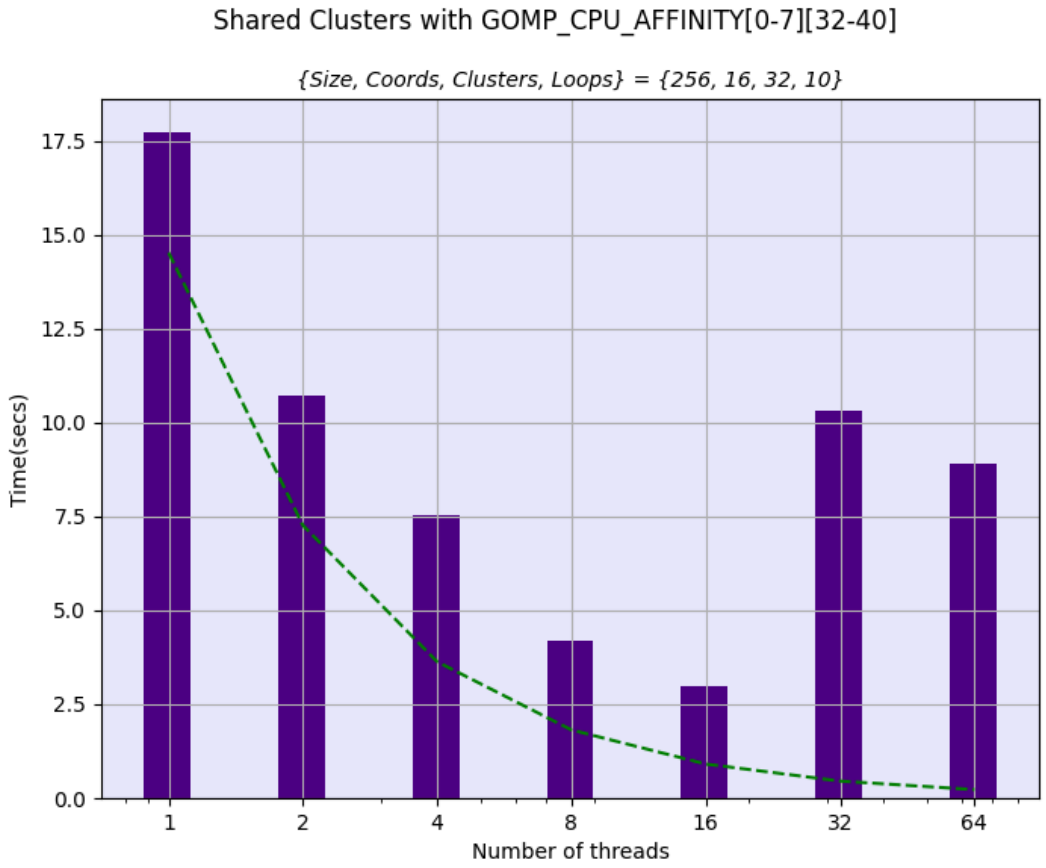
Αποτελέσματα



Παρατηρούμε σημαντική βελτίωση στην κλιμάκωση μέχρι 8 νήματα όμως μετά σταματάει να κλιμακώνει ο αλγόριθμος λόγω της δομής που έχει ο sandman. Για 16 νήματα και πάνω δεν μπορούμε να τα κάνουμε 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 κλιμάκωση σταματάει. Παραθέτουμε το τελικό script υποβολής ακολούθως:

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_kmeans

## Output and error files
#PBS -o gomp_hyper_kmeans.out
#PBS -e gomp_hyper_kmeans.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab09/kmeans

Size=256
Coords=16
Clusters=32
Loops=10

for i in 1 2 4 8 16 32 64; do
    export OMP_NUM_THREADS=$i

    if [[ $i -eq 16 ]]; then
        export GOMP_CPU_AFFINITY="$(seq -s, 0 7),$(seq -s, 32 40)"
    else
        export GOMP_CPU_AFFINITY="$(seq -s, 0 $((i - 1)))"
    fi

    ./kmeans_omp_naive -s $Size -n $Coords -c $Clusters -l $Loops
done
```

Αποτελέσματα



2) Copied Clusters & Reduce

Υλοποίηση

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

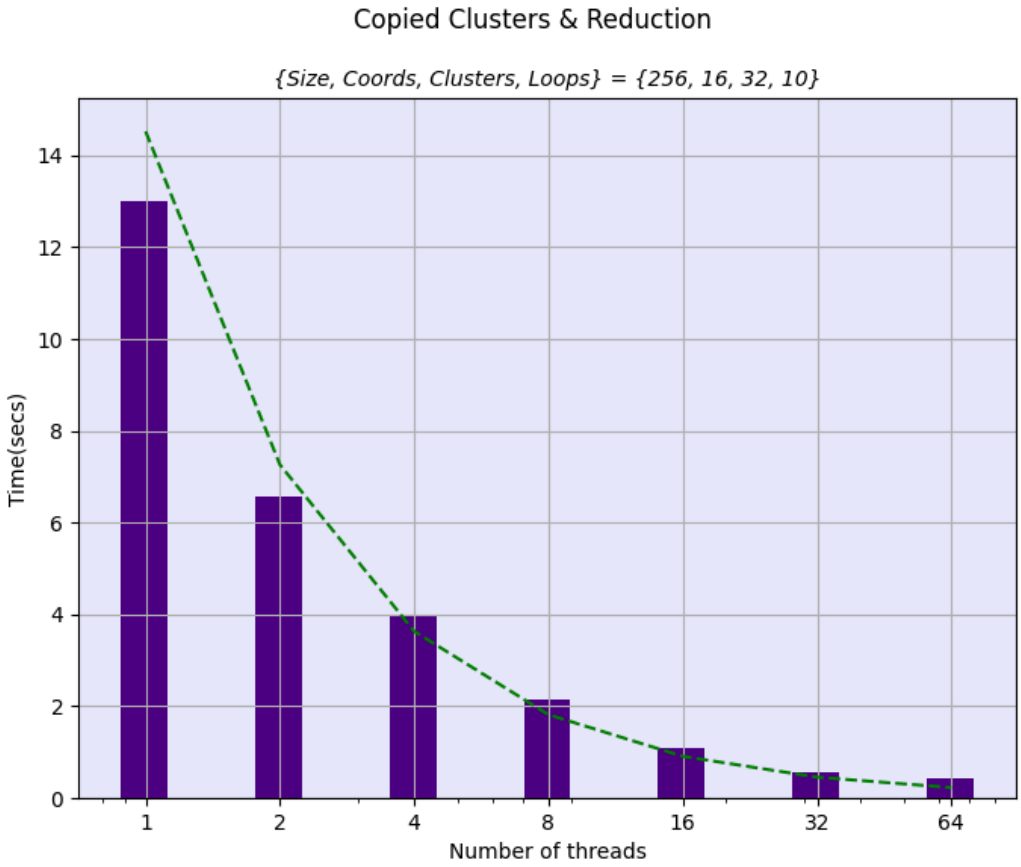
omp_reduction_kmeans1.c

```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] */
      double * clusters)         /* out: [numClusters][numCoords] */
{
  int i, j, k;
  int index, loop=0;
  double timing = 0;

  double delta;          // fraction of objects whose clusters change in each loop
  int * newClusterSize; // [numClusters]: no. objects assigned in each new cluster
  double * newClusters;  // [numClusters][numCoords]
  int nthreads;         // no. threads

  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 array reduction on them.
75    int * local_newClusterSize[nthreads];  // [nthreads][numClusters]
76    double * local_newClusters[nthreads];   // [nthreads][numClusters][numCoords]
77
78    /*
79     * Hint for false-sharing
80     * This is noticed when numCoords is low (and neighboring local_newClusters exist close to each other).
81     * Allocate local cluster data with a "first-touch" policy.
82     */
83    // Initialize local (per-thread) arrays (and later collect result on global arrays)
84    for (k=0; k<nthreads; k++)
85    {
86            local_newClusterSize[k]   =   (typeof(*local_newClusterSize))   calloc(numClusters, 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
114     int thread_id;
115
116       #pragma omp parallel for private(i, j, thread_id, index) shared(local_newClusters, local_newClusterSize) reduction(+:delta) schedule(static)
117     for (i=0; i<numObjs; i++)
118     {
119       thread_id = omp_get_thread_num();
120
121       // find the array index of nearest cluster center
122       index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
123
124       // if membership changes, increase delta by 1
125       if (membership[i] != index)
126         delta += 1.0;
127
128       // assign the membership to object i
129       membership[i] = index;
130
```

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

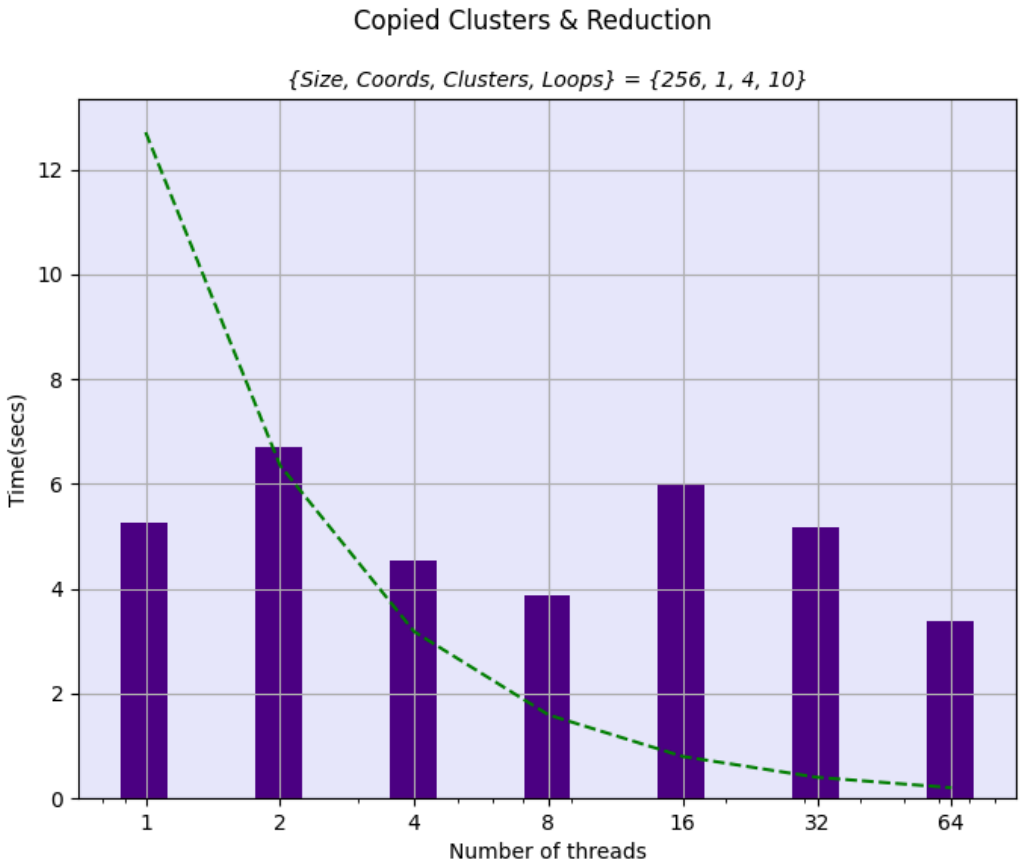
Αποτελέσματα



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 καθυστρεί τον αλγόριθμο.

Δοκιμές με μικρότερο dataset

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



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

First-touch Policy

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

Υλοποίηση

omp_reduction_kmeans.c

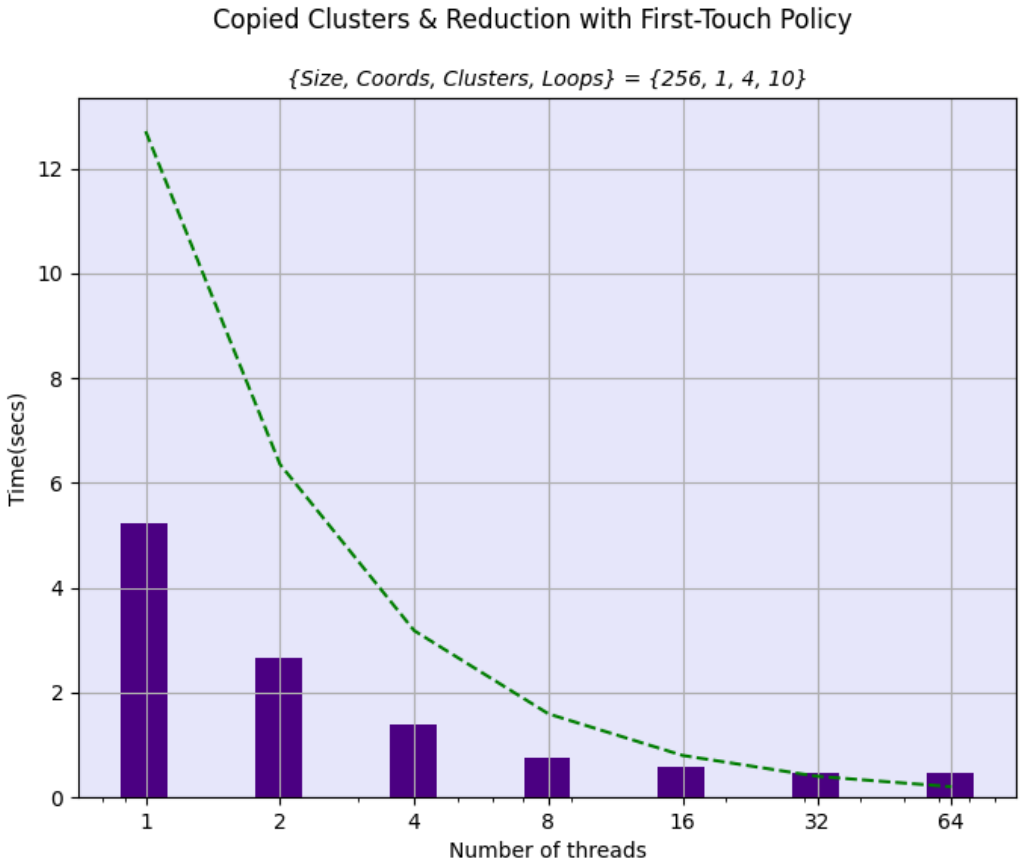
```c
#include <stdio.h>
#include <stdlib.h>
#include "kmeans.h"
/*
 * TODO: include openmp header file
 */

// 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] */
      double * clusters)         /* out: [numClusters][numCoords] */
{
  int i, j, k;
  int index, loop=0;
  double timing = 0;

  double delta;          // fraction of objects whose clusters change in each loop
  int * newClusterSize; // [numClusters]: no. objects assigned in each new cluster
  double * newClusters;  // [numClusters][numCoords]
  int nthreads;         // no. threads

  nthreads = omp_get_max_threads();
  printf("OpenMP Kmeans - Reduction\t(number of threads: %d)\n", nthreads);

  // initialize membership
  for (i=0; i<numObjs; i++)
    membership[i] = -1;

  // initialize newClusterSize and newClusters to all 0
  newClusterSize = (typeof(newClusterSize)) calloc(numClusters, sizeof(*newClusterSize));
  newClusters = (typeof(newClusters))  calloc(numClusters * numCoords, sizeof(*newClusters));

  // Each thread calculates new centers using a private space. After that, thread 0 does an array reduction on them.
  int * local_newClusterSize[nthreads];  // [nthreads][numClusters]
  double * local_newClusters[nthreads];   // [nthreads][numClusters][numCoords]

  /*
   * Hint for false-sharing
   * This is noticed when numCoords is low (and neighboring local_newClusters exist close to each other).
   * Allocate local cluster data with a "first-touch" policy.
   */

  timing = wtime();
  do {
    // before each loop, set cluster data to 0
    for (i=0; i<numClusters; i++) {
      for (j=0; j<numCoords; j++)
```

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

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

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

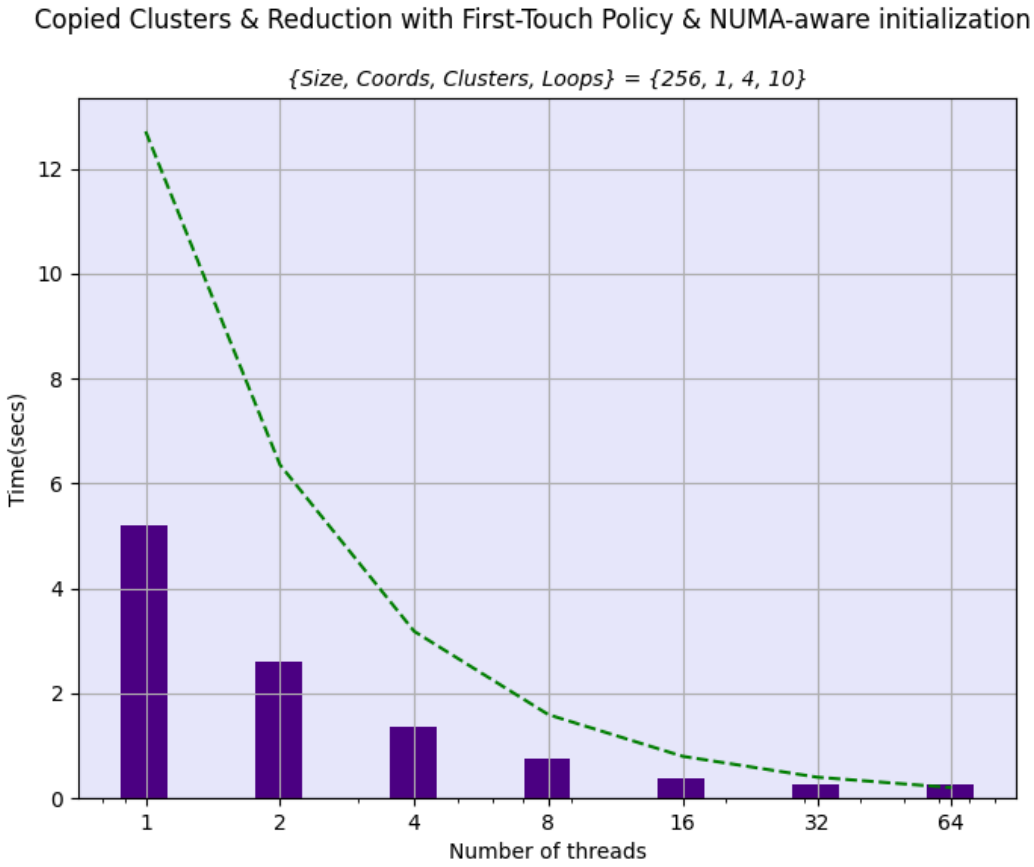
Τροποποιούμε το file_io.c που δίνεται :

file_io.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>     /* strtok() */
#include <sys/types.h>  /* open() */
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>     /* read(), close() */
// TODO: remove comment from following line
#include <omp.h>

#include "kmeans.h"

double * dataset_generation(int numObjs, int numCoords)
{
  double * objects = NULL;
  long i, j;
  // Random values that will be generated will be between 0 and 10.
  double val_range = 10;

  /* allocate space for objects[][] and read all objects */
  objects = (typeof(objects)) malloc(numObjs * numCoords * sizeof(*objects));

  /*
   * Hint : Could dataset generation be performed in a more "NUMA-Aware" way?
   *        Need to place data "close" to the threads that will perform operations on them.
   *        reminder : First-touch data placement policy
   */
  int nthreads = omp_get_max_threads();
  int chunk = numObjs / nthreads;
  int thread_id, start_offs, end_offs;

  #pragma omp parallel private(i, j, thread_id, start_offs, end_offs) shared(nthreads, chunk, objects, numObjs, numCoords, val_range)
  {
    //set the binding to cores manually

    thread_id = omp_get_thread_num();
    start_offs = thread_id * chunk;
    end_offs = (thread_id == nthreads-1) ? numObjs : start_offs + chunk;

    for (i=start_offs; i<end_offs; i++)
    {
      unsigned int seed = i;
      for (j=0; j <numCoords; j++)
      {
        objects[i*numCoords + j] = (rand_r(&seed) / ((double) RAND_MAX)) * val_range;
        if (_debug && i == 0)
          printf("object[i=%ld][j=%ld]=%f\n",i,j,objects[i*numCoords + j]);
      }
    }
  }
    return objects;
}
```

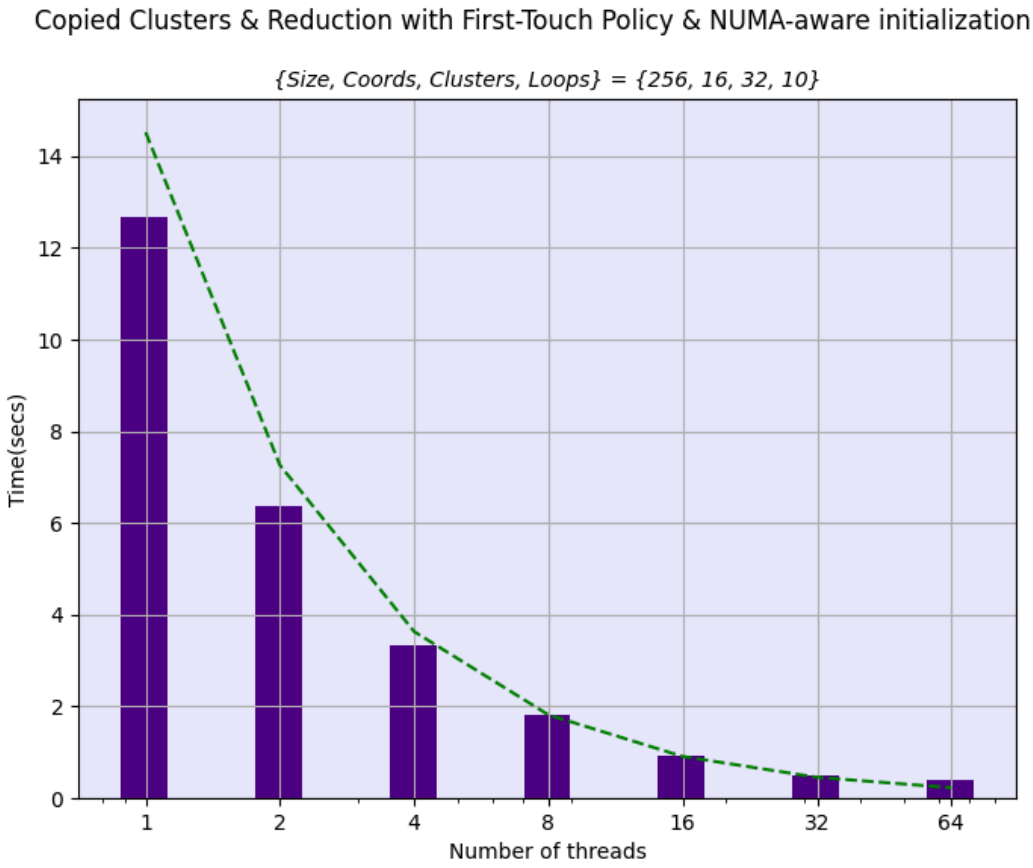
)

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.

FLOYD WARSHALL

1) Recursive

Υλοποίηση

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

fw_sr.c

```c
/*
 * Recursive implementation of the Floyd-Warshall algorithm.
 * command line arguments: N, B
 * N = size of graph
 * B = size of submatrix when recursion stops
 * works only for N, B = 2^k
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "util.h"
#include <omp.h>

inline int min(int a, int b);
void FW_SR (int **A, int arow, int acol,
            int **B, int brow, int bcol,
            int **C, int crow, int ccol,
            int myN, int bsize);

int main(int argc, char **argv)
{
  int **A;
  int i,j;
  struct timeval t1, t2;
  double time;
  int B=16;
  int N=1024;

  if (argc !=3){
    fprintf(stdout, "Usage %s N B \n", argv[0]);
    exit(0);
  }

  N=atoi(argv[1]);
  B=atoi(argv[2]);

  A = (int **) malloc(N*sizeof(int *));
  for(i=0; i<N; i++) A[i] = (int *) malloc(N*sizeof(int));

  graph_init_random(A,-1,N,128*N);
  //enable nested task generation
  omp_set_nested(1);
  // default is equal to 1
  omp_set_max_active_levels(64);

  gettimeofday(&t1,0);

  #pragma omp parallel
  {
    #pragma omp single
    {
      FW_SR(A,0,0, A,0,0,A,0,0,N,B);
    }
  }
  gettimeofday(&t2,0);
```

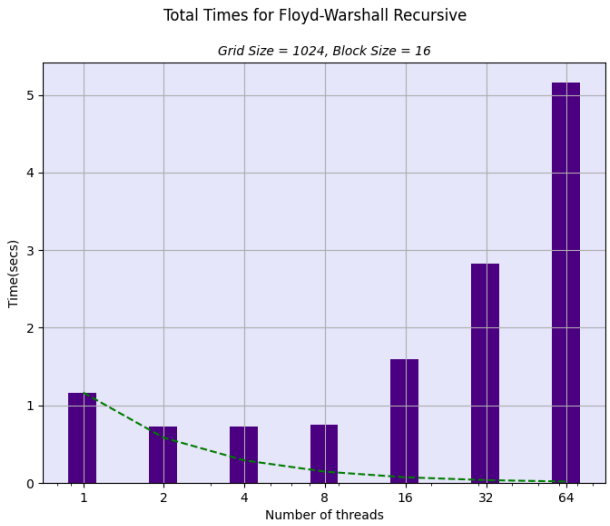
```
57
58    time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)/1000000;
59    printf("FW_SR,%d,%d,%.4f\n", N, B, time);
60
61    /*
62    for(i=0; i<N; i++)
63      for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
64    */
65
66    return 0;
67  }
68
69  inline int min(int a, int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

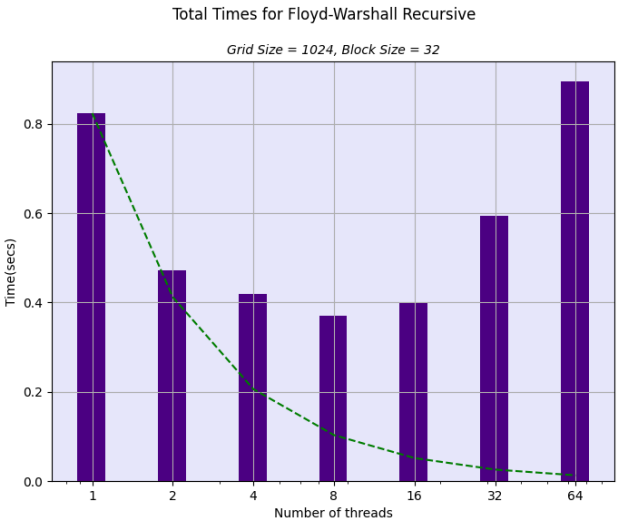
Πειραματιστήκαμε σχετικά με την βέλτιστη τιμή του BSIZE τρέχοντας τις προσομοιώσεις που ακολουθούν. Διαισθητικά η optimal τιμή οφείλει να εκμεταλλεύεται πλήρως το cache size και δεδομένου ότι έχουμε τετράγωνο grid για 1 recursive call που δημιουργεί 4 sub-blocks μεγέθους B θα είναι Bopt = sqrt(cache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

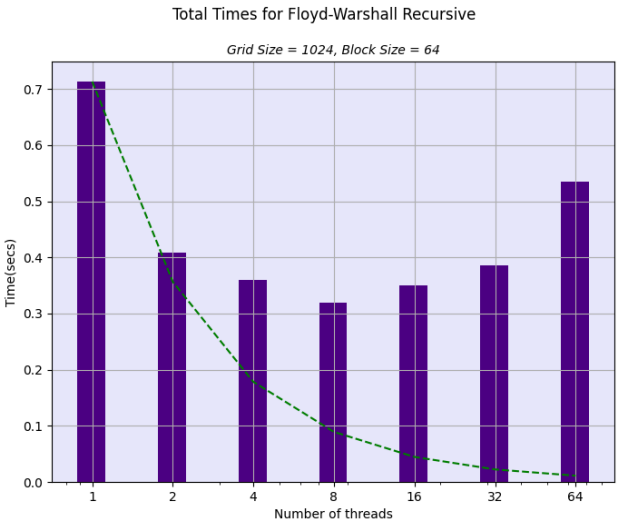
```
#!/bin/bash
## Give the Job a descriptive name #PBS -N run_fw
## Output and error files #PBS -o run_fw_recursive.out #PBS -e run_fw_recursive.err
## How many machines should we get? #PBS -l nodes=1:ppn=8
##How long should the job run for? #PBS -l walltime=00:10:00
## Start
## Run make in the src folder (modify properly)
module load openmp/1.8.3
cd /home/parallel/parlab09/a2/FW

./fw $SIZE

export OMP_NESTED=TRUE
export OMP_MAX_ACTIVE_LEVELS=64

for SIZE in 1024 2048 4096; do
    for BSIZE in 16 32 64 128 256; do
        echo -e "\nBSIZE=${BSIZE}\n"
        for n in 1 2 4 8 16 32 64; do
            export OMP_NUM_THREADS=${n}
            echo -e "\nNumber of threads: ${n}"
            ./fw_sr $SIZE $BSIZE
        done
    done
done
```

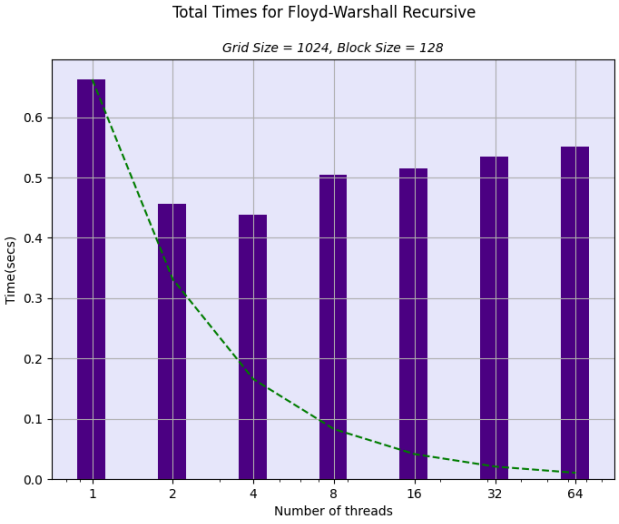
Αποτελέσματα

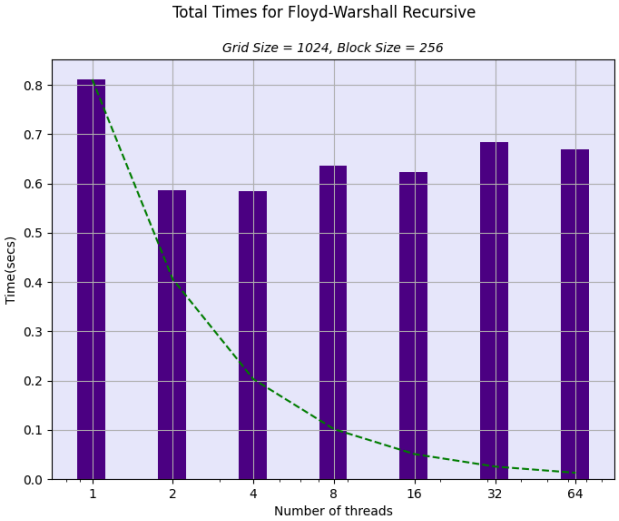
{N = 1024}



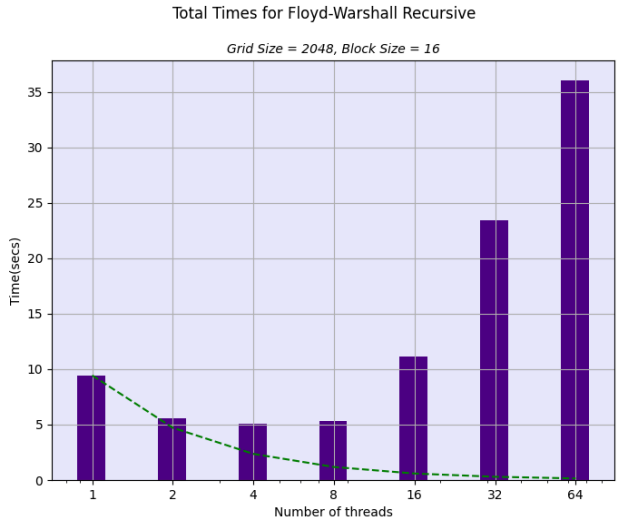

Total Times for Floyd-Warshall Recursive
Grid Size = 1024, Block Size = 32
Time(secs)
0.0
0.2
0.4
0.6
0.8
1
2
4
8
16
32
64
Number of threads

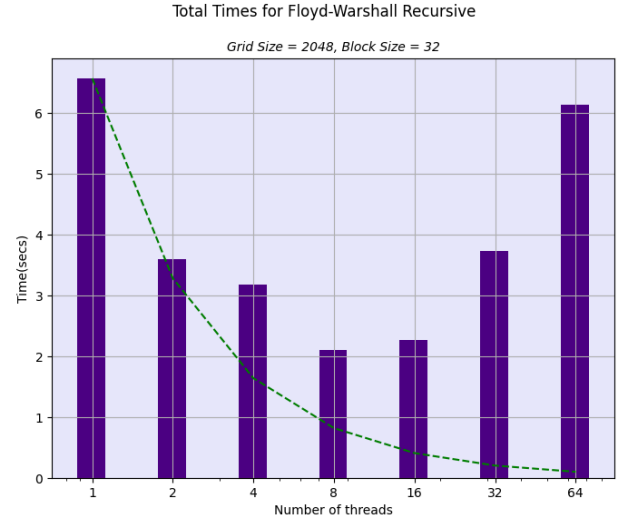

Total Times for Floyd-Warshall Recursive
Grid Size = 1024, Block Size = 64
Time(secs)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
1
2
4
8
16
32
64
Number of threads

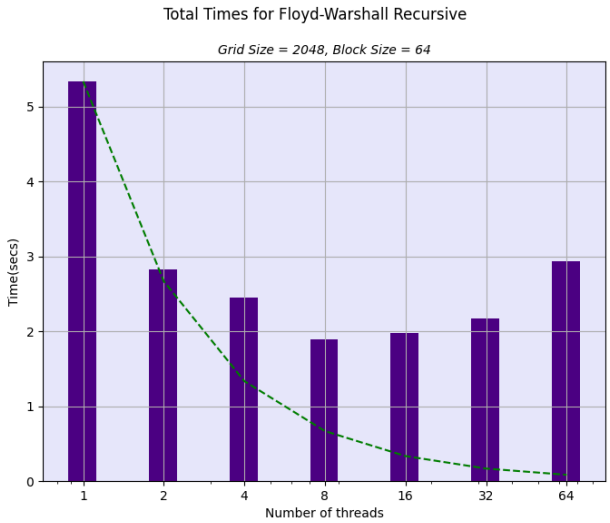

Total Times for Floyd-Warshall Recursive
Grid Size = 1024, Block Size = 128
Time(secs)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
2
4
8
16
32
64
Number of threads

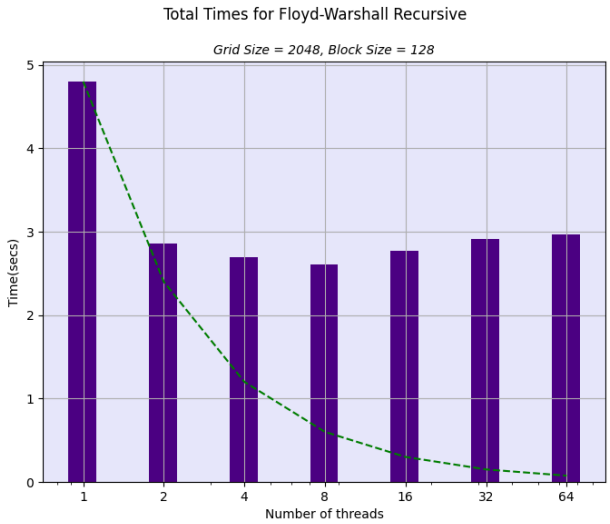

Total Times for Floyd-Warshall Recursive
Grid Size = 1024, Block Size = 256
Time(secs)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
4
8
16
32
64
Number of threads

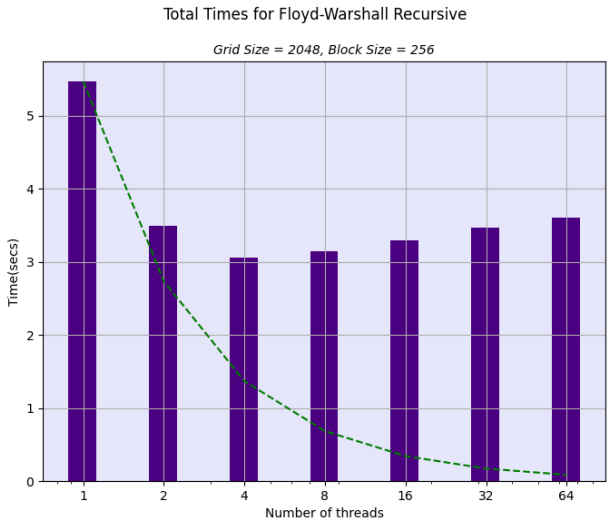
{N = 2048}


Total Times for Floyd-Warshall Recursive
Grid Size = 2048, Block Size = 16
Time(secs)
0
5
10
15
20
25
30
35
1
2
4
8
16
32
64
Number of threads

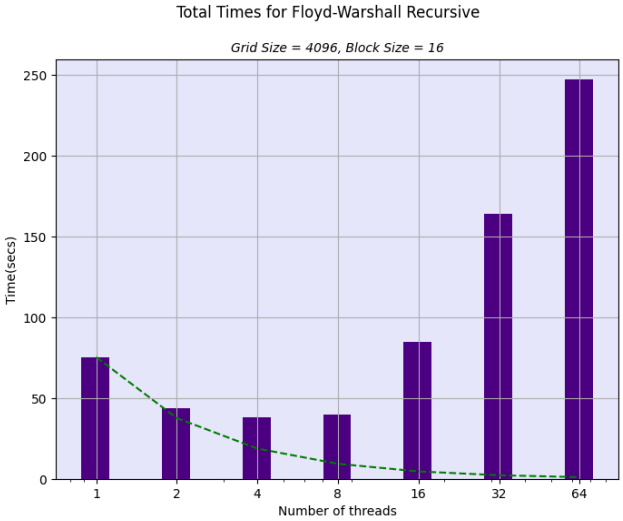

Total Times for Floyd-Warshall Recursive
Grid Size = 2048, Block Size = 32
Time(secs)
0
1
2
3
4
5
6
1
2
4
8
16
32
64
Number of threads

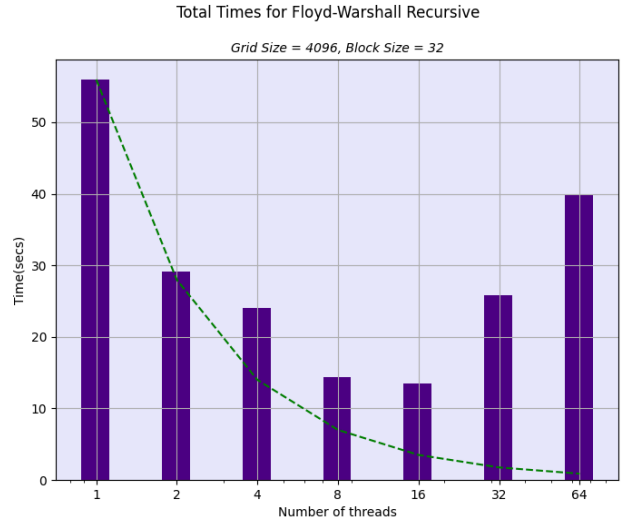

Total Times for Floyd-Warshall Recursive
Grid Size = 2048, Block Size = 64
Time(secs)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads

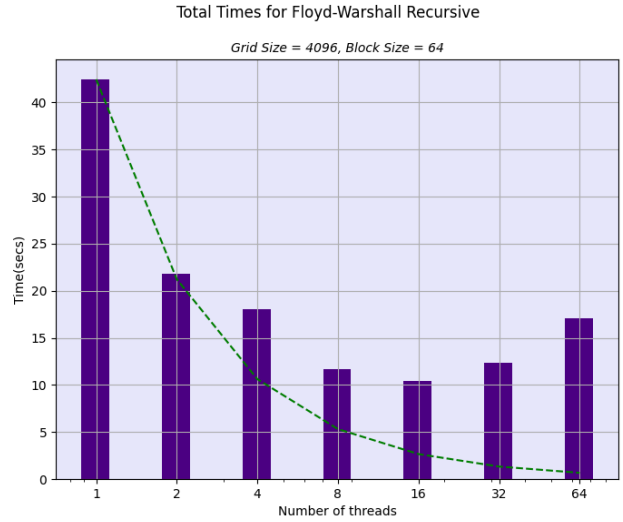

Total Times for Floyd-Warshall Recursive
Grid Size = 2048, Block Size = 128
Time(secs)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads

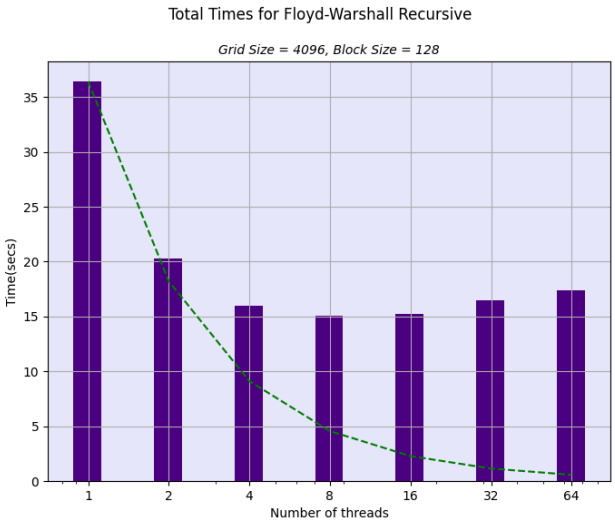

Total Times for Floyd-Warshall Recursive
Grid Size = 2048, Block Size = 256
Time(secs)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads

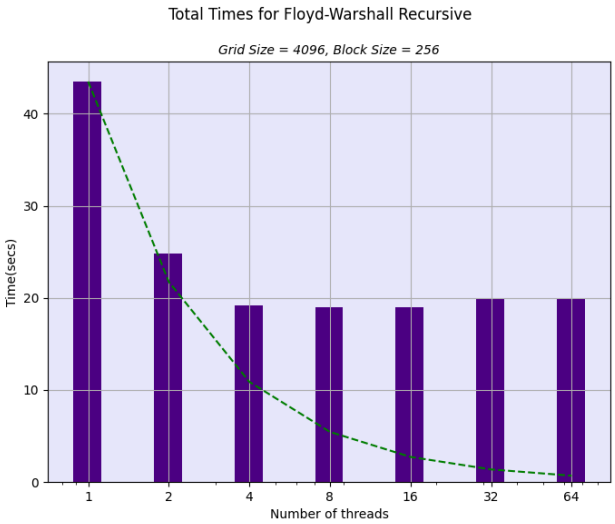
{ N = 4096 }


Total Times for Floyd-Warshall Recursive
Grid Size = 4096, Block Size = 16
Time(secs)
Number of threads
0
50
100
150
200
250
1
2
4
8
16
32
64


Total Times for Floyd-Warshall Recursive
Grid Size = 4096, Block Size = 32
Time(secs)
Number of threads
0
10
20
30
40
50
1
2
4
8
16
32
64


Total Times for Floyd-Warshall Recursive
Grid Size = 4096, Block Size = 64
Time(secs)
Number of threads
0
5
10
15
20
25
30
35
40
1
2
4
8
16
32
64





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 intrisincs 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 αυτήν την εκδοχή είναι **1.39 secs!**

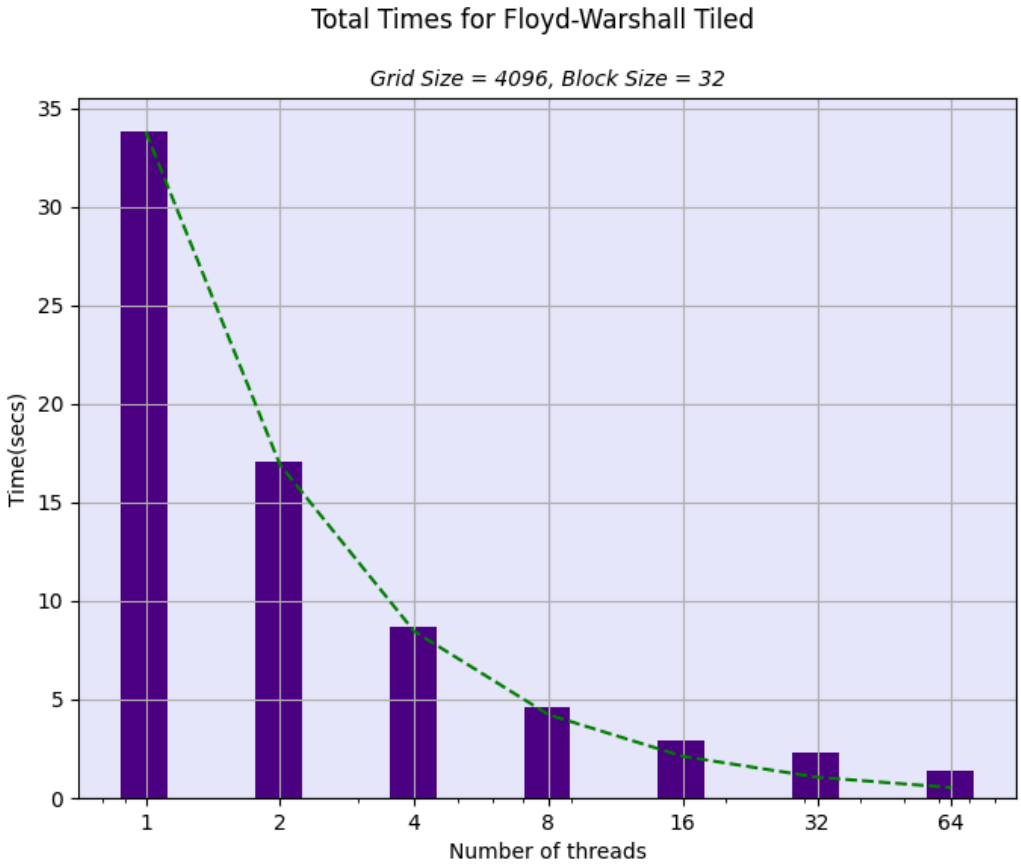
fw_smd.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <immintrin.h>  // For SSE2 intrinsics
#include <omp.h>

inline void FW(int **A, int K, int I, int J, int B);

int main(int argc, char **argv)
{
    int **A;
    int i, j, k;
    struct timeval t1, t2;
    double time;
    int B = 64;
    int N = 1024;

    if (argc != 3) {
        fprintf(stdout, "Usage %s N B\n", argv[0]);
        exit(0);
    }

    N = atoi(argv[1]);
    B = atoi(argv[2]);

    // Allocate memory for A with 32-byte alignment
    posix_memalign((void**)&A, 32, N * sizeof(int*));
    for (i = 0; i < N; ++i) {
        posix_memalign((void**)&A[i], 32, N * sizeof(int));
    }

    // Initialize the graph with random values
    graph_init_random(A, -1, N, 128 * N);

    // Start timer
    gettimeofday(&t1, 0);

```

```
    // Main loop of the Floyd-Warshall algorithm with tiling
    for (k = 0; k < N; k += B) {
        #pragma omp parallel
        {
            #pragma omp single
            {
                FW(A, k, k, k, B);
            }
            #pragma omp for nowait
            for (i = 0; i < k; i += B)
                FW(A, k, i, k, B);

            #pragma omp for nowait
            for (i = k + B; i < N; i += B)
                FW(A, k, i, k, B);

            #pragma omp for nowait
            for (j = 0; j < k; j += B)
                FW(A, k, k, j, B);

            #pragma omp for nowait
            for (j = k + B; j < N; j += B)
                FW(A, k, k, j, B);

            #pragma omp barrier

            #pragma omp for collapse(2) nowait
            for (i = 0; i < k; i += B)
                for (j = 0; j < k; j += B)
                    FW(A, k, i, j, B);

            #pragma omp for collapse(2) nowait
            for (i = 0; i < k; i += B)
                for (j = k + B; j < N; j += B)
                    FW(A, k, i, j, B);

            #pragma omp for collapse(2) nowait
            for (i = k + B; i < N; i += B)
                for (j = 0; j < k; j += B)
                    FW(A, k, i, j, B);

            #pragma omp for collapse(2) nowait
            for (i = k + B; i < N; i += B)
                for (j = k + B; j < N; j += B)
                    FW(A, k, i, j, B);

            #pragma omp barrier
        }
    }

    // Stop timer and calculate execution time
    gettimeofday(&t2, 0);
    time = (double)((t2.tv_sec - t1.tv_sec) * 1000000 + t2.tv_usec - t1.tv_usec) / 1000000;
    fprintf(stdout, "FW_TILED,%d,%d,%.4f\n", N, B, time);

    // Free the memory
    for (i = 0; i < N; i++) {
        _mm_free(A[i]);  // Free each row
    }
    _mm_free(A);  // Free the pointer array

    return 0;
}

inline void FW(int **A, int K, int I, int J, int B)
{
    int i, j, k;

    // Iterate over a block of tiles (3D loop over the block)
    for (k = K; k < K + B; k++) {
        for (i = I; i < I + B; i++) {
                // _mm_prefetch((const char*)&A[i][j], _MM_HINT_T0);
                // _mm_prefetch((const char*)&A[k][j], _MM_HINT_T0);
                // _mm_prefetch((const char*)&A[i][j + 16], _MM_HINT_T0);
                // _mm_prefetch((const char*)&A[k][j + 16], _MM_HINT_T0);

```

```
                __m128i A_i_k = _mm_load_si128((__m128i*)&A[i][k]);

                    for (j = J; j < J + B; j+=16){

                    __m128i A_i_j = _mm_load_si128((__m128i*)&A[i][j]);
                    __m128i A_k_j = _mm_load_si128((__m128i*)&A[k][j]);

                    __m128i A_plus = _mm_add_epi32(A_i_k, A_k_j);
                    __m128i result = _mm_min_epi32(A_i_j, A_plus);

                    _mm_store_si128((__m128i*)&A[i][j], result);

                    // next chunk
                    A_i_j = _mm_load_si128((__m128i*)&A[i][j+4]);
                    A_k_j = _mm_load_si128((__m128i*)&A[k][j+4]);

                    A_plus = _mm_add_epi32(A_i_k, A_k_j);
                    result = _mm_min_epi32(A_i_j, A_plus);

                    _mm_store_si128((__m128i*)&A[i][j+4], result);

                    //next chunk
                    A_i_j = _mm_load_si128((__m128i*)&A[i][j+8]);
                    A_k_j = _mm_load_si128((__m128i*)&A[k][j+8]);

                    A_plus = _mm_add_epi32(A_i_k, A_k_j);
                    result = _mm_min_epi32(A_i_j, A_plus);

                    _mm_store_si128((__m128i*)&A[i][j+8], result);

                    //next chunk
                    A_i_j = _mm_load_si128((__m128i*)&A[i][j+12]);
                    A_k_j = _mm_load_si128((__m128i*)&A[k][j+12]);

                    A_plus = _mm_add_epi32(A_i_k, A_k_j);
                    result = _mm_min_epi32(A_i_j, A_plus);

                    _mm_store_si128((__m128i*)&A[i][j+12], result);

                    // if(j == J)
                    //     _mm_prefetch((const char*)&A[i+1][k], _MM_HINT_T0);
                }
            }
            // if (k + 1 < K + B) {
            //     _mm_prefetch((const char*)&A[i][k + 1], _MM_HINT_T0);
            // }
        }
    }
```

Αποτελέσματα



Παραθέτουμε αναλυτικά και τους βέλτιστους χρόνους:

Number of threads: 1
FW_TILED,4096,32,33.8411

Number of threads: 2
FW_TILED,4096,32,17.0405

Number of threads: 4
FW_TILED,4096,32,8.7231

Number of threads: 8
FW_TILED,4096,32,4.5795

Number of threads: 16
FW_TILED,4096,32,2.9022

Number of threads: 32
FW_TILED,4096,32,2.3016

Number of threads: 64
FW_TILED,4096,32,1.3925

Παράρτημα

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

results.py

```python
import numpy as np
import matplotlib.pyplot as plt
#import pandas
import re

regex_total = r"\(total\s*=\s*([\d.]+)s\)"
regex_loop = r"\(per\s+loop\s*=\s*([\d.]+)s\)"

total_times = []
loop_times = []

with open("results.txt", 'r') as file:
    for line in file:
        match_total = re.search(regex_total, line)
        match_loop = re.search(regex_loop, line)
        if (match_total is not None and match_loop is not None):
            total_times.append(float(match_total.group(1)))
            loop_times.append(float(match_loop.group(1)))


seq_totals = total_times[0:2]
seq_loop = loop_times[0:2]

total_times = total_times[2::]
loop_times = loop_times[2::]

print("Sequential Times: ", seq_totals, seq_loop)
print("Total Times:", total_times)
print("Loop Times:", loop_times)

threads = [1,2,4,8,16,32,64]
nthreads = len(threads)
titles = ["Shared Clusters (naive)",
          "Shared Clusters with GOMP_CPU_AFFINITY set",
          "Copied Clusters & Reduction",
          "Copied Clusters & Reduction",
          "Copied Clusters & Reduction with First-Touch Policy",
          "Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization",
          "Copied Clusters & Reduction with First-Touch Policy & NUMA-aware initialization",
          "Shared Clusters with GOMP_CPU_AFFINITY[0-7][32-40]"]

subtitles = ["{Size, Coords, Clusters, Loops} = {256, 16, 32, 10}",
             "{Size, Coords, Clusters, Loops} = {256, 1, 4, 10}",
             "{Size, Coords, Clusters, Loops} = {256, 16, 32, 10}"]

for i in range(0,8):
    x_axis = threads
    y_axis = total_times[i*nthreads:i*nthreads+nthreads]
    if (i==3 or i==4 or i==5) : seq_time = seq_totals[1]
    else: seq_time = seq_totals[0]
    print(f"Times for version{i}: ", y_axis)
    plt.figure(figsize=(8,6))
    plt.gca().set_facecolor("#e6e6fa")
    plt.xscale('log')
    widths = 0.6*np.diff(threads, prepend=threads[0] / 2) * 0.8
    plt.bar(x_axis, y_axis, width=widths, color="#4b0082", align="center")
    plt.xticks(x_axis, [str(t) for t in threads])
    plt.plot(x_axis, [seq_time/t for t in threads], color ='g', linestyle='--')
    plt.suptitle(titles[i], size = 12)
    plt.title(subtitles[int(i/3)], fontstyle = 'oblique', size = 10)
    plt.xlabel("Number of threads")
    plt.ylabel("Time(secs)")
    plt.grid('y')
    plt.savefig(f"fig{i}.png")
    plt.clf()
```

plots.py

```python
# import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
#import pandas


total_times = []
loop_times = []

with open("sandman.out", 'r') as file:
    for line in file:
        if (line.startswith("omp_num_threads")):
            t = float(line.split(',')[-1])
            if (t is not None):
                total_times.append(t)


print("Total Times:", total_times)
print("Loop Times:", loop_times)

threads = [1,2,4,8,16,32,64]
GridSize = [1024, 2048, 4096]
BSizes = [256, 128, 64, 32, 16]
nthreads = len(threads)

for grid in range (0,len(GridSize)):
    for bsize in range(0,len(BSizes)):
        start_idx = grid*(nthreads*len(BSizes)) + nthreads*bsize
        end_idx = start_idx + nthreads
        x_axis = threads
        y_axis = total_times[start_idx:end_idx]
        print(f"Times for S={GridSize[grid]}, BS={BSizes[bsize]}: ", y_axis)
        plt.figure(figsize=(8,6))
        plt.gca().set_facecolor("#e6e6fa")
        plt.xscale('log')
        widths = 0.6*np.diff(threads, prepend=threads[0] / 2) * 0.8
        plt.bar(x_axis, y_axis, width=widths, color="#4b0082", align="center")
        plt.xticks(x_axis, [str(t) for t in threads])
        plt.plot(x_axis, [y_axis[0]/t for t in threads], color ='g', linestyle='--')
        plt.suptitle("Total Times for Floyd-Warshall Recursive", size = 12)
        plt.title(f"Grid Size = {GridSize[grid]}, Block Size = {BSizes[bsize]}", fontstyle =
'oblique', size = 10)
        plt.xlabel("Number of threads")
        plt.ylabel("Time(secs)")
        plt.grid('y')
        plt.savefig(f"fig{GridSize[grid]}_{BSizes[bsize]}.png")
        plt.clf()
```

Αμοιβαίος Αποκλεισμός-Κλειδώματα

Στο συγκεκριμένο ερώτημα καλούμαστε να αξιολογήσουμε τους διαφορετικούς τρόπους υλοποίησης κλειδωμάτων για αμοιβαίο αποκλεισμό.
Μας δίνονται έτοιμες όλες οι υλοποίησεις των κλειδωμάτων. Για την εκτέλεση του συγκεκριμένου data set (Size = 32, Coords = 16, Clusters = 32, Loops = 10) στον scirouter χρησιμοποιήσαμε το ακόλουθο script :

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_kmeans

## Output and error files
#PBS -o run_kmeans.out
#PBS -e run_kmeans.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab09/a2_new/a2/kmeans

export SIZE=32
export COORDS=16
export CLUSTERS=32
export LOOPS=10

for n in 1 2 4 8 16 32 64; do
    export OMP_NUM_THREADS=$n
    echo "Setting OMP_NUM_THREADS=$n" >&2
    export GOMP_CPU_AFFINITY="0-$(($n - 1))"

    echo "Running ./kmeans_omp_array_lock" >&2
    ./kmeans_omp_array_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_clh_lock" >&2
    ./kmeans_omp_clh_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_critical" >&2
    ./kmeans_omp_critical -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_naive" >&2
    ./kmeans_omp_naive -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_nosync_lock" >&2
    ./kmeans_omp_nosync_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_pthread_mutex_lock" >&2
    ./kmeans_omp_pthread_mutex_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_pthread_spin_lock" >&2
    ./kmeans_omp_pthread_spin_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_tas_lock" >&2
    ./kmeans_omp_tas_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
    echo "Running ./kmeans_omp_ttas_lock" >&2
    ./kmeans_omp_ttas_lock -s $SIZE -n $COORDS -c $CLUSTERS -l $LOOPS
done
```

Τεχνικές συγχρονισμού

1) pthread_mutex_lock
Σε αυτήν την τεχνική χρησιμοποιείται ένα κλείδωμα αμοιβαίου αποκλεισμού. Αν ένα νήμα προσπαθήσει να δεσμεύσει το κλείδωμα και να μπει στο κρίσιμο τμήμα και αποτύχει, τότε μπλοκάρει μέχρι το κλείδωμα να απελευθερωθεί και ξαναπροσπαθήσει τότε. Είναι σαν να μπαίνει σε μια αίθουσα αναμονής και να ξαναπροσπαθεί να δεσμεύσει το κλείδωμα μόνο όταν το αφήσει κάποιος.

2) pthread_spin_lock
Σε αυτήν την τεχνική, όταν ένα νήμα προσπαθεί να δεσμεύσει το κλείδωμα, όσο δεν τα καταφέρνει, συνεχίζει να προσπαθεί παρ'όλο που δεν απελευθερώθηκε από την προηγούμενη φορά που το

δοκίμασε. Δηλαδή εκτελεί busy waiting.

3) tas_lock
Σε αυτήν την τεχνική χρησιμοποιείται η ατομική εντολή test and set. Όταν εκτελείται θέτει το κλείδωμα(μεταβλητή state) σε 1 και γυρνάει την προηγούμενη τιμή της(μεταβλητή test). Αν η προηγούμενη τιμή είναι 0 τότε το νήμα δέσμευσε επιτυχώς το κλείδωμα. Συνήθως αυτήν την τεχνική την χρησιμοποιούμε με while(tas), οπότε είναι busy waiting και δημιουργεί μεγάλη συμφόρηση στο δίαυλο λόγω του πρωτοκόλλου συνάφειας της κρυφής μνήμης.

4) ttas_lock
Σε αυτήν την τεχνική το νήμα πρώτα διαβάζει το κλείδωμα, όσο δεν είναι ελεύθερο απλά περιμένει. Μόνο όταν φαίνεται ελεύθερο θα προσπαθήσει να το δεσμεύσει με την εντολή test_and_set 1 φορά μόνο. Αν δεν τα καταφέρει τότε ξαναρχίζει να διαβάζει μέχρι να ελευθερωθεί. Αυτή η τεχνική χρησιμοποιεί λιγότερο τον δίαυλο σε σχέση με την απλή test_and_set αλλά σε πολλά νήματα συνεχίζει να προκαλεί μεγάλη συμφόρηση. Γενικά επιδέχεται βελτίωση με εκθετική οπισθοχώρηση αλλά δεν το εξετάζουμε σε αυτήν την άσκηση.

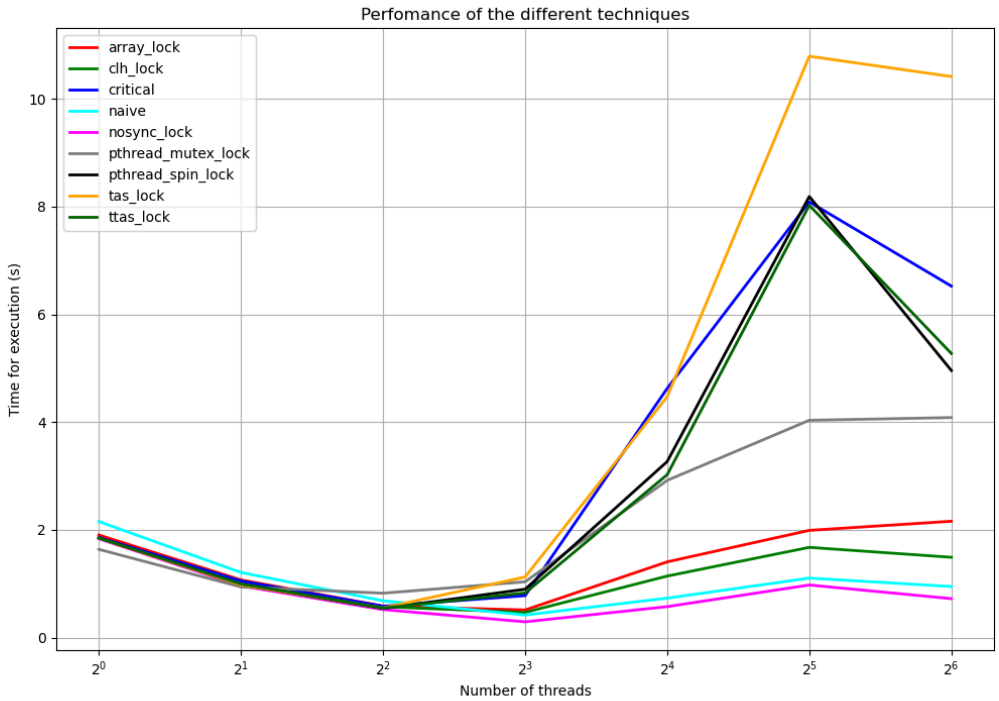
5) array_lock
Σε αυτήν την τεχνική κάθε νήμα έχει μια δική του μεταβλητή slot, ένα global πίνακα flag και που είναι τώρα το τέλος της ουράς. Κάθε φορά που προσπαθεί ένα νήμα να πάρει το κλείδωμα, παίρνει το τέλος της ουράς και κάνει ατομική αύξηση κατά 1, θέτει αυτό ως δικό του slot και περιμένει πότε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

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

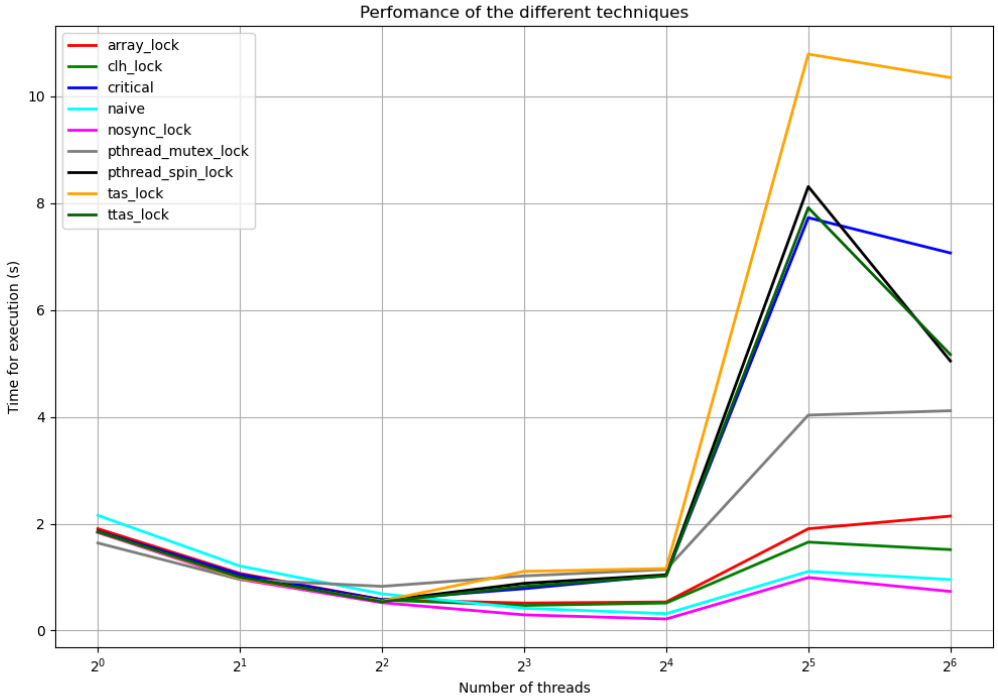
7) pragma omp critical
Θα αξιολογήσουμε και την επίδοση της βιβλιοθήκης του OpenMP για το κρίσιμο τμήμα.

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

Αποτελέσματα



Παρατηρούμε πως :
1) Την χειρότερη επίδοση την έχει η τεχνική tas καθώς συνέχεια θέτει τιμές στο κλείδωμα και δημιουργεί την περισσότερη συμφόρηση στο δίαυλο.
2) Η χρήση του omp critical έχει πολύ μεγάλο κόστος υλοποίησης και είναι η δεύτερη χειρότερη.
3) Οι τεχνικές ttas και pthread_spin_lock έχουν παρόμοια απόδοση. Η ttas τα πηγαίνει αρκετά καλύτερα από την tas για πολλά νήματα.
4) Η καλύτερη έτοιμη υλοποίηση από βιβλιοθήκη είναι η pthread_mutex_lock όπως αναμένεται.
5) Οι τεχνικές array_lock, clh_lock κλιμακώνουν πολύ καλά για πολλά νήματα καθώς κάθε νήμα που θέλει να μπει στο κρίσιμο τμήμα εκτελεί πάντα συγκεκριμένο αριθμό αναθέσεων σε κοινές μεταβλητές. Η clh τα πάει καλύτερα καθώς έχει μόνο ένα κοινό δείκτη και όχι ολόκληρο πίνακα.
6) Η χρήση του omp atomic βλέπουμε πως βελτιστοποιεί πάρα πολύ τον συγχρονισμό που χρειάζεται και τα πηγαίνει σχεδόν το ίδιο καλά με το κάτω όριο που είναι η εκτέλεση χωρίς συγχρονισμό.
7) Οι 3 καλύτερες τεχνικές για αυτήν την άσκηση(array, clh, omp atomic) κλιμακώνουν πολύ καλά για 1 cluster πυρήνων του sandman, δηλαδή 8 νήματα. Αν χρησιμοποιούσαμε hyperthreading για τα 16 νήματα, δηλαδή βάζαμε τα 8 τελευταία νήματα εκτός των 64 λογικών, θα δούμε κλιμάκωση και για 16 νήματα όπως φαίνεται παρακάτω :


Perfomance of the different techniques
array_lock
clh_lock
critical
naive
nosync_lock
pthread_mutex_lock
pthread_spin_lock
tas_lock
ttas_lock
Time for execution (s)
Number of threads
0
2
4
6
8
10
2^0
2^1
2^2
2^3
2^4
2^5
2^6

Ταυτόχρονες Δομές δεδομένων

Σε αυτό το ερώτημα εξετάζουμε πως κλιμακώνουν διάφορες ταυτόχρονες υλοποιήσεις για μια απλά συνδεδεμένη λίστα.
Οι ταυτόχρονες υλοποιήσεις που θα εξετάσουμε είναι οι εξής :

1) Coarse-grain locking
Σε αυτήν την υλοποίηση υπάρχει ένα γενικό κλείδωμα για όλη την δομή. Για κάθε προσθήκη ή αφαίρεση στοιχείου στη λίστα, το νήμα προσπαθεί να δεσμεύσει το κλείδωμα και να κάνει την κατάλληλη αλλαγή. Είναι πολύ απλό στην υλοποίηση όμως δεν θα κλιμακώσει καθόλου καθώς όλοι περιμένουν το ίδιο κλείδωμα και δεν εκμεταλλευόμαστε καθόλου πως δεν θα θέλουν όλοι να αλλάξουν κοντινά μέρη της λίστας.

2) Fine-grain locking
Σε αυτήν την υλοποίηση υπάρχει ένα κλείδωμα για κάθε στοιχείο της λίστας. Ο τρόπος διάσχισης είναι hand-over-hand locking δηλαδή ένα νήμα προσπαθεί να δεσμεύσει τον επόμενο, όταν τα καταφέρει, αφήνει τον προηγούμενο. Μπορεί να δουλέψει καλύτερα από την coarse grain σε συγκεκριμένες περιπτώσεις αλλά το σημαντικότερο πρόβλημα είναι πως αν ένα νήμα θέλει να αλλάξει κάτι που βρίσκεται νωρίς στη λίστα, μπλοκάρει όλα τα άλλα νήματα που θέλουν να ψάξουν ή αλλάξουν κάτι που είναι πιο μετά στη λίστα.

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

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

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

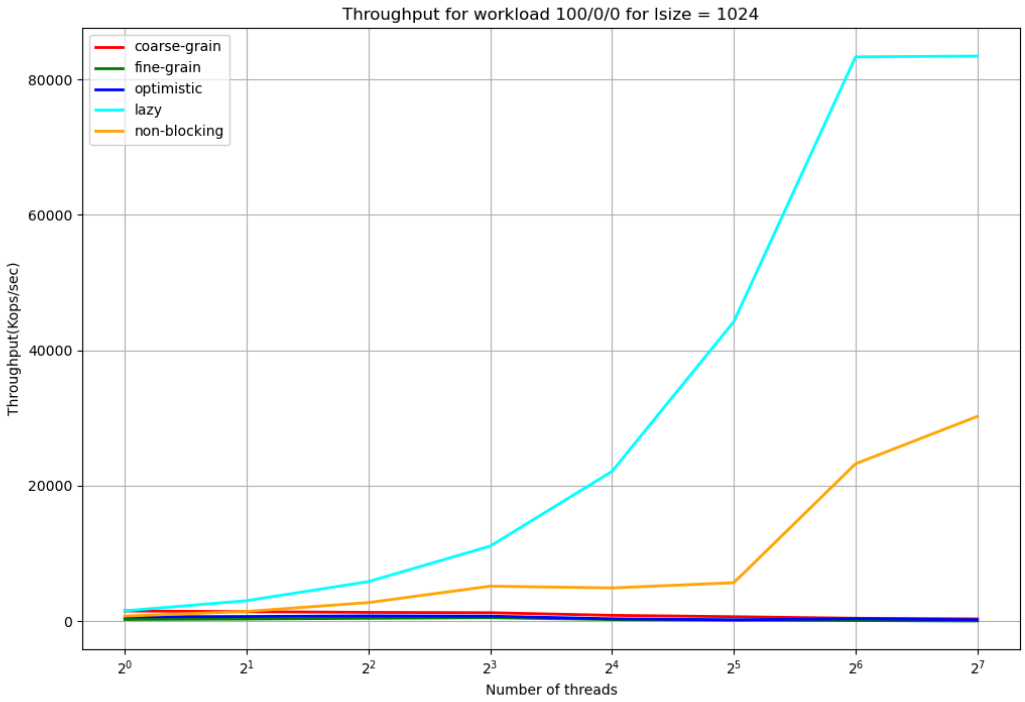
Μας δίνονται έτοιμες όλες οι παραπάνω ταυτόχρονες υλοποιήσεις. Για την ζητούμενη εκτέλεση, το σειριακό πρόγραμμα εκτελέστηκε μόνο με 1 thread αλλιώς θα υπάρχει πρόβλημα, για 128 νήματα χρησιμοποιήθηκε oversubscription. Δηλαδή η μεταβλητή MT_CONF τέθηκε σε 0,1,...63,0,1,,...63 ώστε να δημιουργηθούν και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

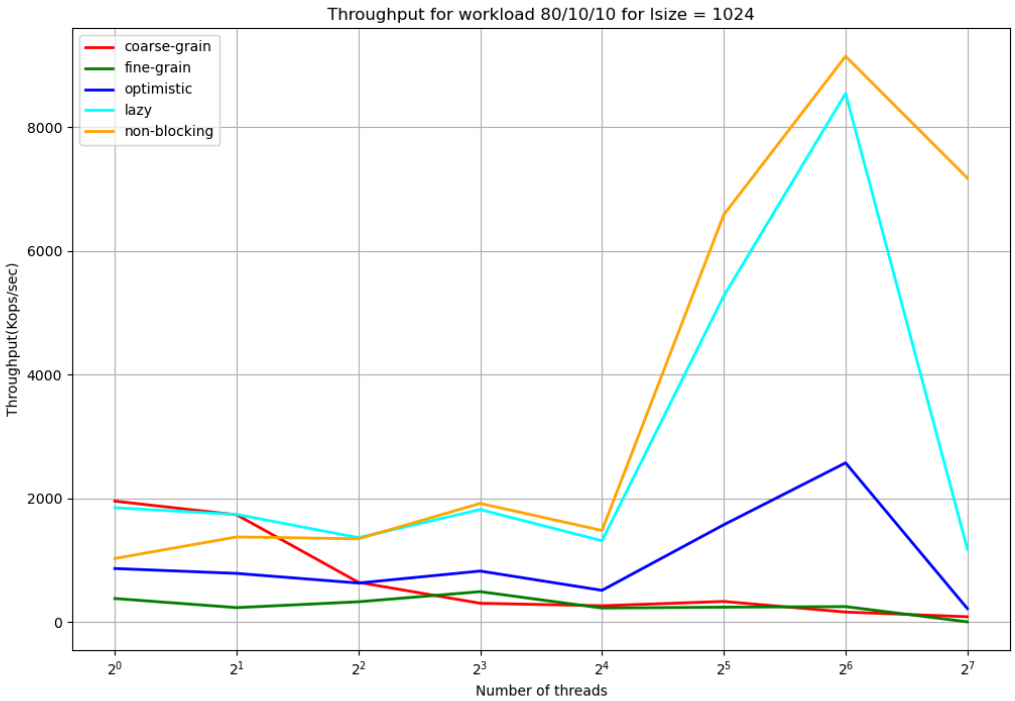
```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_conc_ll

## Output and error files
#PBS -o run_conc_ll.out
#PBS -e run_conc_ll.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

##How long should the job run for?
#PBS -l walltime=01:00:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab09/a2_new/a2/conc_ll

choices=(
    "100 0 0"
    "80 10 10"
    "20 40 40"
    "0 50 50"
)

for lsize in 1024 8192; do
  export LSIZE=$lsize
  echo "LSIZE=$LSIZE"
  for choice in "${choices[@]}"; do
    read -r CONTAINS_PCT ADD_PCT REMOVE_PCT <<< "$choice"
    export MT_CONF="0"
    echo "serial"
    ./x.serial $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
    for n in 1 2 4 8 16 32 64 128; do
      if [ "$n" -eq 128 ]; then
        # Special case for n=128 for over subscription
        export MT_CONF="$(seq -s, 0 63),$(seq -s, 0 63)"
      else
        # Default case for n=1, 2, 4, ..., 64
        export MT_CONF=$(seq -s, 0 $((n-1)))
      fi
        echo "coarse-grain"
        ./x.cgl $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
        echo "fine-grain"
        ./x.fgl $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
        echo "optimistic"
        ./x.opt $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
        echo "lazy"
        ./x.lazy $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
        echo "non-blocking"
        ./x.nb $LSIZE $CONTAINS_PCT $ADD_PCT $REMOVE_PCT
    done
  done
done
```

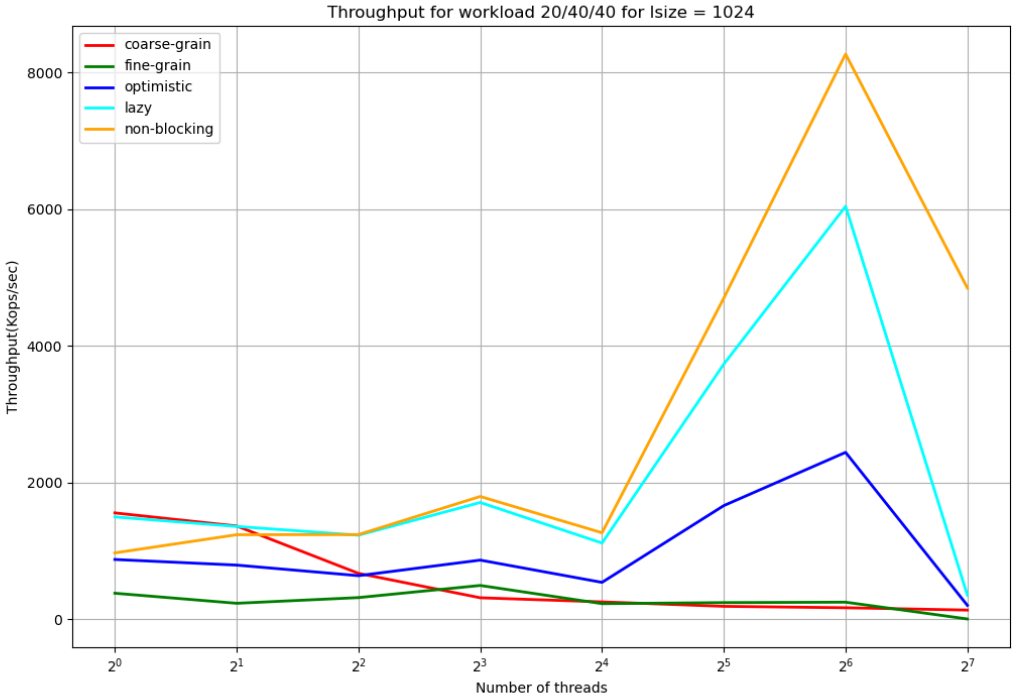
Αποτελέσματα

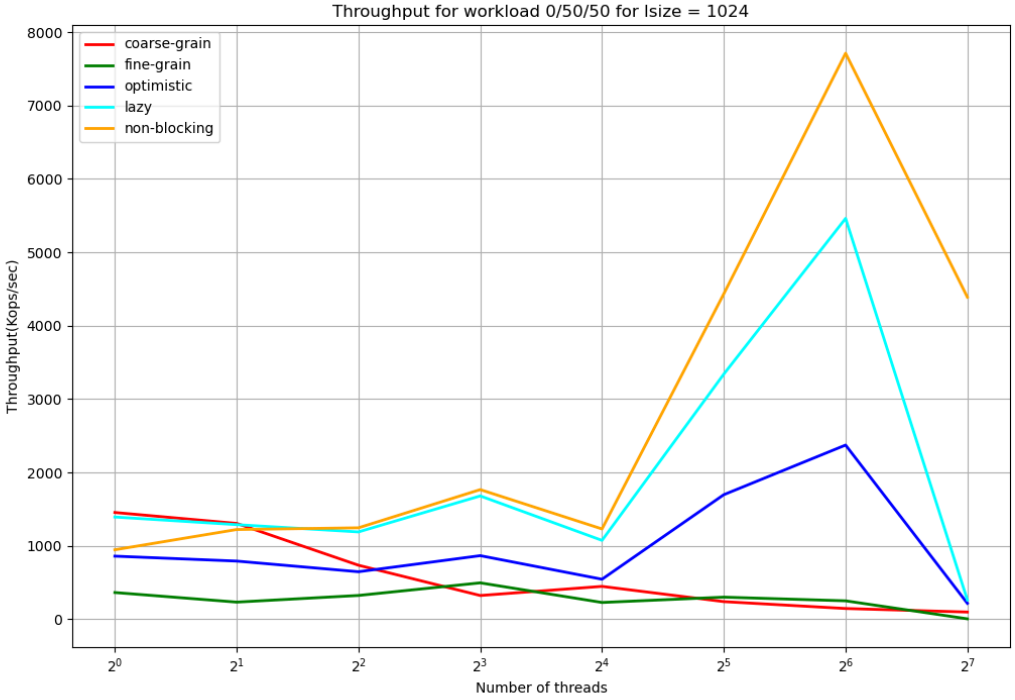
Παρουσιάζονται τα Kops/sec για την κάθε περίπτωση κάθε και το κανονικοποιημένο Kops/sec δηλαδή ανά νήμα.

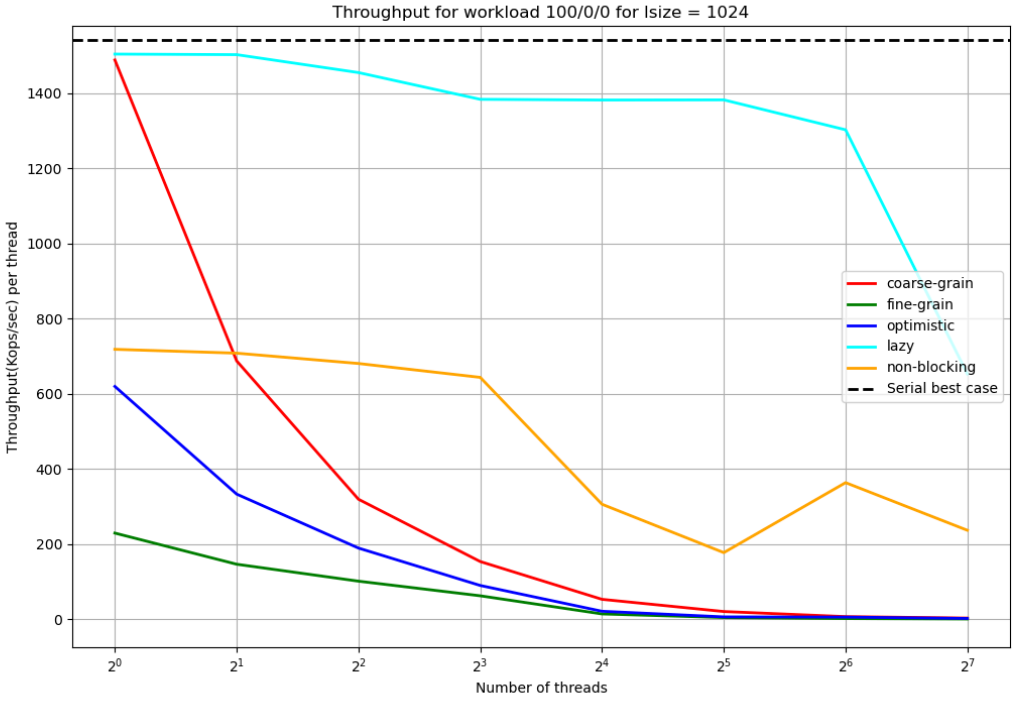
Για μήκος λίστας 1024

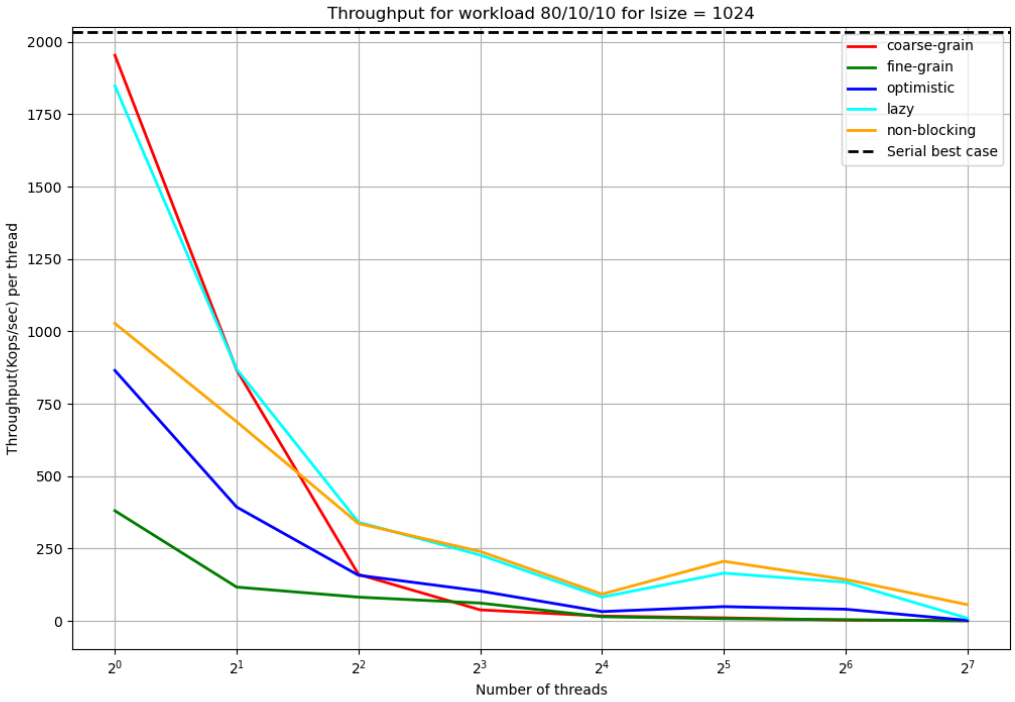


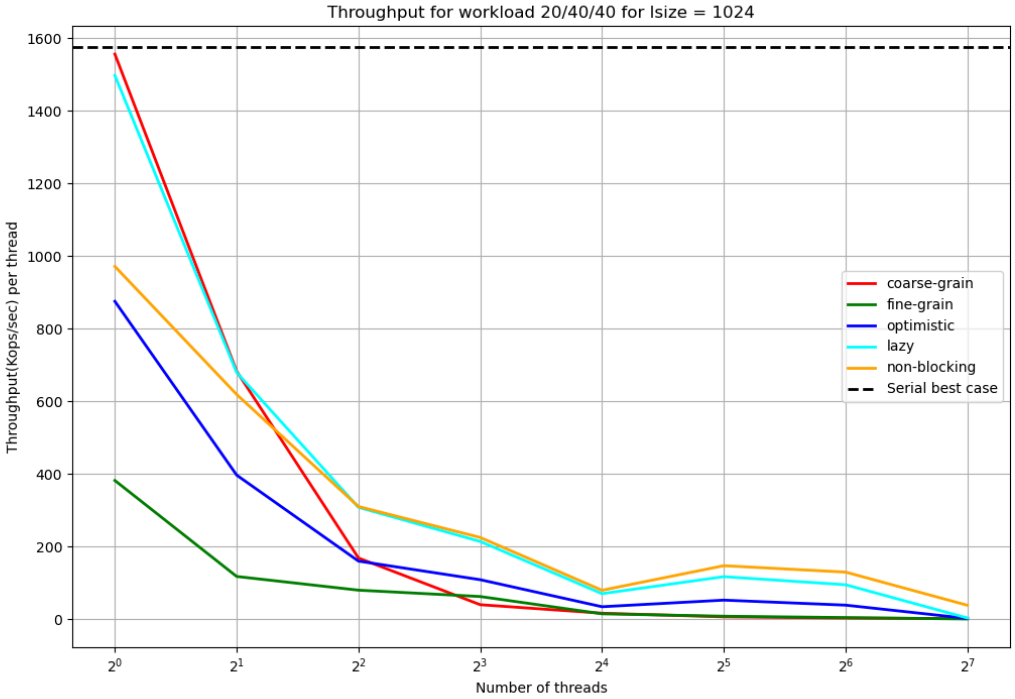


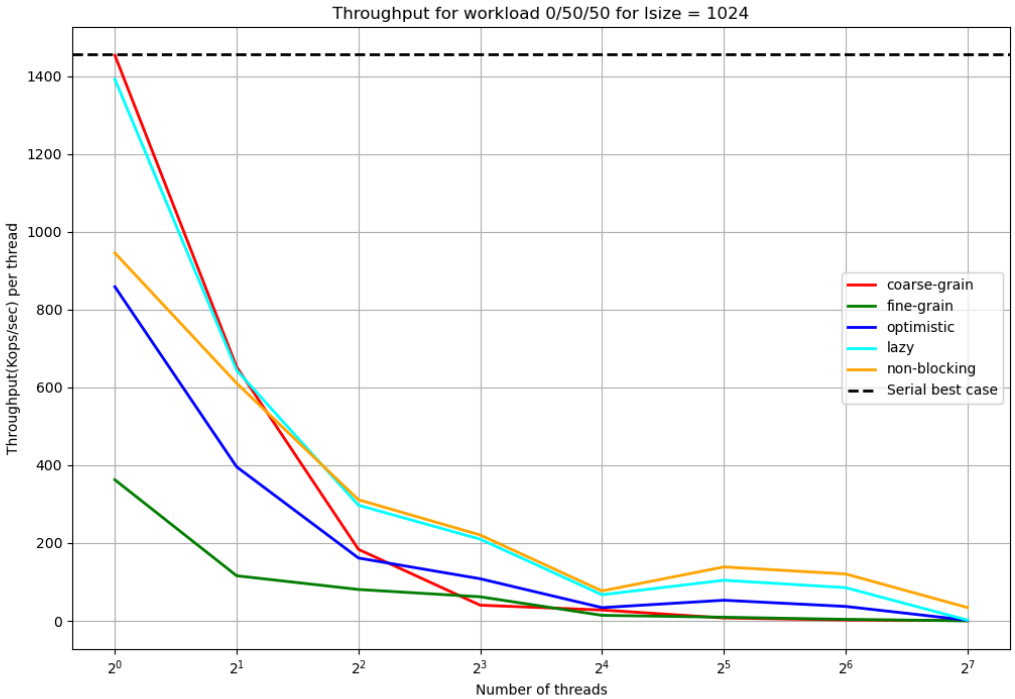

Throughput for workload 20/40/40 for lsize = 1024
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Throughput(Kops/sec)
0
2000
4000
6000
8000
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
Number of threads


Throughput for workload 0/50/50 for lsize = 1024
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Throughput(Kops/sec)
0
1000
2000
3000
4000
5000
6000
7000
8000
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
Number of threads

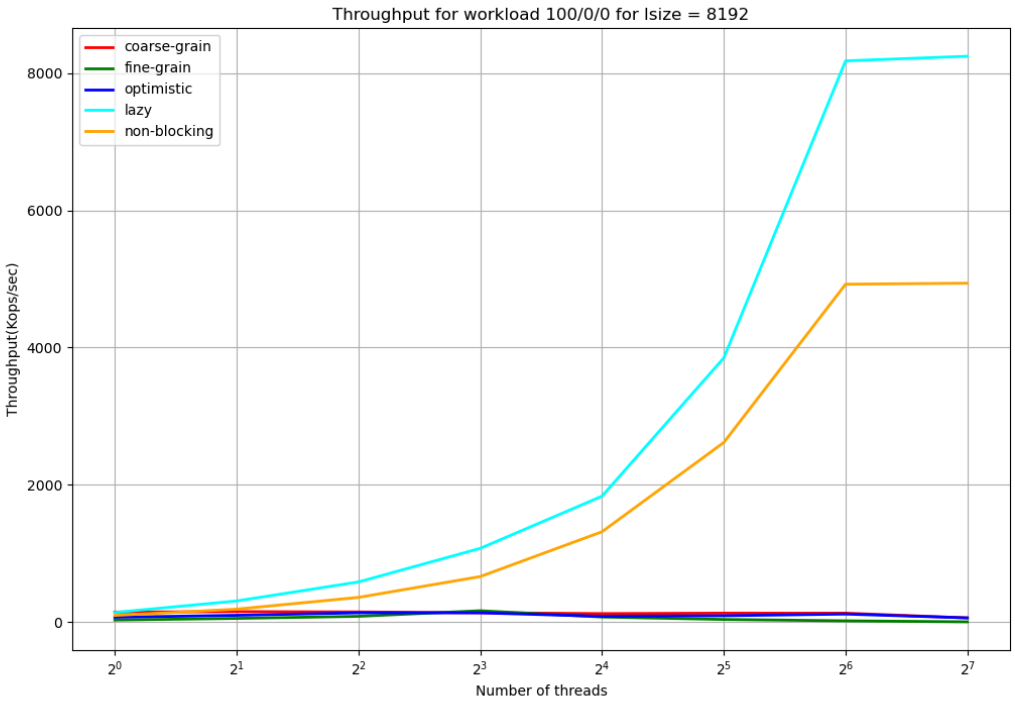

Throughput for workload 100/0/0 for lsize = 1024
Throughput(Kops/sec) per thread
Number of threads
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Serial best case

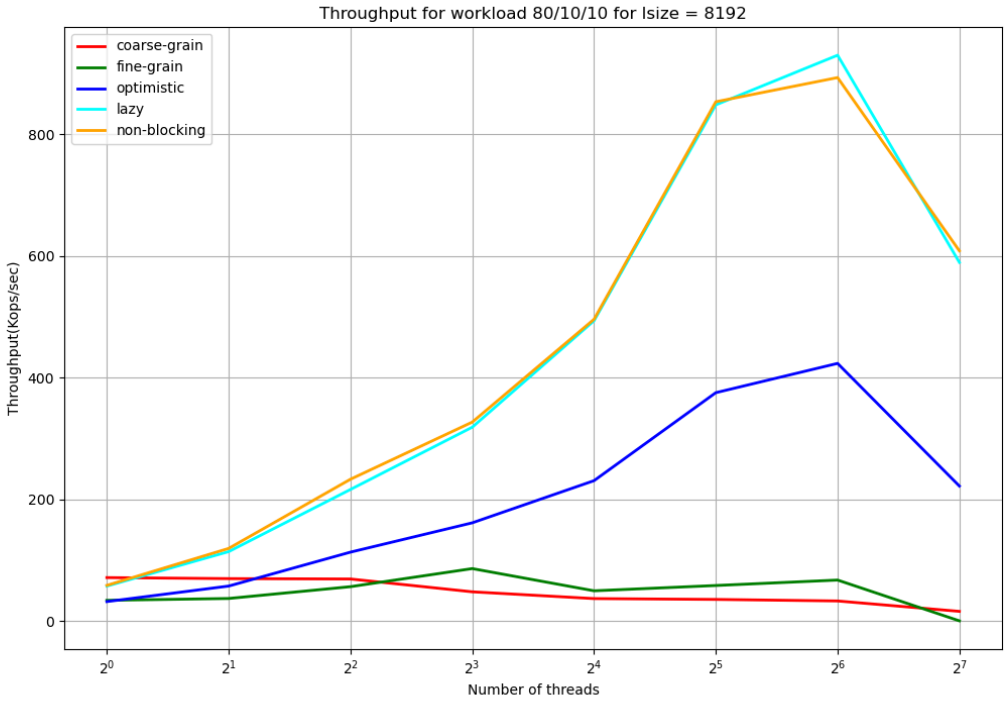

Throughput for workload 80/10/10 for lsize = 1024
Throughput(Kops/sec) per thread
Number of threads
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Serial best case

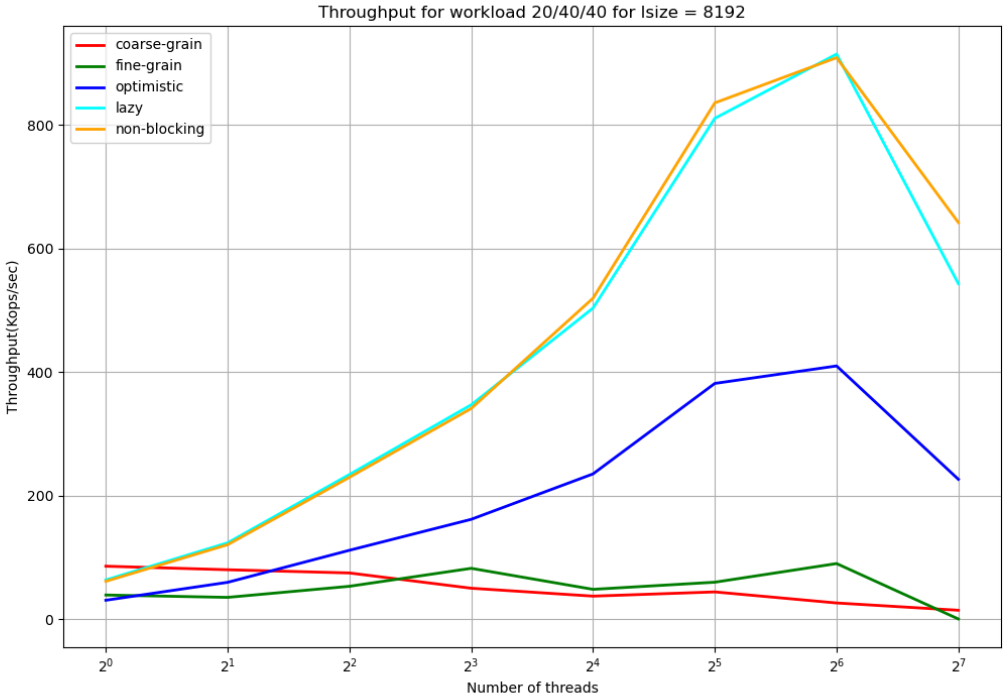

Throughput for workload 20/40/40 for lsize = 1024
Throughput(Kops/sec) per thread
Number of threads
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Serial best case

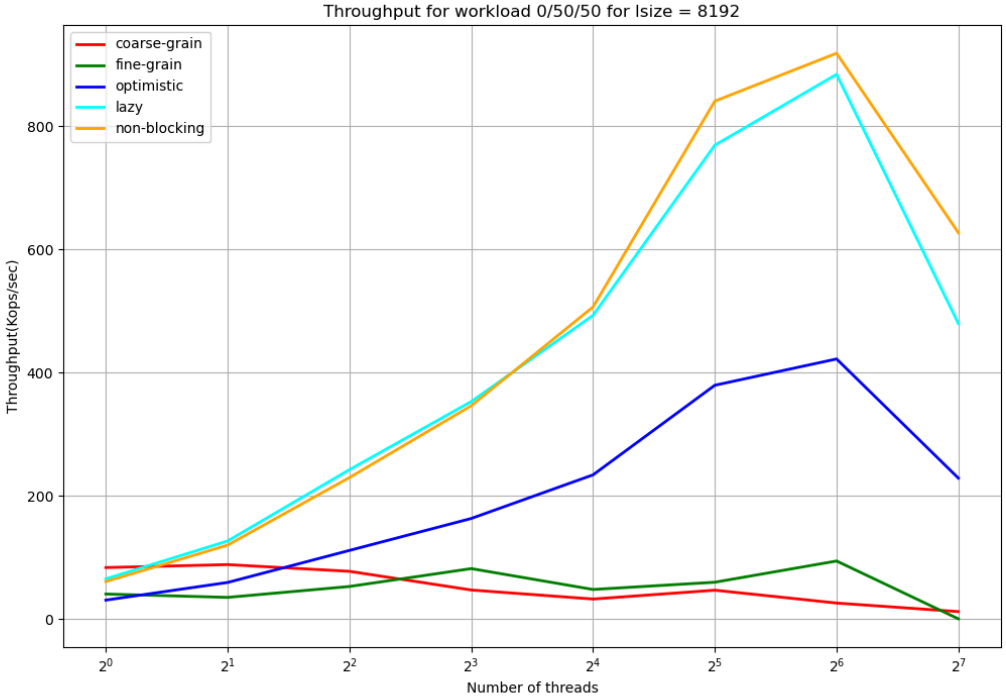

Throughput for workload 0/50/50 for lsize = 1024
Throughput(Kops/sec) per thread
Number of threads
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Serial best case

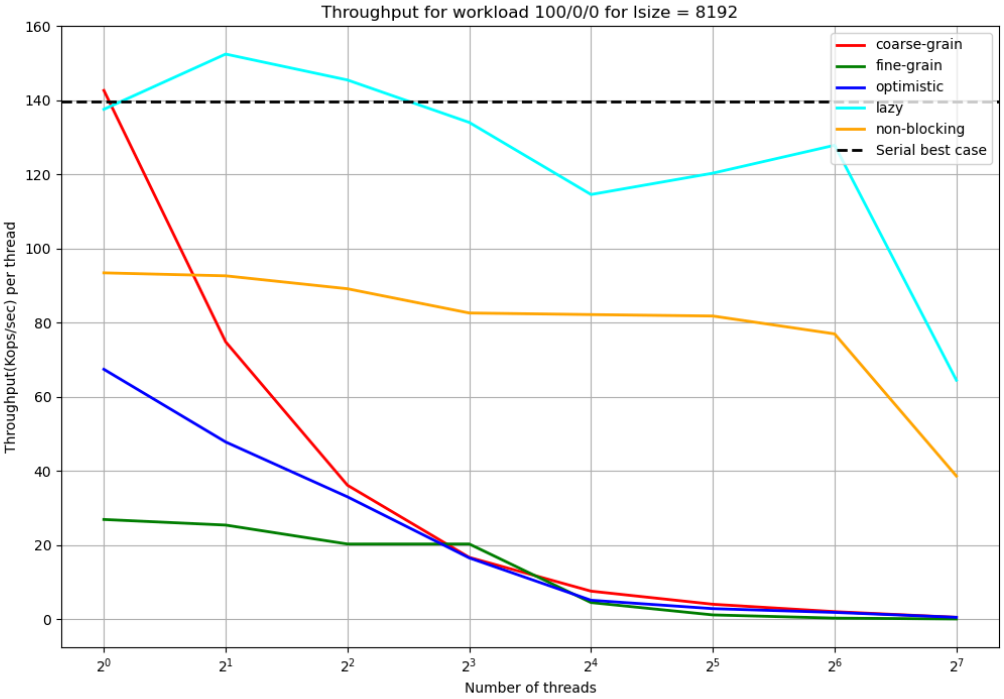
Για μήκος λίστας 8192

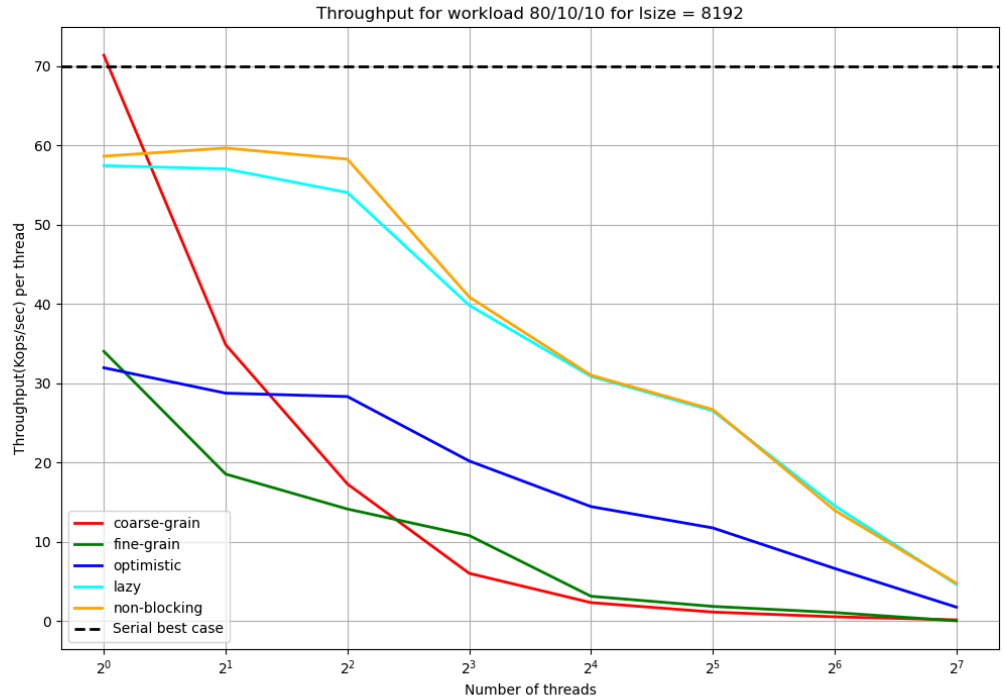


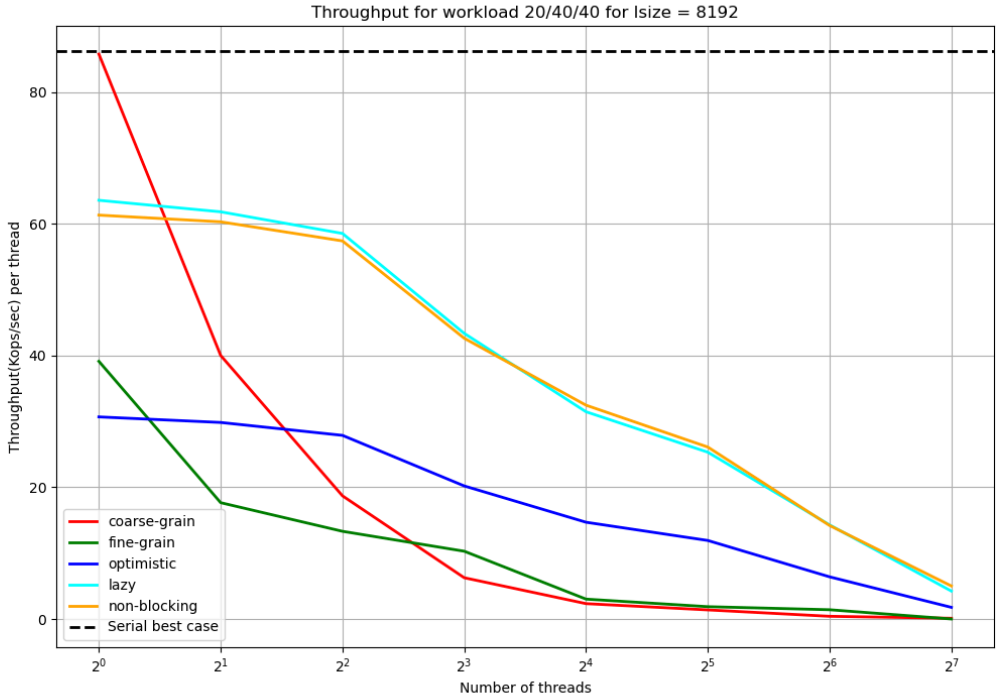


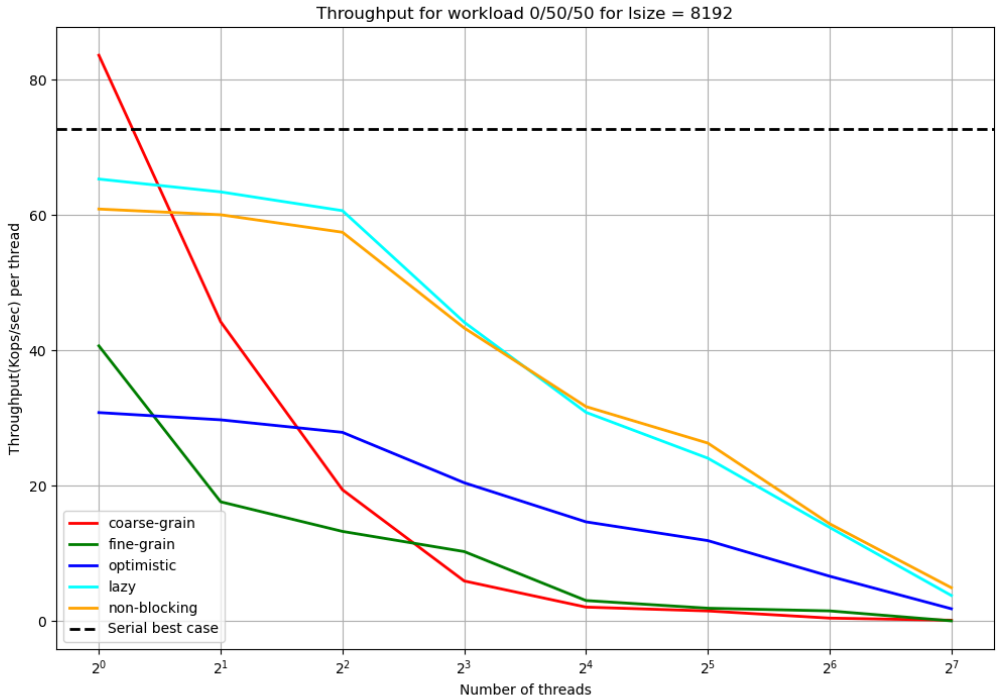

Throughput for workload 20/40/40 for lsize = 8192
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Throughput(Kops/sec)
0
200
400
600
800
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
Number of threads


Throughput for workload 0/50/50 for lsize = 8192
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Throughput(Kops/sec)
0
200
400
600
800
2^0
2^1
2^2
2^3
2^4
2^5
2^6
2^7
Number of threads


Throughput for workload 100/0/0 for lsize = 8192
Throughput(Kops/sec) per thread
Number of threads
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Serial best case


Throughput for workload 80/10/10 for lsize = 8192
Throughput(Kops/sec) per thread
Number of threads
coarse-grain
fine-grain
optimistic
lazy
non-blocking
Serial best case





Παρατηρούμε πως :
1) Στην περίπτωση που έχουμε μόνο contains την χειρότερη επίδοση την έχει η fine grain που έχει τεράστιο overhead καθώς έχει κλείδωμα για κάθε κόμβο, ενώ η coarse grain που έχει ένα global κλείδωμα τα πάει καλύτερα.

2) Στην ίδια περίπτωση για το μεγάλο μέγεθος λίστας οι coarse grain, fine grain, optimistic έχουν φυσιολογικές επιδόσεις μέχρι 8 νήματα, με το που χρησιμοποιήσουμε παραπάνω από 1 cluster με μη κοινή L3 cache τα πάνε χάλια. Αυτό συμβαίνει διότι διασχίζουν την λίστα πολλές φορές για κάθε αλλαγή και αυτό σε συνδυασμό με το πρωτόκολλο συνάφεια κρυφής μνήμης προκαλεί τεράστιες καθυστερήσεις.
3) Η non blocking έχει πιο κακή επίδοση στη μικρή λίστα ενώ στη μεγάλη είναι το ίδιο καλή με την lazy. Αυτό είναι λογικό καθώς η φυσική διαγραφή δεν γίνεται πάντα.
4) Η lazy synchronization έχει τις καλύτερες επιδόσεις σχεδόν σε όλες τις περιπτώσεις και φτάνει πολύ κοντά στο ιδανικό όταν έχουμε μόνο contains. Αυτό συμβαίνει διότι διατρέχει μόνο 1 φορά τη λίστα για κάθε πράξη και δεν δεσμεύει περιττά κλειδώματα.
5) Δεν υπάρχουν σημαντικές διαφορές ανάμεσα στα υπόλοιπα workloads καθώς η contain είναι πιο wait-free μέθοδος και άμα κάνουν όλοι το ίδιο δεν περιμένουν τόσο.
6) Σε όλες τις περιπτώσεις το oversubscription είναι κακή ιδέα καθώς η επίδοση ή μένει ίδια(όταν έχουμε μόνο contains) ή πέφτει αισθητά από τα 64 νήματα. Βλέπουμε πως όταν εμπλακεί το λειτουργικό για το scheduling η επίδοση πέφτει πολύ.

Παράρτημα

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

scraping_kmeans.py

```python
import os
import matplotlib.pyplot as plt

#replace with the actual path
folder_path = "../"
file_name = "run_kmeans_hyper.err"

total_path = os.path.join(folder_path, file_name)

with open(total_path, "r") as file:
    data = file.read()

omp_num_threads = 1
current_running = 'array_lock'
#we need just one the two, but may we need both
results_dict= {}
results_array = {}
for line in data.splitlines():
    words = line.split()
    if len(words) == 0:
        continue
    #set the nthreads, set what is running or the exec time
    if words[0] == 'Setting':
        omp_num_threads = int(words[1].split('=')[-1])
    if words[0] == 'Running':
        temp = words[1].split('_')[2:]
        current_running = '_'.join(temp)
    if words[0] == 'nloops':
        temp_res = float(words[5][0:-2])
        if current_running not in results_dict:
            results_dict[current_running] = {}
            results_array[current_running] = []
        results_dict[current_running][omp_num_threads] = temp_res
        results_array[current_running].append(temp_res)


nthreads = [1, 2, 4, 8, 16, 32, 64]
colours = ['red', 'green', 'blue', 'cyan', 'magenta', 'gray', 'black', 'orange', 'darkGreen']

plt.figure(figsize=(12, 8))
counter = 0
for technique in results_array.keys():
        plt.plot(nthreads,  results_array[technique],  linewidth=2,  color=colours[counter],
label=technique)
    counter += 1
plt.xlabel('Number of threads')
plt.xscale('log', base=2)
plt.ylabel('Time for execution (s)')
plt.title("Perfomance of the different techniques")
plt.grid(True)
plt.legend(loc='best')
plt.savefig("kmeans_results_hyper.png")
plt.close()
```

scraping_conc_ll.py

```python
import os
import matplotlib.pyplot as plt
import numpy as np

#replace with the actual path
folder_path = "../"
file_name = "run_conc_ll.out"
output_dir = "./"
```

```
9
10  total_path = os.path.join(folder_path, file_name)
11
12  with open(total_path, "r") as file:
13      data = file.read()
14
15  lsize = 1024
16  current_running = 'serial'
17  nthreads = 1
18  workload = '100/0/0'
19  res_1024 = {}
20  res_8192 = {}
21  for line in data.splitlines():
22      words = line.split()
23      if len(words) == 0:
24          continue
25      #set lsize
26      if words[0].split('=')[0] == 'LSIZE':
27          lsize = int(words[0].split('=')[1])
28      #get and save the result
29      if words[0] == 'Nthreads:':
30          #only need if we didn't run them in ascending order
31          nthreads = int(words[1])
32          workload = words[5]
33          throughput = float(words[-1])
34          if lsize == 1024:
35              if current_running not in res_1024.keys():
36                  res_1024[current_running] = {}
37              if workload not in res_1024[current_running].keys():
38                  res_1024[current_running][workload] = []
39              res_1024[current_running][workload].append(throughput)
40          elif lsize == 8192:
41              if current_running not in res_8192.keys():
42                  res_8192[current_running] = {}
43              if workload not in res_8192[current_running].keys():
44                  res_8192[current_running][workload] = []
45              res_8192[current_running][workload].append(throughput)
46      #save the current runner for the next result
47      if len(words) == 1 and len(line.split('=')) == 1:
48          current_running=words[0]
49
50  nthreads = [1, 2, 4, 8, 16, 32, 64, 128]
51  colours = ['red', 'green', 'blue', 'cyan', 'orange']
52  workloads = ['100/0/0', '80/10/10', '20/40/40', '0/50/50']
53
54  #for a specific size plot all the workloads in a different plot
55  for i in range(0, len(workloads)):
56      plt.figure(figsize=(12,8))
57      counter = 0
58      for technique in res_1024.keys():
59          if technique == 'serial':
60              continue
                        plt.plot(nthreads,   res_1024[technique][workloads[i]],   linewidth=2,
61  color=colours[counter], label=technique)
62          counter += 1
63
64      plt.xlabel('Number of threads')
65      plt.xscale('log', base=2)
66      plt.ylabel('Throughput(Kops/sec)')
67      #plt.yscale('log', base=10)
68      plt.title(f"Throughput for workload {workloads[i]} for lsize = 1024")
69      plt.grid(True)
70      plt.legend(loc='best')
71      sanitized_workload = workloads[i].replace('/', '_')
72      plt.savefig(f"{output_dir}/conc_ll_1024_{sanitized_workload}.png")
73      plt.close()
74
75      #plot the normalized perfomance as well
76      plt.figure(figsize=(12,8))
77      counter = 0
78      for technique in res_1024.keys():
79          if technique == 'serial':
80              continue
```

```
        temp = np.array(res_1024[technique][workloads[i]])
        #divide by nthreads so we have kops / thread
        temp = np.divide(temp, nthreads)

        plt.plot(nthreads, temp, linewidth=2, color=colours[counter], label=technique)
        counter += 1
    #add serial for comparison
    serial_res = res_1024['serial'][workloads[i]][0]
    plt.axhline(y=serial_res, color='black', linestyle='--', linewidth=2, label='Serial best case')

    plt.xlabel('Number of threads')
    plt.xscale('log', base=2)
    plt.ylabel('Throughput(Kops/sec) per thread')
    #plt.yscale('log', base=10)
    plt.title(f"Throughput for workload {workloads[i]} for lsize = 1024")
    plt.grid(True)
    plt.legend(loc='best')
    sanitized_workload = workloads[i].replace('/', '_')
    plt.savefig(f"{output_dir}/conc_ll_norm_1024_{sanitized_workload}.png")
    plt.close()

#do for 8192 as well
for i in range(0, len(workloads)):
    plt.figure(figsize=(12,8))
    counter = 0
    for technique in res_8192.keys():
        if technique == 'serial':
            continue
                    plt.plot(nthreads,   res_8192[technique][workloads[i]],   linewidth=2, color=colours[counter], label=technique)
        counter += 1

    plt.xlabel('Number of threads')
    plt.xscale('log', base=2)
    plt.ylabel('Throughput(Kops/sec)')
    #plt.yscale('log', base=10)
    plt.title(f"Throughput for workload {workloads[i]} for lsize = 8192")
    plt.grid(True)
    plt.legend(loc='best')
    sanitized_workload = workloads[i].replace('/', '_')
    plt.savefig(f"{output_dir}/conc_ll_8192_{sanitized_workload}.png")
    plt.close()

    #plot the normalized perfomance as well
    plt.figure(figsize=(12,8))
    counter = 0
    for technique in res_8192.keys():
        if technique == 'serial':
            continue
        temp = np.array(res_8192[technique][workloads[i]])
        #divide by nthreads so we have kops / thread
        temp = np.divide(temp, nthreads)

        plt.plot(nthreads, temp, linewidth=2, color=colours[counter], label=technique)
        counter += 1
    #add serial for comparison
    serial_res = res_8192['serial'][workloads[i]][0]
    plt.axhline(y=serial_res, color='black', linestyle='--', linewidth=2, label='Serial best case')

    plt.xlabel('Number of threads')
    plt.xscale('log', base=2)
    plt.ylabel('Throughput(Kops/sec) per thread')
    #plt.yscale('log', base=10)
    plt.title(f"Throughput for workload {workloads[i]} for lsize = 8192")
    plt.grid(True)
    plt.legend(loc='best')
    sanitized_workload = workloads[i].replace('/', '_')
    plt.savefig(f"{output_dir}/conc_ll_norm_8192_{sanitized_workload}.png")
    plt.close()
```