Συστήματα Παράλληλης Επεξεργασίας
Δεύτερη Εργαστηριακή Άσκηση

Αναστασία-Χριστίνα Λίβα 03119029
Γιώργος Μυστριώτης 03119065
Νικόλας Σταματόπουλος 03119020

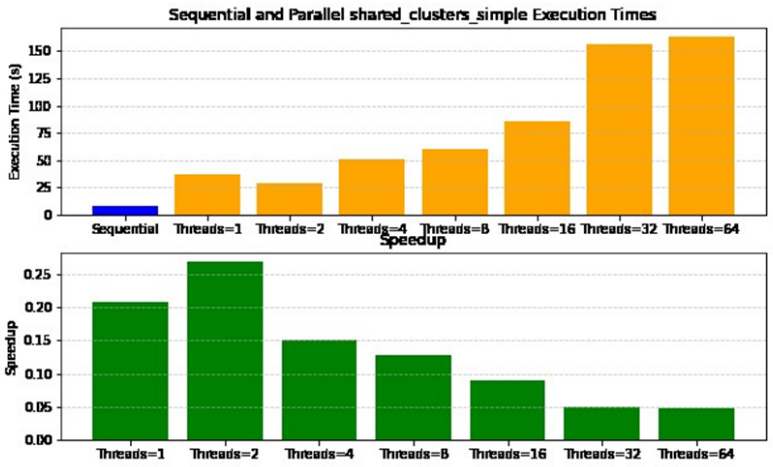
Νοέμβριος 2023

Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Shared Clusters

1. Παρατηρούμε ότι η παραλληλοποίηση του αλγορίθμου με τη χρήση κλειδωμάτων δεν επιφέρει καμία επιτάχυνση στην ταχύτητα εκτέλεσης. Αντιθέτως, βλέπουμε από τα διαγράμματα ότι όσο περισσότερο αυξάνονται τα threads που χρησιμοποιούμε, τόσο περισσότερο επιβραδύνεται το πρόγραμμα μας. Αυτό οφείλεται στο γεγονός ότι η χρήση κλειδωμάτων προκαλεί overhead, καθώς ο ανταγωνισμός για την κατάκτηση του lock αφήνει πολλά threads να κάνουν busy wait όσο περιμένουν να εισέλθουν στο κρίσιμο τμήμα, καταναλώνοντας περισσότερο χρόνο σε αναμονή παρά σε υπολογισμούς. Συγκεκριμένα στην περίπτωσή μας, οπού το κρίσιμο τμήμα είναι μικρό αλλά η «ζήτηση» για το κλείδωμα είναι μεγάλη (1 στον πίνακα newClusterSize και $numCoords στον πίνακα newClusters για κάθε object), το overhead που δημιουργείται είναι συγκρίσιμο με το χρόνο εκτέλεσης των υπολογισμών και γι' αυτό αναιρεί την επιτάχυνση του παραλληλισμού.

Έτσι βλέπουμε τα παρακάτω αποτελέσματα:



Σχήμα 1: shared_clusters_simple Results

Ο κώδικας του shared_clusters_simple φαίνεται παρακάτω:

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

// 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;
    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]
```

```
63      int nthreads;          // no. threads
64
65      pthread_mutex_t lock_cluster_size;
66      pthread_mutex_t lock_new_cluster;
67
68      nthreads = omp_get_max_threads();
69      printf("OpenMP Kmeans - Naive\t(number of threads: %d)\n", nthreads
        );
70
71      //initialize mutexes
72      pthread_mutex_init(&lock_cluster_size, NULL);
73      pthread_mutex_init(&lock_new_cluster, NULL);
74
75      // initialize membership
76      for (i=0; i<numObjs; i++)
77          membership[i] = -1;
78
79      // initialize newClusterSize and newClusters to all 0
80      newClusterSize = (typeof(newClusterSize)) calloc(numClusters,
        sizeof(*newClusterSize));
81      newClusters = (typeof(newClusters))  calloc(numClusters * numCoords
        , sizeof(*newClusters));
82
83      timing = wtime();
84
85      char *affinity = getenv("GOMP_CPU_AFFINITY");
86
87      if (affinity != NULL) {
88          printf("GOMP_CPU_AFFINITY: %s\n", affinity);
89      } else {
90          printf("GOMP_CPU_AFFINITY is not set.\n");
91      }
92
93      do {
94          // before each loop, set cluster data to 0
95          for (i=0; i<numClusters; i++) {
96              for (j=0; j<numCoords; j++)
97                  newClusters[i*numCoords + j] = 0.0;
98              newClusterSize[i] = 0;
99          }
100
101         delta = 0.0;
102
103         /*
104          * TODO: Detect parallelizable region and use appropriate
        OpenMP pragmas
105          */
106         #pragma omp parallel for shared(numObjs, newClusterSize,
        membership) private(i, index)
```

```
107         for (i=0; i<numObjs; i++) {
108             // find the array index of nearest cluster center
109             index = find_nearest_cluster(numClusters, numCoords, &
    objects[i*numCoords], clusters);
110 
111             // if membership changes, increase delta by 1
112             if (membership[i] != index)
113                 delta += 1.0;
114 
115             // assign the membership to object i
116             membership[i] = index;
117 
118             // update new cluster centers : sum of objects located
    within
119             /*
120              * TODO: protect update on shared "newClusterSize" array
121              */
122 
123             pthread_mutex_lock(&lock_cluster_size);
124             newClusterSize[index]++;
125             pthread_mutex_unlock(&lock_cluster_size);
126             #pragma omp parallel for shared(newClusters, objects,
    numCoords) private(j)
127             for (j=0; j<numCoords; j++) {
128                 /*
129                  * TODO: protect update on shared "newClusters" array
130                  */
131                 pthread_mutex_lock(&lock_new_cluster);
132                 newClusters[index*numCoords + j] += objects[i*numCoords
     + j];
133                 pthread_mutex_unlock(&lock_new_cluster);
134             }
135         }
136 
137         // average the sum and replace old cluster centers with
    newClusters
138         #pragma omp parallel for shared(numClusters, newClusterSize,
    numCoords, clusters, newClusters) private(i)
139         for (i=0; i<numClusters; i++) {
140             if (newClusterSize[i] > 0) {
141                 #pragma omp parallel for shared(numCoords,
    newClusterSize, clusters, newClusters) private(j)
142                 for (j=0; j<numCoords; j++) {
143                     clusters[i*numCoords + j] = newClusters[i*numCoords
     + j] / newClusterSize[i];
144                 }
145             }
146         }
147 
```

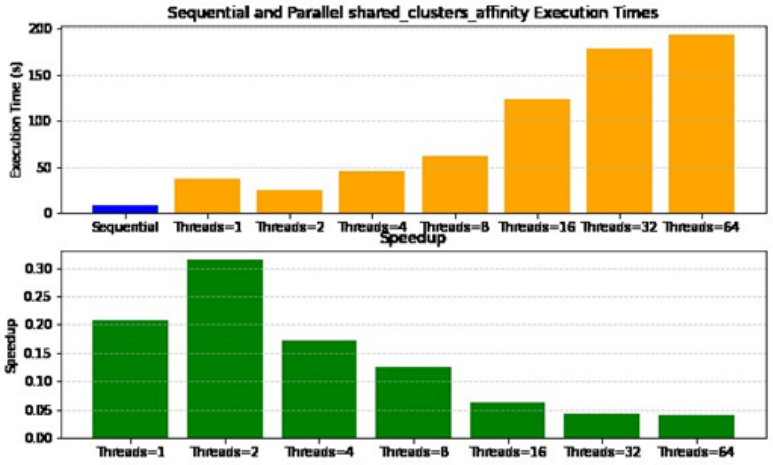
```
148         // Get fraction of objects whose membership changed during this
    loop. This is used as a convergence criterion.
149         delta /= numObjs;
150 
151         loop++;
152         printf("\r\tcompleted loop %d", loop);
153         fflush(stdout);
154     } while (delta > threshold && loop < loop_threshold);
155     timing = wtime() - timing;
156     printf("\n        nloops = %3d   (total = %7.4fs)  (per loop = %7.4
    fs)\n", loop, timing, timing/loop);
157 
158     free(newClusters);
159     free(newClusterSize);
160 }
```

2. Ορίζοντας τη μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY επιτυγχάνουμε thread binding, δηλαδή καθορίζουμε σε ποιον πυρήνα του μηχανήματος θα τρέξει το κάθε thread και το «προσδένουμε» σε αυτόν για όλη την εκτέλεση του προγράμματος. Όπως επιβεβαιώνεται και από τα αντίστοιχα διαγράμματα χρόνων εκτέλεσης και speedup, αυτό επιφέρει βελτιωμένη απόδοση σε σχέση με το προηγούμενο ερώτημα. Αυτό δικαιολογείται από τα εξής:

α′) Με πρόσδεση των νημάτων σε συγκεκριμένους πυρήνες ελαχιστοποιείται ο ανταγωνισμός για την κρυφή μνήμη και

β′) Ελαχιστοποιείται η καθυστέρηση λόγω εναλλαγής πυρήνων (context switching)

Έτσι παίρνουμε τα παρακάτω αποτελέσματα:



Σχήμα 2: shared_clusters_simple + GOMP_CPU_AFFINITY Results

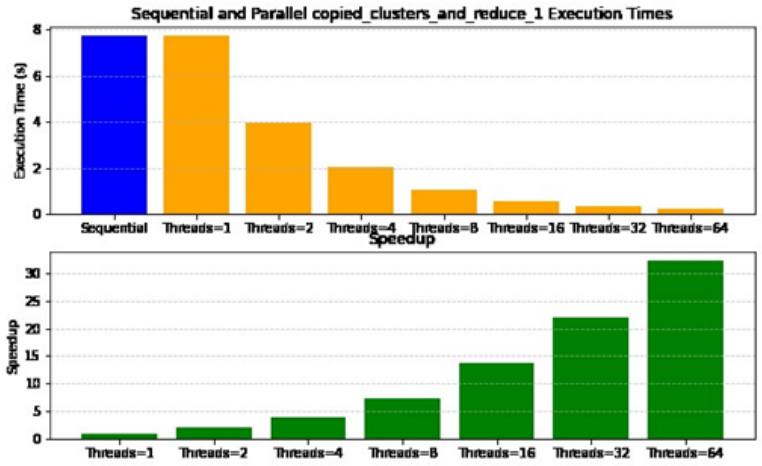
Copied clusters and reduce

1. Μπορούμε να δούμε ότι η τακτική της χρήσης τοπικών πινάκων για το κάθε νήμα και η μετέπειτα αναγωγή τους σε κοινό πίνακα (copy and reduce) επιφέρει ικανοποιητική βελτίωση της απόδοσης. Επισημαίνουμε τις εξής παρατηρήσεις:

α′) Ο αλγόριθμος φαίνεται να κάνει καλύτερο scale-up για 1 έως 8 threads (διπλασιασμός των threads προκαλεί και διπλασιασμό του speedup), ενώ καθώς αυξάνουμε τα threads πάνω από το 8 δεν έχουμε τόσο καλή κλιμάκωση. Αυτό μπορεί να εξηγηθεί από το γεγονός ότι το μηχάνημα Sandman έχει 4 nodes των 8 cores και συνεπώς τα 8 νήματα τρέχουν στο ίδιο node και έχουμε τοπικότητα στις cache όταν πάμε να κάνουμε το reduction.

β′) Δεν παρατηρούμε βελτίωση στην ταχύτητα εκτέλεσης όταν αυξάνουμε τα threads από 32 σε 64. Αυτό οφείλεται στο γεγονός ότι το μηχάνημα Sandman χρησιμοποιεί hyper-threading, δηλαδή έχει 32 πραγματικούς πυρήνες οι οποίοι γίνονται 64 «εικονικοί». Αυτό είναι χρήσιμο για cloud πλατφόρμες, όπου τρέχουν δύο διαφορετικές εφαρμογές που πιθανότατα έχουν διαφορετικές απαιτήσεις σε πόρους, αλλά στην περίπτωσή μας τα δύο νήματα μπαίνουν ουσιαστικά σειριακά στον κοινό επεξεργαστή που μοιράζονται.

Έτσι με τη χρήση τοπικών πινάκων και της αναγωγής έχουμε τα παρακάτω αποτελέσματα:



Σχήμα 3: copied_clusters_reduce + GOMP_CPU_AFFINITY Results

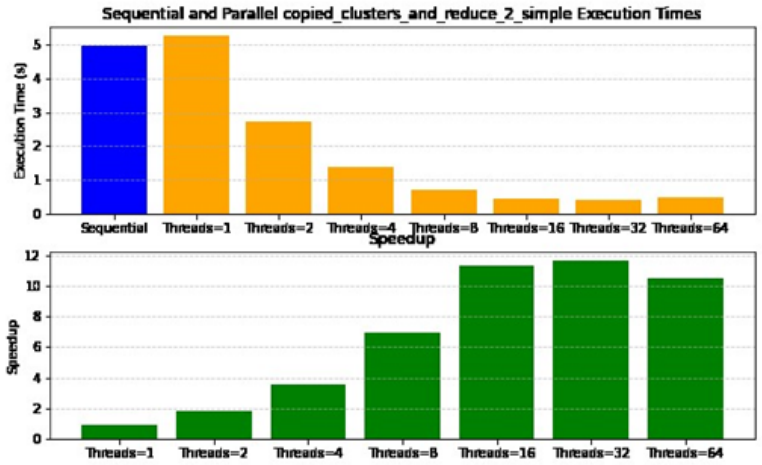
2. Με το configuration Size, Coords, Clusters, Loops = 256, 1, 4, 10 αναμέναμε να παρατηρήσουμε μείωση στο scalability του αλγορίθμου. Αυτό θα οφειλόταν στο μικρότερο αριθμό Clusters*Coords, που οδηγεί τα νήματα να επενεργούν επάνω σε δεδομένα που βρίσκονται στην ίδια cache line με αυτά άλλων νημάτων. Έτσι, όταν ένα νήμα τροποποιεί μία μεταβλητή, ολόκληρη η γραμμή της κρυφής μνήμης στην οποία βρίσκεται αυτή η μεταβλητή ενημερώνεται, παρά το γεγονός ότι τα υπόλοιπα νήματα μπορεί να μην χρειάζεται να ενημερώσουν τις δικές τους μεταβλητές. Λόγω cache coherency πρωτοκόλλων οι cache lines των άλλων πυρήνων που μοιράζονται το συγκεκριμένο cache line καθίστανται invalid, ακόμα και αν δεν χρησιμοποιούν τις ίδιες θέσεις μνήμης της cache line. Ως αποτέλεσμα, όταν κάνουν χρήση της cache line, παρότι δεν χρησιμοποιούν τις ίδιες θέσεις μνήμης, θα πρέπει να ζητήσουν την ανανεωμένη cache line από τον πυρήνα που την άλλαξε, δημιουργώντας μεγάλη κίνηση στο bus. Αυτό το φαινόμενο False-Sharing προκαλεί ικανή καθυστέρηση στην εκτέλεση του προγράμματος. Το πρόβλημα αυτό μπορεί να αντιμετωπιστεί με δύο τρόπους:

α′) First-Touch: Το first touch επιτρέπει στους πυρήνες να κρατήσουν στην cache τους τα cache lines τα οποία κάνουν αυτοί access πρώτοι. Έτσι ο memory manager, όταν γίνεται access ένα variable διαμοιράζει memory space, έτσι ώστε να μειωθεί η πιθανότητα το cache line που περιέχει το variable να μοιράζεται σε δύο πυρήνες. Πλέον, αφού δεν υπάρχει διαμοιραζόμενο cache line, το cache coherency protocol δεν ενεργοποιείται και η κίνηση στο bus είναι πολύ μικρότερη.

β′) Padding: Το padding προσθέτει επιπλέον «άχρηστο» χώρο μεταξύ μεταβλητών ή data structures με σκοπό να αποφύγει αυτά να βρίσκονται στο ίδιο cache line. Με αυτόν τον τρόπο αποφεύγουμε να έχουμε πολλές μεταβλητές στο ίδιο cache line, κάτι που θα μπορούσε να οδηγήσει πολλούς πυρήνες να μοιράζονται το ίδιο cache line και να έχουμε και πάλι cache coherency slowdowns. Με το να μην μοιράζονται τα cache lines μεταξύ πυρήνων, το cache coherency protocol δεν ενεργοποιείται και η κίνηση στο bus είναι πάλι πολύ μικρότερη.

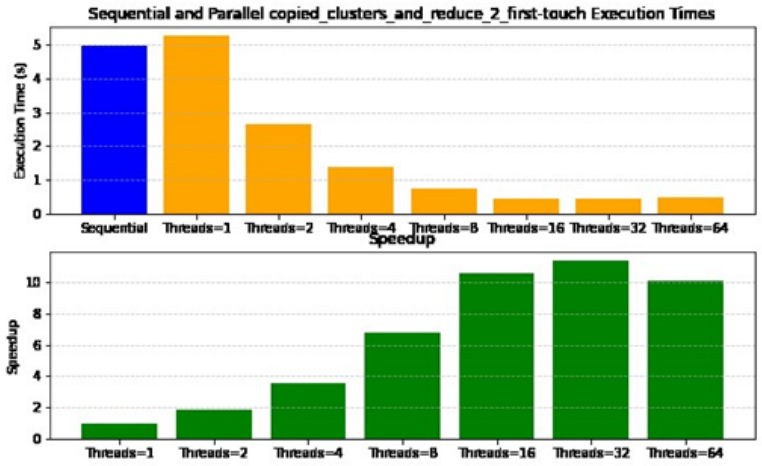
Ωστόσο, όταν εφαρμόσαμε τις παραπάνω τεχνικές δεν παρατηρήσαμε βελτίωση στην απόδοση του προγράμματος. Αυτό οφείλεται στο γεγονός ότι την ανάθεση μνήμης για τους local πίνακες του κάθε νήματος την υλοποιήσαμε με calloc, η οποία στα linux έχει υλοποιηθεί ώστε για μικρά μεγέθη μνήμης να κάνει όντως allocate 0 (αυτό, βέβαια, είναι implementation specific και θα μπορούσε να υλοποιείται αλλιώς). Ως αποτέλεσμα, κάθε thread που καλεί την calloc κάνει όντως touch επάνω στη μνήμη, οπότε αντιμετωπίζεται το πρόβλημα του False-Sharing.

Έτσι τα αποτελέσματα μας χρησιμοποιώντας το first touch της calloc ήταν:



Σχήμα 4: copied_clusters_reduce + GOMP_CPU_AFFINITY / Calloc First Touch Results

Ενώ με manual first touch είχαμε παρόμοια (λίγο καλύτερα) αποτελέσματα:



Σχήμα 5: copied_clusters_reduce + GOMP_CPU_AFFINITY / Manual First Touch Results

Ο κώδικας που χρησιμοποιήθηκε για το copied clusters - reduce είναι ο εξής:

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

// 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]
```

```
63      int nthreads;           // no. threads
64
65      nthreads = omp_get_max_threads();
66      printf("OpenMP Kmeans - Reduction\t(number of threads: %d)\n",
        nthreads);
67
68      // initialize membership
69      for (i=0; i<numObjs; i++)
70          membership[i] = -1;
71
72      // initialize newClusterSize and newClusters to all 0
73      newClusterSize = (typeof(newClusterSize)) calloc(numClusters,
        sizeof(*newClusterSize));
74      newClusters = (typeof(newClusters))  calloc(numClusters * numCoords
        , sizeof(*newClusters));
75
76      // Each thread calculates new centers using a private space. After
        that, thread 0 does an array reduction on them.
77      int * local_newClusterSize[nthreads];  // [nthreads][numClusters]
78      double * local_newClusters[nthreads];   // [nthreads][numClusters][
        numCoords]
79
80      /*
81       * Hint for false-sharing
82       * This is noticed when numCoords is low (and neighboring
        local_newClusters exist close to each other).
83       * Allocate local cluster data with a "first-touch" policy.
84       */
85      // Initialize local (per-thread) arrays (and later collect result
        on global arrays)
86
87
88      #pragma omp parallel for private(k)
89      for (k=0; k<nthreads; k++)
90      {
91          local_newClusterSize[k] = (typeof(*local_newClusterSize))
        calloc(numClusters, sizeof(**local_newClusterSize));
92          local_newClusters[k] = (typeof(*local_newClusters)) calloc(
        numClusters * numCoords, sizeof(**local_newClusters));
93
94          // first-touch
95          for (i=0; i<numClusters; i++) {
96                  for (j=0; j<numCoords; j++)
97                      local_newClusters[k][i*numCoords + j] = 0.0;
98                  local_newClusterSize[k][i] = 0;
99              }
100     }
101
102     int flag = 0; // simply a flag to not initialize to 0 on the first
```

```
    loop
103     timing = wtime();
104     do {
105         // before each loop, set cluster data to 0
106         for (i=0; i<numClusters; i++) {
107             #pragma omp parallel for private(j)
108             for (j=0; j<numCoords; j++)
109                 newClusters[i*numCoords + j] = 0.0;
110             newClusterSize[i] = 0;
111         }
112 
113         delta = 0.0;
114 
115         #pragma omp parallel private(index,i,j,k) reduction(+:delta)
116         {
117         /*
118          * TODO: Initiliaze local cluster data to zero (separate for
    each thread)
119          */
120 
121            #pragma omp for
122            for (k=0; k<nthreads; k++) {
123                #pragma omp parallel for private(i)
124                for (i=0; i<numClusters; i++) {
125                    #pragma omp parallel for private(j)
126                    for (j=0; j<numCoords; j++)
127                        local_newClusters[k][i*numCoords + j] = 0.0;
128                    local_newClusterSize[k][i] = 0;
129                }
130            }
131 
132         #pragma omp for
133         for (i=0; i<numObjs; i++)
134         {
135             // find the array index of nearest cluster center
136             index = find_nearest_cluster(numClusters, numCoords, &
    objects[i*numCoords], clusters);
137 
138             // if membership changes, increase delta by 1
139             if (membership[i] != index)
140                 delta += 1.0;
141 
142             // assign the membership to object i
143             membership[i] = index;
144 
145             // update new cluster centers : sum of all objects located
    within (average will be performed later)
146             /*
147              * TODO: Collect cluster data in local arrays (local to
```

```
each thread)
148              *       Replace global arrays with local per-thread
149              */
150             int thread_id = omp_get_thread_num();
151             local_newClusterSize[thread_id][index]++;
152             for (j=0; j<numCoords; j++)
153                 local_newClusters[thread_id][index*numCoords + j] +=
    objects[i*numCoords + j];
154         }
155
156         /*
157          * TODO: Reduction of cluster data from local arrays to shared.
158          *       This operation will be performed by one thread
159          */
160         }
161
162         for (k=0; k<nthreads; k++){
163             for(i=0; i<numClusters; i++)
164                 if(local_newClusterSize[k][i] > 0){
165                     newClusterSize[i] += local_newClusterSize[k][i];
166                     for(j=0; j<numCoords; j++)
167                         newClusters[i*numCoords + j] +=
    local_newClusters[k][i*numCoords + j];
168                 }
169         }
170
171         // average the sum and replace old cluster centers with
    newClusters
172
173         for (i=0; i<numClusters; i++) {
174             if (newClusterSize[i] > 0) {
175                 //#pragma omp parallel for
176                 for (j=0; j<numCoords; j++) {
177                     clusters[i*numCoords + j] = newClusters[i*numCoords
     + j] / newClusterSize[i];
178
179                 }
180            }
181         }
182
183         // Get fraction of objects whose membership changed during this
     loop. This is used as a convergence criterion.
184         delta /= numObjs;
185
186         loop++;
187         printf("\r\tcompleted loop %d", loop);
188         fflush(stdout);
189     } while (delta > threshold && loop < loop_threshold);
190     timing = wtime() - timing;
```

```
191     printf("\n        nloops = %3d   (total = %7.4fs)  (per loop = %7.4
        fs)\n", loop, timing, timing/loop);
192
193     for (k=0; k<nthreads; k++)
194     {
195         free(local_newClusterSize[k]);
196         free(local_newClusters[k]);
197     }
198     //free(local_newClusters);
199     //free(local_newClusterSize);
200     free(newClusters);
201     free(newClusterSize);
202 }
```

Αμοιβαίος Αποκλεισμός - Κλειδώματα

Στην άσκηση αυτή μελετάμε διάφορα κλειδώματα στον αλγόριθμο K-means. Συγκεκριμένα μελετάμε τις εξής 9 περιπτώσεις:

1. Naive: χρησιμοποιείται το OpenMP directive #pragma omp atomic για την προστασία του κρίσιμου τμήματος.
2. Naive-critical: χρησιμοποιείται το OpenMP directive #pragma omp critical για την προστασία του κρίσιμου τμήματος.
3. nosync_lock: Η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα. Ωστόσο, θα χρησιμοποιηθεί ως άνω όριο για την αξιολόγηση της επίδοσης των υπόλοιπων κλειδωμάτων.
4. pthread_mutex_lock: Το pthread_mutex_t κλείδωμα που παρέχει η βιβλιοθήκη Pthreads.
5. pthread_spin_lock: Το pthread_spinlock_t κλείδωμα που παρέχει η βιβλιοθήκη Pthreads.
6. tas_lock: Το test-and-set κλείδωμα χρησιμοποιεί atomic operations με σκοπό να ελέγξει την τιμή μιας μεταβλητής (κλειδώματος) και να την αλλάξει, όλα σε μια uninterruptible λειτουργία.
7. ttas_lock: Το test-and-test-and-set κλείδωμα όμοια με το tas_lock χρησιμοποιεί atomic operations για τον ίδιο σκοπό, όμως επιπλέον αρχικά ελέγχει αν το lock είναι διαθέσιμο και άρα δεν προσπαθεί συνέχεια να πάρει και να αλλάξει την τιμή του. Αυτό θα το προσπαθήσει μόνο εάν το lock ελευθερωθεί πρώτα. Όσο αυτό δεν συμβαίνει είμαστε σε busy-wait κατάσταση.

8. array_lock: Το array-based κλείδωμα αποτελεί ουσιαστικά έναν κυκλικό πίνακα στον οποίο τα threads μας καταλαμβάνουν κυκλικά slots και παίρνουν διαδοχικά το κλείδωμα, αντί να ανταγωνίζονται για το global κλείδωμα.

9. clh_lock: Ένα είδος κλειδώματος που στηρίζεται στη χρήση μίας συνδεδεμένης λίστας.

Εκτελώντας make στο directory μας παίρνουμε τα απαραίτητα εκτελέσιμα για την εκτέλεση του K-means με τα διάφορα κλειδώματα. Στη συνέχεια μέσω ενός bash script τρέχουμε τα εκτελέσιμα παίρνοντας τα αποτελέσματα για τιμές: {Size, Coords, Clusters, Loops} = {32, 16, 16, 10} για threads = {1, 2, 4, 8, 16, 32, 64}. Τα αποτελέσματα φαίνονται στο παρακάτω διάγραμμα:

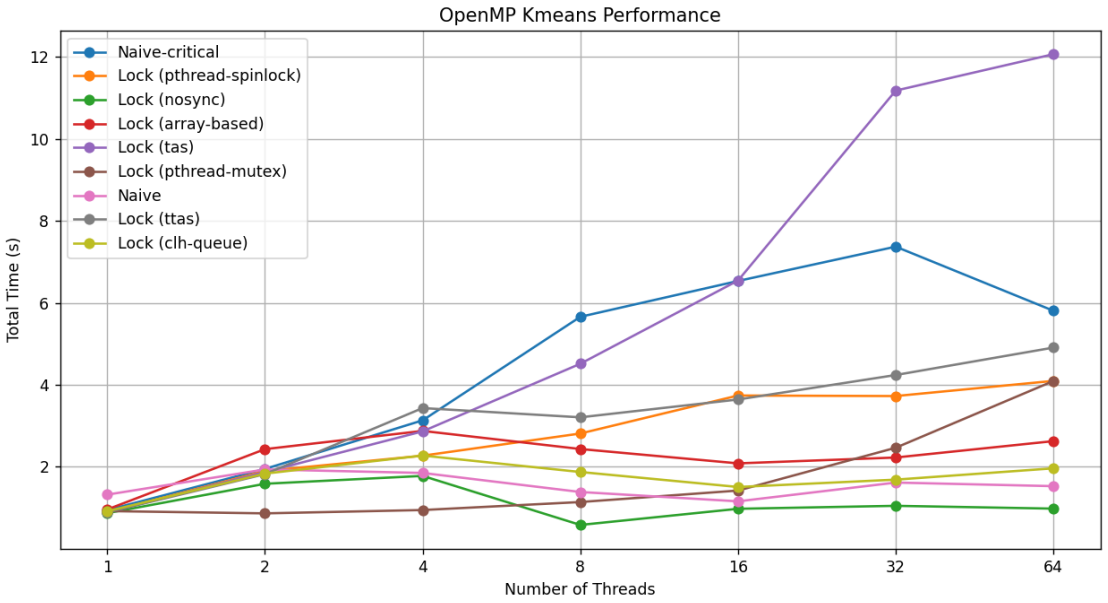


Figure 6: K-means & Locks | {Size, Coords, Clusters, Loops} = {32, 16, 16, 10}

Παρακάτω σχολιάζουμε τα performance values που παίρνουμε σε κάθε περίπτωση:

1. Naive: Όπως αναφέραμε χρησιμοποιούμε το #pragma omp atomic για την προστασία του κρίσιμου τμήματος. Βλέπουμε πως οι χρόνοι που πετυχαίνει αυτό το κλείδωμα είναι πάρα πολύ καλοί και σχετικά σταθεροί όσον αφορά τον αριθμό των threads. Αυτό επιτυγχάνεται στη συγκεκριμένη υλοποίηση με χρήση ατομικών εντολών που υποστηρίζονται απευθείας από το hardware. Με αυτό τον τρόπο επιτυγχάνονται οι γρήγοροι χρόνοι.

2. Naive-critical: Όπως αναφέραμε χρησιμοποιούμε το #pragma omp critical για την προστασία του κρίσιμου τμήματος. Αντίθετα με το #pragma omp atomic, το #pragma omp critical δεν χρησιμοποιεί ατομικές εντολές, αλλά θέτει το critical section προσβάσιμο μόνο από ένα thread τη φορά. Αυτό όπως μπορούμε να δούμε και από το διάγραμμα οδηγεί σε αρκετά μεγάλους χρόνους, όσο περισσότερα threads έχουμε, λόγω των περισσότερων collisions. Έτσι το #pragma omp critical μας οδηγεί σε ένα από τα χειρότερα performances από τις 9 επιλογές, ιδιαίτερα για πολλά threads.

3. nosync_lock: Όπως αναφέραμε η συγκεκριμένη υλοποίηση δεν παρέχει αμοιβαίο αποκλεισμό οπότε δεν παράγει και σωστά αποτελέσματα. Έτσι είναι μια από τις πιο γρήγορες υλοποιήσεις, ειδικά για πολλά threads, αφού δεν εξετάζονται collisions.

4. pthread_mutex_lock: Το pthread_mutex_t κλείδωμα που παρέχει η βιβλιοθήκη Pthreads. Όπως βλέπουμε για μικρούς αριθμούς threads πετυχαίνει πολύ καλούς χρόνους. Αυτό συμβαίνει επειδή η υλοποίηση αυτή χρησιμοποιεί atomic operations υλοποιημένες σε hardware, που επιτρέπουν στα threads να μπουν σε sleep state, μέχρι να είναι διαθέσιμο το lock. Παρόλαυτα, όταν έχουμε πολλά threads ο ανταγωνισμός για το κλείδωμα είναι μεγάλος με αποτέλεσμα για αυτές τις τιμές threads να έχουμε χειρότερους χρόνους.

5. pthread_spin_lock: Το pthread_spinlock_t κλείδωμα που παρέχει η βιβλιοθήκη Pthreads. Όπως βλέπουμε η υλοποίηση αυτή έχει παρόμοια συμπεριφορά με το pthread_mutex_t, με πιο ομαλό (και άρα πιο γρήγορο) scale-up. Αυτό συμβαίνει επειδή τα spinlocks εκτελούν busy-wait στα cores, αντί να μπαίνουν με ατομικές εντολές (που είναι και πολύ πιο γρήγορες) σε sleep state. Έτσι η έλλειψη ατομικών εντολών σε συνδυασμό με το busy-wait μας δίνουν χειρότερους χρόνους σε σχέση με το mutex. Παρατηρώντας καλύτερα για 64 threads (χρήση hyper-threading στους επεξεργαστές: 2 virtual cores για κάθε physical core), βλέπουμε πως συγκλίνουν στην ίδια τιμή. Αυτό μας οδηγεί στο συμπέρασμα πως από ένα σημείο και μετά παρότι τα spinlocks δεν χρησιμοποιούν atomic operations, τα collisions προκαλούν το μεγαλύτερο μέρος της καθυστέρησης, λόγω του ανταγωνισμού για το lock.

6. tas_lock: Το test-and-set κλείδωμα βλέπουμε πως έχει ίσως τους χειρότερους χρόνους από τις 9 υλοποιήσεις, ειδικά για μεγάλο αριθμό threads. Αυτό συμβαίνει επειδή κάθε thread είναι σε busy-wait loop εκτελώντας atomic operation για να διαβάσει την τιμή του lock και να προσπαθήσει να το πάρει αν αυτό είναι διαθέσιμο. Αυτό προφανώς προκαλεί μεγάλο overhead και πολλά collisions, ειδικά για μεγάλο αριθμό threads.

7. ttas_lock: Το test-and-test-and-set κλείδωμα όπως βλέπουμε παρουσιάζει αρ-

κετά καλύτερους χρόνους από το tas_lock. Αυτό οφείλεται στο γεγονός πως, πριν προσπαθήσει το κάθε thread με atomic operation να ελέγξει και να αλλάξει(πάρει) το lock, ελέγχει αρχικά αν το lock είναι διαθέσιμο. Αν είναι, μόνο τότε εκτελεί την ατομική εντολή για να διεκδικήσει το lock. Έτσι έχουμε λιγότερο overhead και άρα καλύτερους χρόνους.

8. array_lock: Το array-based κλείδωμα όπως βλέπουμε δίνει πολύ καλούς χρόνους. Αυτό επιτυγχάνεται μέσω της μορφής του κλειδώματος αυτού. Τα threads καταλαμβάνουν κυκλικά slots παίρνοντας διαδοχικά το global lock. Έτσι ο ανταγωνισμός γίνεται για τα κυκλικά slots και όχι για το global lock. Άρα έχουμε λιγότερα collisions, αλλά λίγο μεγαλύτερο overhead για την δημιουργία και ανανέωση του lock. Έτσι δεν παίρνουμε τους καλύτερους χρόνους, αλλά πολύ ικανοποιητικούς, οι οποίοι μάλιστα δεν αυξάνονται πολύ για μεγάλο αριθμό threads, για τον λόγο που αναφέραμε.

9. clh_lock: Το clh_lock όπως βλέπουμε δίνει πολύ καλούς χρόνους. Τα threads που επιθυμούν να πάρουν το lock μπαίνουν ως nodes στην λίστα και εκτελούν loop σε μια τοπική μεταβλητή περιμένοντας ο πρωηγούμενος κόμβος να του δηλώσει πως ελευθερώθηκε το lock. Και σε αυτή την περίπτωση αποφεύγουμε το contention για το global lock, με τα collisions να γίνονται για την είσοδο στην ουρά. Έτσι έχουμε λιγότερο contention και πετυχαίνουμε γρήγορους χρόνους, ακόμα και για μεγάλο αριθμό threads. Ο λόγος που πετυχαίνουμε καλύτερο αποτέλεσμα από το array-based lock είναι ότι έχουμε μικρότερο overhead να διατηρήσουμε την λίστα updated απ' ότι έναν κυκλικό πίνακα.

Καταλήγοντας τα καλύτερα αποτελέσματα (εξαιρώντας το nosync_lock) δίνονται από τις υλοποιήσεις Naive(#pragma omp atomic), clh-queue_lock και array-based_lock. Η #pragma omp atomic υλοποίηση εκμεταλεύεται με χρήση γρήγορων ατομικών εντολών υλοποιημένες σε επίπεδο hardware. Αντίστοιχα για τα δύο locks αποφεύγουμε τις πολλές συγκρούσεις στο global lock έχοντας ένα lock structure πάνω στο οποίο ανταγωνίζονται τα threads.

Αντίθετα, τα χειρότερα αποτελέσματα είναι για τις υλοποιήσεις Naive-critical(#pragma omp critical) και tas_lock. Το #pragma omp critical έχει κακό performance, επειδή αρχικά δεν κάνει χρήση atomic operations και επειδή απλώς επιτρέπει στο critical section μόνο ένα thread τη φορά. Η λογική αυτή προκαλεί πολλά collisions δίνοντας αργούς χρόνους. Το tas_lock από την άλλη, χρησιμοποιεί atomic operations, όμως κάνει busy-wait σε αυτές. Δηλαδή τα threads προσπαθούν συνεχώς να διεκδικήσουν το lock (ακόμα και αν αυτό δεν είναι διαθέσιμο) και να εκτελέσουν το atomic operation. Έτσι οδηγούμαστε σε πολλές συγκρούσεις με μεγάλο overhead.

Τέλος παρουσιάζουμε ένα διάγραμμα που δείχνει την απόδοση των υλοποιήσεων σε σχέση με την nosync_lock (λάθος) υλοποίηση για διάφορες τιμές threads:

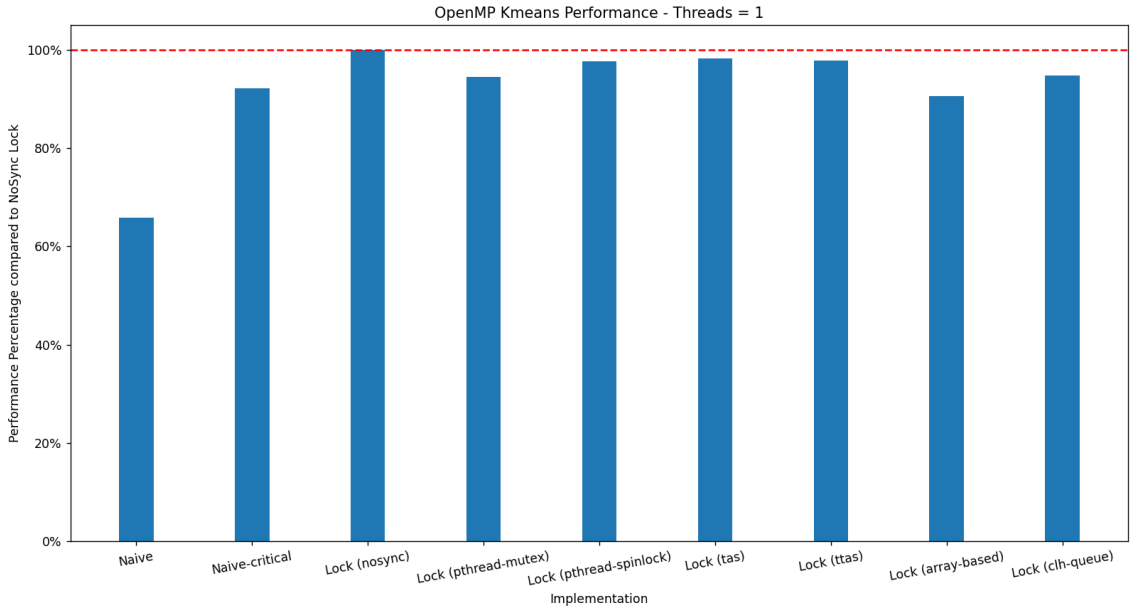


Figure 7: Implementations Performance in relation to NoSync Lock | Threads = 1

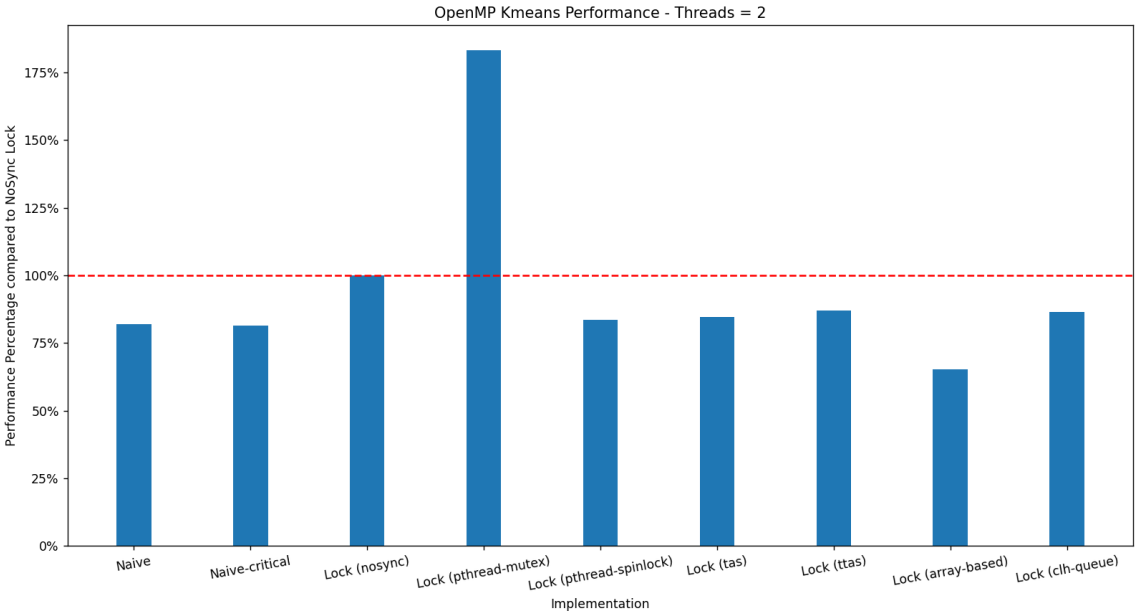


Figure 8: Implementations Performance in relation to NoSync Lock | Threads = 2

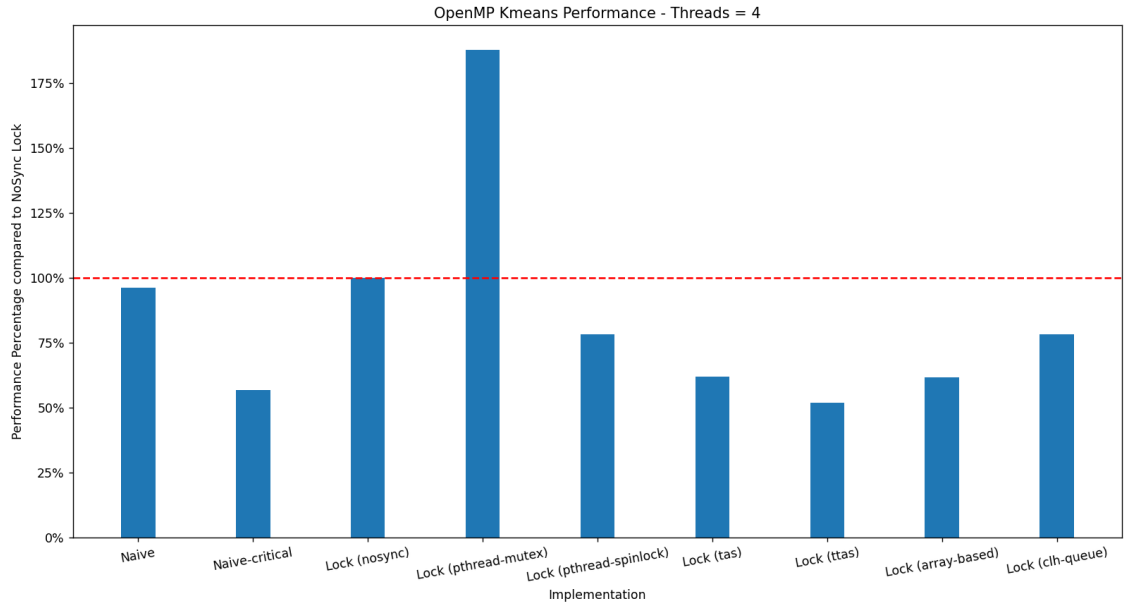


Figure 9: Implementations Performance in relation to NoSync Lock | Threads = 4

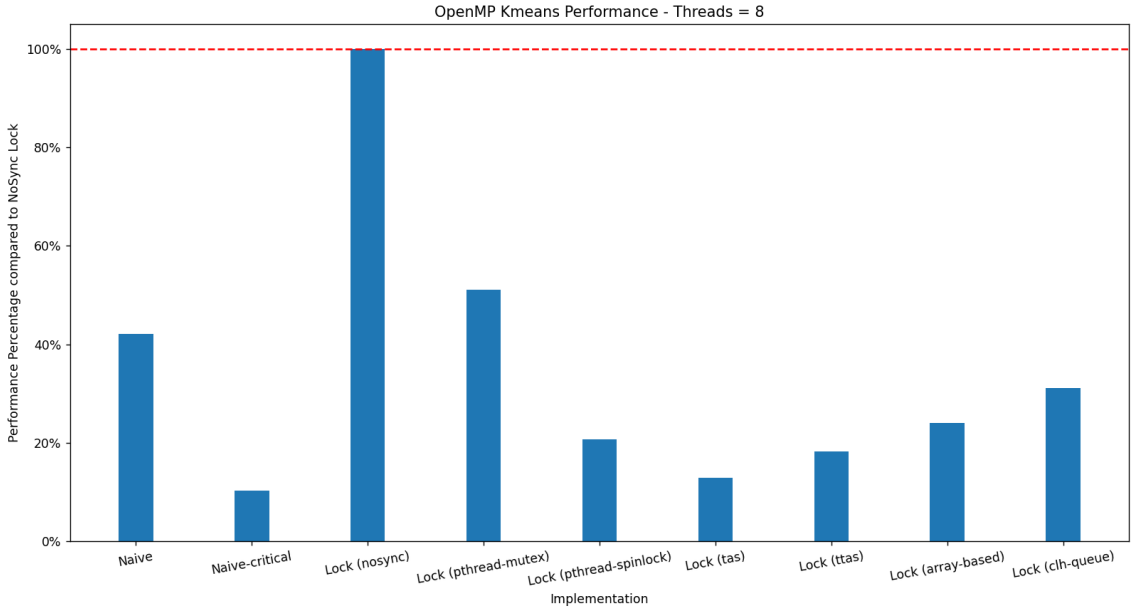


Figure 10: Implementations Performance in relation to NoSync Lock | Threads = 8

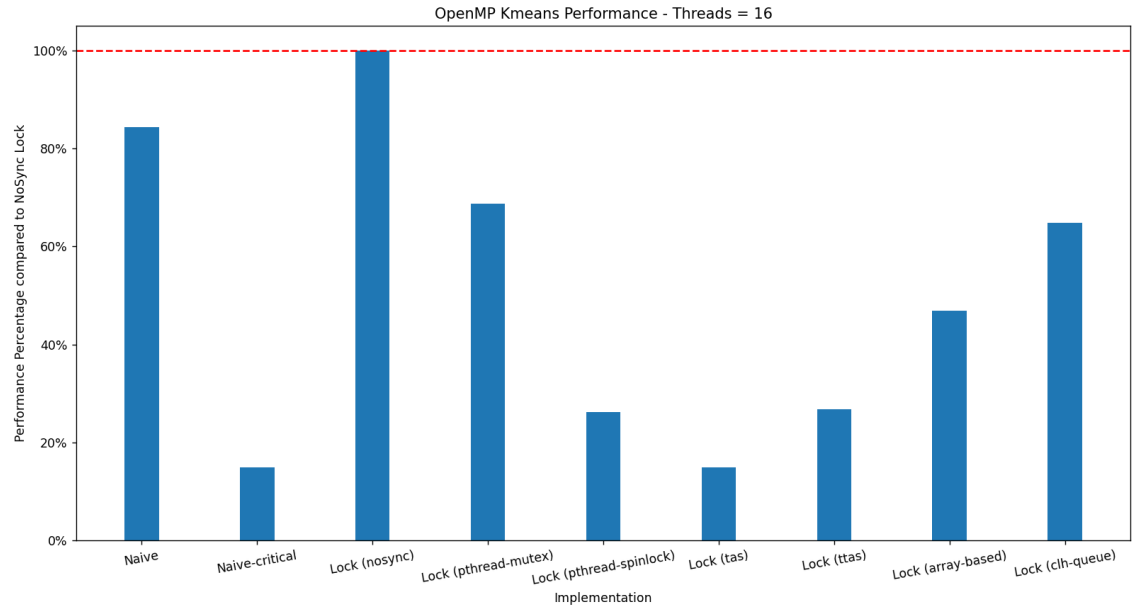


Figure 11: Implementations Performance in relation to NoSync Lock | Threads = 16

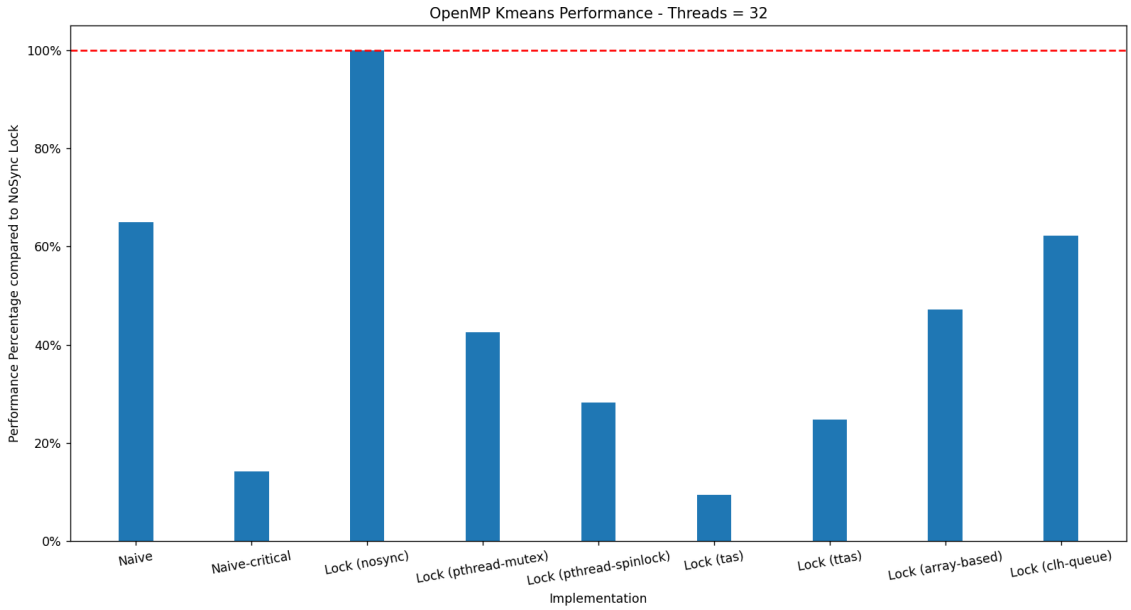


Figure 12: Implementations Performance in relation to NoSync Lock | Threads = 32

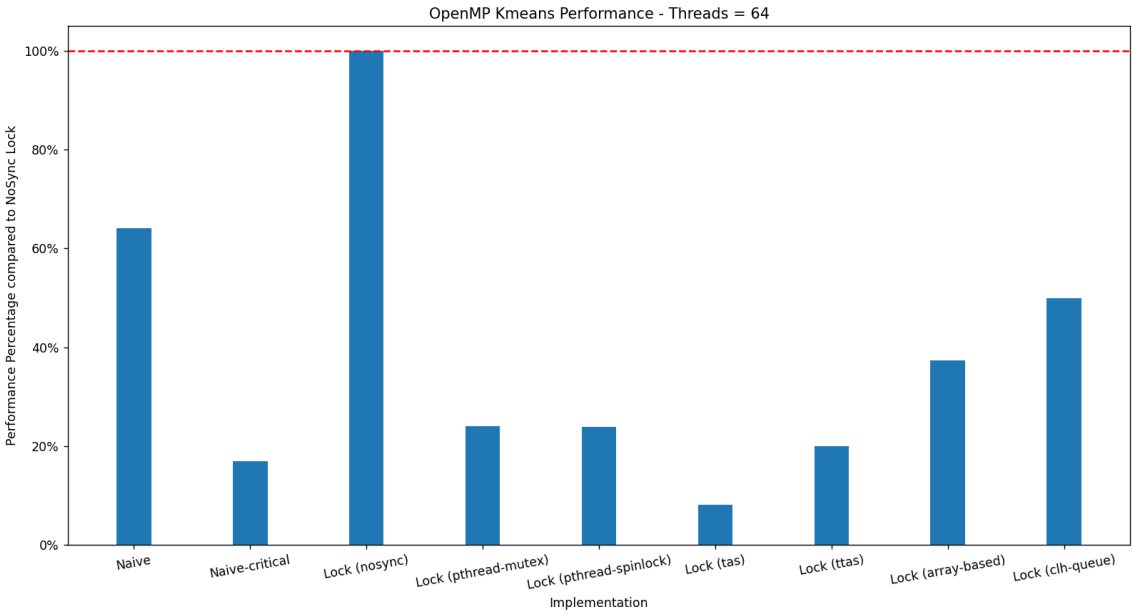


Figure 13: Implementations Performance in relation to NoSync Lock | Threads = 64

Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Recursive Floyd Warshall

Αρχικά παραλληλοποιούμε τη recursive έκδοση του αλγορίθμου. Μπορούμε να παρατηρήσουμε πως η βασική ιδέα στον αλγόριθμο είναι να φτάσουμε σε ένα block size ίσο με το block size που περνάμε ως argument και τότε σε αυτό το μικρό block τρέχουμε τον αλγόριθμο για τη συγκεκριμένη περιοχή. Για να φτάσουμε σε αυτές τις μικρές περιοχές κάνουμε αναδρομικές κλήσεις της συνάρτησης μας, χωρίζοντας το αρχικό μας ταμπλό σε 4 περιοχές, το οποίο απαιτεί την αναδρομική κλήση 8 συναρτήσεων. Έτσι για να πετύχουμε επιτάχυνση μπορούμε να ορίσουμε κλήσεις ως ένα διαφορετικό task, προσέχοντας όμως το γεγονός πως όλοι γράφουν στον πίνακα A. Η επιτάχυνση που πετυχαίνουμε οφείλεται σε μεγάλο βαθμό στο γεγονός πως δουλεύοντας σε μία μικρή περιοχή σε κάθε task επιτυγχάνουμε cache locality. Έτσι, επαναχρησιμοποιούμε τα ίδια δεδομένα κατά την αναδρομή, δηλαδή μειώνουμε το reuse distance, χωρίς να φέρνουμε συνέχεια νέα cache lines.

Τα αποτελέσματα φαίνονται στο κοινό διάγραμμα για τους αλγορίθμους Floyd Warshall στο τέλος του αρχείου.

Ο κώδικας που χρησιμοποιήθηκε για τον Recursive Floyd Warshall είναι ο εξής:

```
/*
 * 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;
```

```
26          double time;
27          int B=16;
28          int N=1024;
29
30          if (argc !=3){
31                  fprintf(stdout, "Usage %s N B \n", argv[0]);
32                  exit(0);
33          }
34
35          N=atoi(argv[1]);
36          B=atoi(argv[2]);
37
38          A = (int **) malloc(N*sizeof(int *));
39          for(i=0; i<N; i++) A[i] = (int *) malloc(N*sizeof(int));
40
41          graph_init_random(A,-1,N,128*N);
42
43          gettimeofday(&t1,0);
44          FW_SR(A,0,0, A,0,0,A,0,0,N,B);
45          gettimeofday(&t2,0);
46
47          time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
    tv_usec)/1000000;
48          printf("FW_SR,%d,%d,%.4f\n", N, B, time);
49
50          /*
51             for(i=0; i<N; i++)
52             for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
53           */
54
55          return 0;
56  }
57
58  inline int min(int a, int b)
59  {
60          if(a<=b)return a;
61          else return b;
62  }
63
64  void FW_SR (int **A, int arow, int acol,
65              int **B, int brow, int bcol,
66              int **C, int crow, int ccol,
67              int myN, int bsize)
68  {
69          int k,i,j;
70
71          /*
72           * The base case (when recursion stops) is not allowed to be
    edited!
```

```
         * What you can do is try different block sizes.
         */
        if(myN<=bsize)
                for(k=0; k<myN; k++)
                        for(i=0; i<myN; i++)
                                for(j=0; j<myN; j++)
                                        A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
        else {
                FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
                #pragma omp parallel
                {
                        #pragma omp single
                        {
                                #pragma omp task shared(A, B, C)
                                FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
                                FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
                                #pragma omp taskwait
                        }
                }

                FW_SR(A,arow+myN/2, acol+myN/2,B,brow+myN/2, bcol,C, crow, ccol+myN/2, myN/2, bsize);
                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);

                #pragma omp parallel
                {
                        #pragma omp single
                        {
                                #pragma omp task shared(A,B,C)
                                FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
                                FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
                        }
                }

                FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);

        }
}
```

Tiled Floyd Warshall

Στη συνέχεια παραλληλοποιούμε τον αλγόριθμο Tiled Floyd – Warshall. Ο αλγόριθμος αυτός χωρίζει το ταμπλό μας σε μικρότερα με βάση το block size που ορίζουμε και σε κάθε ένα από αυτά εκτελεί τον standard Floyd – Warshall. Με τον διαχωρισμό αυτό μπορούμε να πετύχουμε cache locality στους πυρήνες μας, αλλά και parallelization, αφού εργαζόμαστε κάθε φορά σε διαφορετικές περιοχές.

Ο κώδικας που χρησιμοποιήθηκε για τον Tiled Floyd Warshall είναι ο εξής:

```
/*
 * Tiled version of the Floyd-Warshall algorithm.
 * command-line arguments: N, B
 * N = size of graph
 * B = size of tile
 * works only when N is a multiple of B
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "util.h"
#include <omp.h>

inline int min(int a, int b);
inline void FW(int **A, int K, int I, int J, int N);

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]);

        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);

        gettimeofday(&t1,0);
```

```
40
41          for(k=0;k<N;k+=B){
42                  FW(A,k,k,k,B);
43                  #pragma omp parallel shared(A, B, k)
44                  {
45
46                          #pragma omp for nowait
47                          for(i=0; i<k; i+=B)
48                                  FW(A,k,i,k,B);
49
50                          #pragma omp for nowait
51                          for(i=k+B; i<N; i+=B)
52                                  FW(A,k,i,k,B);
53
54                          #pragma omp for nowait
55                          for(j=0; j<k; j+=B)
56                                  FW(A,k,k,j,B);
57
58                          #pragma omp for nowait
59                          for(j=k+B; j<N; j+=B)
60                                  FW(A,k,k,j,B);
61
62                  }
63
64                  #pragma omp parallel shared(A, B, k)
65                  {
66
67                          #pragma omp for collapse(2) nowait
68                          for(i=0; i<k; i+=B)
69                                  for(j=0; j<k; j+=B)
70                                          FW(A,k,i,j,B);
71
72                          #pragma omp for collapse(2) nowait
73                          for(i=0; i<k; i+=B)
74                                  for(j=k+B; j<N; j+=B)
75                                          FW(A,k,i,j,B);
76
77                          #pragma omp for collapse(2) nowait
78                          for(i=k+B; i<N; i+=B)
79                                  for(j=0; j<k; j+=B)
80                                          FW(A,k,i,j,B);
81
82                          #pragma omp for collapse(2) nowait
83                          for(i=k+B; i<N; i+=B)
84                                  for(j=k+B; j<N; j+=B)
85                                          FW(A,k,i,j,B);
86
87                  }
88          }
```

```
89          gettimeofday(&t2,0);
90
91          time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.
    tv_usec)/1000000;
92          printf("FW_TILED,%d,%d,%.4f\n", N,B,time);
93
94          /*
95             for(i=0; i<N; i++)
96             for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
97           */
98
99          return 0;
100 }
101
102 inline int min(int a, int b)
103 {
104         if(a<=b)return a;
105         else return b;
106 }
107
108 inline void FW(int **A, int K, int I, int J, int N)
109 {
110         int i,j,k;
111
112         for(k=K; k<K+N; k++)
113                 for(i=I; i<I+N; i++)
114                         for(j=J; j<J+N; j++)
115                                 A[i][j]=min(A[i][j], A[i][k]+A[k][j]);
116
117 }
```

Αποτελέσματα

Παρακάτω έχουμε τα αποτελέσματα για όλους τους αλγορίθμους Floyd Warshall:

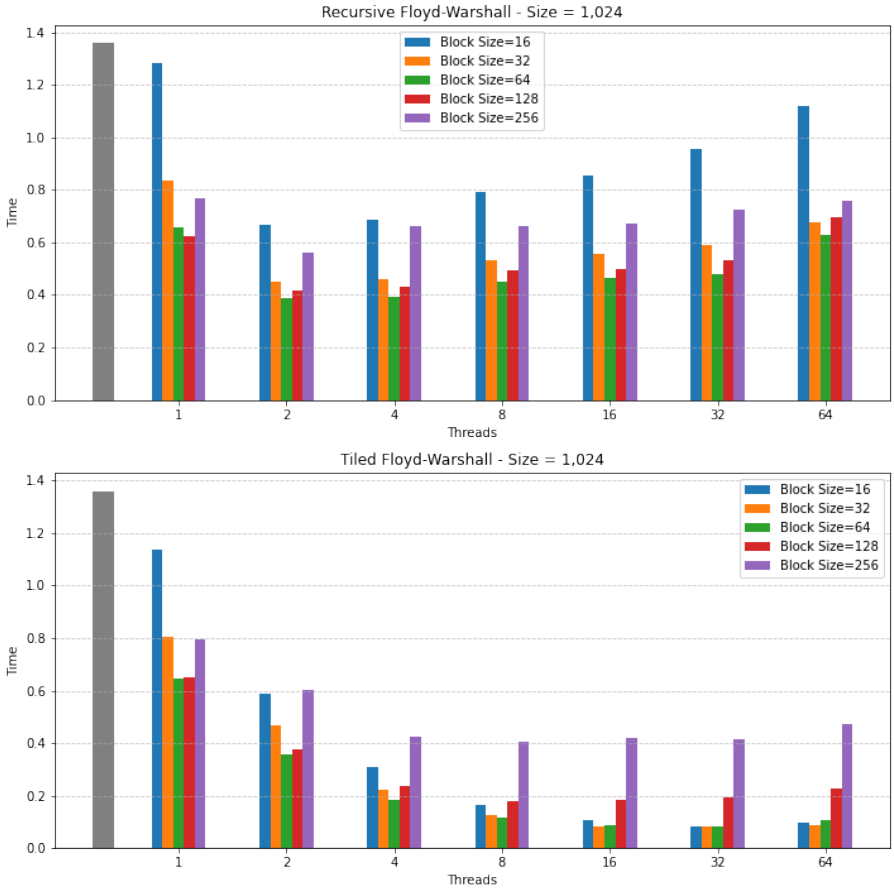


Figure 14: Floyd Warshall - Size: 1024 Results

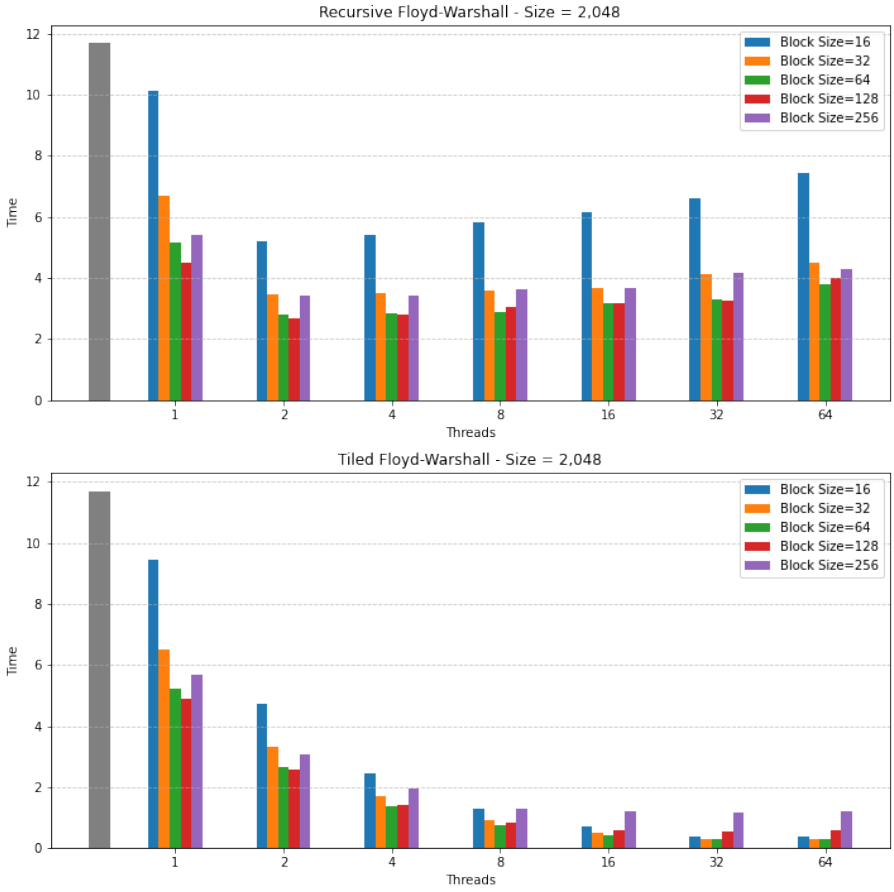


Figure 15: Floyd Warshall - Size: 2048 Results

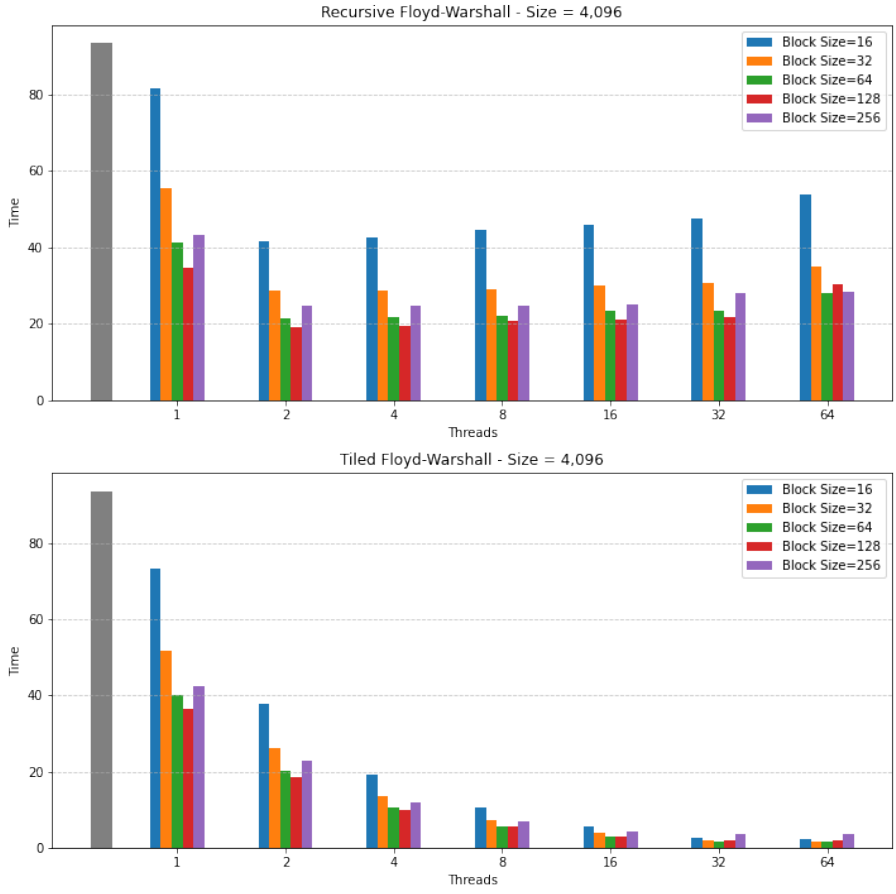


Figure 16: Floyd Warshall - Size: 4096 Results

Ταυτόχρονες Δομές Δεδομένων

Στη συνέχεια μελετάμε διάφορους τύπους κλειδωμάτων πάνω σε δομές δεδομένων. Δηλαδή, μελετάμε τρόπους κλειδώματος μιας δομής δεδομένων την οποία κάνουν access πολλαπλά threads. Στην περίπτωσή μας αυτή η δομή είναι μία απλά συνδεδεμένη ταξινομημένη λίστα.

Ακολουθώντας τις οδηγίες εκτέλεσης για τους διάφορους συνδυασμούς παραμέτρων παίρνουμε τα εξής αποτελέσματα (τα οποία μορφοποιούμε σε διαγράμματα):

Size: 1024 | For Different Workloads (Read-Add-Remove)

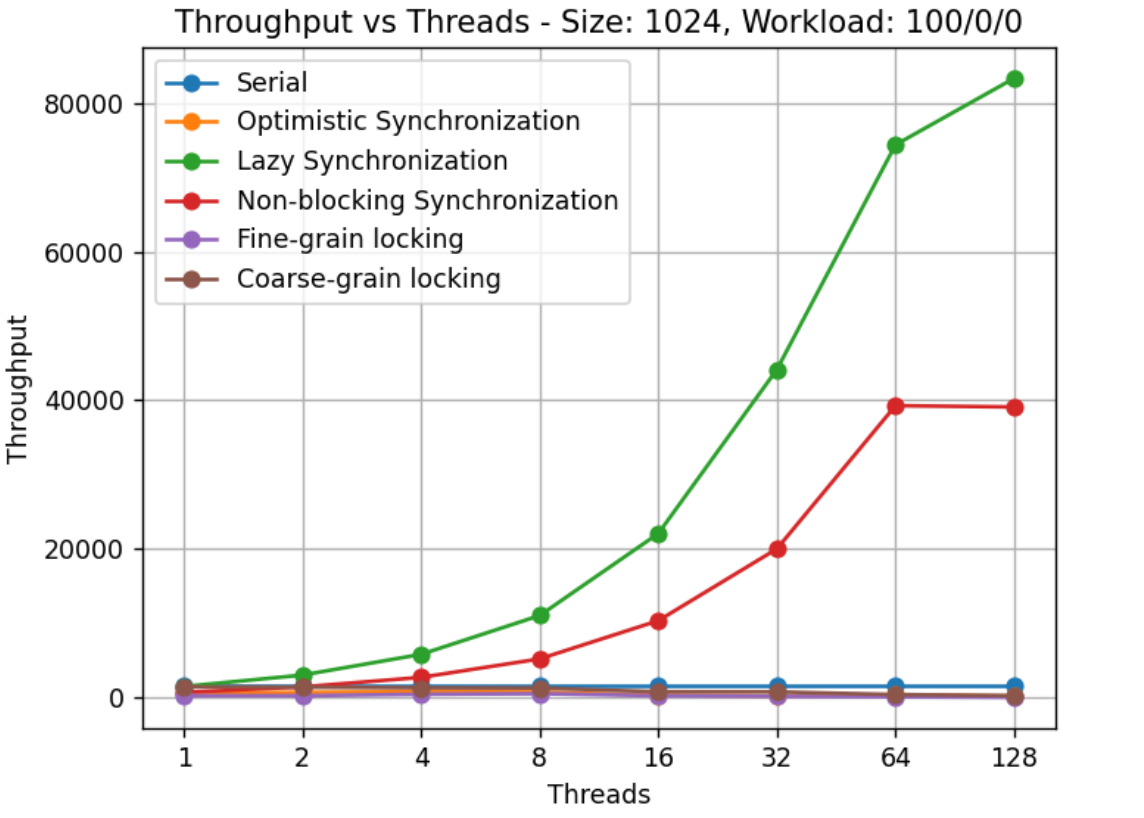


Figure 17: Throughput vs Threads for different parallel implementations
Size: 1024 - Workload: 100% Read / 0% Add / 0% Remove

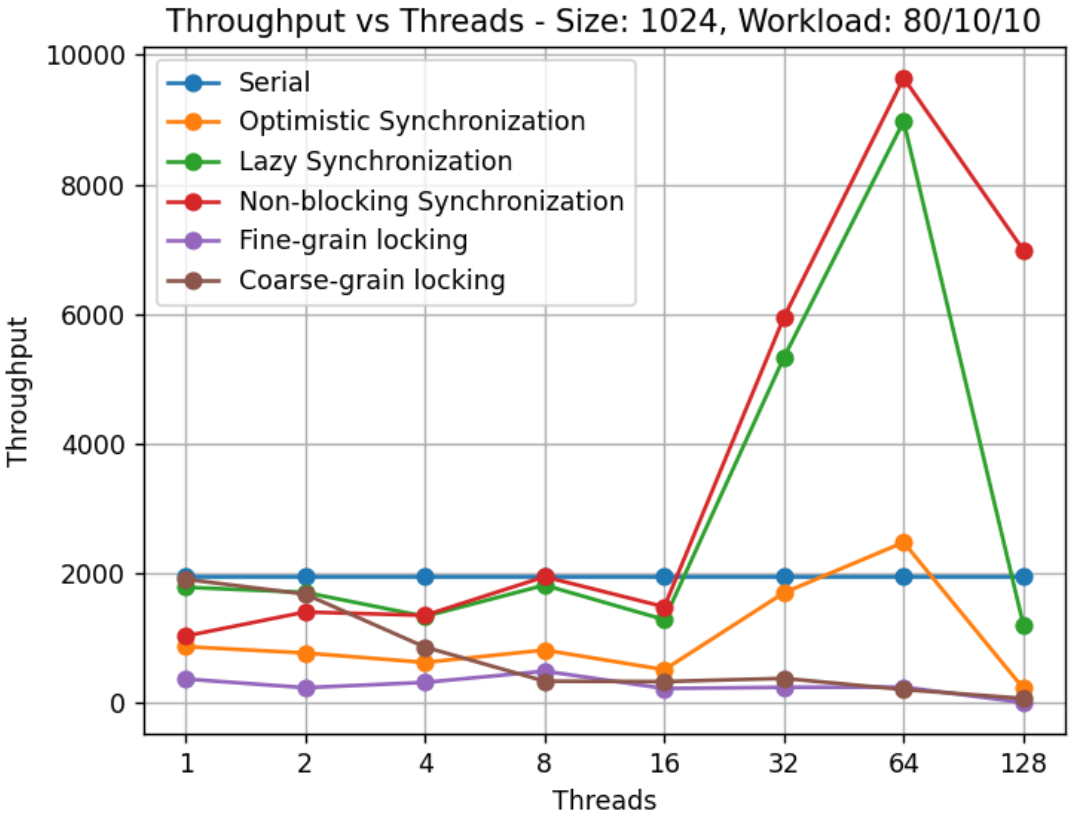


Figure 18: Throughput vs Threads for different parallel implementations
Size: 1024 - Workload: 80% Read / 10% Add / 10% Remove

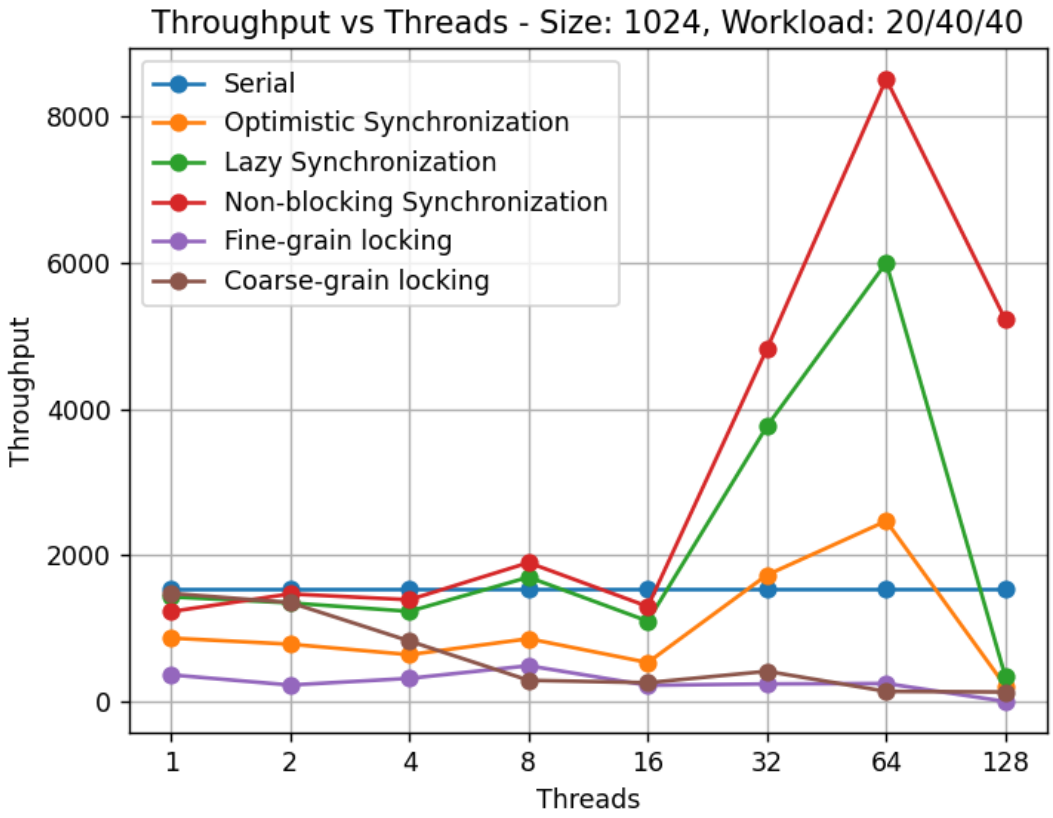


Figure 19: Throughput vs Threads for different parallel implementations
Size: 1024 - Workload: 20% Read / 40% Add / 40% Remove

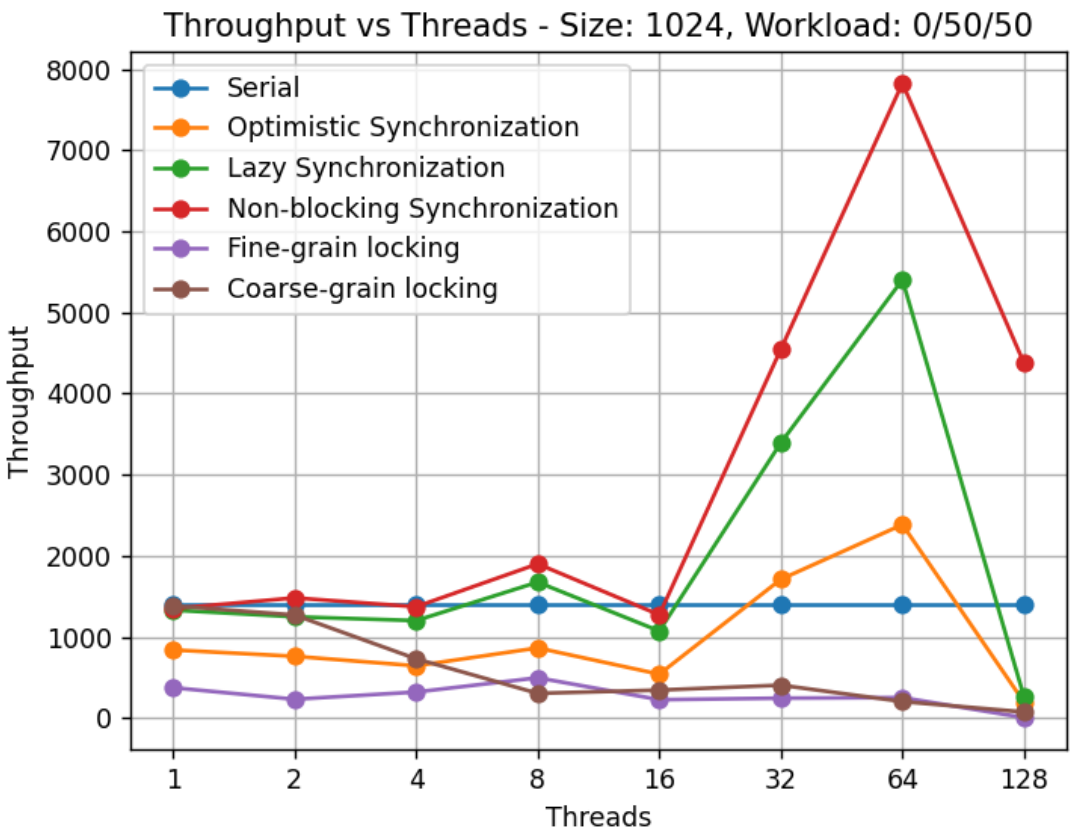


Figure 20: Throughput vs Threads for different parallel implementations
Size: 1024 - Workload: 0% Read / 50% Add / 50% Remove

Size: 8192 | For Different Workloads (Read-Add-Remove)

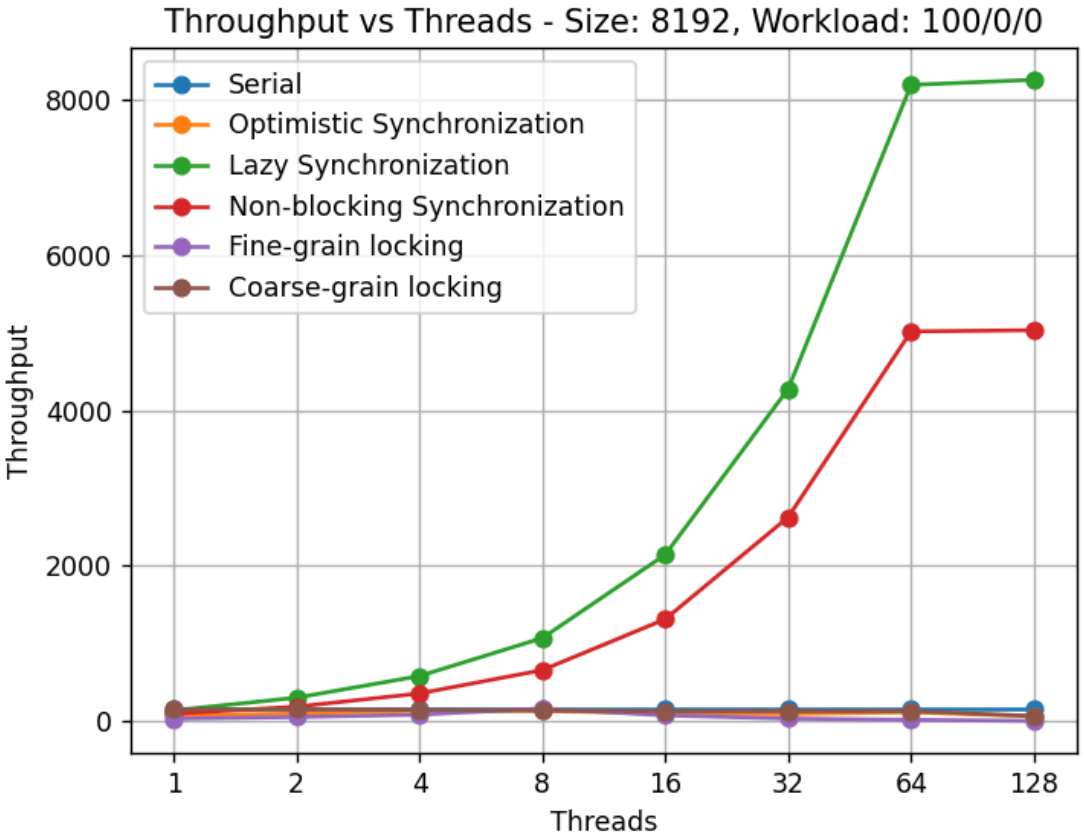


Figure 21: Throughput vs Threads for different parallel implementations
Size: 8192 - Workload: 100% Read / 0% Add / 0% Remove

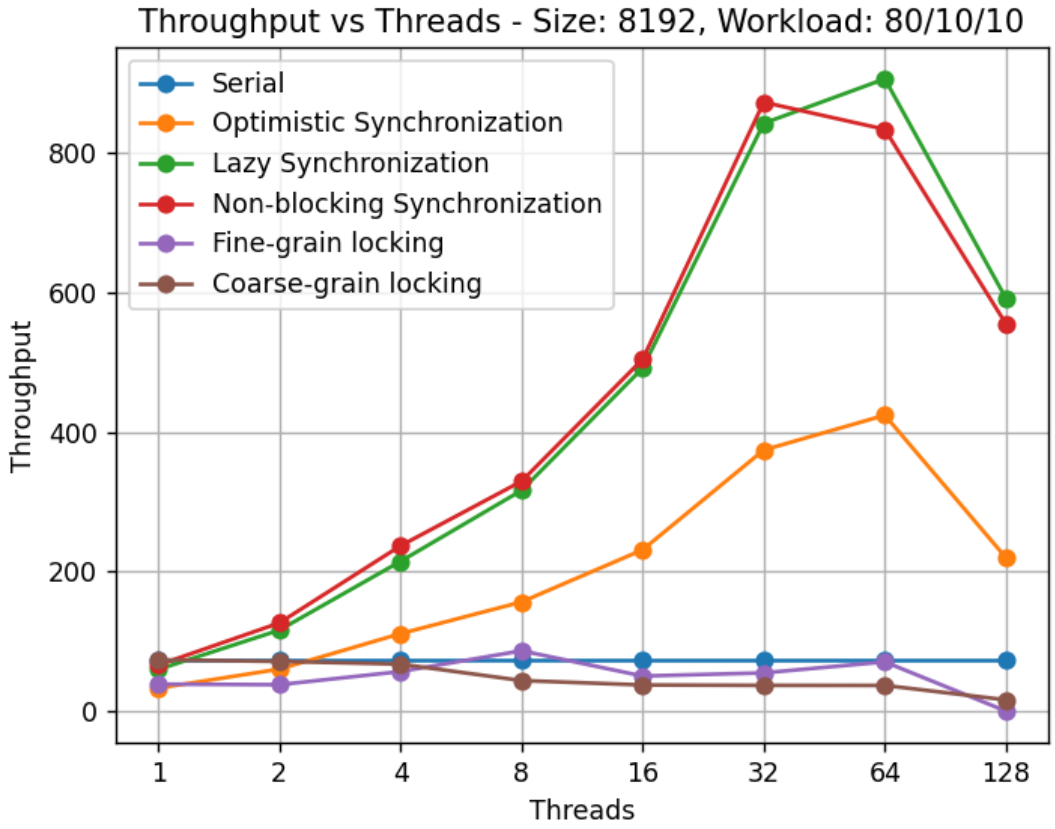


Figure 22: Throughput vs Threads for different parallel implementations
Size: 8192 - Workload: 80% Read / 10% Add / 10% Remove

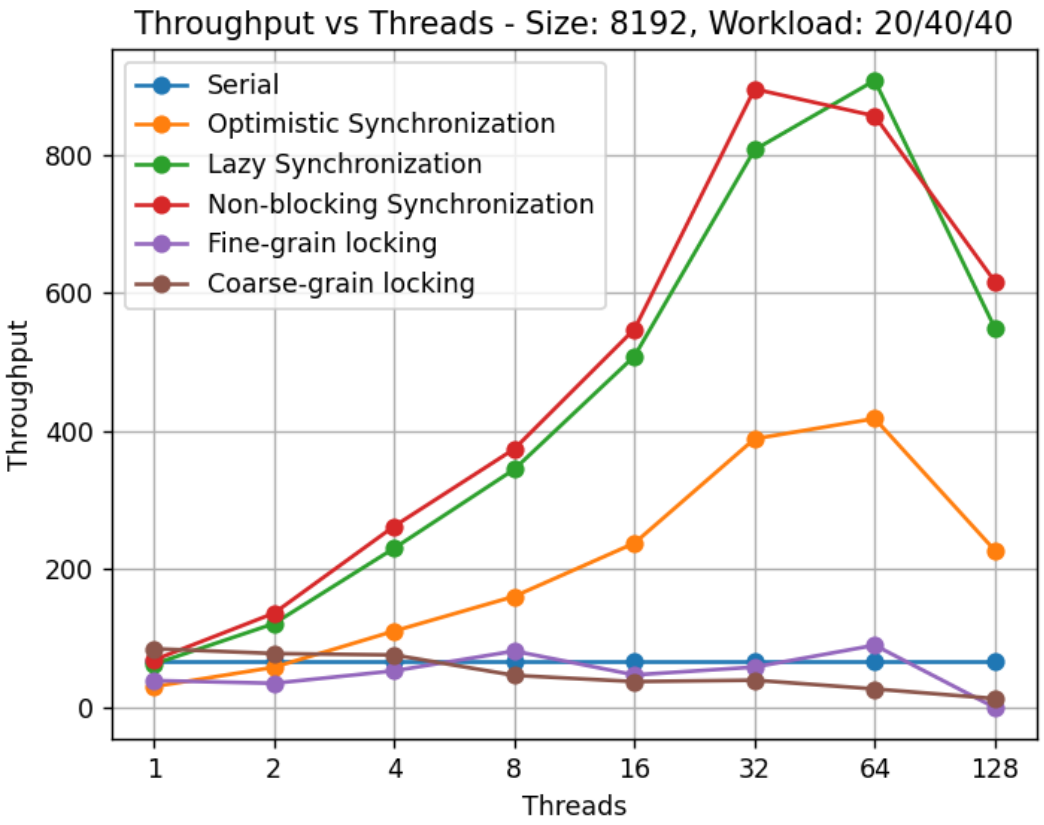


Figure 23: Throughput vs Threads for different parallel implementations
Size: 8192 - Workload: 20% Read / 40% Add / 40% Remove

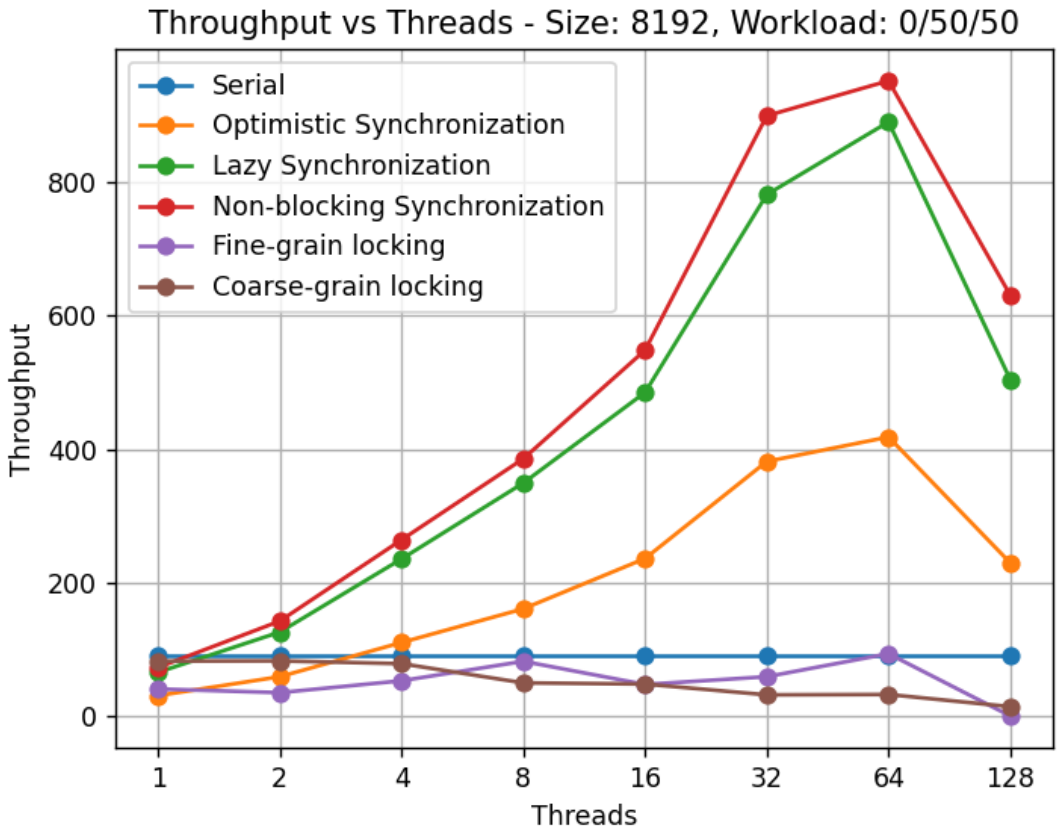


Figure 24: Throughput vs Threads for different parallel implementations
Size: 8192 - Workload: 0% Read / 50% Add / 50% Remove

Με βάση τα παραπάνω διαγράμματα μπορούμε να κάνουμε μερικές παρατηρήσεις για την κάθε υλοποίηση, αλλά και να ερμηνεύσουμε τα αντίστοιχα αποτελέσματα.

1. Serial: Η Serial (δηλαδή σειριακή) έκδοση κάνει χρήση ενός μόνο thread εκτελόντας τον κώδικα για το access της δομής δεδομένων σειριακά. Αυτό επιβεβαιώνεται στα αποτελέσματα αφού σε κάθε περίπτωση έχουμε ένα σταθερό throughput ανεξαιρέτως του αριθμού των threads που χρησιμοποιούνται. Το μόνο που αξίζει να αναφέρουμε είναι πως το throughput δεν αλλάζει ιδιαίτερα ανάλογα με το workload. Για παράδειγμα έχουμε περιπτώσεις που μπορεί να εκτελούμε πολλά reads ή πολλά adds και το throughput να είναι περίπου το ίδιο, παρότι τα reads είναι πολύ πιο "ακριβά" σε operations. Αυτό συμβαίνει επειδή το throughput μετρά Operations/Seconds, και άρα ουσιαστικά δείχνει την δυναμική του core στην σειριακή έκδοση.

2. Coarse-grain locking: Η τεχνική Coarse-grain locking κλειδώνει για κάθε διεργασία ολόκληρη τη δομή δεδομένων μέχρις ότου αυτή να τελειώσει την ενέργεια που θέλει και να απελευθερώσει την δομή. Όπως μπορούμε να φανταστούμε αυτό είναι ιδιαίτερα κακό αφού κάθε ένα thread θα παίρνει το lock ακόμα και αν κάποιο άλλο thread θέλει να εκτελέσει μια ενέργεια που δεν το επηρεάζει. Από τα διαγράμματα μπορούμε να δούμε πως το coarse-grain locking αποδίδει περίπου όπως το σειριακό, αφού τα μόνα επιπλέον computations είναι για τη διεκδίκηση του lock. Επιπλέον για 128 threads επειδή πρέπει να δρομολογήσουμε πλέον τα threads στα 64 virtual threads έχουμε ακόμα μικρότερο throughput. Όσο περισσότερα threads έχουμε, τόσο μεγαλύτερη η διεκδίκηση για το lock και άρα τόσο μικρότερο το throughput σε σχέση με τη σειριακή υλοποίηση.

3. Fine-grain locking: Η τεχνική Fine-grain locking χρησιμοποιεί πολλαπλά κλειδώματα για την δομή ώστε να μπορούμε να έχουμε παράλληλο access σε αυτή. Στην περίπτωσή μας κάθε κόμβος της λίστας θα έχει και ένα κλείδωμα. Όπως φανταζόμαστε για να διαβάσουμε ένα στοιχείο θα πρέπει διαδοχικά να περάσουμε από τα κλειδώματα. Αν όμως θέλουμε να προσθέσουμε ή να διαγράψουμε, χρειαζόμαστε κλειδώματα μιας περιοχής. Αντίστοιχα, το γεγονός πως για να διατρέξουμε την λίστα, απαιτεί να πάρουμε διαδοχικά τα κλειδώματα, δημιουργεί πολλά προβλήματα στην περίπτωση που μια περιοχή (ειδικά στην αρχή) έχει συχνά adds και deletes. Όπως είναι λογικό, λοιπόν, έχουμε στην πλειοψηφία των περιπτώσεων χειρότερη απόδοση από το serial implementation, λόγω αυτού ακριβώς του blocking μηχανισμού, όταν έχουμε μεγάλο αριθμό threads. Από την άλλη για μικρό αριθμό threads, μπορεί να μην έχουμε τόσες συγκρούσεις, αλλά ούτε τότε βλέπουμε ιδιαίτερα μεγάλα gains σε σχέση με την σειριακή έκδοση. Αυτό συμβαίνει επειδή με λίγα threads δεν μπορούμε να πετύχουμε μεγάλη παραλληλοποίηση, αλλά και επειδή ακόμα και με λίγα threads, η σειριακή

απόκτηση των locks δημιουργεί μεγάλη καθυστέρηση. Συνεπώς ότι κερδίζουμε από παραλληλοποίηση, χάνουμε από τον blocking μηχανισμό.

4. Optimistic Synchronization: Το Optimistic Synchronization έχει το πλεονέκτημα, πως για προσθήκες και διαγραφές η αναζήτηση της θέσης που θα γίνει η προσθήκη ή η διαγραφή γίνεται χωρίς διεκδίκηση του lock. Μόλις βρεθεί η θέση τότε διεκδικούνται τα locks της περιοχής και γίνεται έλεγχος αν έχει αλλάξει κάτι στην περιοχή μέχρι να παρθούν τα locks. Αντίστοιχα, για τα reads έχουμε κανονικά διεκδίκηση των διαδοχικών locks. Άρα, όπως φαίνεται και στα διαγράμματα, όταν έχουμε μόνο reads τα αποτελέσματά μας είναι πανομοιότυπα με το Fine-grain locking. Σε περιπτώσεις όμως που έχουμε adds και removes, παρατηρούμε μια μεγάλη αύξηση στο throughput, σε σχέση με προηγούμενες υλοποιήσεις, αφού adds και removes σε διαφορετικά σημεία της λίστας μπορούν να γίνουν ταυτόχρονα. Και πάλι παρατηρούμε πως για 128 threads, λόγω της δρομολόγησης των threads στα virtual threads του επεξεργαστή αλλά και του μεγάλου ανταγωνισμού, έχουμε μείωση του throughput.

5. Lazy Synchronization: Το Lazy Synchronization βασίζεται σε ένα μηχανισμό που χωρίζει τις αφαιρέσεις κόμβων σε δύο μέρη: στην λογική αφαίρεση του κόμβου (γρήγορη διαδικασία) και στην φυσική αφαίρεση του κόμβου (αργή διαδικασία). Για παράδειγμα, με ένα dirty bit μπορούμε να θεωρήσουμε κόμβους άκυρους (διεγραμμένους) και να τους αφαιρέσουμε εντελώς κάποια άλλη στιγμή. Παράλληλα τα reads, όπως μπορούμε να δούμε γίνονται πολύ γρήγορα (workload 100/0/0). Αυτό συμβαίνει επειδή γίνεται χρήση read-write locks, τα οποία επιτρέπουν πολλούς αναγνώστες ή ένα μόνο εγγραφέα. Έτσι σε περιπτώσεις με reads πετυχαίνουμε τα καλύτερα αποτελέσματα, ενώ αντίστοιχα όταν έχουμε adds και removes έχουμε επίσης καλό throughput λόγω του optimization που αναφέραμε αρχικά. Τέλος, όπως και στο Optimistic Synchronization, έχουμε την λογική της αναζήτησης χωρίς locks για εγγραφή/διαγραφή. Έτσι το implementation αυτό μας δίνει ίσως τα καλύτερα αποτελέσματα από τις 6 επιλογές. Και πάλι, όμως, παρατηρούμε πως για 128 threads, λόγω της δρομολόγησης των threads στα virtual threads του επεξεργαστή αλλά και του μεγάλου ανταγωνισμού για πολλά adds και removes, έχουμε μείωση του throughput.

6. Non-blocking Synchronization: Το non-blocking Synchronization, σύμφωνα και με το όνομά του, δεν κάνει χρήση locks πάνω στη δομή δεδομένων. Αντίθετα, χρησιμοποιεί ατομικές εντολές υλοποιημένες στο υλικό ή λογισμικό, μέσω των οποίων εκτελεί τα read, add και remove. Έτσι δεν υπάρχει ανάγκη απο συγχρονισμό. Όπως βλέπουμε και από τα διαγράμματα, η υλοποίηση αυτή μας δίνει πολύ καλά αποτελέσματα, ειδικά όταν έχουμε πολλά adds και removes, αφού αυτά εκτελούνται ατομικά. Και πάλι, όμως, παρατηρούμε πως για 128 threads, λόγω της δρομολόγησης των threads στα virtual threads του επεξερ-

γαστή έχουμε μείωση του throughput.