2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Shared Clusters:

Η πρώτη υλοποίηση που καλούμαστε να συμπληρώσουμε είναι η naive υλοποίηση.
Ζητείται το configuration:

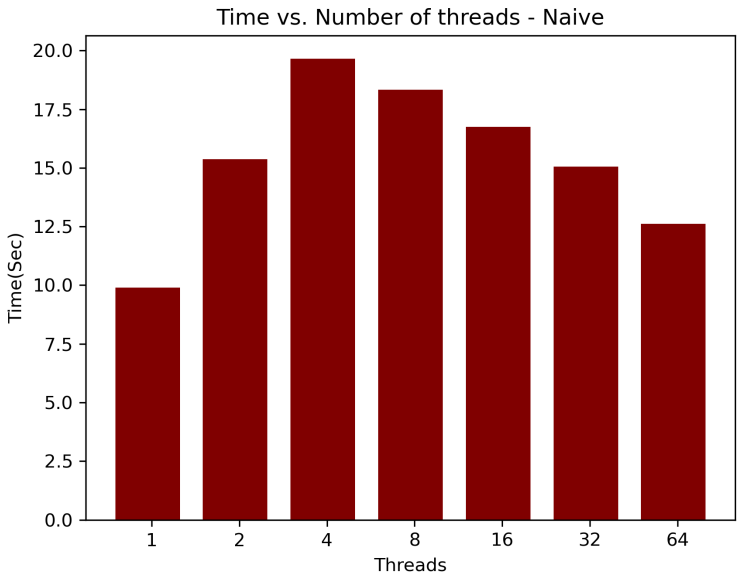
$$\{\text{Size, Coords, Clusters, Loops}\} = \{256, 16, 16, 10\}$$

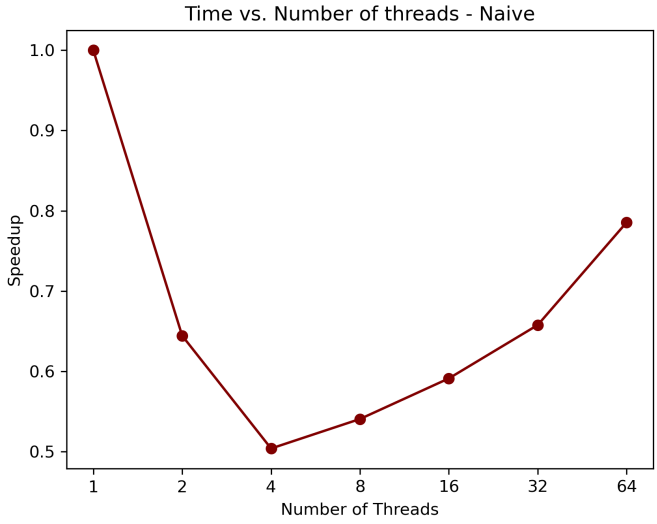
Καλούμαστε να παραλληλοποιήσουμε το πιο υπολογιστικα βαρύ κομμάτι του κώδικα που είναι η εύρεση του κοντινότερου cluster για κάθε σημείο του dataset. Ο υπολογισμός γίνεται με τη συνάρτηση find_nearest_cluster().

```
        */

        #pragma omp parallel for shared(newClusters, objects, membership,clusters,newClusterSize,numCoords,delta) private(i,j,index)
        for (i=0; i<numObjs; i++) {
            // find the array index of nearest cluster center
            index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);

            // if membership changes, increase delta by 1
            if (membership[i] != index)
                #pragma omp atomic
                delta += 1.0;

            // assign the membership to object i
            membership[i] = index;

            // update new cluster centers : sum of objects located within
            /*
             * TODO: protect update on shared "newClusterSize" array
             */
            #pragma omp atomic
            newClusterSize[index]++;


            for (j=0; j<numCoords; j++)
                /*
                 * TODO: protect update on shared "newClusters" array
                 */
                #pragma omp atomic
                newClusters[index*numCoords + j] += objects[i*numCoords + j];

        }
```

Στη παραπάνω υλοποίηση παραλληλοποιούμε το loop με #pragma omp parallel for. Κλειδώνουμε τη με ατομική εντολή τη μεταβλητή delta όταν τα threads επιθυμούν να αυξήσουν τον αριθμό του count των αλλαγών στο membership, ενώ κλειδώνουμε με ατομική εντολή και το πίνακα μεγέθους των clusters, το newClusterSize και τον πίνακα που έχει τα κέντρα των clusters, το newClusters. Η ατομική εντολή προσδιορίζεται απο το #pragma omp atomic. Παρατηρούμε στο διάγραμμα χρόνου εκτέλεσης προς τον αριθμό των νημάτων τα παρακάτω αποτελέσματα. Για τα benchmarks χρησιμοποιούμε το μηχάνημα sandman που μας δίνει τη δυνατότητα να χρησιμοποιήσουμε 64 threads.

Παρατηρούμε πως φαίνεται η σειριακή υλοποίηση είναι η γρηγορότερη εκ των υπολοίπων, αυτό οφείλεται στο ανταγωνισμό για το lock των atomic instructions που έχουμε στο κώδικα, συνολικά γίνονται πολλές προσβάσεις στους πίνακες newClusters και newClusterSize. Όταν κλειδώνει η πρόσβαση από ένα νήμα, κλειδώνει όλος ο πίνακας για τα άλλα νήματα οπότε έχουμε συνωστισμό. Επίσης ο χρόνος εκτέλεσης χειροτερεύει και απο φαινόμενα όπως ότι τα δεδομένα τα οποία θέλει ο κάθε επεξεργαστής δεν είναι pinned στη cache αυτού.



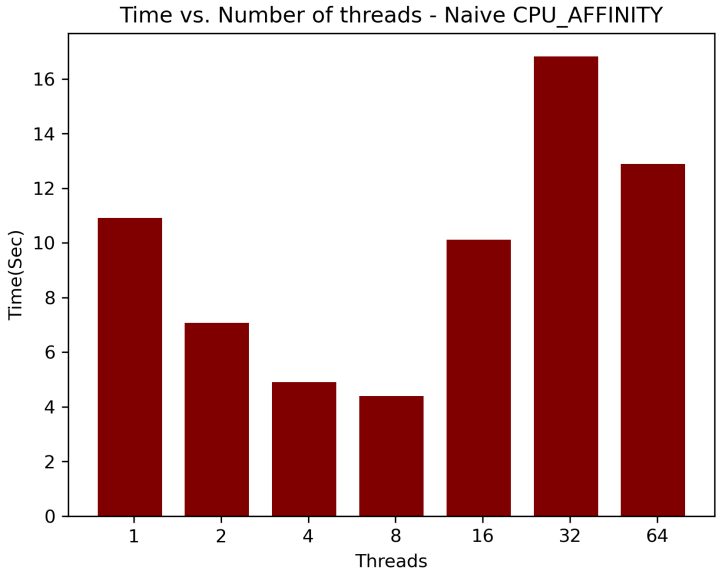


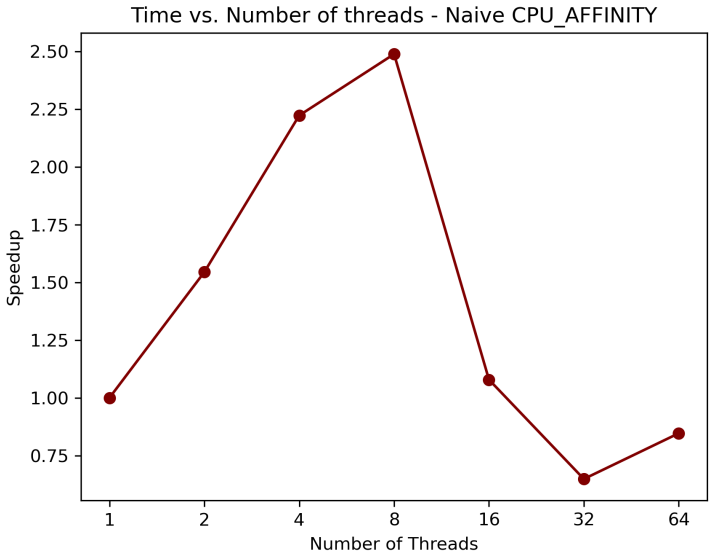
Το ζήτημα του pinning των δεδομένων το αντιμετωπίζουμε με την επιλογή GOMP_CPU_AFFINITY:

```
for i in 1 2 4 8 16 32 64
do
    export OMP_NUM_THREADS=${i}
    export GOMP_CPU_AFFINITY="0-63"
```

Έτσι προσδένουμε τα δεδομένα του κάθε επεξεργαστή στη cache του, βελτιώνοντας το locality των δεδομένων. Συγκεκριμένα στους επεξεργαστές 0 έως 31. Παρατηρούμε όπως είναι αναμενόμενο καλύτερους χρόνους και βελτίωση σε χρόνο στις παράλληλες

υλοποιήσεις σε σχέση με τη σειριακή. Χάνονται αυτά τα πλεονεκτήματα απο 16 threads και μετά λόγω αυξημένου contention. Καλύτερη επίδοση παρατηρούμε στα 8 threads.





Copied and Reduced Clusters:

Ορίζουμε τα τοπικά clusters που θα χρησιμοποιηθούν για να μαζέψουν τα επιμέρους αριθμητικά αποτελέσματα για την ανανέωση των “γενικών” clusters, όπως φαίνεται στο

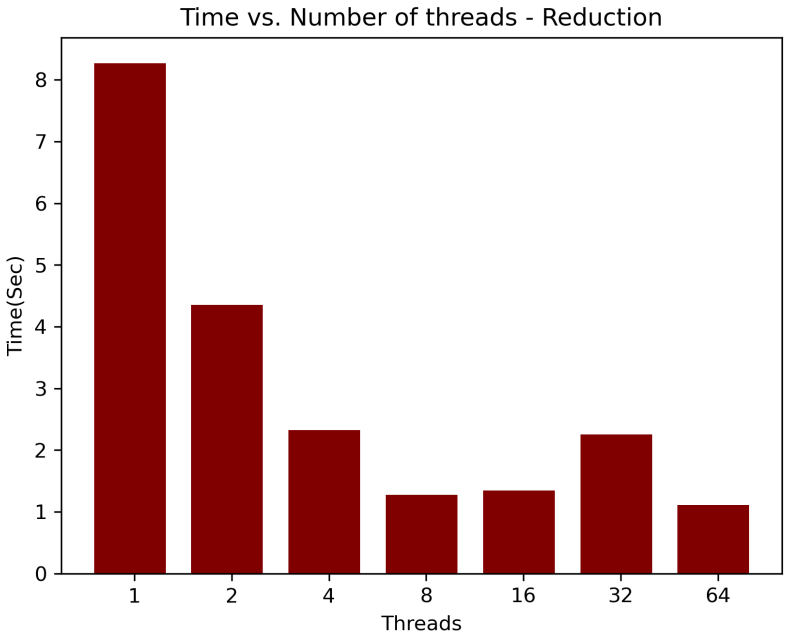
πρώτο TODO. Στη συνέχεια υλοποιούμε τη παραλληλοποίηση της εύρεσης των κοντινότερων clusters.

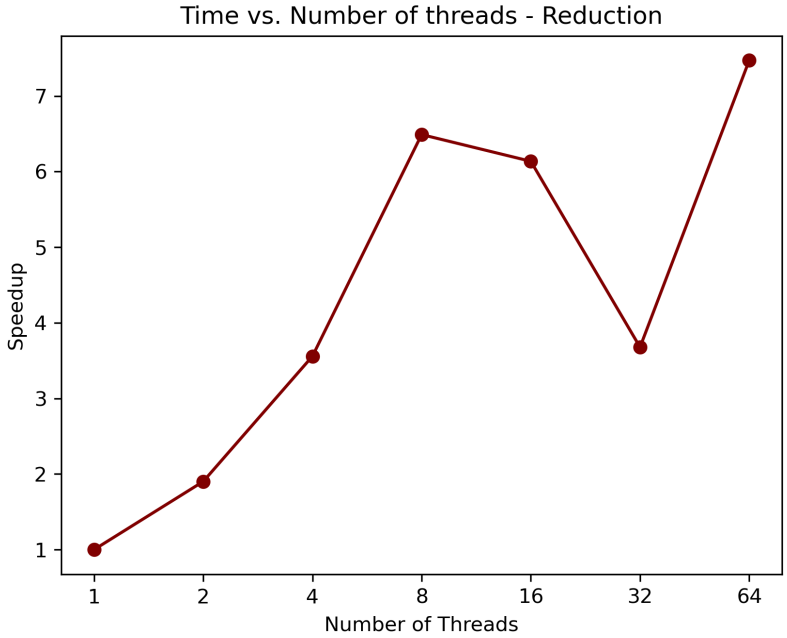
```
    /*
     * TODO: Initiliaze local cluster data to zero (separate for each thread)
     */
    for(k=0; k<nthreads; k++){
       for (i=0; i<numClusters; i++) {
           for (j=0; j<numCoords; j++)
               local_newClusters[k][i*numCoords + j] = 0.0;
           local_newClusterSize[k][i] = 0;
    }
    }
    #pragma omp parallel shared(local_newClusters, objects,delta,membership,clusters,local_newClusterSize,numCoords) private(i,j,index)
    {
    #pragma omp for
    for (i=0; i<numObjs; i++)
    {
        // find the array index of nearest cluster center
        index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);

        // if membership changes, increase delta by 1
        if (membership[i] != index)
            #pragma omp atomic
            delta += 1.0;

        // assign the membership to object i
        membership[i] = index;
```

Αντι να έχουμε ένα global array για clusters όπως στη naive υλοποίηση και να δημιουργείται ένα bottleneck λόγω των προσβάσεων που επιθυμούν να πραγματοποιήσουν τα threads και να ανανέωσουν τόσο το newClusters όσο και το newClusterSize καθώς τότε κλειδώνεται όλος ο πίνακας τώρα το κάθε thread γράφει σε δικά του τοπικά arrays και το master thread στο τέλος μαζεύει αποτελέσματα με #pragma omp master. Έτσι προκύπτει και η υλοποίηση του reduction.

```
        // update new cluster centers : sum of all objects located within (average will be performed later)
        /*
         * TODO: Collect cluster data in local arrays (local to each thread)
         *       Replace global arrays with local per-thread
         */
        int tid = omp_get_thread_num();
        local_newClusterSize[tid][index]++;
        for (j=0; j<numCoords; j++)
            local_newClusters[tid][index*numCoords + j] += objects[i*numCoords + j];

    }

    /*
     * TODO: Reduction of cluster data from local arrays to shared.
     *       This operation will be performed by one thread
     */
    #pragma omp barrier
    #pragma omp master
    {
        for (k=1; k<nthreads; k++)
        {
            for (i=0; i<numClusters; i++)
            {
                for (j=0; j<numCoords; j++) {
                    local_newClusters[0][i*numCoords + j] += local_newClusters[k][i*numCoords + j];
                }

                local_newClusterSize[0][i] += local_newClusterSize[k][i];
    }
    }
```

Στο reduction implementation έχουμε σχεδόν γραμμικό speedup μεχρι και 8 threads, απο 16 και μετα δεν παρατηρείται βελτίωση στο χρόνο (ακόμα υπάρχει και χειροτέρευση για 64 threads).

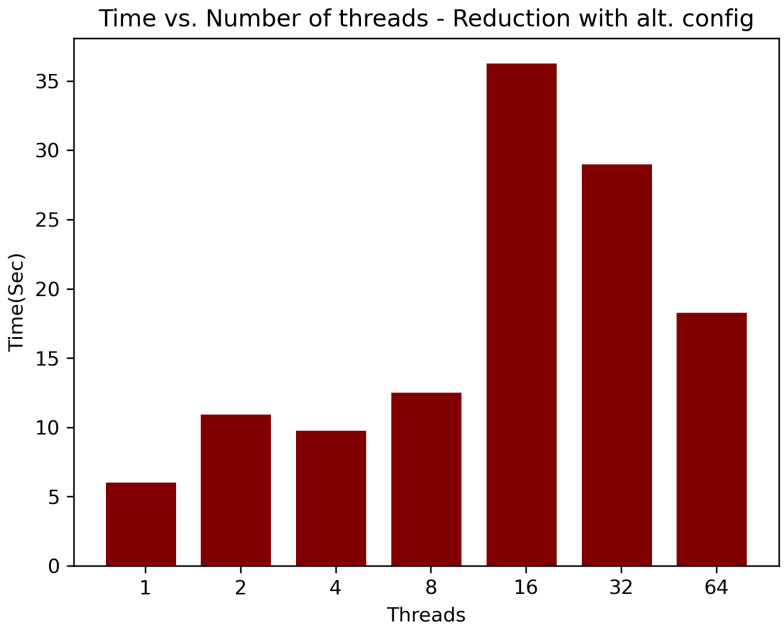

Time vs. Number of threads - Reduction
Time(Sec)
0
1
2
3
4
5
6
7
8
1
2
4
8
16
32
64
Threads

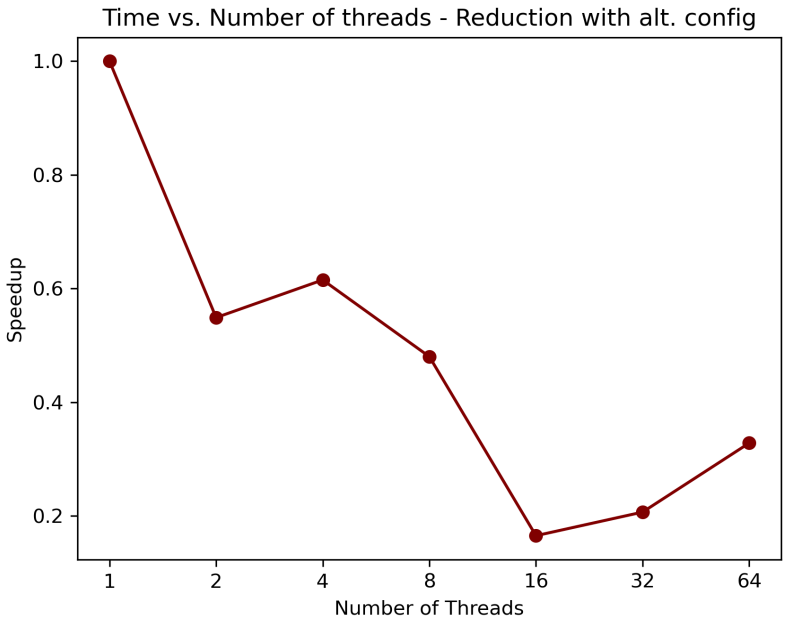

Time vs. Number of threads - Reduction
Speedup
1
2
3
4
5
6
7
1
2
4
8
16
32
64
Number of Threads

Ζητείται να συγκρίνουμε και με μια εναλλακτική αρχικοποίηση:

{Size, Coords, Clusters, Loops} = {256, 1, 4, 10}

Παρατηρούμε στη παραμετροποίηση αυτή πως ο καλύτερος χρόνος προκύπτει απο τη σειριακή υλοποίηση.Διαπιστώνουμε ότι οι χρόνοι εκτέλεσης και η κλιμάκωση δεν είναι ικανοποιητική. Αυτό οφείλεται στον μικρό αριθμό των clusters, ο οποίος οδηγεί στο να βρίσκονται περισσότερες γραμμές των πινάκων local_newClusters και local_newClusterSize στην ίδια περιοχή της cache μνήμης. Αποτέλεσμα αυτού είναι το φαινόμενο του False Sharing, κατά το οποίο η εγγραφή δεδομένων από ένα νήμα συμπίπτει χωρικά με αυτές των άλλων νημάτων, καθώς γράφονται στην ίδια cache line. Σε επόμενη εγγραφή τα νήματα πρέπει να κάνουν invalidate τη cache line που θέλουν να γράψουν και ύστερα να φορτώσουν εκ νέου τα δικά τους δεδομένα. Όλο αυτο δημιουργεί αργοπορία λόγω της μετακίνησης δεδομένων από κύρια μνήμη και ανάποδα, που χειροτερεύει με τον αριθμό των νημάτων.
Ο καλύτερος χρόνος που πετύχαμε είναι 5.79 sec στη σειριακή υλοποίηση.





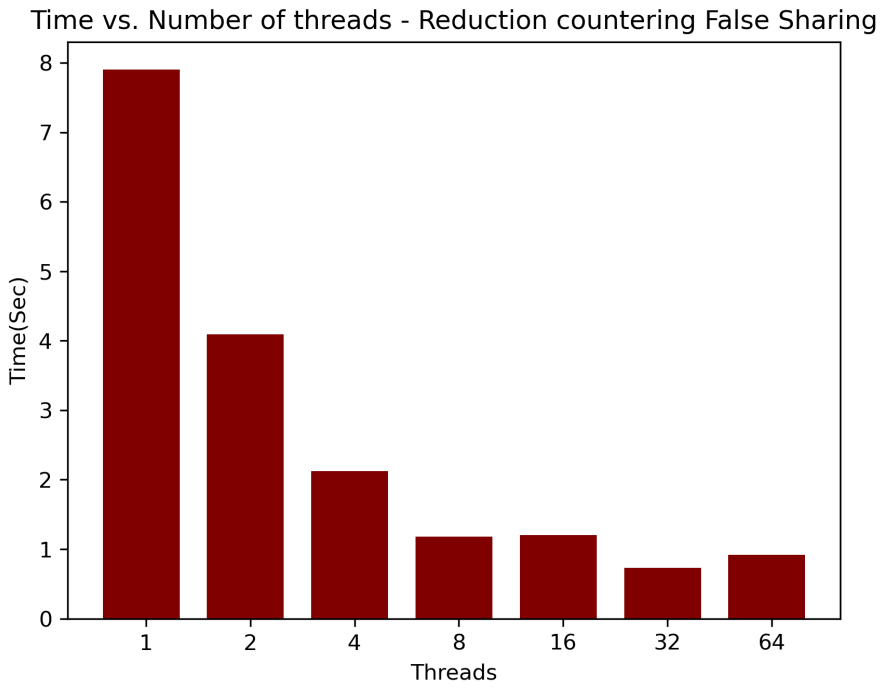
Για να αποφύγουμε αυτό το φαινόμενο, θα προσθέσουμε μηδενικά στο τέλος κάθε τμήματος των πινάκων που αντιστοιχεί σε κάποιο thread (padding), ώστε να συμπληρωθεί το επιθυμητό μήκος cache line. Έτσι, αποφεύγουμε να επεξεργάζονται το ίδιο cache line δύο ή περισσότερα threads. Αυτό θα γίνει κατά το allocation:

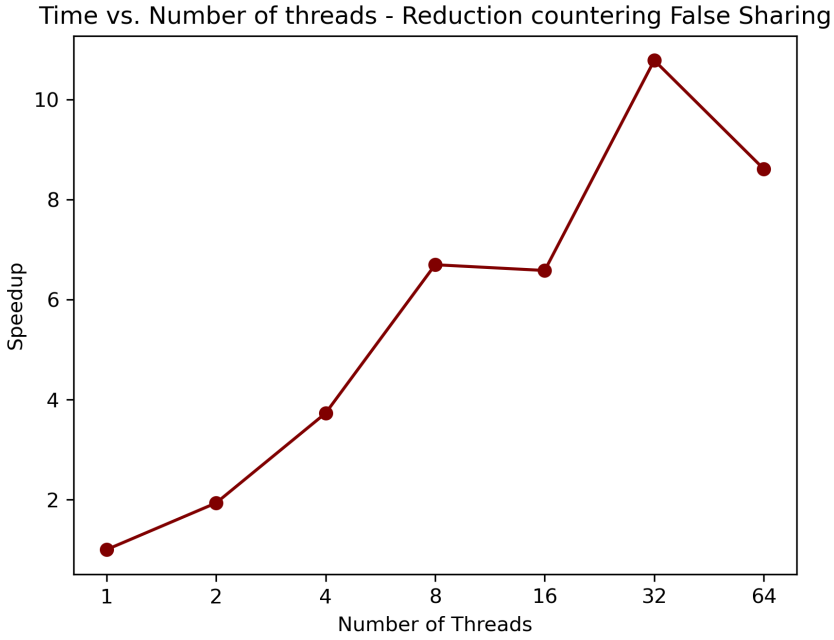
Η πολιτική "first touch" στα συστήματα NUMA ορίζει ότι η μνήμη που ανατίθεται σε ένα νήμα προέρχεται από την περιοχή μνήμης που είναι πιο κοντά στον επεξεργαστή που εκτελεί αυτο νήμα για πρώτη φορά. Αυτό βελτιστοποιεί την απόδοση, καθώς ελαχιστοποιείται η καθυστέρηση πρόσβασης στη μνήμη. Έχουμε θέσει όπως είχε ζητηθεί GOMP_CPU_AFFINITY=’0-63’ για τα ερωτήματα του reduction και ετσι τα νήματα προσδένονται στις CPU που πρωτοέφεραν τα δεδομένα τους. Τώρα όλα τα νήματα που θα χρειάζονται το δεδομένο αυτό δε θα το πηγαινοφερνουν μεταξύ της RAM των nodes που εκείνα βρίσκονται και αυτής της αρχικής ανάθεσης.

Το μεγαλύτερο ίσως optimization είναι η παραλληλοποίηση του allocation των local_newClusters και local_newClusterSize, ετσι το κάθε νήμα κάνει allocate blocks με private τρόπο, επομένως μειώνεται αρκετά η πιθανότητα να πέσουν blocks από δύο ξεχωριστά νήματα στην ίδια Cache Line. Τα παραπάνω τα υλοποιούμε με #pragma omp parallel for private (k).

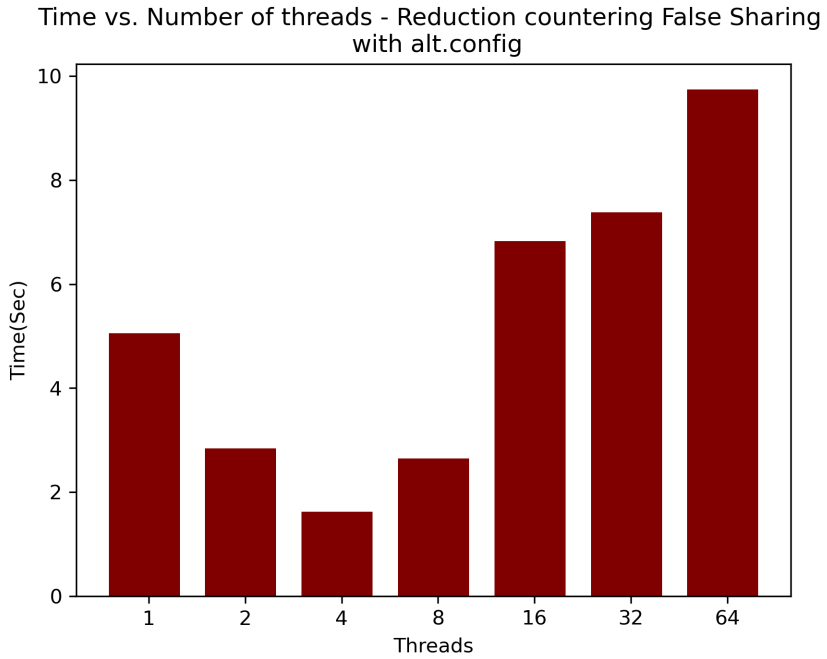
```c
#define CACHE_LINE_SIZE 64
#define PADDING_INT ((numClusters/(CACHE_LINE_SIZE/sizeof(int))+1)*sizeof(int) - numClusters)
#define PADDING_DOUBLE (((numClusters*numCoords)/(CACHE_LINE_SIZE/sizeof(double))+1)*sizeof(double) - numClusters*numCoords)

// 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.
 */
// Initialize local (per-thread) arrays (and later collect result on global arrays)
#pragma omp parallel for private (k)
for (k=0; k<nthreads; k++)
{

local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters+PADDING_INT, sizeof(**local_newClusterSize));

local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters*numCoords+PADDING_DOUBLE, sizeof(**local_newClusters));
}
```

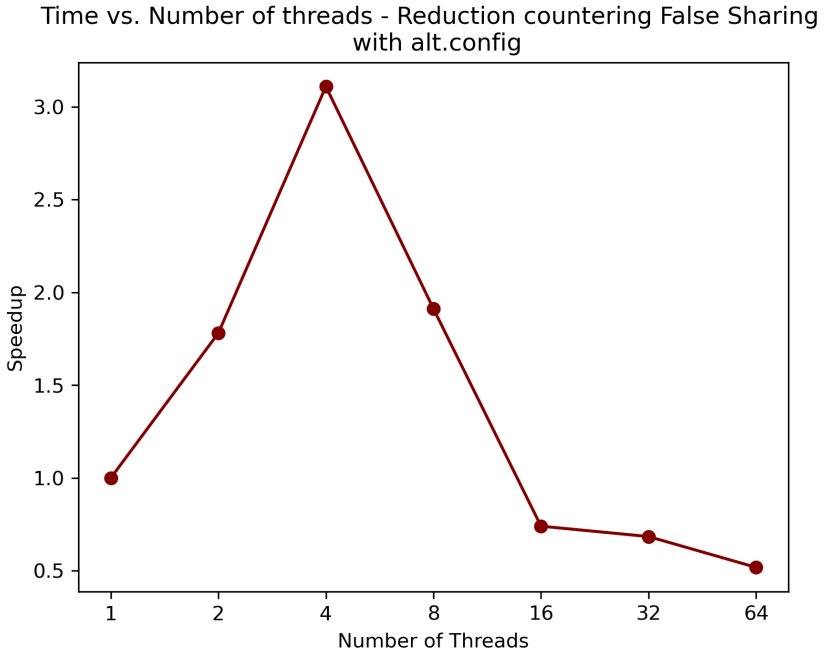
Παρατηρούμε πως και στις δύο περιπτώσεις έχουμε αισθητά βελτιωμένο χρόνο εκτέλεσης:





Ειδικά στη περίπτωση του alternative configuration επειδή το μέγεθος της cache line είναι 64 bytes, συμπεραίνουμε πως ο πίνακας των localClusters για ένα νήμα χωράει ακριβώς στη μισή cache line. Με συνέπεια το ακριβώς διπλανό νήμα να φέρνει το δικο του localClusters στην ίδια γραμμή κάνοντας συνεχώς invalidate τα άλλα αντίγραφα των cache lines και ξαναφορτώνοντας με τα δικά του δεδομένα. Με το padding όμως αναγκάζουμε κάθε επόμενο localClusters να πάει στην απο κάτω cache line και η διαφορά είναι εμφανής.


Time vs. Number of threads - Reduction countering False Sharing with alt.config
Time(Sec)
0
2
4
6
8
10
1
2
4
8
16
32
64
Threads


Time vs. Number of threads - Reduction countering False Sharing with alt.config
Speedup
0.5
1.0
1.5
2.0
2.5
3.0
1
2
4
8
16
32
64
Number of Threads

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

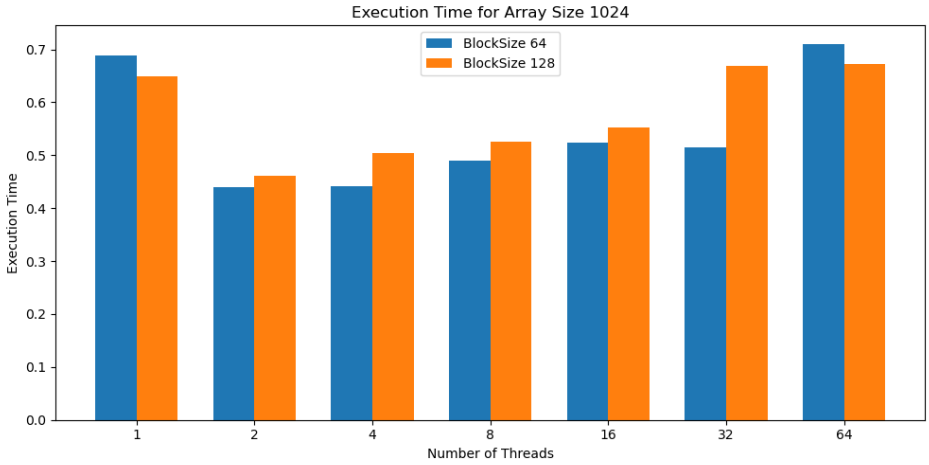
Σε αυτό το ερώτημα παραλληλοποιούμε τον αλγόριθμο fw_sr.c και πραγματοποιήούμε μετρήσεις για μεγέθη πινάκων 1024x1024, 2048x2048 και 4096x4096 για threads = {1, 2, 4, 8, 16, 32, 64} στο μηχάνημα sandman.

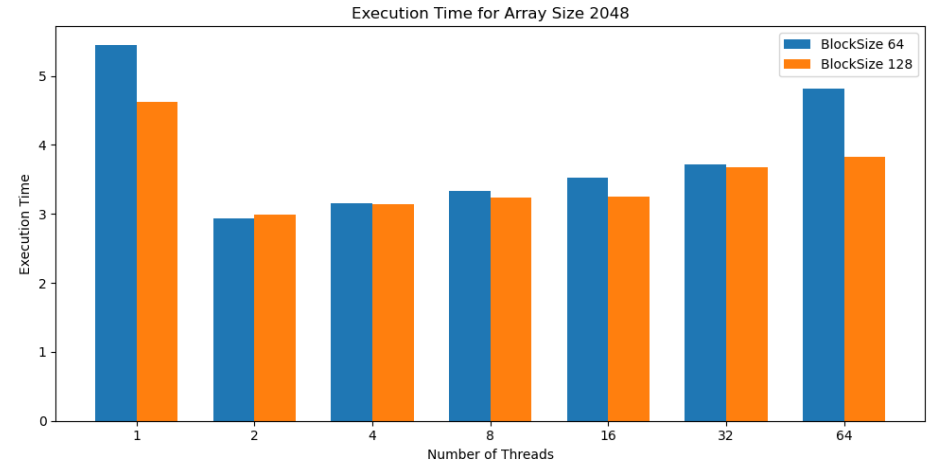
```
    if(myN<=bsize)
       for(k=0; k<myN; k++)
          for(i=0; i<myN; i++)
             for(j=0; j<myN; j++)
                A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
    else {
       FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize);
       #pragma omp parallel
                    {
                        #pragma omp single
                              {
                                  #pragma omp task
                                            FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize);
                                 #pragma omp task
                                 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
                                            FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
                                 #pragma omp task
                                 FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2, myN/2, bsize);
                                  #pragma omp taskwait
                              }
                    }
      FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize);
    }
}
```

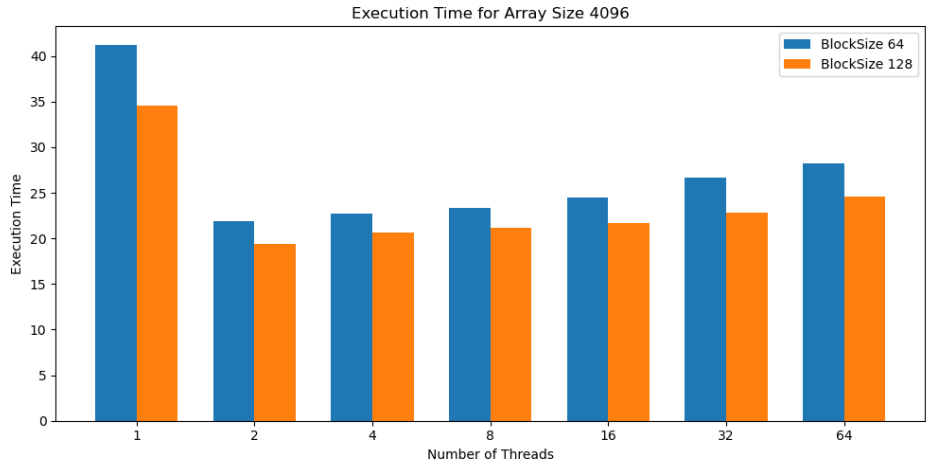
Παραλληλοποιούμε τον κώδικα με χρήση task. Βλέπουμε πως λόγω των εξαρτήσεων μπορούμε να παραλληλοποιήσουμε μόνο τις {2,3} και {6,7} αναδρομικές κλήσεις της συνάρτησης FW_SR.

Δοκιμάζουμε και μετράμε χρόνο εκτέλεσης για N = {1024,2048,4096} και B = {64,128} με Threads= {1,2,4,8,16,32,64}

Παρακάτω λαμβάνουμε τα bar plots για καθε array configuration:



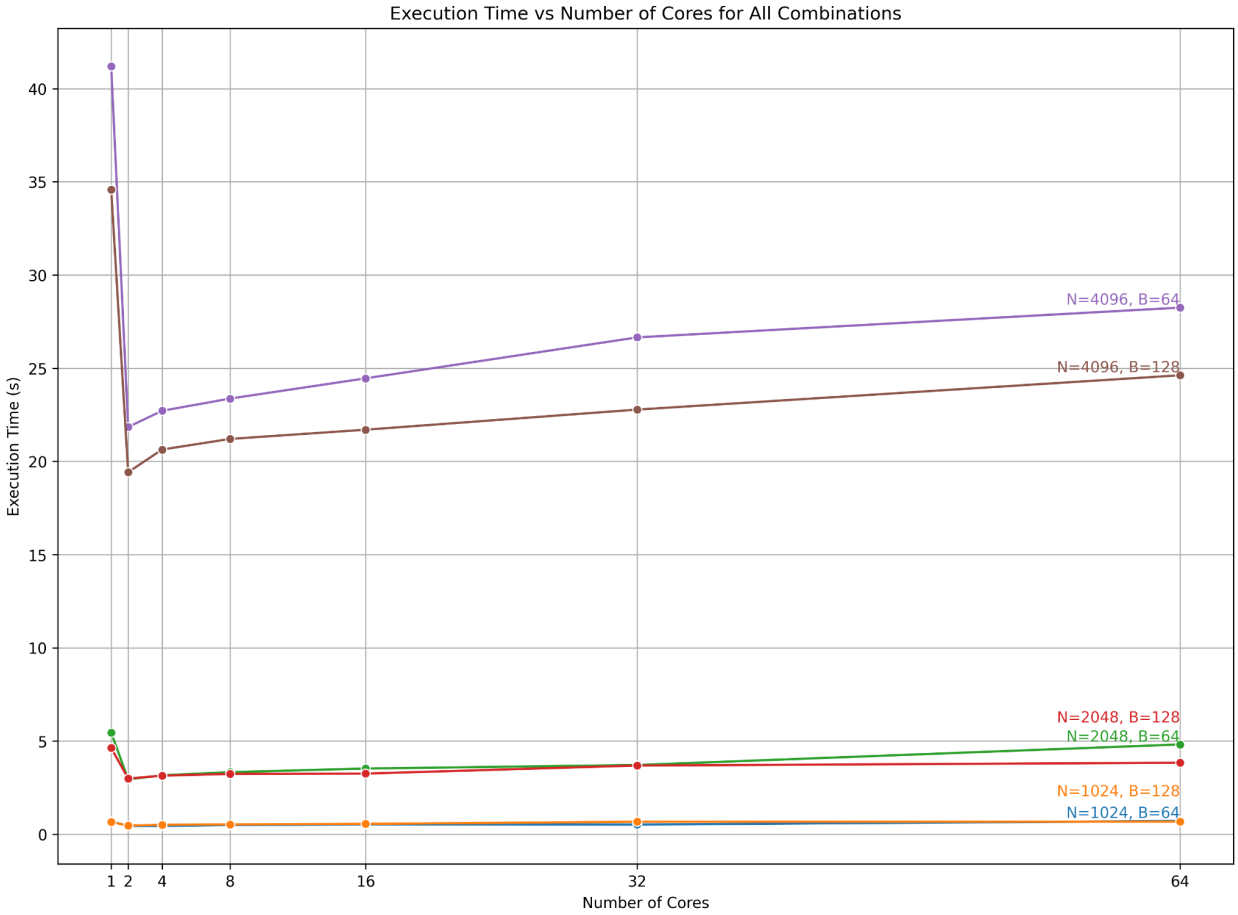




Για 64 threads έχουμε εμφανώς χειρότερη απόδοση ως προς τη κλιμάκωση μέχρι και τα 32 threads, αυτο συμβαίνει διότι λόγω του hyperthreading, το hyperthreading επιχειρεί να έχει δύο

pipelines στον ίδιο πυρήνα και να εκμεταλλευτεί το οτι οι δύο αυτές διεργασίες δε θα θελουν ταυτόχρονα τους ίδιους πόρους τη CPU και έτσι μπορούμε να εκμεταλλευτούμε τους νεκρούς αυτούς χρόνους(π.χ cache misses). Στη προκειμένη όμως έχουμε threads στον ίδιο πυρήνα να δουλεύουν στο ίδιο block και επομένως το overhead μειώνει κατα πολύ την απόδοση.

Αναπαριστούμε επίσης σε ένα συνολικό plot τα execution times:



Παρατηρούμε πως βέλτιστη επίδοση έχουμε πάντα στους δύο πυρήνες ανεξαρτήτως των παραμέτρων N και B. Αυτό οφείλεται στο οτι μπορούμε να παραλληλοποιήσουμε μεχρι δυο αναδρομικές κλήσεις την φορά, άρα παραπάνω πυρήνες δεν θα δώσουν περαιτέρω επιτάχυνση καθώς δεν υπάρχουν tasks να τους αξιοποιήσουν.

Επίσης παρατηρούμε πως στα μικρά N το B δεν έχει σημαντικό ρόλο σε αντίθεση με το N = 4096 όπου το B=128 είναι εμφανώς καλύτερο.