Συστήματα Παράλληλης Επεξεργασίας

Άσκηση 2

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

Ομάδα	**parlab04**
Θεόδωρος Αράπης	el18028
Εμμανουήλ Βλάσσης	el18086
Παναγιώτης Παπαδέας	el18039

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Shared Clusters

Ζητούμενα

1. Ζητείται η παραλληλοποίηση του αλγορίθμου K-means με χρήση των απαραίτητων εντολών συγχρονισμού της βιβλιοθήκης OpenMP, με σκοπό την αποδοτικότερη και ορθότερη διαμοίραση των δεδομένων μεταξύ των threads. Στη συνέχεια, ζητείται η πραγματοποίηση μετρήσεων για το configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} και threads = {1, 2,4, 8, 16, 32, 64} στο μηχάνημα sandman καθώς και τα διαγράμματα barplot χρόνου εκτέλεσης και το αντίστοιχο speedup plot.
2. Στο ζητούμενο αυτό καλούμαστε να αξιοποιήσουμε τη μεταβλητή περιβάλλοντος *GOMP_CPU_AFFINITY,* να επαναλάβουμε τις μετρήσεις και να συγκρίνουμε τα αποτελέσματα με προηγουμένως.

Υλοποίηση

1)

Ο αλγόριθμος αυτός λειτουργεί ως εξής:

- Για κάθε αντικείμενο, βρίσκει αρχικά το cluster που βρίσκεται πιο κοντά του (συγκρίνοντας τις ευκλείδειες αποστάσεις του αντικειμένου με κάθε cluster), αποθηκεύει το index του cluster αυτού, αυξάνει την μεταβλητή δέλτα για το convergence (αν διαφέρει με το προηγούμενο index), αυξάνει το μέγεθος του cluster και προσθέτει τις συντεταγμένες του αντικειμένου στον πίνακα με τα αθροίσματα *newClusters* (για το αντίστοιχο cluster).
- Ύστερα, για κάθε cluster, ενημερώνει τον πίνακα *clusters* (που περιέχει τις συντεταγμένες των κέντρων όλων των clusters) με τον μέσο όρο των συντεταγμένων των αντικειμένων που ανήκουν σε αυτό (διαιρώντας τα αθροίσματα στον πίνακα *newClusters* με το πλήθος των αντικειμένων που ανήκουν εκεί) και υπολογίζει το δέλτα (διαιρώντας το με το πλήθος των αντικειμένων).

Οι παραπάνω διαδικασίες επαναλαμβάνονται ωσότου η τιμή του δέλτα περάσει ένα ορισμένο κατώφλι σε λιγότερο από δέκα κύκλους.

Μελετώντας τον κώδικα και τον αλγόριθμο, επιλέξαμε να παραλληλοποιήσουμε το *for loop* που εκτελεί τις λειτουργίες που περιγράφηκαν στο πρώτο bullet point παραπάνω, καθώς δεν παρουσιάζουν εξαρτήσεις δεδομένων, με χρήση του directive ***#pragma omp parallel for***.

Διατηρούμε ως διαμοιραζόμενες (***shared***) μεταβλητές:

- τον πίνακα με τα μεγέθη των clusters (*newClustersSize*),
- τον πίνακα με τα αθροίσματα των συντεταγμένων που ανήκουν στο ίδιο cluster (*newClusters*),
- το πλήθος των αντικειμένων (*numObjs*),
- το πλήθος των συντεταγμένων που ορίζουν την διάσταση του αντικειμένου (*numCoords*),
- τον πίνακα που αποθηκεύει τον index του cluster στο οποίο διατάχτηκε το αντικείμενο στην προηγούμενη επανάληψη (*membership*),
- τον πίνακα με τις συντεταγμένες των αντικειμένων (*objects*),
- το δέλτα (*delta*) και
- τον πίνακα με τις τελικές συντεταγμένες των cluster (*clusters*)

Διατηρούμε ως ***private*** τις μεταβλητές:

- (*i*), δείκτης επαναληπτικού βρόγχου
- (*j*), δείκτης φωλιασμένου επαναληπτικού βρόγχου
- (*index*), προσωρινή μεταβλητή του νέου δείκτη κοντινότερου γείτονα

Παρατηρούμε, όμως, ότι με την παραλληλοποίηση αυτή διεγείρονται ορισμένα race conditions.

Αυτά είναι τα εξής:

- Η αύξηση της τιμής *newClustersSize[index]* (αύξηση μεγέθους cluster)
- Η επεξεργασία του της τιμής *newClusters[index*numCoords + j]* (πρόσθεση της συντεταγμένης *j* του αντικειμένου *i* στη ήδη υπάρχουσα τιμή του πίνακα)

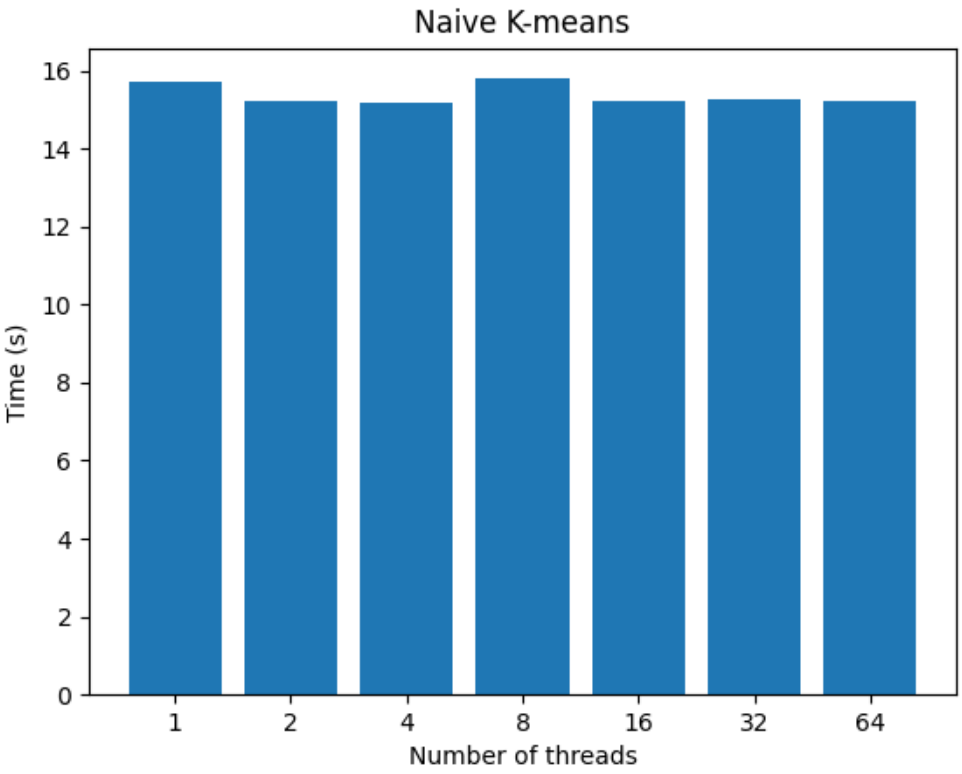
(Φαίνεται όμως ότι και η αύξηση του δέλτα εμφανίζει race conditions, παρόλο που δεν υποδεικνύεται)

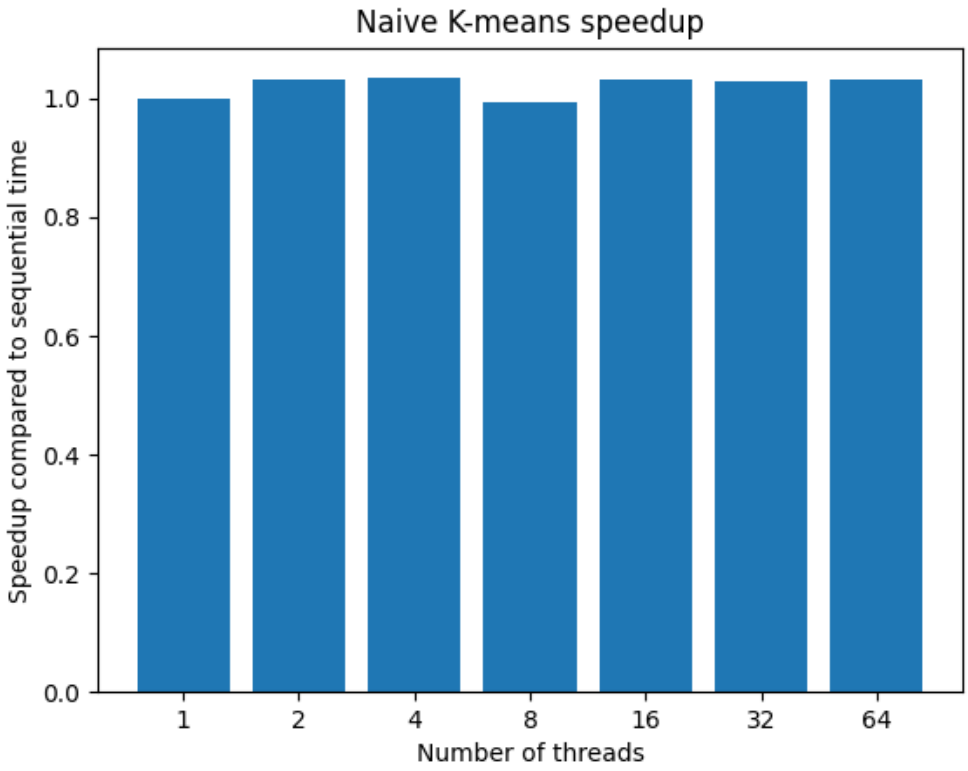
Συνεπώς, τις παραπάνω περιπτώσεις race condition θα τις αντιμετωπίσουμε με χρήση του directive ***#pragma omp atomic***, το οποίο παρέχει ατομική πρόσβαση σε μία θέση μνήμης για κάθε thread.

Η υλοποίηση του κώδικα φαίνεται ακολούθως:

```
90          #pragma omp parallel for shared (newClusterSize, newClusters, numObjs, numCoords, membership, objects, delta, clusters) private (i, j, index)
91          for (i=0; i<numObjs; i++) {
92              // find the array index of nearest cluster center
93              index = find_nearest_cluster(numClusters, numCoords, &objects[i*numCoords], clusters);
94
95              // if membership changes, increase delta by 1
96              if (membership[i] != index){
97               //   #pragma omp atomic
98                  delta += 1.0;
99               }
100             // assign the membership to object i
101             membership[i] = index;
102
103             // update new cluster centers : sum of objects located within
104             /*
105              * TODO: enforce atomic access to shared "newClusterSize" array
106              */
107             #pragma omp atomic
108             newClusterSize[index]++;
109             for (j=0; j<numCoords; j++)
110                 /*
111                  * TODO: enforce atomic access to shared "newClusters" array
112                  */
113                 #pragma omp atomic
114                 newClusters[index*numCoords + j] += objects[i*numCoords + j];
115         }
```

Τα αποτελέσματα συνοψίζονται στα παρακάτω γραφήματα:


Naive K-means
Time (s)
0
2
4
6
8
10
12
14
16
1
2
4
8
16
32
64
Number of threads


Naive K-means speedup
Speedup compared to sequential time
0.0
0.2
0.4
0.6
0.8
1.0
1
2
4
8
16
32
64
Number of threads

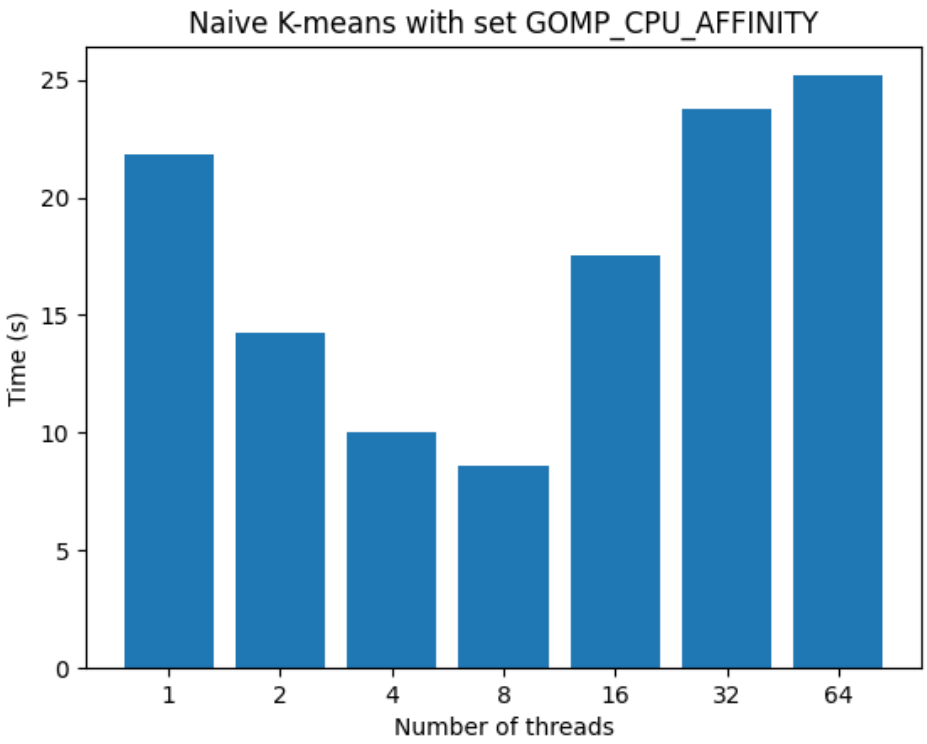
Παρατηρούμε ότι δεν έχουμε χρονική επιτάχυνση σε σχέση με τη σειριακή εκτελεση του προγράμματος.

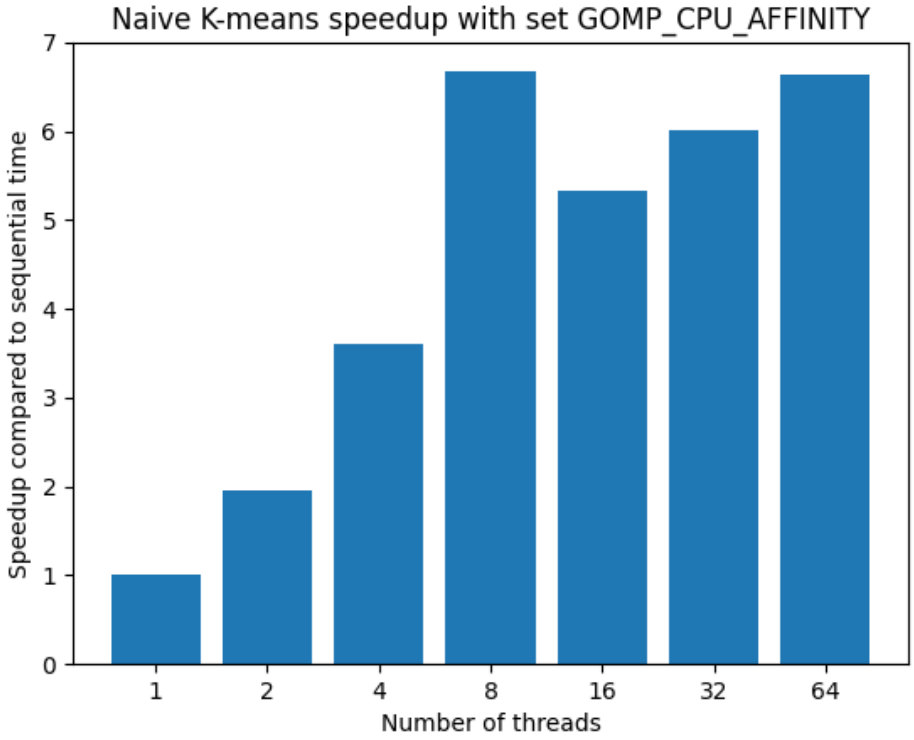
2)

Θα αξιοποιήσουμε τώρα τη μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY. Με τη μεταβλητή αυτή μπορούμε να κάνουμε bind συγκεκριμένα threads σε συγκεκριμένους πυρήνες. Στην περίπτωσή μας, το μηχάνημα sandman έχει 32 πυρήνες και κάθε πυρήνας μπορεί να επεξεργαστεί μέχρι δύο threads παράλληλα. Όταν τρέχει ένα πρόγραμμα, τα threads που δημιουργούνται μπορούν να ανατεθούν σε οποιονδήποτε επεξεργαστή. Συνεπώς, όταν τρέχει το πρόγραμμα που υλοποιεί τον k-means παράλληλα, τα δημιουργούμενα threads μπορούν να τοποθετηθούν σε οποιοδήποτε πυρήνα, όπως, για παράδειγμα, να ανατεθούν δύο threads στον ίδιο πυρήνα, με αποτέλεσμα να μην κατανέμεται σωστά το παράλληλο φορτίο ώστε να αξιοποιήσει μέγιστα την αρχιτεκτονική πυρήνων του sandman. Οπότε, στο script, το οποίο υποβάλουμε στο sandman, προσθέτουμε την εντολή “export GOMP_CPU_AFFINITY="0-31"” η οποία τοποθετεί τα threads στους πυρήνες με round-robin λογική.

```
module load openmp
cd /home/parallel/parlab04/a2/2.1
export SIZE=256
export COORDS=16
export CLUSTERS=16
export LOOPS=10
export GOMP_CPU_AFFINITY="0-31"
```

Τα αποτελέσματα που λάβαμε είναι:


Naive K-means with set GOMP_CPU_AFFINITY
Time (s)
0
5
10
15
20
25
1
2
4
8
16
32
64
Number of threads


Naive K-means speedup with set GOMP_CPU_AFFINITY
Speedup compared to sequential time
0
1
2
3
4
5
6
7
1
2
4
8
16
32
64
Number of threads

Εδώ παρατηρούμε ότι μέχρι τους 8 πυρήνες πετυχαίνουμε επιτάχυνση του προγράμματος, ενώ για περισσότερους από 8 πυρήνες η επίδοση μειώνεται. Αυτό μπορεί να εξηγηθεί από το γεγονός ότι ο χρόνος για τη δημιουργία και την επικοινωνία μεταξύ των threads υπερτερεί του χρόνου που κερδίζουμε από την περαιτέρω παραλληλοποίηση. Συγκεκριμένα, κατά την εκτέλεση των εντολών που βρίσκονται σε #pragma omp atomic, θα πρέπει όλα τα άλλα threads να περιμένουν την ολοκλήρωση ενός thread για να ανανεώσουν τις δικές τους τιμές.

Copied Clusters and Reduce

Ζητούμενα

1. Ζητείται η παραλληλοποίηση του αλγορίθμου K-means χρησιμοποιώντας τώρα τοπικούς (αντιγραμμένους) πίνακες για κάθε νήμα και συνδυάζοντας τα αποτελέσματα με reduction στον πίνακα newClusters. Στη συνέχεια, επαναλαμβάνουμε τις ίδιες μετρήσεις με πριν και συγκρίνουμε τα αποτελέσματα.
2. Στο ερώτημα αυτό θα συγκρίνουμε τα configurations {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} και {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}. Ύστερα θα μελετήσουμε το φαινόμενο false sharing που συναντάται και θα επιχειρήσουμε να το επιλύσουμε.

Υλοποίηση

1)

Ο παραλληλοποιημένος αλγόριθμος με χρήση αντιγραμμένων πινάκων και reduction λειτουργεί ως εξής:

Ορίζουμε και έναν πίνακα local_newClusters[N][numClusters · numCoords], ο οποίος περιέχει N (πλήθος threads) πίνακες newClusters, έναν για κάθε thread ώστε να επεξεργάζεται τοπική περιοχή μνήμης και να αποφύγουμε τα race conditions. Όμοια, δημιουργούμε και έναν πίνακα local_newClustersSize[N][numClusters]. Ο αλγόριθμος επαναλαμβάνεται όσο το δέλτα δεν ξεπερνάει ένα ορισμένο κατώφλι ή όταν ολοκληρωθούν οι κύκλοι που ορίζονται ως κατώφλι. Τα βήματα είναι παρόμοια με προηγουμένως, μόνο που αυτήν την φορά αντί για τους πίνακες newClusters και newClustersSize επεξεργαζόμαστε τους πίνακες local_newClusters και

local_newClustersSize. Ξεκινώντας την επανάληψη του αλγορίθμου, αρχικοποιούμε τους προαναφερθέντες πίνακες.

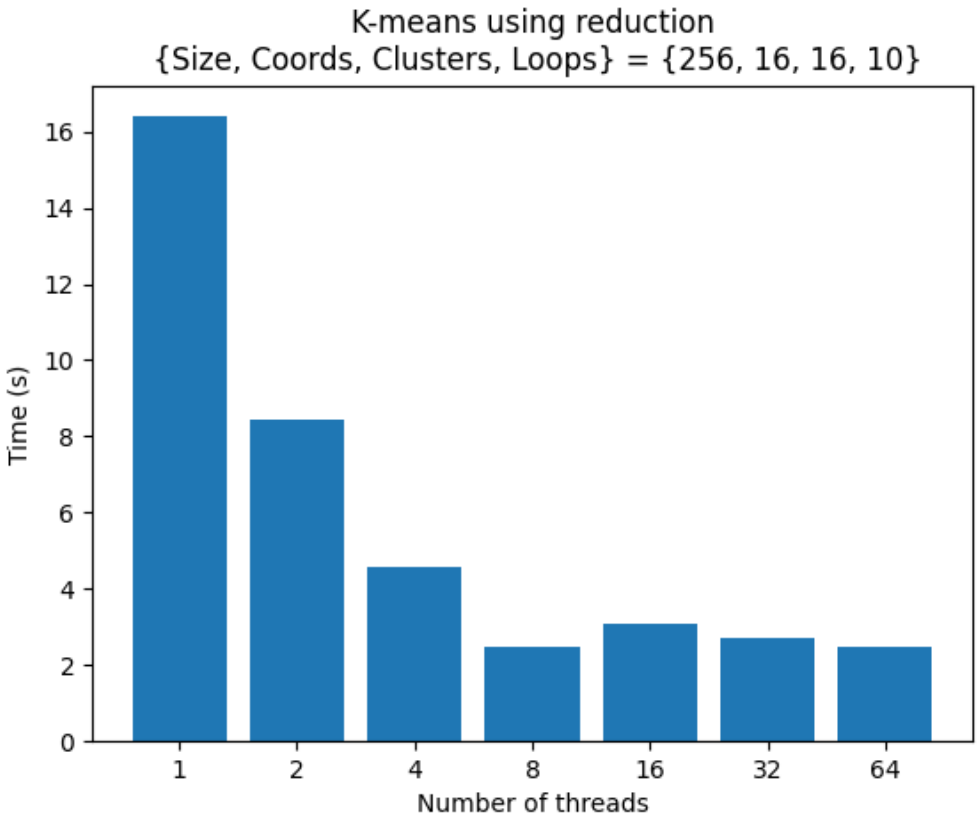
```
/*
 * TODO: Initiliaze local cluster data to zero (separate for each thread)
 */
for (k=0; k<nthreads; k++){
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++)
            local_newClusters[k][i*numCoords + j] = 0.0;
    local_newClusterSize[k][i] = 0;
    }
}
```

Στη συνέχεια, χρησιμοποιούμε το directive ***#pragma omp parallel,*** προκειμένου να ορίσουμε την παράλληλη περιοχή (φαίνεται στην εικόνα που ακολουθεί) και ύστερα θα αξιοποιήσουμε πάλι το ***#pragma omp for*** ώστε να παραλληλοποιήσουμε το ίδιο for loop με προηγουμένως, διατηρώντας τώρα ως διαμοιραζόμενες μεταβλητές τους νέους local πίνακες. Όπως εξηγήσαμε προηγουμένως, δεν έχουμε race conditions άρα δεν χρειαζόμαστε ατομική πρόσβαση στα δεδομένα. Μετά την ολοκλήρωση του parallel for loop, το οποίο πραγματοποιεί συγχρονισμό (implicit barrier), γίνεται το reduction των αθροισμάτων που υπολόγισε κάθε νήμα σε ένα συνολικό άθροισμα για κάθε cluster, και οι τελικές τιμές αποθηκεύονται σε έναν κοινό πίνακα. Επιλέξαμε αυτός ο πίνακας, αντί του newClusters που παρέχεται, να είναι ο local_newClusters[0], προκειμένου να αποφύγουμε τη χρήση έξτρα μεταβλητών. Τέλος, το reduction αυτό θέλουμε να εκτελεστεί μόνο από ένα thread, γι' αυτό θα αξιοποιήσουμε το directive ***#pragma omp master***, ώστε το αρχικό thread να υπολογίσει τα τελικά αθροίσματα. Από εκεί και πέρα η υλοποίηση του κώδικα είναι ίδια με πριν.

Το παραλληλοποιημένο κομμάτι του κώδικα είναι:

```
#pragma omp parallel shared (local_newClusterSize, local_newClusters, clusters, delta, numObjs, numCoords, membership, objects) private (i, j, k, index)
{
    #pragma omp for
    for (i=0; i<numObjs; i++)
    {
        k = omp_get_thread_num();

        // 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 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
         */
        //newClusterSize[index]++;
        local_newClusterSize[k][index]++;
        for (j=0; j<numCoords; j++)
            //newClusters[index*numCoords + j] += objects[i*numCoords + j];
            local_newClusters[k][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 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];
            }
        }

        // average the sum and replace old cluster centers with newClusters
        for (i=0; i<numClusters; i++) {
            for (j=0; j<numCoords; j++) {
                //k = omp_get_thread_num();
                if (local_newClusterSize[0][i] > 1)
                    clusters[i*numCoords + j] = local_newClusters[0][i*numCoords + j] / local_newClusterSize[0][i];
            }
        }
    }
}
```

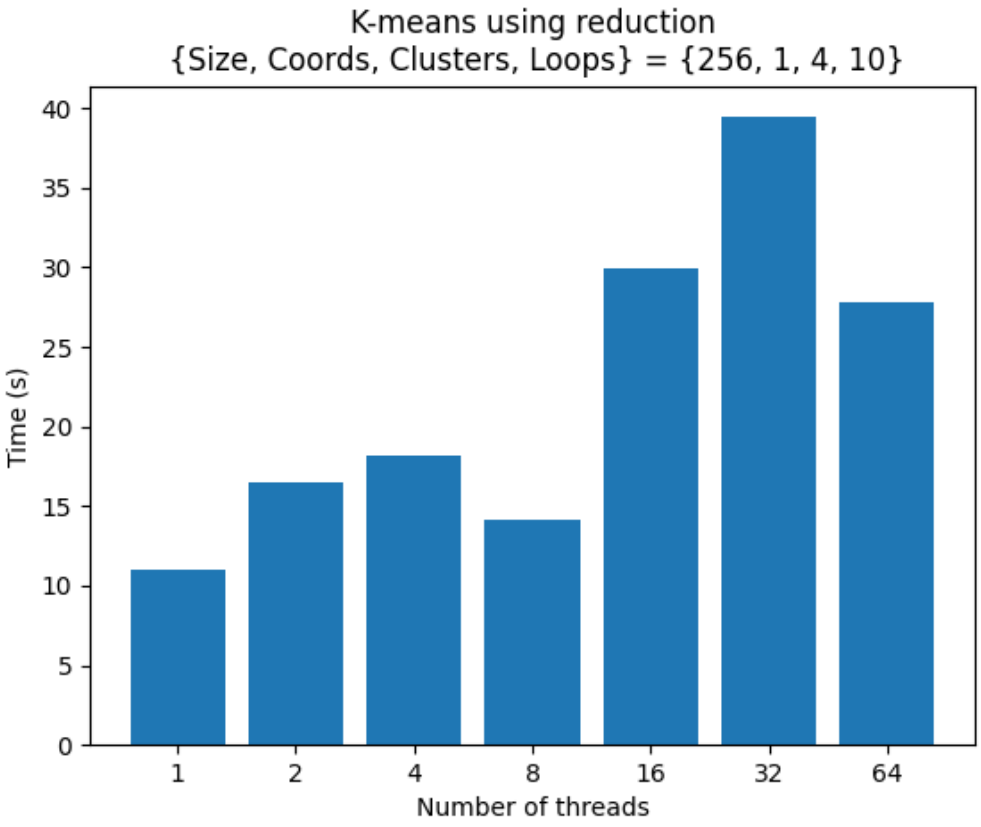
Θα πραγματοποιήσουμε τώρα τις ζητούμενες μετρήσεις στο μηχάνημα sandman εφαρμόζοντας το configuration: {Size, Coords, Clusters, Loops} = {256, 16, 16, 10}, για τα ακόλουθα πλήθη threads: {1, 2, 4, 8, 16, 32, 64}. Τα αποτελέσματα συνοψίζονται στο παρακάτω γράφημα:



Για 2, 4, και 8 πυρήνες έχουμε σχεδόν γραμμικό speedup, ενώ για περισσότερους πυρήνες η επίδοση δεν βελτιώνεται. Αυτό μπορεί να εξηγηθεί από το γεγονός ότι πλέον το μεγαλύτερο μέρος του χρόνου εκτέλεσης γίνεται μέσα στο #pragma omp master που τρέχει το master thread, το οποίο δεν μπορεί να παραλληλοποιηθεί (νόμος του Amdahl).

2)

Στο σημείο αυτό θα πραγματοποιήσουμε αντίστοιχες μετρήσεις με πριν για το configuration: {Size, Coords, Clusters, Loops} = {256, 1, 4, 10} και θα συγκρίνουμε το scalability με το προηγούμενο configuration (ζητούμενο 1):



Σε αυτό το configuration παρατηρούμε ότι ο ταχύτερος χρόνος εκτέλεσης προκύπτει για μόνο 1 πυρήνα. Αντιθέτως, για παραπάνω από 8 πυρήνες ο χρόνος εκτέλεσης αυξάνεται σημαντικά.

Αυτό συμβαίνει διότι στην περίπτωση του εξεταζόμενου configuration έχουμε έντονο φαινόμενο false sharing. Πιο ειδικά, οι πίνακες local_newClustersSize και local_newClusters, στους οποίους γίνονται εγγραφές στην παράλληλη περιοχή, είναι τύπου int (4 bytes) και float (4 bytes) και τα μεγέθη τους είναι N×Clusters και N×Clusters ×Coords, αντίστοιχα, όπου N το πλήθος των threads. Οι πίνακες αυτοί είναι αποθηκευμένοι σειριακά στην μνήμη και επομένως (υποθέτοντας ότι η cache line των επεξεργαστών έχει αρκετά μεγάλο μέγεθος, π.χ 32 ή 64 bytes), τα τμήματα τους που έχουν ανατεθεί σε κοντινά, ως προς το id, thread για επεξεργασία, καταλήγουν στην ίδια cache line (την οποία αντιγράφουν οι επεξεργαστές από την κύρια μνήμη στην τοπική τους cache), αφού κάθε τμήμα έχει μέγεθος 4 (bytes)×Clusters = 4×4 bytes και 4 (bytes)× Clusters×Coords = 4×4×16 bytes αντίστοιχα. Συνεπώς, όταν ένα thread επεξεργάζεται το τμήμα του, τότε όλα τα υπόλοιπα threads των οποίων το τμήμα τους βρίσκεται στην ίδια cache line με το προηγούμενο, θα αναγκαστούν να ξαναφορτώσουν το cache line αυτό στην cache, καθυστερώντας έτσι την εκτέλεση του παράλληλου προγράμματος.

```
int paddingBytes = 128;
int paddingElements = paddingBytes/sizeof(int);
int paddingElements2 = paddingBytes/sizeof(float);
int padded_1 = numClusters + (paddingElements-numClusters%paddingElements);
int padded_2 = (numClusters*numCoords)+(paddingElements2-(numClusters*numCoords)%paddingElements2);
```

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:

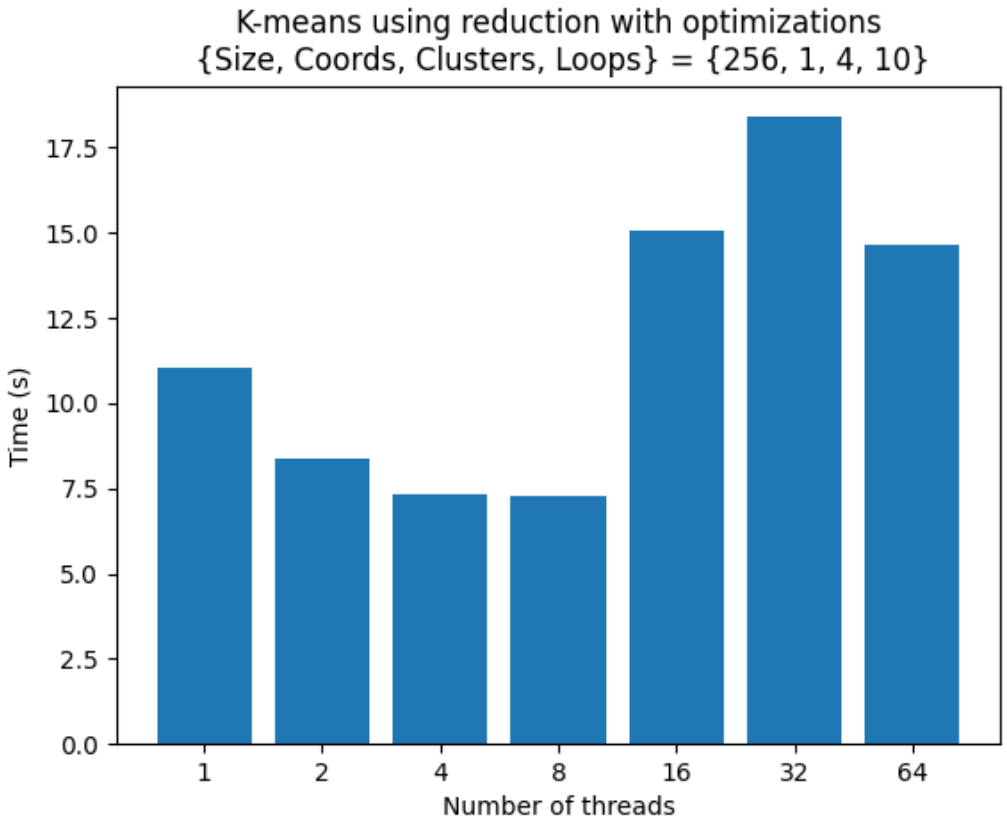
```
/*
 * Hint for false-sharing
 * This is noticed when numCoords is low (and neighboring local_newClusters exist close to each other).
 * Allocate local cluster data with a "first-touch" policy.
 */
// Initialize local (per-thread) arrays (and later collect result on global arrays)
for (k=0; k<nthreads; k++)
{
    local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(padded_1, sizeof(**local_newClusterSize));
    local_newClusters[k] = (typeof(*local_newClusters)) calloc(padded_2, sizeof(**local_newClusters));
}
```

Επιπρόσθετα, όπως γνωρίζουμε, τα συστήματα NUMA υλοποιούν το first-touch policy, το οποίο κάνει allocate τη σελίδα με τα δεδομένα που ζητείται στη κοντινότερη μνήμη προς το thread που «αγγίζει» τα συγκεκριμένα δεδομένα. Στην περίπτωσή μας αρχικοποιούμε (allocate) τους πίνακες local από ένα μόνο thread. Ως εκ τούτου, τα υπόλοιπα threads θα πρέπει να επικοινωνούν με εκείνη την cache στην οποία τα τοποθέτησε το master thread, με αποτέλεσμα τα δεδομένα να διανύουν μεγάλες αποστάσεις (καθώς θα βρίσκονται σε άλλα nodes) και επομένως να επιφέρουν καθυστέρηση.

Προκειμένου να επιλύσουμε αυτό το πρόβλημα, θα παραλληλοποιήσουμε την αρχικοποίηση των πινάκων local_newClustersSize και local_newClusters με την ίδια λογική που παραλληλοποιήσαμε και τον αλγόριθμο, χρησιμοποιώντας το directive ***#pragma omp for schedule(static)***. Έτσι, κάθε thread θα κάνει «first-touch» μόνο το τμήμα των πινάκων που του αντιστοιχεί.

```
#pragma omp parallel shared (local_newClusterSize, local_newClu
{
    #pragma omp for schedule(static)
    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;
        }
    }
```

Αφού εφαρμόσαμε τις παραπάνω αλλαγές, επαναλάβαμε τις ίδιες μετρήσεις και λάβαμε τα παρακάτω αποτελέσματα:



Σε σύγκριση με το προηγούμενο πρόγραμμα, βλέπουμε ότι για παραπάνω από 1 πυρήνα έχουμε περίπου υποδιπλασιασμό του χρόνου εκτέλεσης. Βέβαια, για παραπάνω από 8 πυρήνες ο χρόνος εκτέλεσης πάλι αυξάνεται.

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

Ζητούμενα

Σε αυτό το κομμάτι της άσκησης θα επιχειρήσουμε να παραλληλοποιήσουμε την έκδοση του recursive αλγορίθμου Floyd-Warshall με χρήση OpenMP tasks και θα καταγράψουμε μετρήσεις για διάφορα μεγέθη πινάκων και πλήθη νημάτων, δημιουργώντας το αντίστοιχο barplot για κάθε μέγεθος αξιοποιώντας το μηχάνημα sandman.

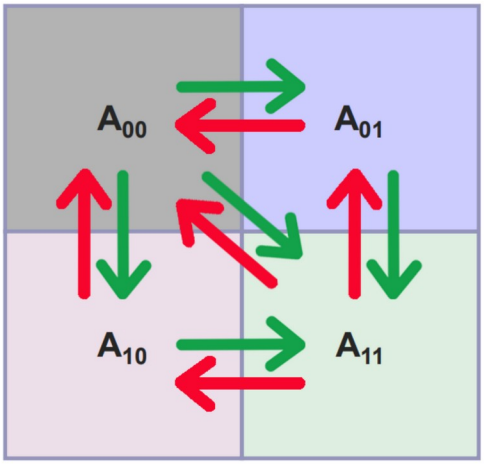
Συγκεκριμένα, τα μεγέθη πινάκων που θα εξετάσουμε είναι {1024x1024, 2048x2048, 4096x4096}, ενώ για το πλήθος των νημάτων είναι {1, 2, 4, 8, 16, 32, 64}.

Υλοποίηση

Μας δίνεται το αρχείο fw_sr.c που περιέχει την recursive έκδοση του αλγορίθμου Floyd-Warshall.

```
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);
    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);
    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);
    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);
}
```

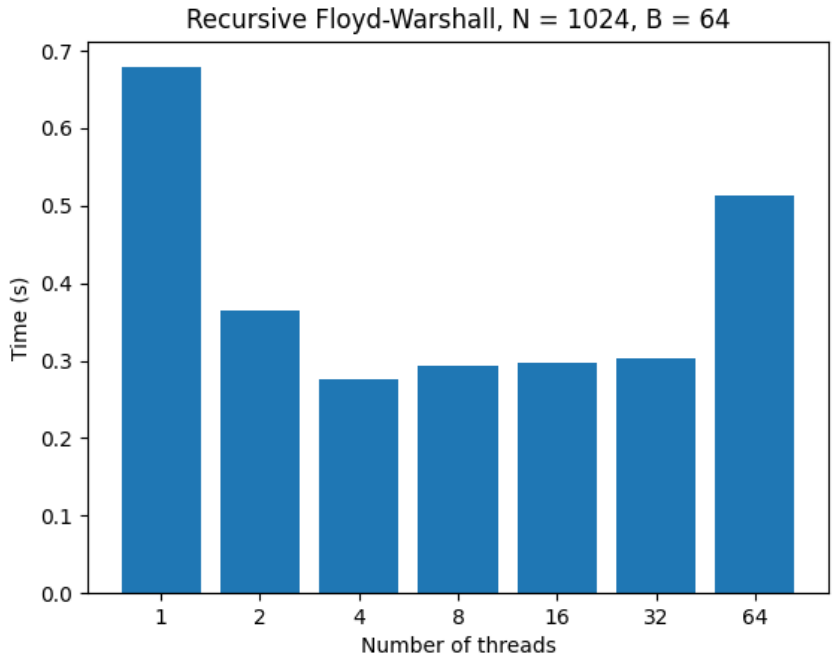
Η υλοποίηση αυτή εμφανίζει συγκεκριμένα υπολογιστικά dependencies, με αποτέλεσμα να χρειάζεται συγκεκριμένη σειρά για τα recursive calls. Πιο ειδικά, παρατηρούμε ότι τα recursive calls επενεργούν σε 4 περιοχές του αρχικού πίνακα γειτνίασης A, διαχωρίζοντάς των έτσι 4 ίσους υποπίνακες όπως φαίνεται στην εικόνα.

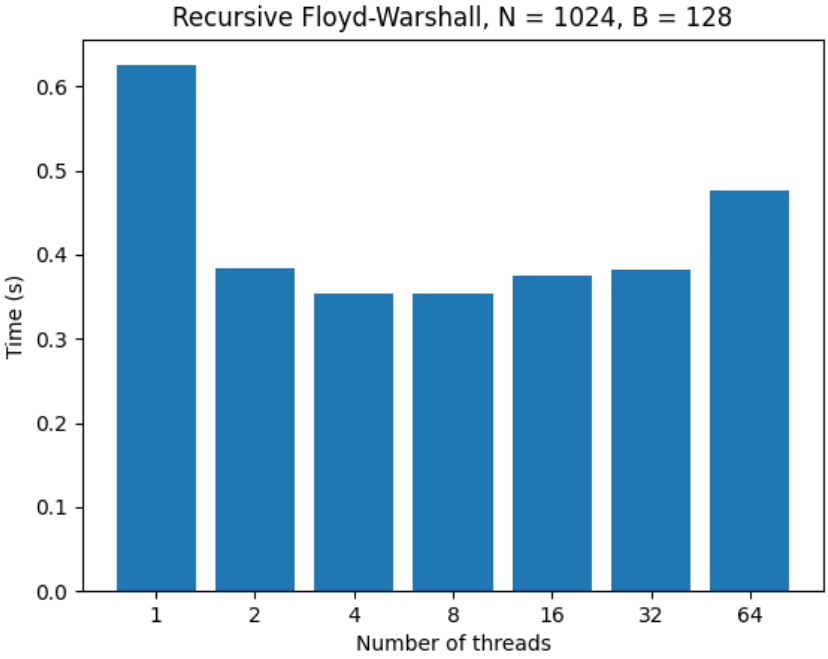


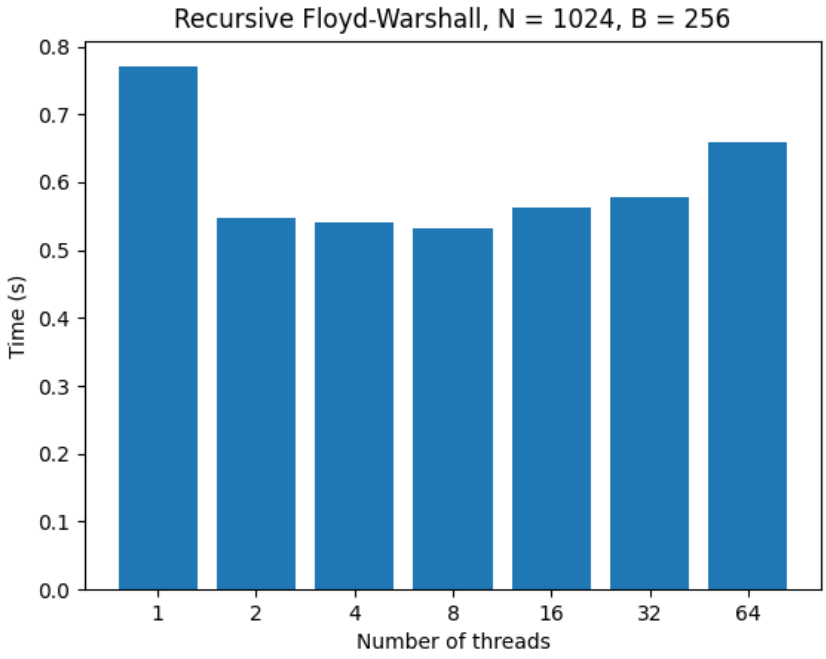
Τα 4 πρώτα recursive calls επενεργούν στον αρχικό πίνακα γειτνίασης A, από τον υποπίνακα A_{00} προς τον υποπίνακα A_{11} (πράσινα βέλη) ενώ τα 4 τελευταία επενεργούν με την αντίστροφη σειρά (κόκκινα βέλη). Ως εκ τούτου, παρατηρούμε ότι οι υποπίνακες A_{01} και A_{10} παρουσιάζουν ανεξαρτησία και άρα τα recursive calls που επενεργούν πάνω τους μπορούν να παραλληλοποιηθούν. Συνεπώς, θα κάνουμε χρήση tasks, ώστε να παράξουμε 2 ζεύγη tasks, καθένα από τα οποία θα εκτελεί υπολογισμούς στους διαγώνιους υποπίνακες A_{01} και A_{10}. Το δεύτερο και τρίτο recursive call αποτελούν το πρώτο ζεύγος tasks, ενώ το έκτο και έβδομο recursive call το δεύτερο ζεύγος tasks. Τέλος, θα πρέπει να συγχρονίσουμε τα task πριν προχωρήσει το πρόγραμμα στην εκτέλεση του υπόλοιπου κώδικα. Η υλοποίησή μας φαίνεται παρακάτω:

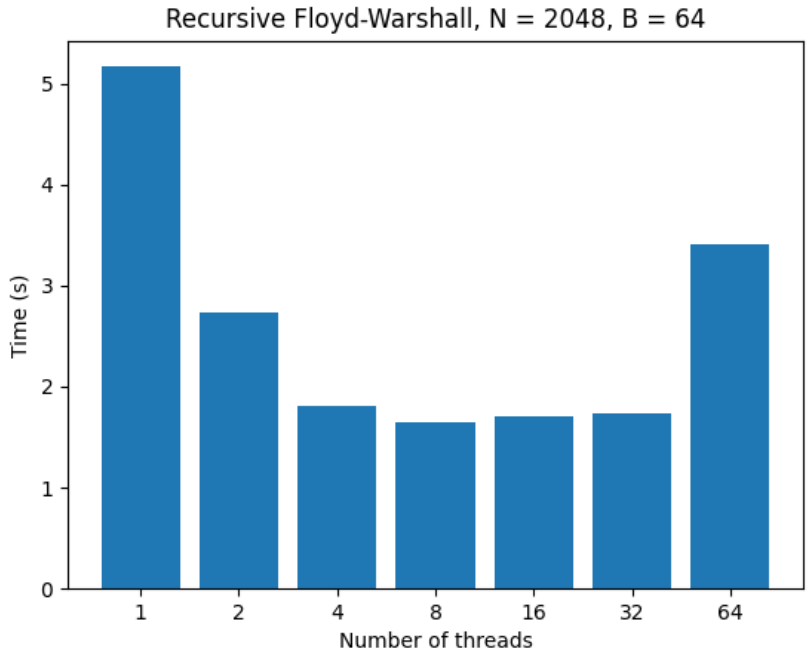
```
if(myN<=bsize)
    for(k=0; k<myN; k++)
        for(i=0; i<myN; i++)
            for(j=0; j<myN; j++)
                A[arow+i][acol+j]=min(A[arow+i][acol+j], B[brow+i][bcol+k]+C[crow+k][ccol+j]);
else
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            FW_SR(A,arow,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

            #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 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);
        }
    }
}
```

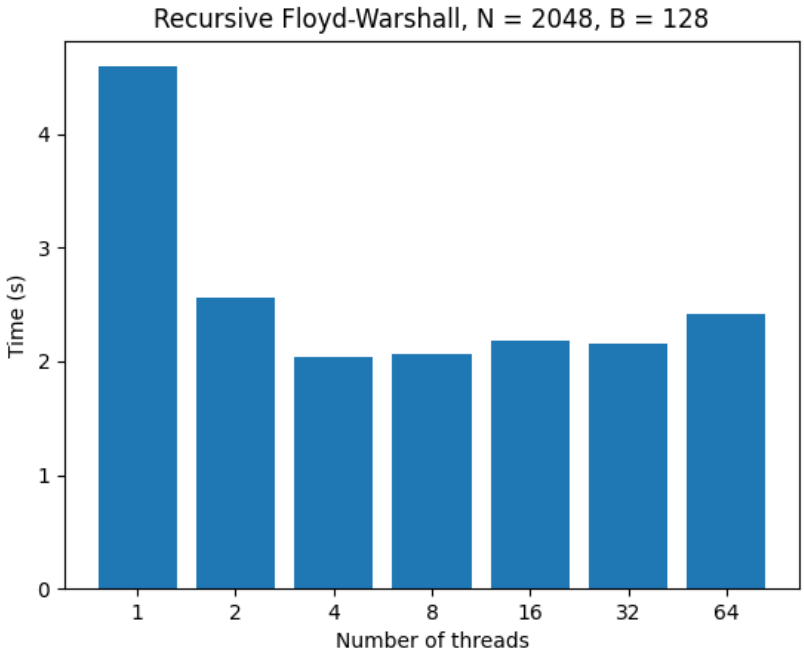
Στη συνέχεια πραγματοποιούμε μετρήσεις για τις τιμές των παραμέτρων που περιγράφηκαν στα ζητούμενα και θα εξετάσουμε τις αντίστοιχες γραφικές παραστάσεις:

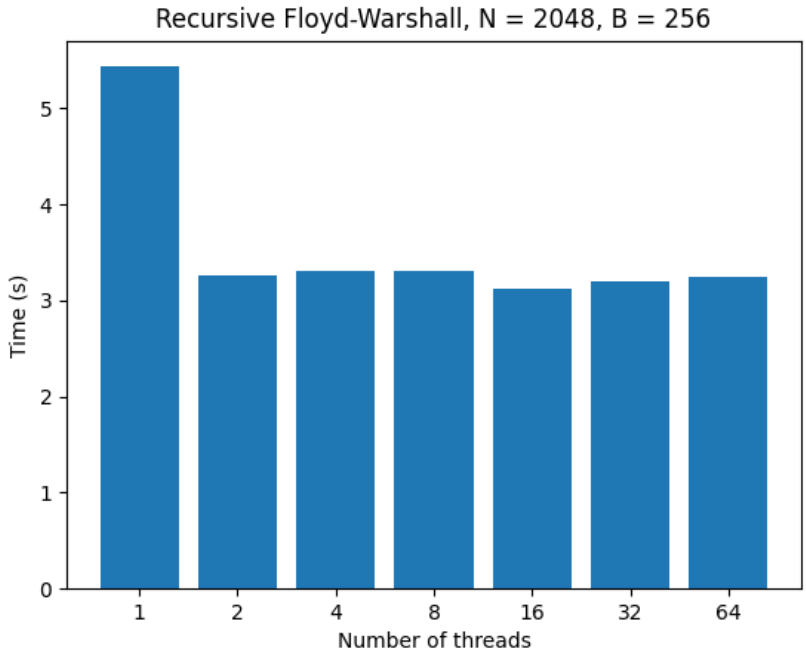


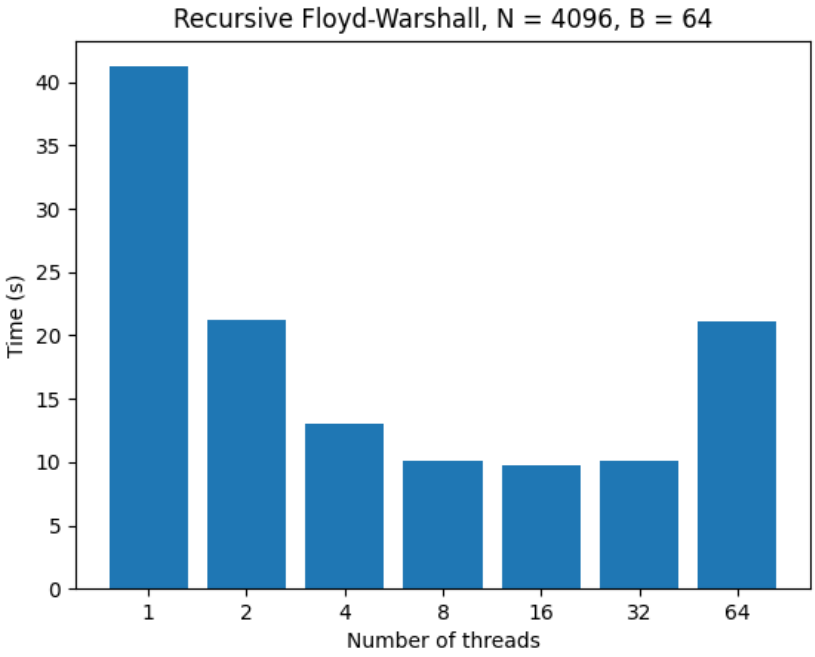

Recursive Floyd-Warshall, N = 1024, B = 128
Time (s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
1
2
4
8
16
32
64
Number of threads

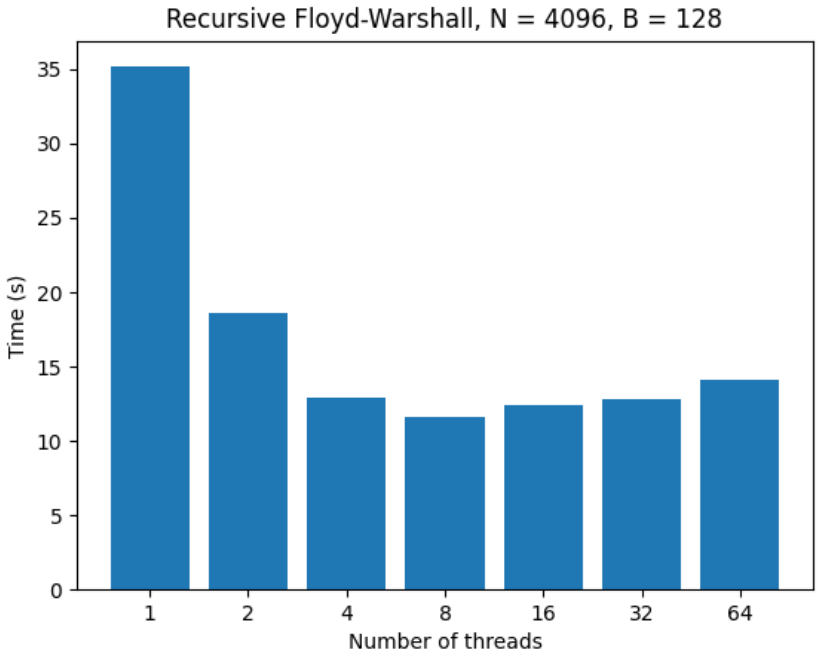

Recursive Floyd-Warshall, N = 1024, B = 256
Time (s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
4
8
16
32
64
Number of threads

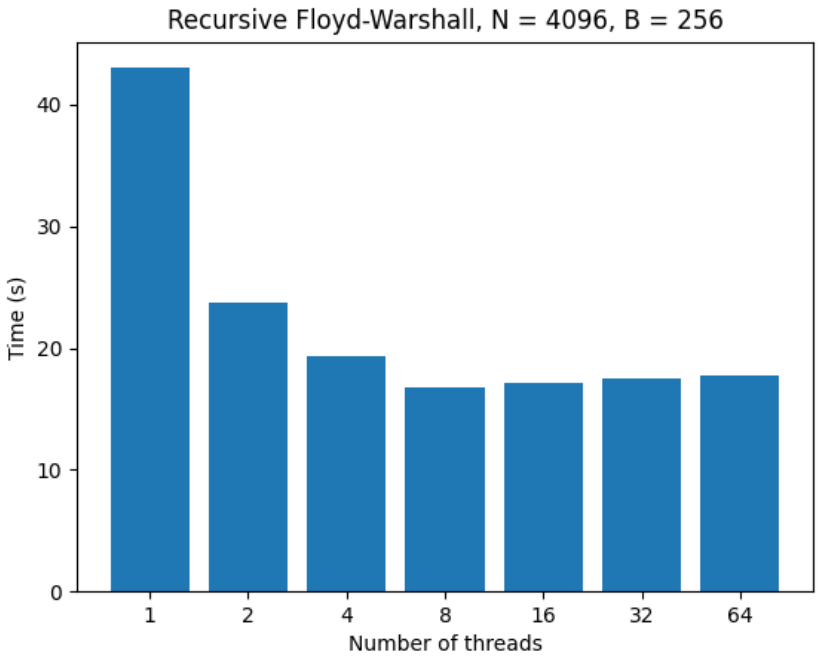

Recursive Floyd-Warshall, N = 2048, B = 64
Time (s)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads


Recursive Floyd-Warshall, N = 2048, B = 128
Time (s)
0
1
2
3
4
1
2
4
8
16
32
64
Number of threads


Recursive Floyd-Warshall, N = 2048, B = 256
Time (s)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads


Recursive Floyd-Warshall, N = 4096, B = 64
Time (s)
0
5
10
15
20
25
30
35
40
1
2
4
8
16
32
64
Number of threads





Σχόλια - Παρατηρήσεις

Αρχικά, βλέπουμε ότι η τιμή του B δεν επηρεάζει σημαντικά τον χρόνο εκτέλεσης σε όλα τα μεγέθη πίνακα, οπότε οι υπόλοιπες παρατηρήσεις αφορούν όλα τα B. Το βέλτιστο B από τα 3 που επιλέξαμε είναι το B = 128. Η αύξηση των διαθέσιμων πυρήνων από 1 σε 2

βελτιώνει σε όλες τις περιπτώσεις τον χρόνο εκτέλεσης. Το speedup με την αύξηση από 1 σε 2 πυρήνες είναι σχεδόν 2, ενώ για N = 1024 παρατηρούμε μικρότερο speedup από ότι για τα μεγαλύτερα N.

Για την αύξηση από 2 σε 4 πυρήνες και από 4 σε 8 πυρήνες δεν έχουμε σημαντικό speedup. Μάλιστα, για πάνω από 8 πυρήνες ο χρόνος εκτέλεσης αυξάνεται, ενώ για B = 64 και 64 threads έχουμε σχεδόν διπλασιασμό του χρόνου εκτέλεσης. Αυτό εξηγείται επειδή για B = 64, στο χαμηλότερο επίπεδο recursion έχουμε 16 blocks για N = 1024, 32 blocks για N = 2048, και 64 blocks για N = 2048, συνεπώς περαιτέρω αύξηση των πυρήνων δεν βελτιώνει την επίδοση.

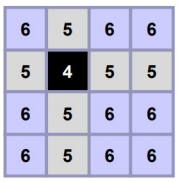
Προαιρετικά

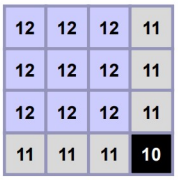
Θα επιχειρήσουμε να παραλληλοποιήσουμε τώρα την έκδοση tiled του αλγορίθμου Floyd-Warshall, να την εκτελέσουμε με τις ίδιες τιμές πινάκων και νημάτων με προηγουμένως και να συγκρίνουμε τα αποτελέσματα με την παραλληλοποιημένη recursive έκδοση του αλγορίθμου.

Η έκδοση tiled λειτουργεί ως εξής:

Ο αρχικός πίνακας γειτνίασης χωρίζεται σε tiles μεγέθους B. Σε κάθε επανάληψη k, ο αλγόριθμος ενημερώνει το k-οστό tile, κατά μήκος της διαγωνίου του αρχικού πίνακα, στη συνέχεια ενημερώνει όλα τα tiles που βρίσκονται στην ίδια γραμμή και αριστερά του, στην ίδια γραμμή και δεξιά του, στην ίδια στήλη και από πάνω του και στην ίδια στήλη και από κάτω του. Τέλος, ενημερώνονται όλα τα υπόλοιπα στοιχεία του πίνακα.

```
for(k=0;k<N;k+=B)
{
    FW(A,k,k,k,B);

    for(i=0; i<k; i+=B)
        FW(A,k,i,k,B);

    for(i=k+B; i<N; i+=B)
        FW(A,k,i,k,B);

    for(j=0; j<k; j+=B)
        FW(A,k,k,j,B);

    for(j=k+B; j<N; j+=B)
        FW(A,k,k,j,B);

    for(i=0; i<k; i+=B)
        for(j=0; j<k; j+=B)
            FW(A,k,i,j,B);

    for(i=0; i<k; i+=B)
        for(j=k+B; j<N; j+=B)
            FW(A,k,i,j,B);

    for(i=k+B; i<N; i+=B)
        for(j=0; j<k; j+=B)
            FW(A,k,i,j,B);

    for(i=k+B; i<N; i+=B)
        for(j=k+B; j<N; j+=B)
            FW(A,k,i,j,B);
}
```

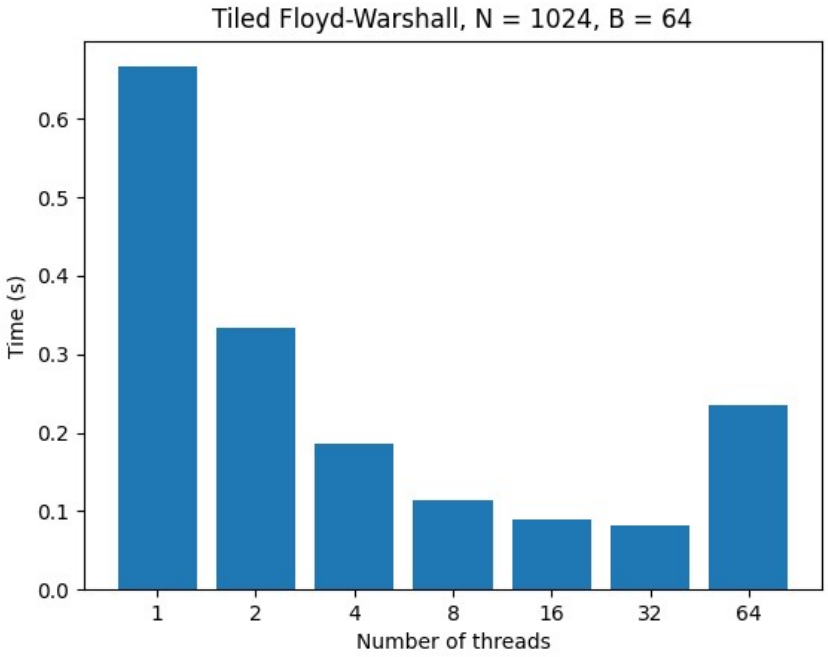


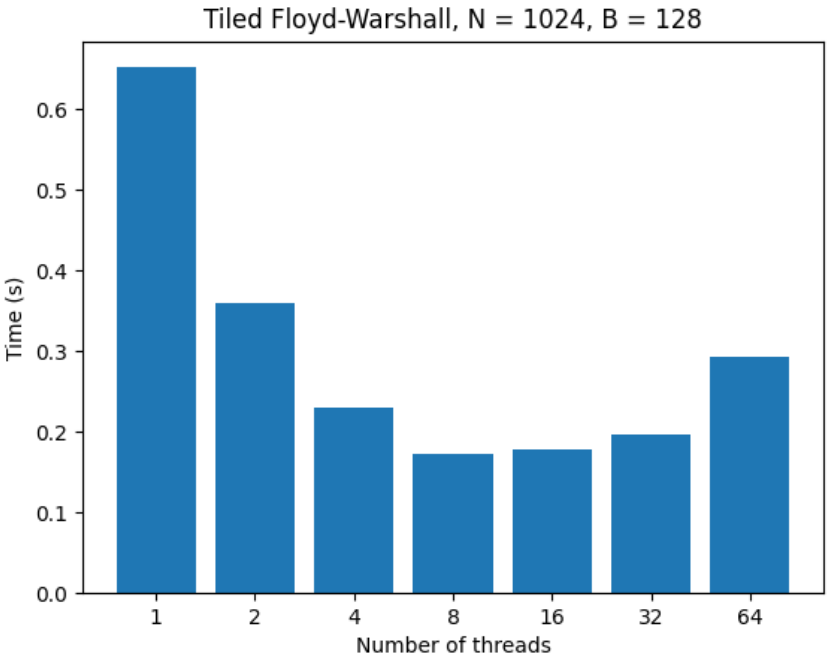


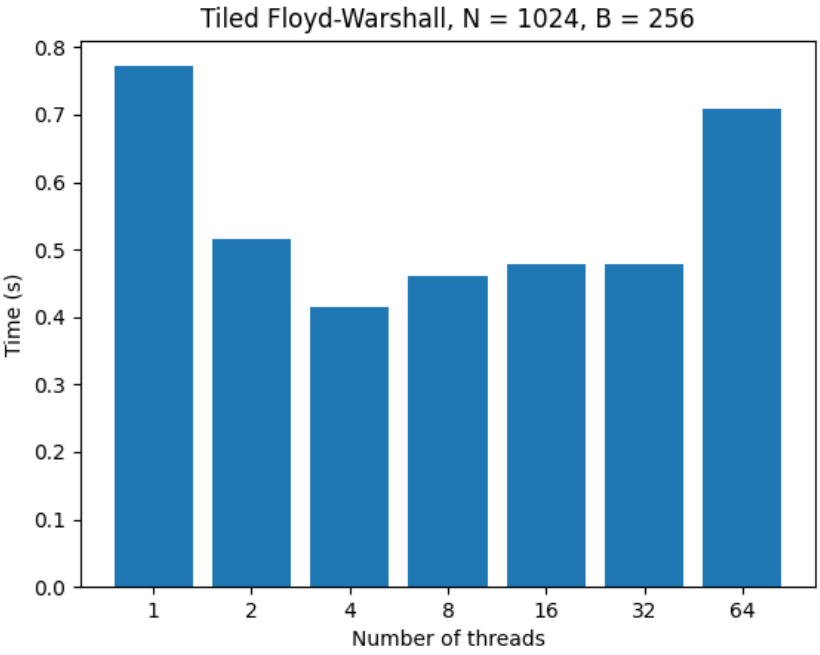
Εύκολα παρατηρούμε ότι τα πρώτα 4 for loops του αλγορίθμου που ενημερώνουν τα tiles της ίδιας γραμμής και της ίδιας στήλης με το k-οστό tile στην k-οστή επανάληψη, μπορούν να εκτελεστούν παράλληλα, καθώς δεν υπάρχουν εξαρτήσεις μεταξύ τους, μιας και τα εξαρτώμενα για τους υπολογισμούς tiles (που βρίσκονται στην από πάνω γραμμή και στην αριστερή στήλη) έχουν υπολογιστεί στην k-1 επανάληψη. Ύστερα, τα υπόλοιπα tiles του πίνακα (τα οποία χωρίζονται από την γραμμή και την στήλη του k-οστού tile) μπορούν και αυτά να υπολογιστούν παράλληλα, αφού πρώτα ολοκληρωθούν η υπολογισμοί της k-οστής σειράς και στήλης, μιας και όλες οι εξαρτήσεις της k-οστής επανάληψης έχουν εκπληρωθεί. Τέλος, απαιτείται και συγχρονισμός μετά τον υπολογισμό των τελευταίων στοιχείων του πίνακα. Η παράλληλη υλοποίηση μας με χρήση της βιβλιοθήκης OpenMP είναι:

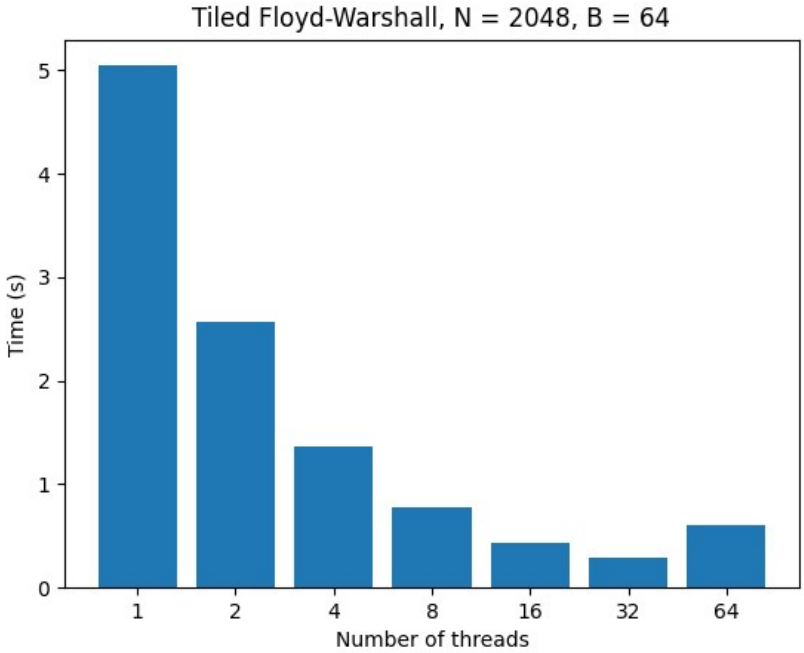
```
for(k=0;k<N;k+=B)
{
    FW(A,k,k,k,B);
    #pragma omp parallel
    {
        #pragma omp single
        {
            for(i=0; i<k; i+=B)
                #pragma omp task
                {FW(A,k,i,k,B);}

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

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

            for(j=k+B; j<N; j+=B)
                #pragma omp task
                {FW(A,k,k,j,B);}
        }
    }
    #pragma omp barrier

    #pragma omp parallel
    {
        #pragma omp single
        {
            for(i=0; i<k; i+=B)
                for(j=0; j<k; j+=B)
                    #pragma omp task
                    {FW(A,k,i,j,B);}

            for(i=0; i<k; i+=B)
                for(j=k+B; j<N; j+=B)
                    #pragma omp task
                    {FW(A,k,i,j,B);}

            for(i=k+B; i<N; i+=B)
                for(j=0; j<k; j+=B)
                    #pragma omp task
                    {FW(A,k,i,j,B);}

            for(i=k+B; i<N; i+=B)
                for(j=k+B; j<N; j+=B)
                    #pragma omp task
                    {FW(A,k,i,j,B);}
        }
    }
    #pragma omp barrier

}
```

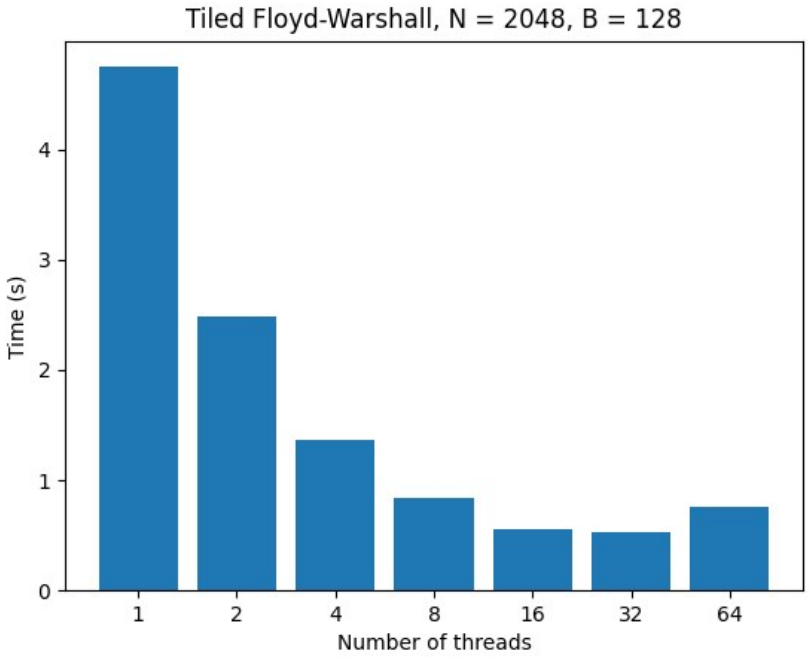
Τα αποτελέσματα συνοψίζονται στα παρακάτω διαγράμματα:

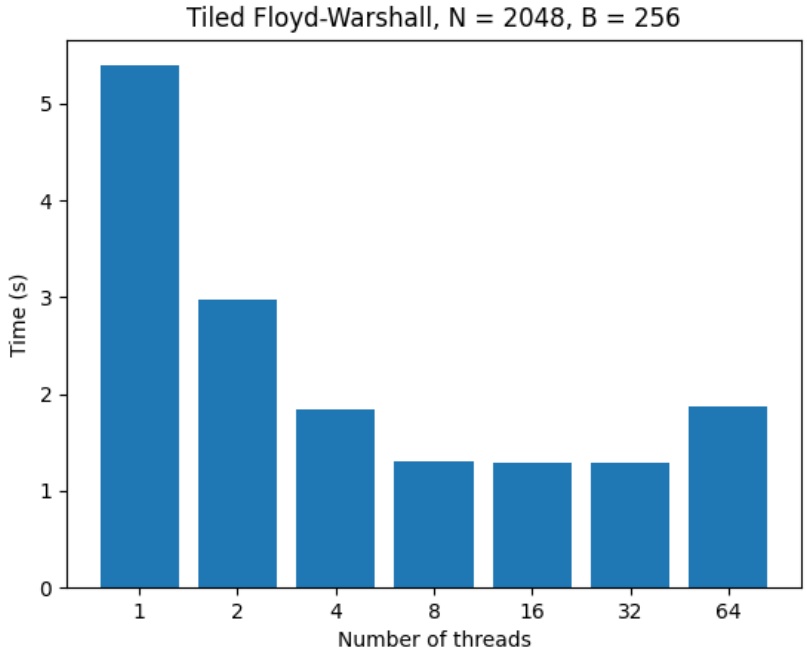


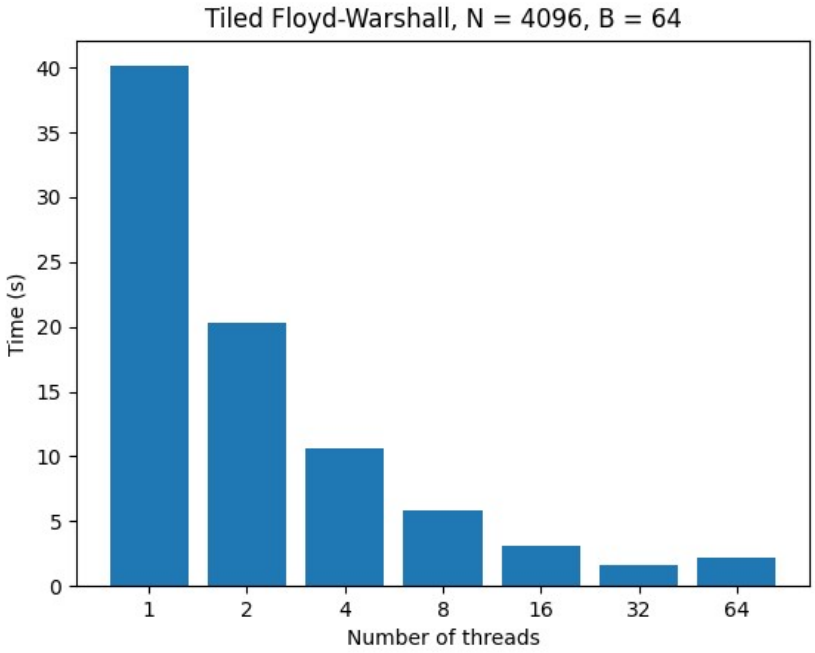


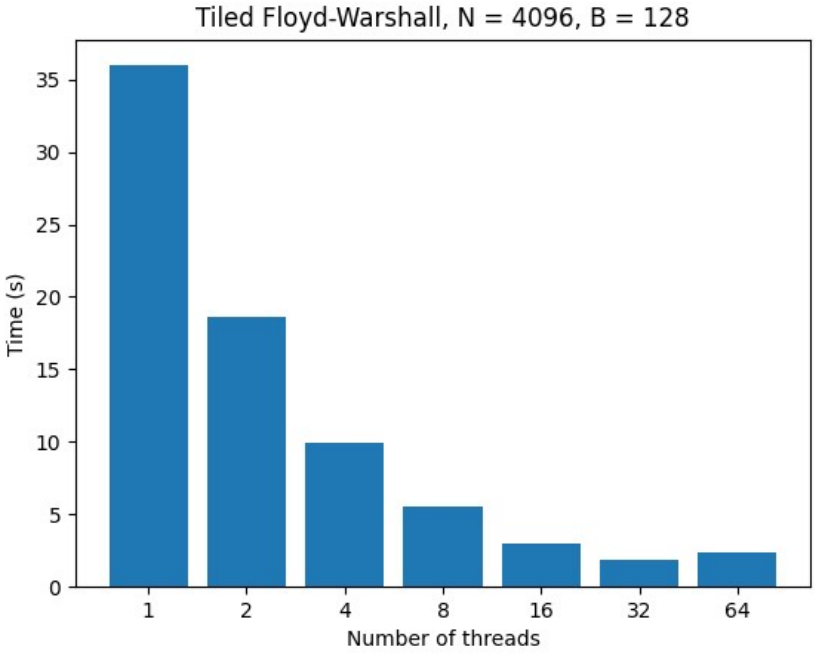

Tiled Floyd-Warshall, N = 1024, B = 256
Time (s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
1
2
4
8
16
32
64
Number of threads

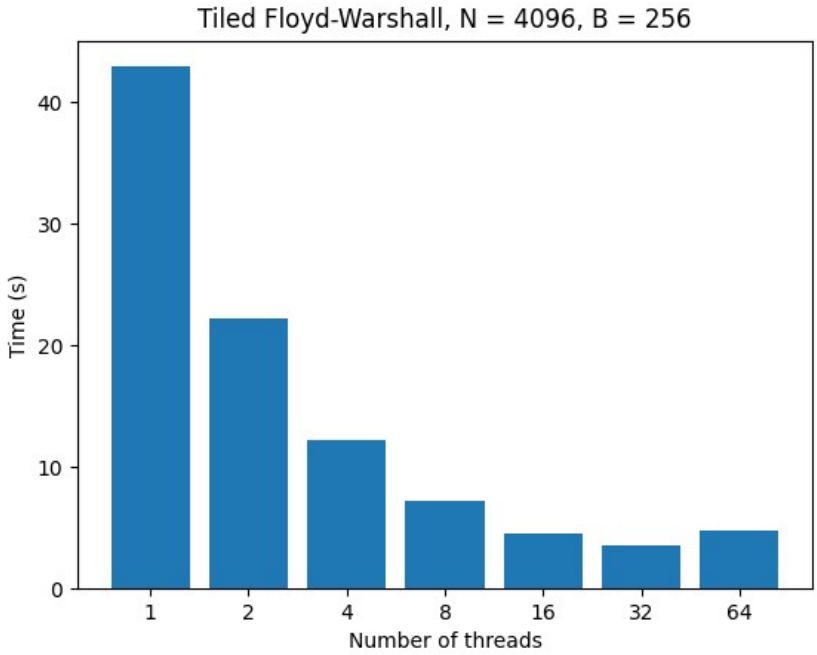

Tiled Floyd-Warshall, N = 2048, B = 64
Time (s)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads


Tiled Floyd-Warshall, N = 2048, B = 128
Time (s)
0
1
2
3
4
1
2
4
8
16
32
64
Number of threads


Tiled Floyd-Warshall, N = 2048, B = 256
Time (s)
0
1
2
3
4
5
1
2
4
8
16
32
64
Number of threads


Tiled Floyd-Warshall, N = 4096, B = 64
Time (s)
0
5
10
15
20
25
30
35
40
1
2
4
8
16
32
64
Number of threads


Tiled Floyd-Warshall, N = 4096, B = 128
Time (s)
0
5
10
15
20
25
30
35
1
2
4
8
16
32
64
Number of threads



Σχόλια - Παρατηρήσεις

Αρχικά, σε όλα τα N έχουμε παρόμοια μορφή του bar plot, οπότε οι παρατηρήσεις μας αφορούν όλες τις μετρήσεις για N = {1024, 2048, 4096}. Σε σχέση με την υλοποίηση με recursion, για 1 πυρήνα δεν υπάρχει σημαντική διαφορά, αλλά για περισσότερους πυρήνες η tiled υλοποίηση έχει πολύ καλύτερο scaling. Εδώ το speedup είναι πολύ καλό για κάθε διπλασιασμό μέχρι και τους 32 πυρήνες. Για 64 πυρήνες δεν έχουμε περαιτέρω βελτίωση του χρόνου Εδώ το καλύτερο B είναι B = 64.

Συγκρίνοντας τις 2 υλοποιήσεις, η recursive εκδοχή που χρησιμοποιεί tasks είναι προγραμματιστικά πιο εύκολη.