Συστήματα Παράλληλης Επεξεργασίας

Εργαστηριακή αναφορά 2
K-means & Floyd-Warshall

Γιάννος Παράνομος - 03118021
Νικήτας Τσίννας - 03118187

Νοέμβριος 2022

1 Εισαγωγή

Σε αυτή την εργαστηριακή άσκηση αναπτύσσουμε δύο παράλληλες εκδόσεις του αλγορίθμου K-Means στο προγραμματιστικό μοντέλο του κοινού χώρου διευθύνσεων με την χρήση του προγραμματιστικού εργαλείου `OpenMP`. Δοκιμάζουμε συνήθεις βελτιώσεις υλοποίησης και αξιολογούμε την τελική επίδοση. Επίσης, προς εξοικίωση της χρήσης των `OpenMP tasks` παραλληλοποιούμε τον αλγόριθμο Floyd-Warshall.

2 Αλγόριθμος K-means

Οι εκδόσεις που αναπτύξαμε είναι δύο: μια βασική έκδοση που παραλληλοποιεί τον σειριακό αλγόριθμο με προσθήκη καταλλήλων εντολών συγχρονισμού ώστε να γινει ορθά η παράλληλη ενημέρωση του πίνακα newClusters με τις νέες υπολογιζόμενες συντεταγμένες των κέντρων των συστάδων, και μια πιο εξελιγμένη έκδοση που χρησιμοποιεί τοπικούς (copied) πίνακες για κάθε νήμα, ώστε να μπορεί να γίνεται η εγγραφή χωρίς συγχρονισμό. Στη δεύτερη έκδοση μετά τους τοπικούς υπολογισμούς τα αποτελέσματα των τοπικών αυτών πινάκων συνδυάζονται (reduction) στον πίνακα newClusters.

2.1 Naive - shared clusters

Παραλληλοποιήσαμε την έκδοση του αλγορίθμου προσθέτοντας την ακόλουθη γραμμή πάνω από το for loop για το iteration των objects προς υπολογισμό των νέων clusters.

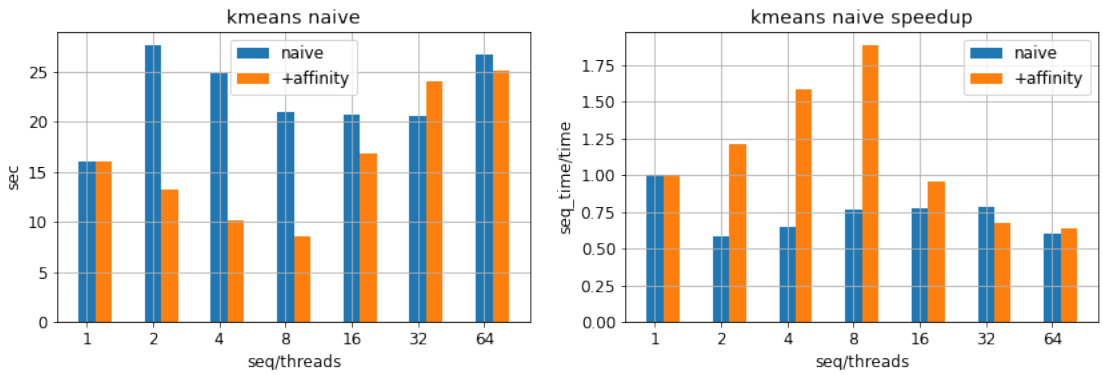
```
    ...
    do {
        // before each loop, set cluster data to 0
        for (i=0; i<numClusters; i++) {
            for (j=0; j<numCoords; j++)
                newClusters[i*numCoords + j] = 0.0;
```

```
            newClusterSize[i] = 0;
        }

        delta = 0.0;

        /*
         * TODO: Detect parallelizable region and use appropriate OpenMP
    pragmas
         */
        #pragma omp parallel for shared(newClusters, newClusterSize,
    membership, delta) private(i, j, index)
        for (i=0; i<numObjs; i++) {
            // find the array index of nearest cluster center
        ...
```

Δόθηκε προσοχή στα shared και private clauses για τον διαχωρισμό των μεταβλητών σε shared και private. Στις μεταβλητές που είναι shared, έχουν κοινή πρόσβαση όλοι οι εργάτες, ενώ στις private, ο κάθε εργάτης έχει την δική του έκδοση.
Σε αυτήν την έκδοση, για τις κοινές μεταβλητές `newClusters` και `newClusterSize` έχουμε αποφύγει πιθανά race conditions, αφού το openmp έχει φροντίσει για αυτό με κατάλληλη χρήση μεθόδων συγχρονισμού (mutexes, locks). Ωστόσο, για ατομικές εντολές μπορεί να χρησιμοποιηθεί το `#pragma omp atomic` clause, το οποίο αυξάνει σημαντικά την επίδοση χωρίς να ακυρώνει τον συγχρονισμό εφόσον η πράξη πρόσθεσης γίνεται μέσω μοναδικής εντολής assembly χωρίς να δημιουργούνται προβλήματα συγρονισμού.

```
...
    #pragma omp atomic
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        /*
        * TODO: enforce atomic access to shared "newClusters" array
        */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
...
```

Η παραπάνω υλοποίηση κώδικα εκτελέστηκε για αριθμό νημάτων 1, 2, 4, 8, 16, 32, 64. Έγινε έλεγχος σωστού συγχρονισμού εφόσον φροντίσαμε για κάθε διαφορετικό αριθμό νημάτων να έχουμε τα ίδια αποτελέσματα για τα final clusters. Τα παρακάτω αποτελέσματα αποτελούν τους χρόνους και το speedup, που εμφανίζει ο κάθε αλγόριθμος, για πίνακα μεγέθους 256.
Έπειτα, επαναλάβαμε τις μετρήσεις έχοντας κάνει export την μεταβλητή περιβάλλοντος `GOMP_CPU_AFFINITY = 0-64` η οποία προσδένει τα νήματα σε συγκεκριμένους πυρήνες για όλη την εκτέλεση (thread binding).
Παρουσιάζονται τα διαγράμματα επίδοσης:



Εστιάζοντας μόνο στις μπλε μπάρες, δηλαδή στην απλή naive έκδοση, όπου τα νήματα δεν προσδένονται απαραίτητα σε κάποιον φυσικό πυρήνα του μηχανήματος, οι επιδόσεις της παραλληλοποιημένης έκδοσης ανεξάρτητα από τον αριθμό νημάτων είναι χειρότερη της σειριακής έκδοσης, εφόσον δεν πετυχαίνουμε καθόλου speedup. Αυτό που υποψηαζόμαστε είναι πως τα νήματα του παραλληλοποιημένου προγράμματος αλλάζουν κατά την διάρκεια της εκτέλεσης το φυσικό περιβάλλον εκτέλεσής τους (δηλαδή τρέχουν σε διαφορετικό πυρήνα του μηχανήματος) και έτσι καταλήγουν συνεχώς σε cache misses. Αντιλαμβανόμαστε πως αυτό υποβαθμίζει σημαντικά την επίδοση έχοντας υπόψη τα γνωστά πρωτόκολλα cache coherence που ακουλουθούνται.

Αντιθέτως, όταν απαιτούμε thread binding μέσω της μεταβλητής περιβάλλοντος `GOM_CPU_AFFINITY` τότε παρατηρούμε σημαντική βελτίωση στα διαγράμματα χρόνου εκτέλεσης και επιτάχυνσης. Πιο συγκεκριμένα, τώρα που τα νήματα είναι προσδεμένα σε έναν επεξεργαστή δεν έχουμε φαινόμενα αποτυχίας της cache και το κάθε νήμα έχει συνεχώς στην διάθεσή του το cache block που χρησιμοποιεί και τροποποιεί συχνά χωρίς να χρειάζεται να γίνεται invalidated, όπως θα γινόταν αν ο ίδιος εργάτης ζητούσε το ίδιο block από διαφορετική cache.
Αξίζει να σημειώσουμε πως το scalability ”σπάει” στα 16 threads, και μάλιστα η επίδοση γίνεται χειρότερη από εκεί και έπειτα από την σειριακή.

2.2 Reduction - copied clusters and reduce

Σε αυτήν την έκδοση, έχουμε δημιουργήσει έναν πίνακα μεγέθους k, δηλαδή όσα είναι τα threads που χρησιμοποιούμε, των μεταβλητών newClusters και newClusterSize που τους έχουμε ονομάσει local_newClusters και local_newCluserSize αντίστοιχα:

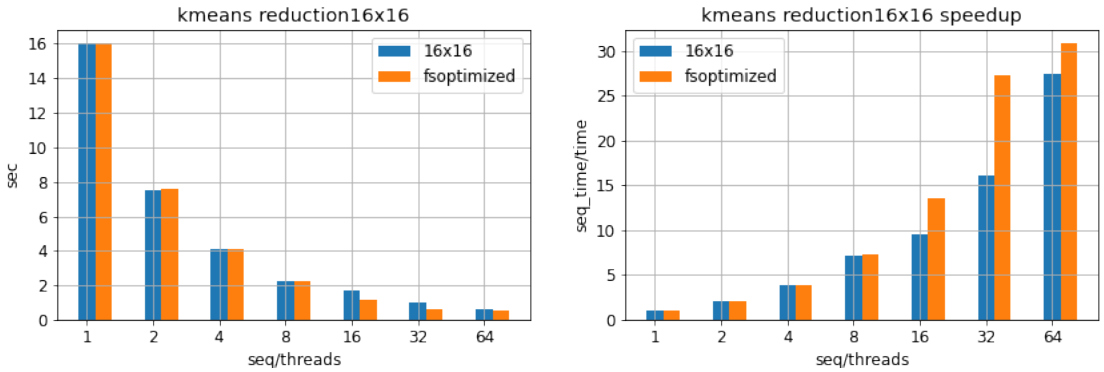
```
1    // Initialize local (per-thread) arrays (and later collect result on
     global arrays)
2    for (k=0; k<nthreads; k++)
3    {
4        local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(
     numClusters, sizeof(**local_newClusterSize));
5        local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters
     * numCoords, sizeof(**local_newClusters));
6    }
```

Η κάθε θέση των νέων πινάκων θα αντιστοιχεί σε ένα νήμα. Έτσι, το κάθε νήμα θα έχει τον δικό του χώρο εργασίας και στο τέλος θα συνδυάζονται όλες οι θέσεις των local πινάκων προς

την δηιουργία των ζητούμενων newClusters και newCLusterSize. Αυτή η τελική διαδικασία ονομάζεται reduction. παρουσιάζεται ο αντίστοιχος κώδικας παρακάτω:

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

Με αυτόν τον τρόπο, λοιπόν, αποφεύγουμε ζητήματα συγχρονισμού με μόνο κόστος την απαίτηση περισσότερης φυσικής μνήμης και την επεξεργασία που χρειάζεται στο τέλος όπου θα πρέπει να συνδυαστούν τα αποτελέσματα των διαφορετικών νημάτων από ένα νήμα. Εδώ η αρχικοποίηση των local μεταβλητών γίνεται από το κύριο νήμα εκτέλεσης, δηλαδή έξω από την περιοχή παραλληλοποίησης. Την έκδοση του προγράμματος όπου λαμβάνεται υπόψη το φαινόμενο του false-sharing την περιγράφουμε παρακάτω και για αυτό δεν σχολιάζουμε προς το παρόν τις πορτοκαλί μπάρες.

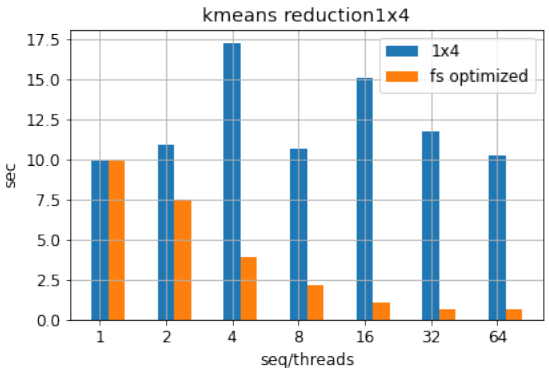


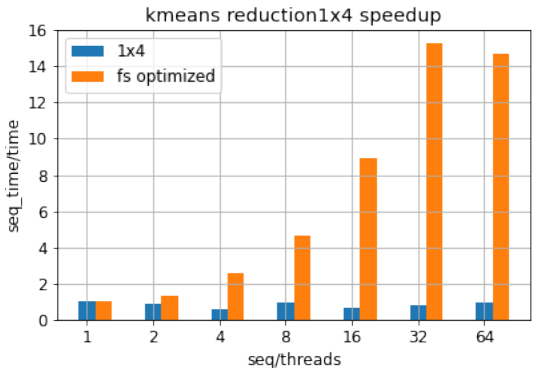
Σε σχέση με την προηγούμενη έκδοση (naive) παρατηρούμε σημαντικές βελτιώσεις. Αρχικά έχουμε exponential scalability και επίσης η καλύτερη επιτάχυνση που παρατηρούμε είναι γύρω στο x30. Επομένως, φαίνεται πως στην συγκεκριμένη περίπτωση, όπου το reduction δεν απαιτεί ιδιαίτερη υπολογιστική ισχύ και η μνήμη μας είναι αρκετή, η εκτέλεση των νημάτων σε ιδιωτικό χώρο μνήμης (εννοούμε τις local μεταβλητές) είναι καλύτερη επιλογή από τον μοιρασμό των global μεταβλητών.

Στην συνέχεια, τροποποιούμε την παρούσα έκδοση του προγράμματος ώστε να λαμβάνουμε υπόψη το φαινόμενο του false-sharing. Το φαινόμενο αυτό συμβαίνει όταν 2 ή παραπάνω επεξεργαστές θέλουν να τροποποιήσουν την ίδια γραμμή cache αλλά σε διαφορετικό byte index. Πιο συγκεκριμένα,

η αχρείαστη εναλλαγή της κατάστασης μίας cache line μεταξύ invalid και valid (exclusive/modified/shared αναλόγως το πρωτόκολλο της cache coherence) δημιουργεί ζήτημα επίδοσης.
Για να λύσουμε το συγκεκριμένο πρόβλημα πρέπει να σκεφτούμε πώς ή μνήμη του μηχανήματος sandman είναι σχεδιασμένη βάση NUMA (non-universal memory access). Δηλαδή, η πρόσβαση στην φυσική μνήμη είναι άμεσα συνδεδεμένη με τον χώρο εκτέλεσης (επεξεργαστή).
Στην αρχική έκδοση του reduction λοιπόν, αρχικοποιώντας τις local μεταβλητές σειριακά, προκαλούμε την αποθήκευση των local μεταβλητών σε σειριακή μνήμη η οποία αντστοιχεί στον έναν επεξεργαστή όπου εκτέλεσε την αρχικοποίηση, έστω τον E1. Όταν "μπαίνουμε" στο παράλληλο μέρος τους προγράμματος, τα threads τα οποία βρίσκονται σε διαφορετικές μονάδες επεξεργασίας, απαιτούν πρόσβαση στις μεταβλητές local οι οποίες είναι πολύ πιθανό να βρίσκονται σε διαφορετικό χώρο μνήμης, δηλαδή στην cache του E1. Επομένως θα πρέπει να μεταφερθεί αυτή η cache line στον χώρο επεξεργασίας του παράλληλου νήματος. Το πρόβλημα έγγυται στο γεγονός πως πολλά νήματα θέλουν πρόσβαση στο ίδιο cache line, για τον λόγο που περιγράψαμε προηγουμένως.
Το πρόβλημα λύνεται όταν τα νήματα δεν έχουν ανάγκη πρόσβασης στην ίδια cache line. Αυτό το καταφέρνουμε αρχικοποιώντας παράλληλα τις local μεταβλητές και όχι σειριακά. Έτσι μειώνουμε το πρόβλημα, αφού το κάθε νήμα αρχικοποιεί την local μεταβλητή στον χώρο μνήμης που αντιστοιχεί στην δική του μονάδα επεξεργασίας. Παραθέτουμε το σημείο κώδικα που περιγράφουμε:

```
...
#pragma omp parallel private(i, j, index) reduction(+:delta)
{
    int k = omp_get_thread_num();
    initialize_local(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 for
    for (i=0; i<numObjs; i++)
    ...
```





Εδώ βλέπουμε και τις 2 εκδόσεις του προγράμματος για configuration μικρών διαστάσεων (1x4). Για μικρές διαστάσεις του πίνακα φαίνεται πως η επίδοση του απλού reduction δεν είναι καθόλου ικανοποιητική. Αυτό συμβαίνει καθώς όλες οι μεταβλητές των Clusters μπορούν να χωρέσουν σε μία cache line, η οποία όταν αρχικοποιείται σειριακά, όλα τα νήματα απαιτούν πρόσβαση στην ίδια και μοναδική cache line.

Αντίθετα, όταν λαμβάνουμε υπόψη το φαινόμενο του false-sharing, γαι τους λόγους που περιγράψαμε αναλυτικά προηγουμένως, παρατηρούμε εξομάλυνση των διαγραμμάτων και ορατή βελτίωση, με scalability break point στα 32 threads.

Για το πρώτο configuration εκτέλεσης, δηλαδή για μέγεθος πίνακα Clusters 256, έχουμε σχετικά ίδιες επιδόσεις μέχρι και τα 8 threads. Από τα 16 threads και μετά παρατηρούμε σημαντικό speedup. Αυτό μπορούμε να το δικαιολογήσουμε, αφού για περισσότερα threads έχουμε περισσότερες απαιτήσεις της ίδιας cache line.

3 Floyd-Warshall

Μας δόθηκαν 3 εκδόσεις του αλγορίθμου FW. Μία standard, μία recursive και μία tiled. Μας ζητείται να παραλληλοποίησουμε την αναδρομική έκδοση του αλγορίθμου (recursive). Ωστόσο εμείς για διερευνητικούς λόγους παραλληλοποιούμε και την standard έκδοση.

3.1 Standard FW

Παρακάτω δίνεται μια υλοποίηση του κλασσικού αλγόριθμου Floyd-Warshall.

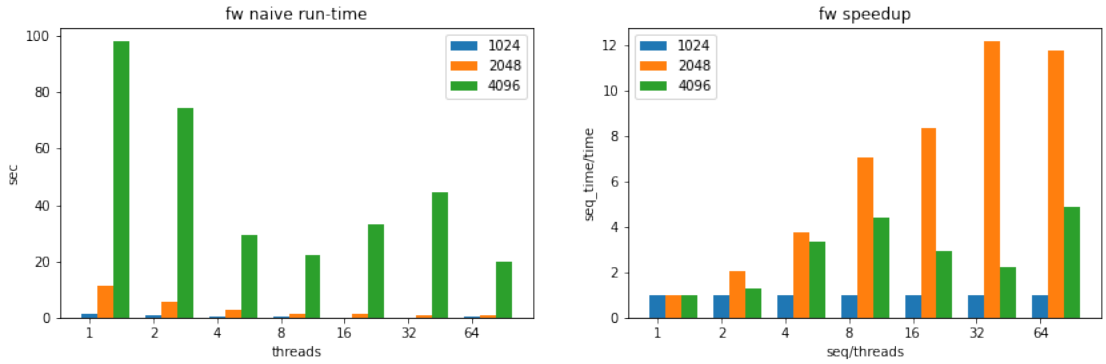
```
for(k=0;k<N;k++)
  for(i=0; i<N; i++)
      for(j=0; j<N; j++)
          A[i][j]=min(A[i][j], A[i][k]+A[k][j]);
```

Εξετάζοντας την κλασική υλοποίηση του αλγορίθμου Floyd-Warshall, μπορούμε να κάνουμε ορισμένες παρατηρήσεις ώστε να το παραλληλοποιήσουμε ορθά. Αρχικά, είναι σαφές ότι το βήμα k+1 απαιτεί την ολοκλήρωση του βήματος k, ουτωσώστε να έχει ανανεωθεί πλήρως ο πίνακας A. Σημειώνουμε ότι σε κάθε βήμα i,j, διατρέχεται ολόκληρος ο πίνακας A. Έπειτα ακριβώς λόγω της ανάγκης διαπέρασης ολόκληρου του πίνακα, προβλέπουμε ότι η παραλληλοποίηση του αλγορίθμου αυτού δεν θα έχει θεαματικά αποτελέσματα, καθώς μπορούμε να παραλληλοποιήσουμε μόνο τους υπολογισμούς για ένα δεδομένο βήμα i,j.

```
for(k=0;k<N;k++)
#pragma omp parallel for shared(A) private(i,j) firstprivate(k,N)
  for(i=0; i<N; i++)
      for(j=0; j<N; j++)
          A[i][j]=min(A[i][j], A[i][k]+A[k][j]);
```

Δίνονται τα διαγράμματα του χρόνου εκτέλεσης και επιτάχυνσης για την παραπάνω παραλληλοποιημένη

εκδοχή του αλγορίθμου.



Από τα παραπάνω αποτελέσματα παρατηρούμε ότι αν και πετύχαμε βελτίωση του χρόνου, με την αύξηση των thread, δεν έχουμε καταφέρει να παραλληλοποιήσουμε το κομμάτι του κώδικα που προσδίδει την μεγαλύτερη καθυστέρηση, δηλαδή την φόρτωση και διαπέραση ολόκληρου του πίνακα A. Το πρόβλημα της υλοποίησης αυτής είναι εμφανές αν παρατηρήσουμε τα αποτελέσματα για τον πίνακα μεγέθους 4096, ο οποίος όπως καταλαβαίνουμε δεν χωράει στην κρυφή μνήμη των thread, με αποτέλεσμα να έχουμε πολύ μεγάλες καθυστερήσεις και κακό scalability στην παραλληλοποίηση.

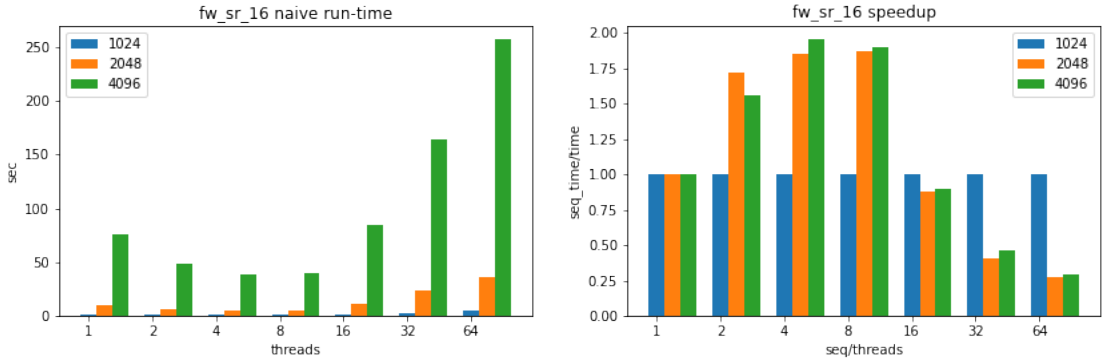
3.2 Recursive FW

Η αναδρομική υλοποίηση, χωρίζει τον πίνακα A αναδρομικά σε ταταρτημόρια, τα οποία εμφανίζουν μεταξύ τους εξαρτήσεις που θα σημειωθούν παρακάτω. Με τον τρόπο αυτό, φτάνουμε σε μια αναδρομική βάση, στην οποία εκτελείται αντίστοιχο κομμάτι κώδικα με την κλασσική υλοποίηση, τώρα όμως μόνο για έναν πίνακα μεγέθους B. Η βασική ιδέα είναι ότι το B πρέπει να πλησιάζει οριακά το μέγεθος της κρυφής μνήμης του κάθε thread, ώστε να μπορούμε να αποδώσουμε τα tasks του υπολογισμού των υποπινάκων αυτών, σε διαφορετικούς εργάτες. Σημειώνουμε ότι τα κομμάτια που μπορούν να παραλληλοποιηθούν είναι οι υπολογισμοί των τεταρτημορίων A12 και A21 καθώς εξαρτώνται και τα δύο είτε από το A11 είτε από το A22 και ποτέ μεταξύ τους. Έτσι ο παραλληλοποιημένος αλγόριθμος θα δομηθεί με τη βοήθεια tasks και taskwait, με χρήση του OpenMP. Τέλος, όπως θα φανεί παρακάτω το B είναι επαρκώς μικρό, ώστε να μην απαιτείται παραλληλοποίση στο τμήμα του αλγορίθμου που εκτελεί τον κλασσικό FW. Η απόφασή μας αυτή υποστηρίζεται από τα αποτελέσματα του παραπάνω ερωτήματος, όπου δεν παρατηρήθηκε βελτίωση επίδοσης όταν πολλά thread συνεισφέρουν στον υπολογισμό του πίνακα μεγέθους 1024.

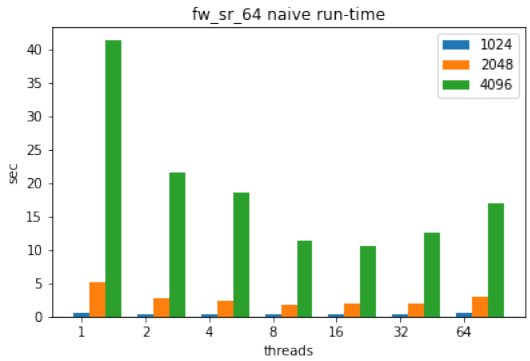
```
FW_Recursive(A) ->
    If A.size < B:
        FW_Classic(A)
    Else:
```

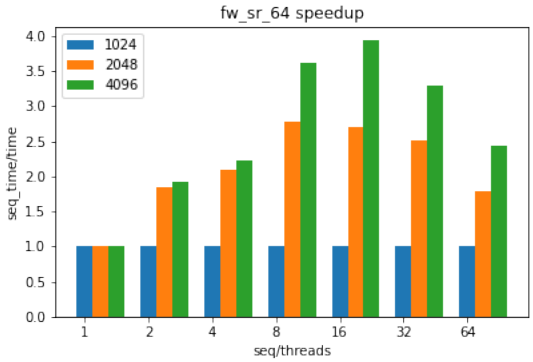
```
FW_Recursive(A11)
#pragma omp taskwait
#pragma omp task
FW_Recursive(A12)
#pragma omp task
FW_Recursive(A21)
#pragma omp taskwait
FW_Recursive(A22)
#pragma omp taskwait
FW_Recursive(A22)
#pragma omp taskwait
#pragma omp task
FW_Recursive(A21)
#pragma omp task
FW_Recursive(A12)
#pragma omp taskwait
FW_Recursive(A11)
```

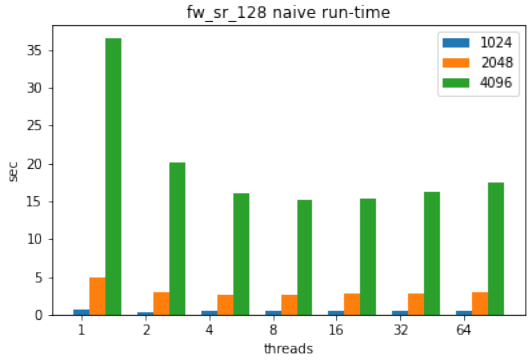
Παρακάτω δίνονται τα αποτελέσματα για χρήση διαφορετικών μεγεθλων B, για κάθε ζητούμενο μέγεθος πίνακα A. Παρατηρούμε ότι ανεξαρτήτως μεγέθους πίνακα, τα αποτελέσματά μας βελτιστοποιούνται για B=128, το οποίο σημαίνει οτί αυτό είναι το μέγεθος που πλησιάζει το περισσότερο το μέγεθος της κρυφής μνήμης των thread και άρα πετυχαίνει την βέλτιστη παραλληλοποίηση.

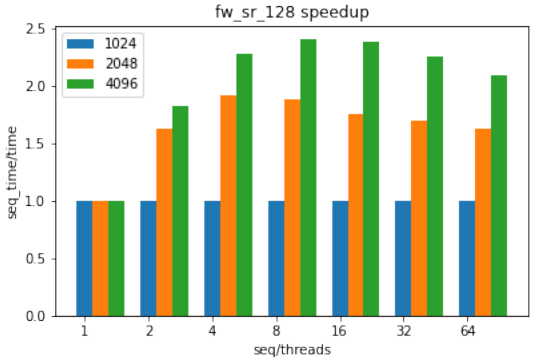


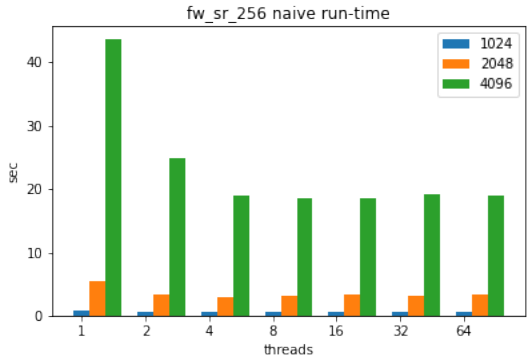
Όπως έχει ήδη αναφερθεί, το κακό scalability είναι αναμενόμενο. Η μικρή τιμή B = 16, είναι πολύ μικρότερη από το όριο της κρυφής μνήμης, με αποτέλεσμα να κάνουμε περισσότερες αναδρομές από ότι είναι απαραίτητο.

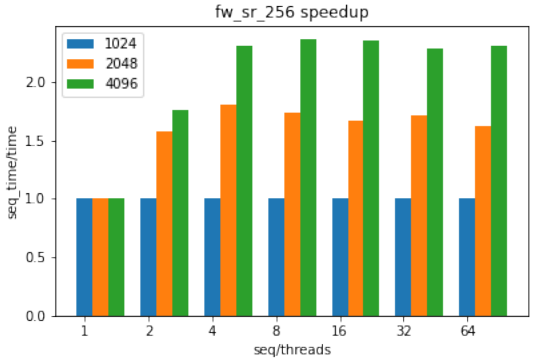












Παρατηρούμε ότι για B = 256, έχουμε μικρή μείωση της επίδοσης συγκριτικά με το B = 128. Έτσι θεωρούμε ότι πλέον έχουμε ξεπεράσει το όριο της κρυφής μνήμης και άρα καταλήγουμε στο B = 128 ως το βέλτιστο μέγεθος του υποπίνακα.