Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη ΗΜΜΥ

Συστηματα Παραλληλης Επεξεργασιας, 2022-23

Εξαμηνιαία Εργασία

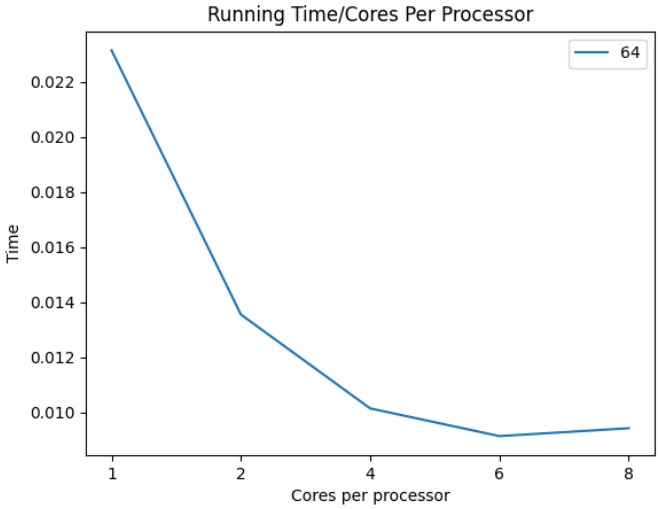
Αποστόλης Σταματής
03118034
Δημήτρης Μητρόπουλος
03118608
Δήμητρα Λεβέντη
03118015

5 Μαρτίου 2023

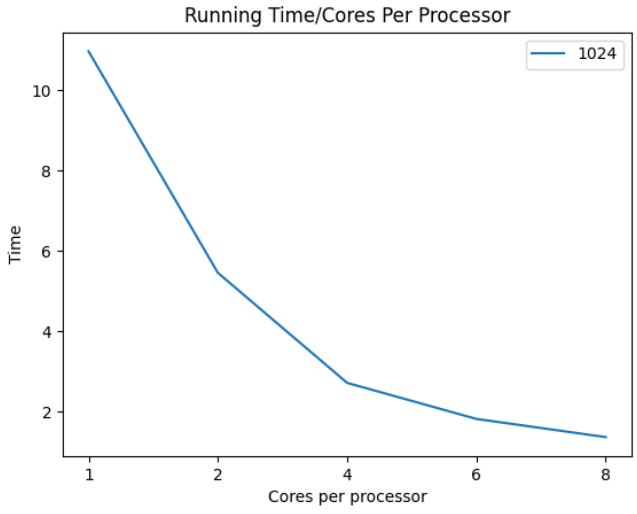
1 Εξοικείωση με το περιβάλλον προγραμματισμού

Τα αποτελέσματα της παραλληλοποίησης παρουσιάζονται παρακάτω σε γραφικές παραστάσεις για Grid size 64x64, 1024x1024 και 4096x4096. Όλα τα προγράμματα έτρεξαν για 1000 γενιές.

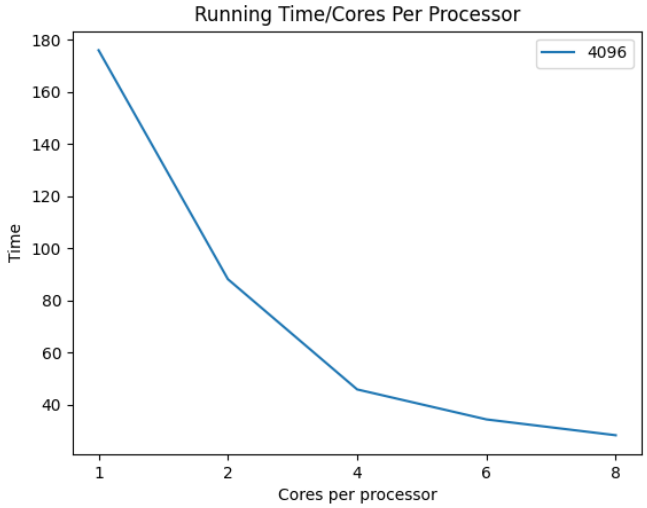
1. Grid size 64x64



2. Grid size 1024x1024



3. Grid size 4096x4096



Με δεδομένα τα παραπάνω παρατηρούμε τα εξής:

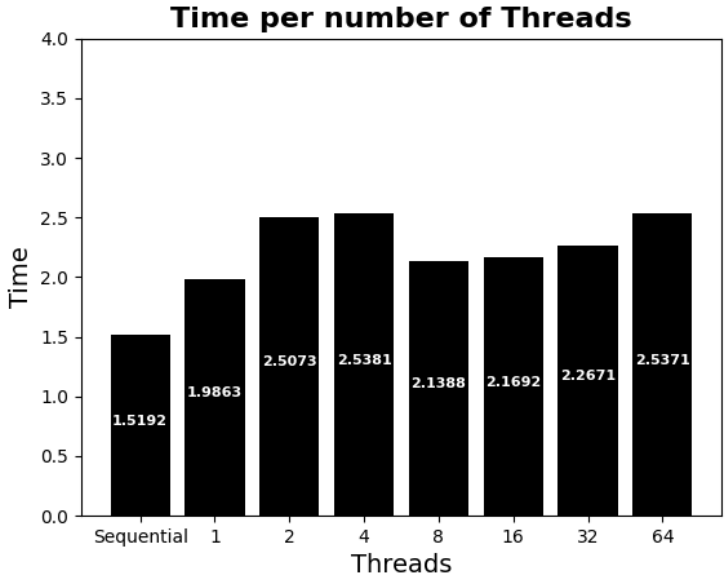
- Grid size 64x64: παρατηρούμε βελτίωση όσο αυξάνονται τα cores, χωρίς όμως να υπάρχει διαφορά μεταξύ 4,6,8 threads. Αυτό οφείλεται στο γεγονός πως για τόσο μικρό grid size, το overhead που προκαλείται από την παραλληλοποίηση αποτελεί bottleneck και η επίδοση δεν βελτιώνεται περαιτέρω.
- Grid size 1024x1024: παρατηρούμε ότι επιτυγχάνεται το βέλτιστο αποτέλεσμα παραλληλοποίησης, αφού υπάρχει γραμμική σχέση μεταξύ αριθμού threads και χρόνου εκτέλεσης.
- Grid size 4096x4096: παρατηρούμε ότι υπάρχει γραμμική βελτίωση για threads 1,2,4 αλλά παρόμοια με την περίπτωση του size 64 η βελτίωση είναι πολύ μικρή για 6,8 threads. Ο λόγος είναι ότι το πολύ μεγάλο grid size χαλάει το locality, οπότε χρειάζεται περισσότερος χρόνος για πρόσβαση στην μνήμη.

2 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

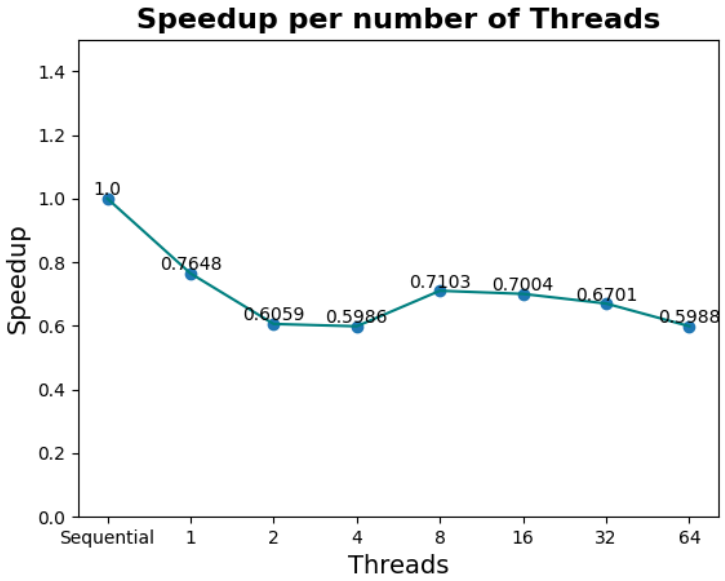
2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου k-means

Shared clusters

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:



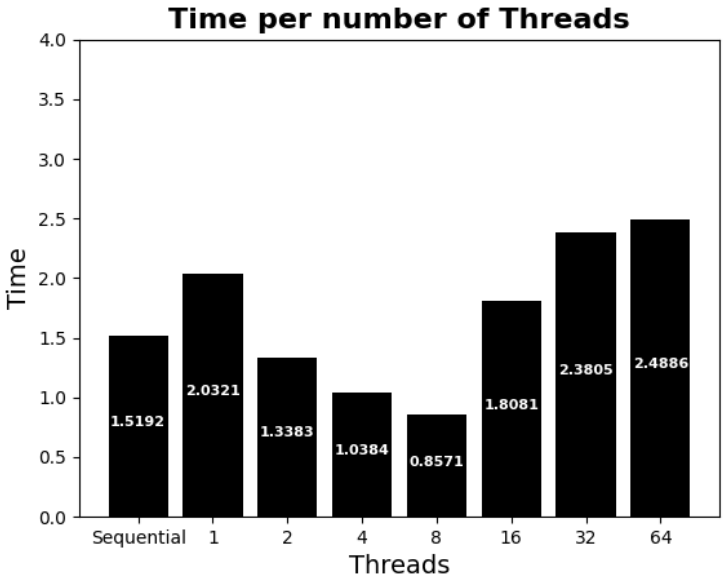
2. Το διάγραμμα speedup διαμορφώνεται ως εξής:



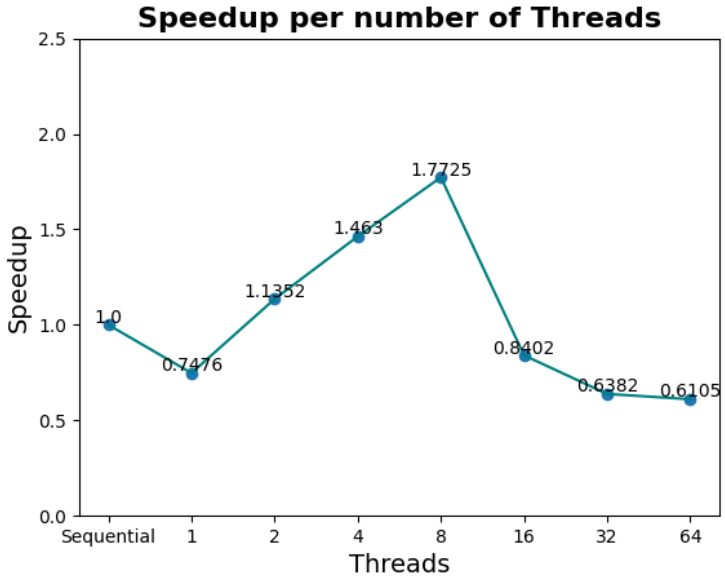
Παρατηρούμε ότι η επίδοση της παράλληλης έκδοσης είναι χειρότερη της σειριακής. Ο χρόνος που χρειάζεται το σειριακό πρόγραμμα είναι λιγότερος από οποιοδήποτε παράλληλο, παρόλο που δίνονται περισότεροι υπολογιστικοί πόροι. Όπως είναι προφανές και οι τιμές speedup των παράλληλων προγραμμάτων δεν ξεπερνούν το 1.
Συγκρίνοντας τα παράλληλα προγράμματα μεταξύ τους, την καλύτερη επίδοση φαίνεται να έχουν οι 8 και 16 threads, από άποψη χρόνου και speedup.
Για μικρό αριθμό threads (2,4), το overhead που απαιτείται για την πρόσβαση στην κοινή μνήμη είναι πολύ μεγάλο και δεν αντισταθμίζεται από την παράλληλη εργασία. Για μεγάλο αριθμό threads (32,64) λόγω του μεγάλου κατακερματισμού εργασίας, δεν αξιοποιούνται κατάλληλα οι caches των επεξεργαστών οπότε απαιτείται περισσότερος χρόνος για προσβάσεις στην μνήμη. Στην περίπτωση των 8 και 16 threads, το tradeoff locality-overhead διαχείρισης πρόσβασης στην κοινή μνήμη, φαίνεται να βρίσκεται σε ισορροπία. Παρόλα αυτά και πάλι δεν είναι αρκετό για να βελτιώσει την επίδοση του παράλληλου προγράμματος.

Η μεταβλητή GOMP_CPU_AFFINITY χρησιμοποιείται για να προσδέσει τα threads σε συγκεκριμένες CPUs. Με τον τρόπο αυτό αξιοποιείται καλύτερα το locality. Τα διαγράμματα έπειτα από την ανάθεση κατάλληλων τιμών στην μεταβλητή GOMP_CPU_AFFINITY διαμορφώνονται ως εξής:

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:



2. Το διάγραμμα speedup διαμορφώνεται ως εξής:



Copied clusters and reduce

Η ιδέα του αλγορίθμου είναι η διατήρηση ενός αντιγράφου του πίνακα clusters και κάθε thread, στο οποίο θα γράφονται τα αποτελέσματα που υπολογίζονται από το εκάστοτε thread. Το πλεονέκτημα της υλοποίησης αυτής είναι ότι δεν απαιτείται συγχρονισμός ανάμεσα στα threads, αφού το κάθε thread γράφει σε διαφορετική θέση μνήμης.
Στο τέλος του υπολογισμού, ο master αναλαμβάνει να συγκεντρώσει όλα τα αποτελέσματα των επιμέρους threads και να υπολογίσει το τελικό αποτέλεσμα. Στην προκειμένη περίπτωση, ο master για να υπολογίσει το κέντρο του κάθε cluster εργάζεται ως εξής:

- Για κάθε συντεταγμένη x_i:
    1. Αθροίζει όλα τα επιμέρους αθροίσματα της i που έχουν υπολογίσει τα διαφορετικά threads, υπολογίζοντας το $\sum_{i \in cluster} x_i$
    2. Υπολογίζει αντίστοιχα το μέγεθος του cluster
    3. Διαιρώντας τα παραπάνω προκύπτει το κέντρο του cluster για την συντεταγμένη x_i

Ορισμένα σημαντικά σημεία του κώδικα που αξίζει να αναφερθούν είναι:

- Update δεδομένων σπό το κάθε cluster

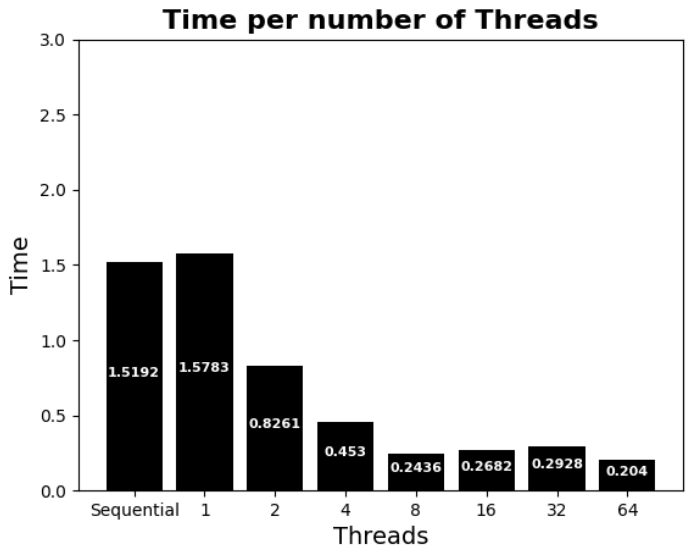
```
int threadid = omp_get_thread_num();
local_newClusterSize[threadid][index]++;
for (j=0; j<numCoords; j++)
    local_newClusters[threadid][index*numCoords + j] +=
    ↪  objects[i*numCoords + j];
```

- Reduction από τον master

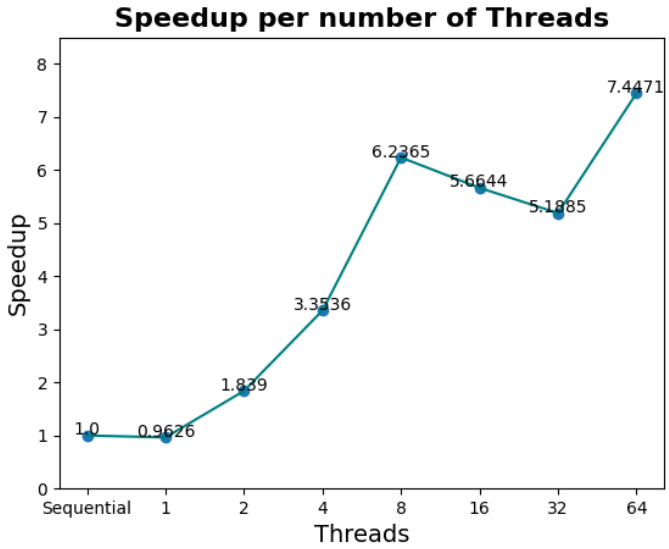
```
for (k=0; k<nthreads; k++) {
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            newClusters[i*numCoords + j] +=
            ↪  local_newClusters[k][i*numCoords + j];
        }
        newClusterSize[i] += local_newClusterSize[k][i];
    }
}
// average the sum and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

Για το configuration {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} οι επιδόσεις που επιτυγχάνονται φαίνονται παρακάτω:

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:

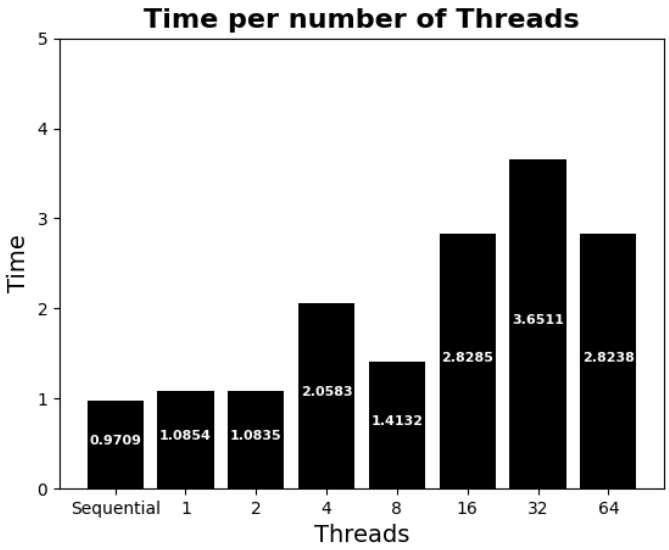


2. Το διάγραμμα speedup διαμορφώνεται ως εξής:

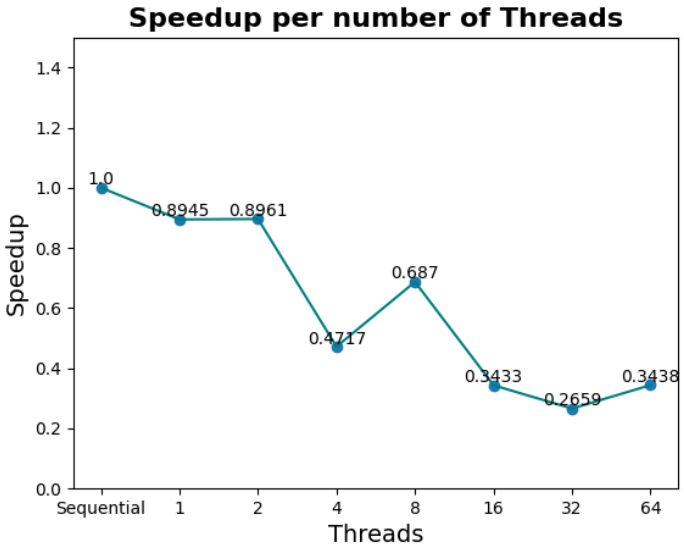


Παρατηρούμε σαφώς καλύτερα αποτελέσματα, αφού σε αντίθεση με πριν η υλοποιήση αποφεύγει τον συγχρονισμό κατά την ενημέρωση των πινάκων, αξιοποιώντας πλήρως την παραλληλία.
Όσον αφορά τα αποτελέσματα για το configuration {Size, Coords, Clusters, Loops} = {256, 1, 4, 10}, φαίνονται παρακάτω:

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:



2. Το διάγραμμα speedup διαμορφώνεται ως εξής:



Παρατηρούμε ότι σε σχέση με το {Size, Coords, Clusters, Loops} = {256, 16, 16, 10} τα αποτελέσματα είναι πολύ χειρότερα, παρόλο που το μέγεθος του input είναι μικρότερο. Αυτό οφείλεται στους εξής παράγοντες:

1. **First-touch policy**: Το thread που ¨ακουμπάει' πρώτη φορά κάποια δεδομένα είναι ιδιοκτήτης της σελίδας στην οποία ανήκουν, δηλαδή τα δεδομένα τοποθετούνται στην μνήμη που βρίσκεται πιο κοντά στο thread. Για να το εκμεταλευτούμε αυτό, θα θέλαμε το κάθε thread να έχει γρήγορη πρόσβαση στην γραμμή των local πινάκων που επεξεργάζεται. Κάνουμε τις εξής αλλαγές:

   - Χρησιμοποιήσουμε malloc για να δεσμεύσουμε την μνήμη αντί για calloc (η οποία κάνει και touch τα δεδομένα)
   - Το κάθε thread αρχικοποιεί την σελίδα των local πινάκων που θα επεξεργαστεί. Αυτό γίνεται με ένα ξεχωριστό parallel for.

2. **False-sharing**: Επειδή το input είναι πολύ μικρό, διπλανές γραμμές των local πινάκων βρίσκονται στην ίδια cache lane. Έτσι όταν κάποιο thread αλλάζει τα δεδομένα της κ γραμμής, ολόκληρη η cache lane γίνεται invalidate, με αποτέλεσμα να θεωρούνται invalid ενδεχομένως και οι γραμμές κ+1, κ+2 κλπ και να πρέπει να ξαναφορτωθούν από την μνήμη. Για να λύσουμε το πρόβλημα αυτό, παρεμβάλλουμε μεταξύ των γραμμών των χρήσιμων γραμμών των πινάκων και κάποιες άδειες γραμμές ως padding. Έτσι, η cache lane περιέχει μόνο 1 χρήσιμη γραμμή, επομένως δεν γίνεται invalidate από αλλαγές που κάνουν άλλα threads στα δικά τους δεδομένα.

Οι παραπάνω παρατηρήσεις οδηγούν στις εξής αλλαγές στην υλοποίηση:

- Η δέσμευση μνήμης γίνεται με τον εξής τρόπο (οι παράμετροι size_offset και cluster_offset υπολογίζονται ανάλογα το μέγεθος της cache lane)

```
int * local_newClusterSize[size_offset*nthreads];  //
↪  [nthreads][numClusters]
float * local_newClusters[cluster_offset*nthreads];   //
↪  [nthreads][numClusters][numCoords]

// Initialize local (per-thread) arrays (and later collect
↪  result on global arrays)
for (k=0; k<cluster_offset*nthreads; k++)
    // use malloc here instead of calloc, so we let the
    ↪  threads initialize the arrays to take advantage of
    ↪  first touch policy
    local_newClusters[k] = (typeof(*local_newClusters))
    ↪  malloc(numClusters * numCoords *
    ↪  sizeof(**local_newClusters));
for (k=0; k<size_offset*nthreads; k++)
    // use malloc here instead of calloc, so we let the
    ↪  threads initialize the arrays to take advantage of
    ↪  first touch policy
    local_newClusterSize[k] =
    ↪  (typeof(*local_newClusterSize)) malloc(numClusters
    ↪  * sizeof(**local_newClusterSize));
```

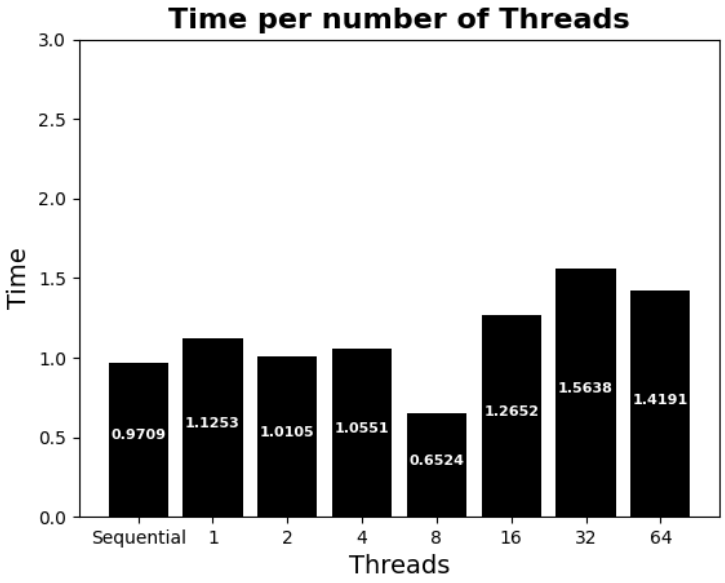
- Η εκμετάλλευση της πολιτικής first touch γίνεται με τον εξής τρόπο:

```
// Reduntant parallel for for first touch policy
#pragma omp parallel for private(j) default(shared)
for (i=0; i<nthreads; i++)
{
// Make each thread initialize its own subarray on
↪  local_newClusters
int k;
int thread_id;
thread_id = omp_get_thread_num();
for (k=0; k<numClusters; k++) {
  for (j=0; j<numCoords; j++)
    local_newClusters[cluster_offset*thread_id]
    ↪  [k*numCoords + j] = 0.0;
    local_newClusterSize[size_offset*thread_id][k] = 0;
  }
}
```

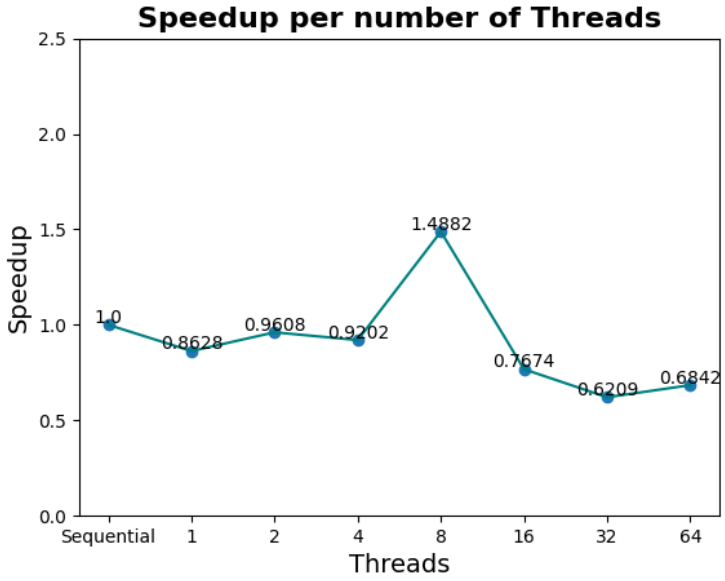
Για να υπολογιστεί η τιμή του padding, χρησιμοποιήθηκαν 2 πιθανά μεγέθη cache lane, 64 και 128 KB. Τα αποτελέσματα για κάθε μία περίπτωση φαίνονται παρακάτω:

Cache lane 64

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:

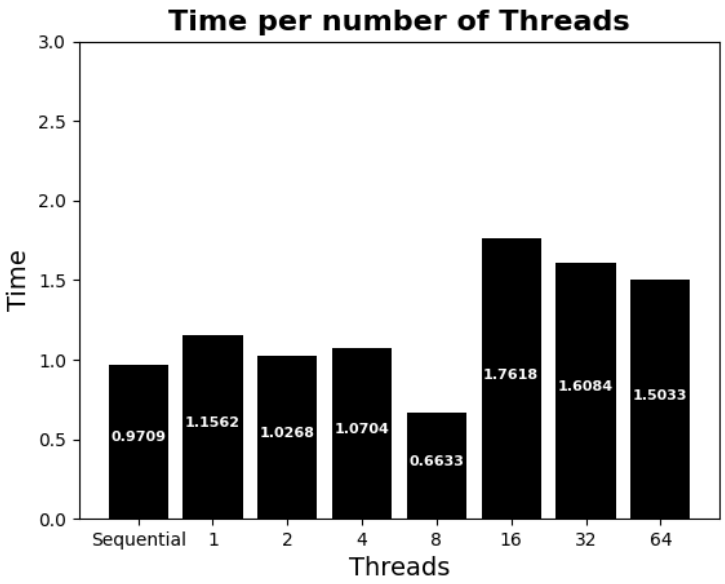


2. Το διάγραμμα speedup διαμορφώνεται ως εξής:

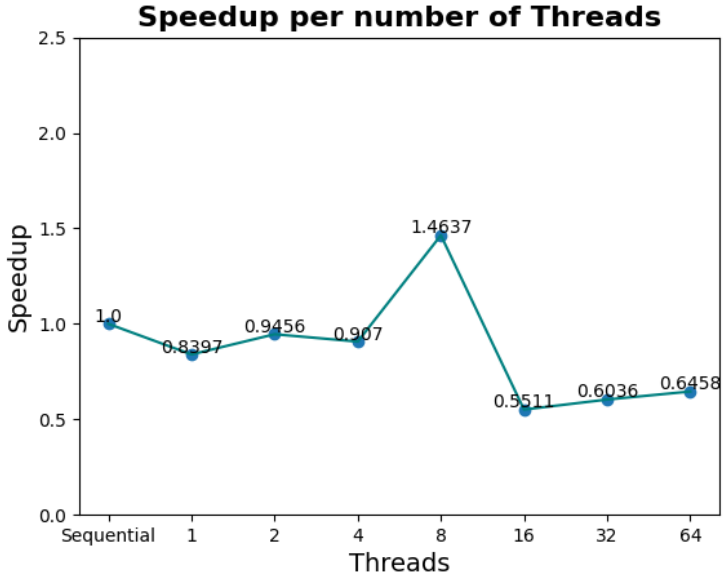


Cache lane 128

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:



2. Το διάγραμμα speedup διαμορφώνεται ως εξής:



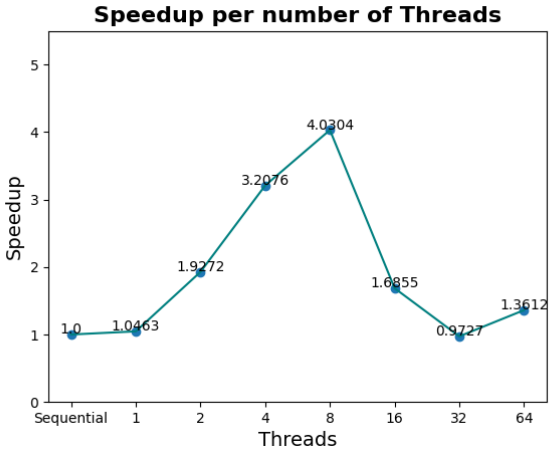
Παρατηρούμε ότι και στις 2 περιπτώσεις το καλύτερο speedup επιτυγχάνεται για 8 threads. Ακόμα, το speedup διπλασιάζεται περίπου σε σχέση με το αντίστοιχο

configuration της προηγούμενης υλοποίησης, χωρίς να πετυχαίνουμε όμως αντίστοιχες τιμές με το configuration {256, 16, 16, 10}.

Αμοιβαίος Αποκλεισμός - Κλειδώματα

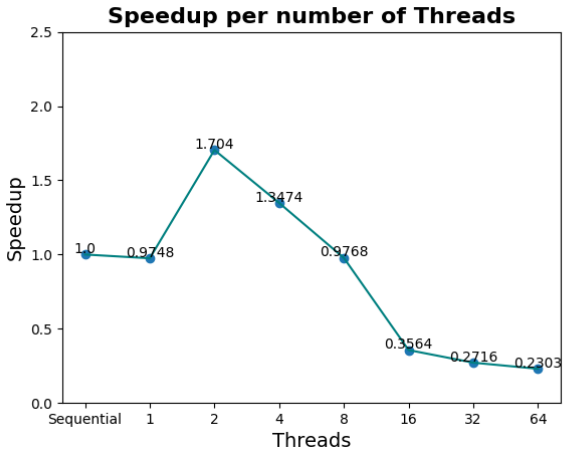
Τα αποτέλεσματα των μετρήσεων για τις δοσμένες υλοποιήσεις παρουσιάζονται παρακάτω Για κάθε κλείδωμα:

1. **no sync (Χωρίς κλείδωμα):**



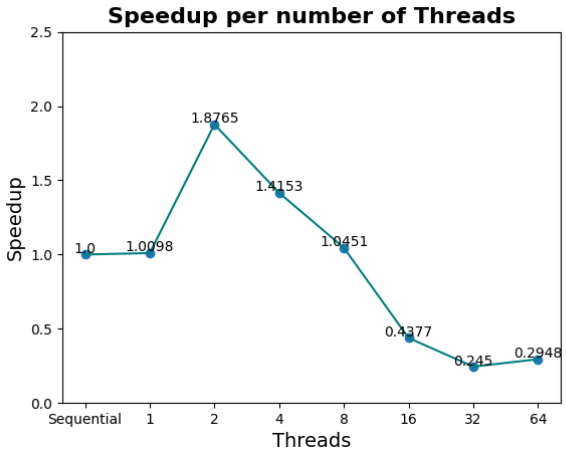
Δεν εξασφαλίζεται αμοιβαίος αποκλεισμός στα διαμοιραζόμενα δεδομένα. Παρατηρείται σημαντική αύξηση του speedup μέχρι και τα 8 threads όμως τα αποτελέσματα του προγράμματος είναι λανθασμένα. Για περισσότερα από 8 threads, η συμφόρηση του bus λόγω cache invalidation στις θέσεις κοινής μνήμης εντός του κρίσιμου τμήματος επιδρά στην απόδοση του προγράμματος.

2. **pthread mutex lock:**



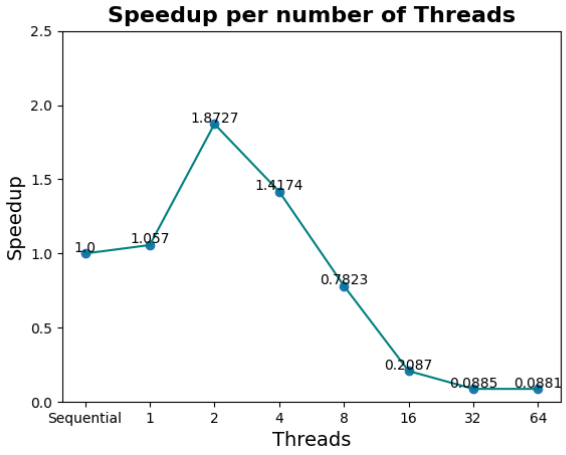
Υλοποιείται από τη βιβλιοθήκη pthread της C. Βασίζεται στην ατομική αύξηση και μείωση της τιμής του αντικειμένου mutex, το οποίο λαμβάνει τιμές 0 ή 1, που αντιστοιχούν σε κατειλημμένο ή ελεύθερο lock αντίστοιχα. Αν μία διεργασία προσπαθήσει να πάρει το lock αλλά αποτύχει, θέτεται σε κατάσταση sleep ώστε ο επεξεργαστής να είναι ελεύθερος προς χρήση από άλλες διεργασίες, και περιμένει ώσπου να ελευθερωθεί το lock. Η λειτουργία του μπορεί να διαφοροποιηθεί με χρήση διάφορων παραμέτρων όμως εμείς το χρησιμοποιούμε με την default μορφή του.
Στην υλοποίηση του kmeans, το pthread mutex παρουσιάζει ικανοποιητικό speedup για 2 threads, όμως η προσθήκη παραπάνω παράλληλων προσβάσεων στο κρίσιμο τμήμα με χρήση παραπάνω threads, έχει σημαντικές επιπτώσεις στον χρόνο εκτέλεσης. Τόσο σημαντικές που για περισσότερα των 8 threads, το speedup έιναι μικρότερο της μονάδας. Αυτό οφείλεται στη συμφόρηση του bus λόγω cache invalidation, όποτε αλλάζει η τιμή του mutex.

3. **pthread spin lock:**



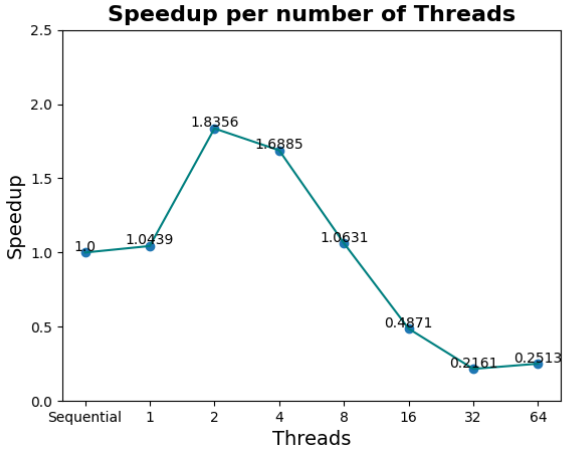
Υλοποιείται από τη βιβλιοθήκη pthread της C. Βασίζεται στην ατομική αύξηση της τιμής της μεταβλητής pthread_spinlock_t, η οποία λαμβάνει τιμές 0 και 1, που αντιστοιχούν σε κατειλημμένο και ελεύθερο lock αντίστοιχα. Σε αυτή την περίπτωση αν μια διεργασία θέλει να πάρει το lock, προσπαθεί έως να τα καταφέρει.
Παρατηρείται αντίστοιχη συμπεριφορά με το κλείδωμα pthread mutex όσον αφορά την επιβάρυνση του χρόνου εκτέλεσης με προσθήκη επιπλέον threads.

4. **tas lock:**



Ατομικά θέτει την μεταβλητή State, η οποία αποτελεί το lock, ίση με 1 και την μεταβλητή Test με την προηγούμενη τιμή που διέτετε η State. Αν η Test έχει μηδενική τιμή, τότε δεσμεύεται το lock, διαφορετικά επαναλλαμβάνει μέχρι να το δεσμεύσει.
Παρατηρείται παρόμοια συμπεριφορά με τα παραπάνω κλειδώματα όσον αφορά την επιβάρυνση του χρόνου εκτέλεσης με προσθήκη επιπλέον threads.

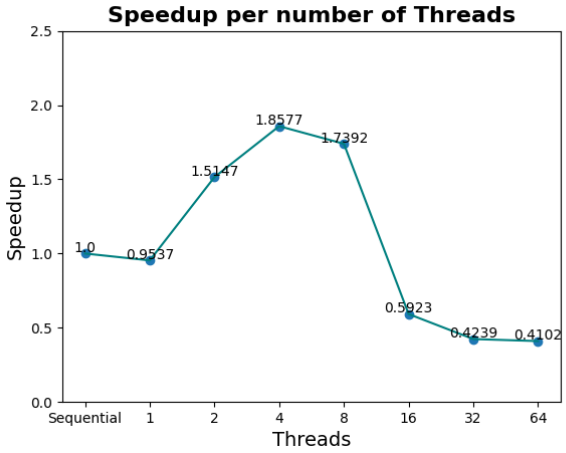
5. **ttas lock:**



Έχει παρόμοια λειτουργία με το tas, με την προσθήκη ενός ακόμα βήματος. Πρωτού διεκδικήσει το lock, διαβάζει την τιμή της state, ώσπου αυτή να

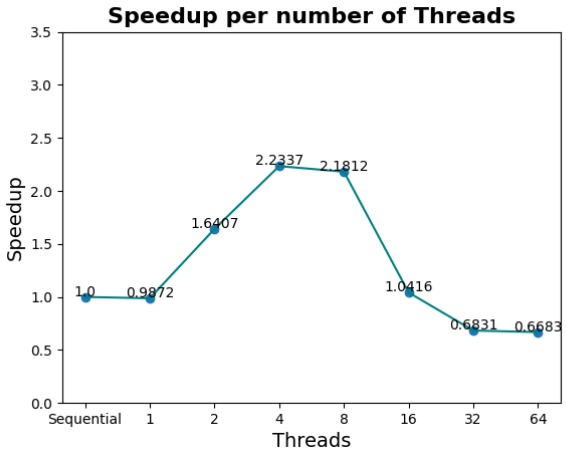
γίνει ίση με 0. Ύστερα προσπαθεί να πάρει το lock. Αν η προσπάθεια είναι ανεπιτυχής επιστρέφει στο πρώτο βήμα.
Και σε αυτή τη περίπτωση παρόμοια συμπεριφορά με τα παραπάνω κλειδώματα όσον αφορά την επιβάρυνση του χρόνου εκτέλεσης με προσθήκη επιπλέον threads.

6. **array lock:**



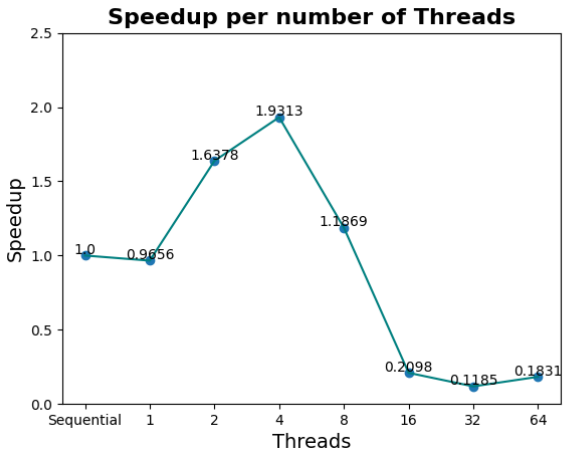
Βασίζεται στην διαχείριση ενός Boolean flag array με χρήση του iterator tail. Για να αποκτήσει ένα thread το lock, αυξάνει ατομικά την τιμή του tail, οπότε λαμβάνει μία θέση στον flag array. Ύστερα διαβάζει επίμονα την θέση που του ανατέθηκε ώσπου να γίνει true. Τότε μπορεί να μπεί στο κρίσιμο τμήμα. Για να απελευθερώσει το lock, το thread κάνει την θέση του στον πίνακα false, και την επόμενη θέση true. Ο πίνακας είναι κυκλικός, οπότε η τιμή του tail υπολογίζεται modulo n, όπου n ο μέγιστος αριθμός threads που μπορεί να τρέχουν ταυτόχρονα.
Εδώ, το μέγιστο speedup παρατηρείται στα 4 και 8 threads, ακολουθώντας την εικόνα της no sync υλοποίησης.

7. **clh lock:**



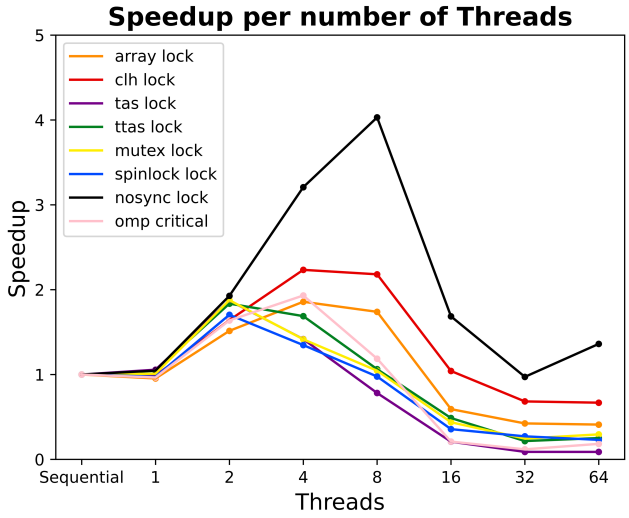
Βασίζεται στην διαχείριση μίας συνδεδεμένης λίστα από αντικείμενα QNode, και του δείκτη στο τελευταίο της στοιχείο tail. Κάθε αντικείμενο εκπροσωπεί την κατάσταση ενός thread που είτε διαθέτει, είτε προσπαθεί να πάρει, το lock. Όταν ένα thread θέλει να αποκτήσει το lock, θέτει το πεδίο locked του QNode που του αντιστοιχεί σε true. Επιπλέον βάζει τον δείκτη tail να δείχνει προς το QNode του και τον πεδίο predecessor του QNode του, να δείχνει στο προηγούμενο tail. Ύστερα διαβάζει επίμονα το πεδίο locked του predecessor, ώσπου να γίνει false. Όταν συμβεί αυτό, το thread θα διαθέτει το lock. Για να απελευθερώσει το lock, αρκεί να θέσει το πεδίο lock του QNode που του αντιστοιχεί σε false.
Αντιστοίχως με την array υλοποίηση, ακολουθεί την εικόνα της nosync.

8. **Με χρήση omp critical:**



Αυτή είναι και η υλοποίηση που χρησιμοποιήσαμε στην εφαρμογή του shared clusters στο προηγούμενο σκέλος της άσκησης (βλ. σελ. 5).

Συνολικά:



Όπως είναι λογικό η καλύτερη απόδοση είναι της no sync υλοποίησης. Αμέσως μετά έρχονται οι queue lock υλοποιήσεις του clh queue lock και array lock. Για 2 και 4 threads, Μεταξύ του clh και array βρίσκεται η υλοποίηση omp critical. Ακολουθούν τα υπόλοιπα locks με το ttas να είναι το πιο αποδοτικό.

Όπως αναφέρθηκε παραπάνω, οι queue lock υλοποιήσεις παρουσιάζουν εικόνα αντίστοιχη του no sync στη συσχέτιση speedup-number of threads, υποδεικνύοντας ότι δεν επιβαρύνουν την υλοποίηση λόγω της συμπεριφοράς τους, όπως κάνουν τα υπόλοιπα locks που χρησιμοποιήθηκαν. Το clh queue lock είναι καλύτερο του array, καθώς εκμεταλλεύεται το locality και το κάθε thread χρειάζεται να κρατάει private αντίγραφο του επόμενου κόμβου, ενώ το array χρησιμοποιεί ένα shared array στο οποίο η πρόσβαση είναι εξ΄ ορισμού πιο αργή. Συνολικά, η υπεροχή των queue locks οφείλεται στο γεγονός ότι αποφεύγουν τις προσβάσεις σε κοινή μνήμη. Τα locks tas, ttas, mutex και spinlock βασίζονται στην πρόσβαση σε μία θέση μνήμης που αποτελεί το lock, επιβαρύνοντας χρονικά το πρόγραμμα λόγω αναγκών cache coherence κάθε φορά που αλλάζει η ιδιοκτησία του lock και άρα και η τιμή του. Αντίθετα στα queue locks, κάθε thread σχετίζεται με 2 θέσεις μνήμης και προκαλεί 1 μόνο invalidation στην cache του επόμενου thread που διεκδικεί το lock.
Όσον αφορά την υλοποίηση με χρήση omp critical, φαίνεται να παρουσιάζει καλή κλιμακοσημότητα έως τα 4 threads, ενώ επιδεινώνεται για περισσότερα των 4 threads, με πολύ κακή απόδοση για περισσότερα των 16. Η συμπεριφορά του διαφέρει τόσο από την no sync υλοποίηση (και τις queue lock που την ακολουθούν), όσο από τα non-queue locks. Ωστόσο, όπως προαναφέρθηκε, στην αποδοτική περιοχή λειτουργίας του (2-4threads), η υλοποίηση omp critical, παρουσιάζει speedup ανάλογο των κυρίαρχων queue locks.

2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall Recursive Αλγόριθμος

Τα κύρια σημεία του κώδικα που αξίζει να αναφερθούν είναι:

- Ο συγχρονισμός με χρήση tasks γίνεται με τον ακόλουθο τρόπο:

```
FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2,
↪  bsize);
#pragma omp task shared(A,B,C) if (0)
{
  #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 task shared(A,B,C) if (0)
{
  #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);
  #pragma omp taskwait
}
FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol,
↪  myN/2, bsize);
```

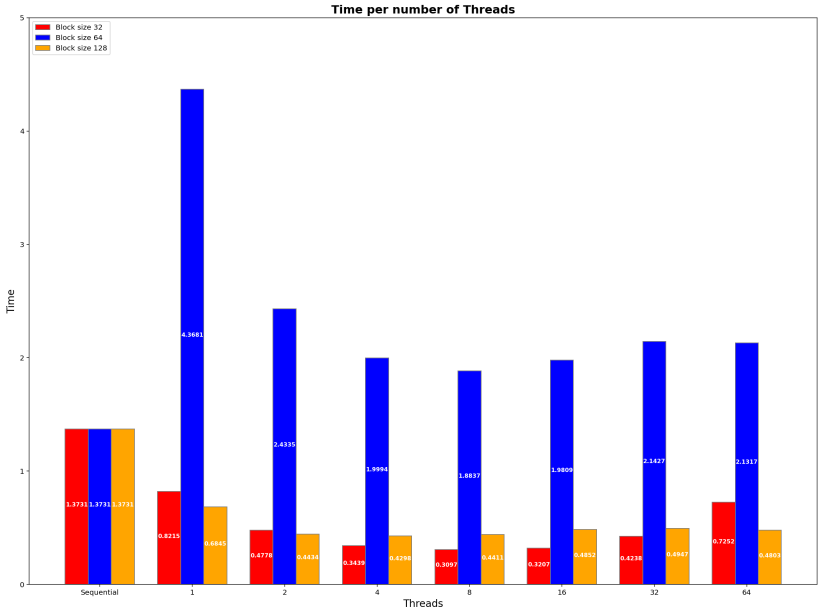
- Η κλήση της συνάρτησης στην main γίνεται με τον ακόλουθο τρόπο:

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

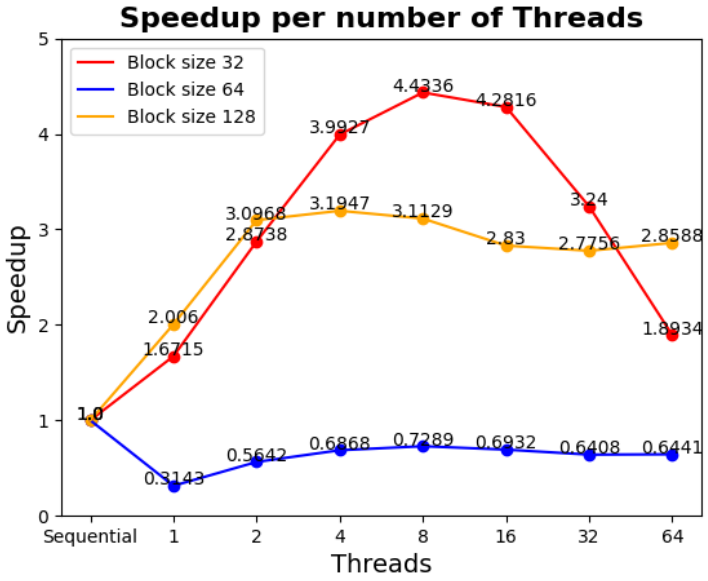
Πραγματοποιήθηκαν μετρήσεις για 3 διαφορετικά μεγέθη της παραμέτρου Block size που αποτελεί την βάση της αναδρομής, 32, 64 και 128.

Μέγεθος Πίνακα 1024

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:

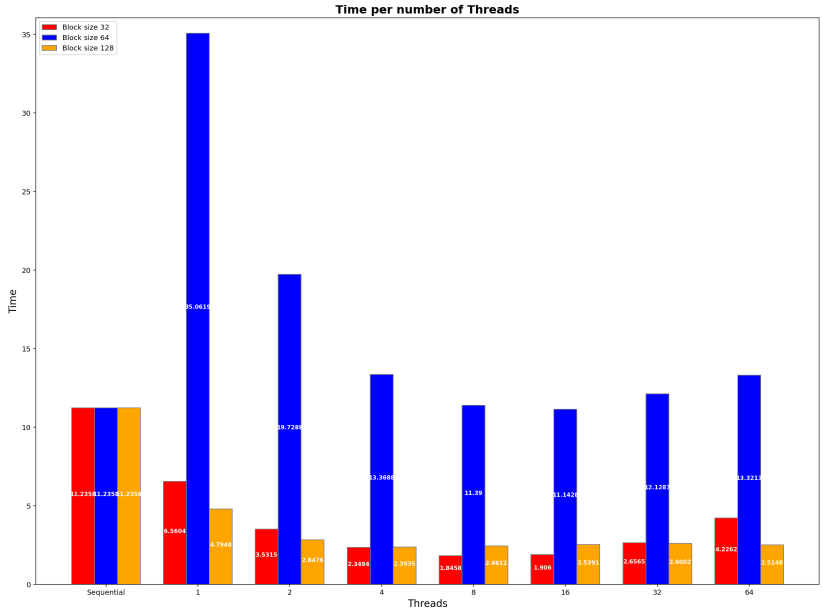


2. Το διάγραμμα speedup διαμορφώνεται ως εξής:

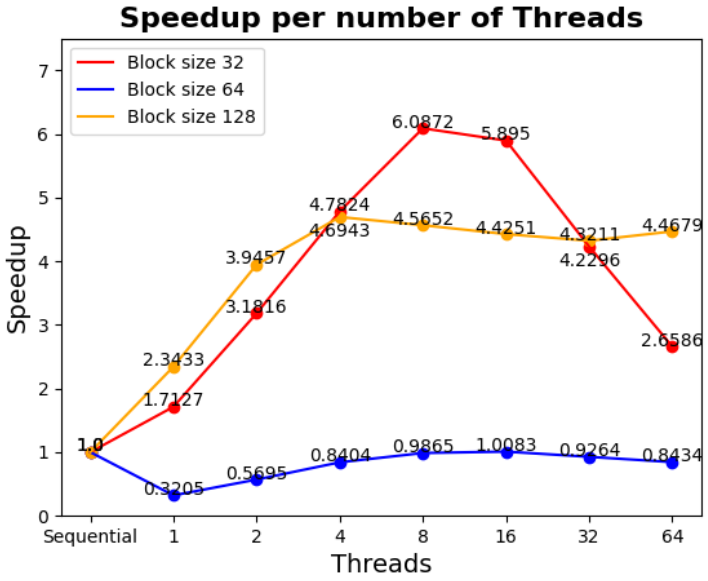


Μέγεθος Πίνακα 2048

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:

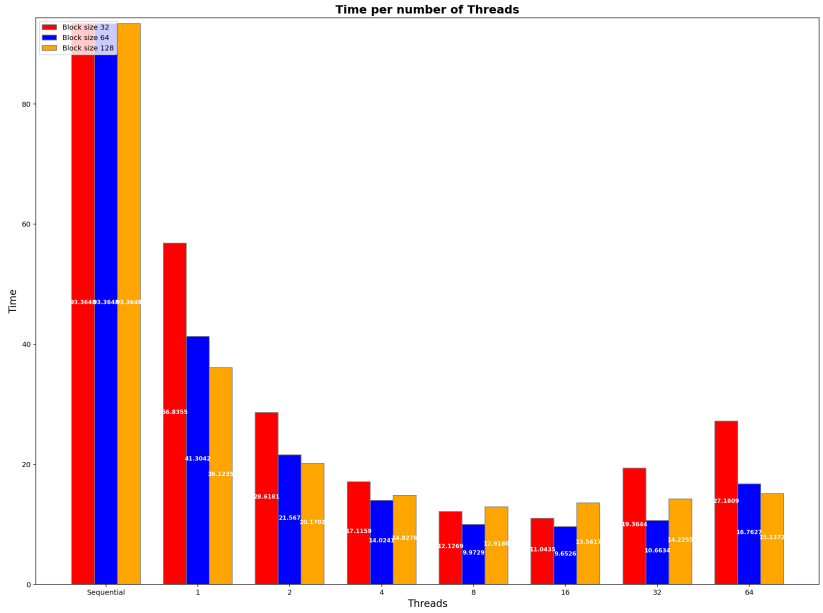


2. Το διάγραμμα speedup διαμορφώνεται ως εξής:

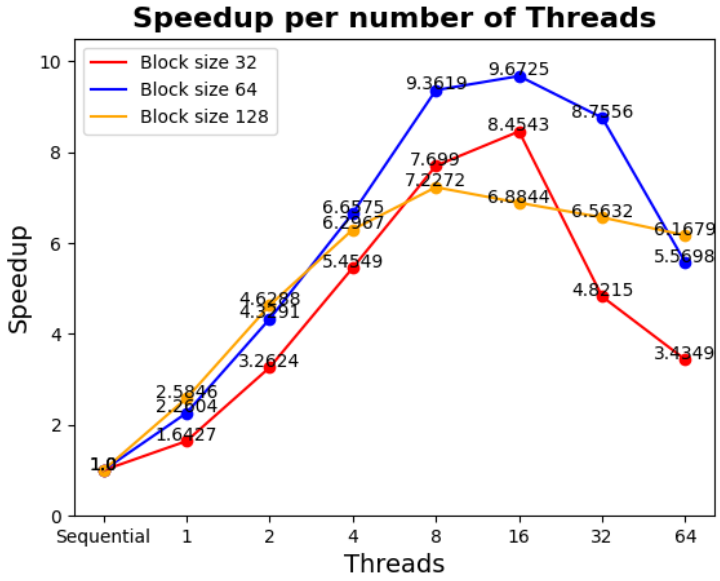


Μέγεθος Πίνακα 4096

1. Το ιστόγραμμα χρόνου διαμορφώνεται ως εξής:



2. Το διάγραμμα speedup διαμορφώνεται ως εξής:

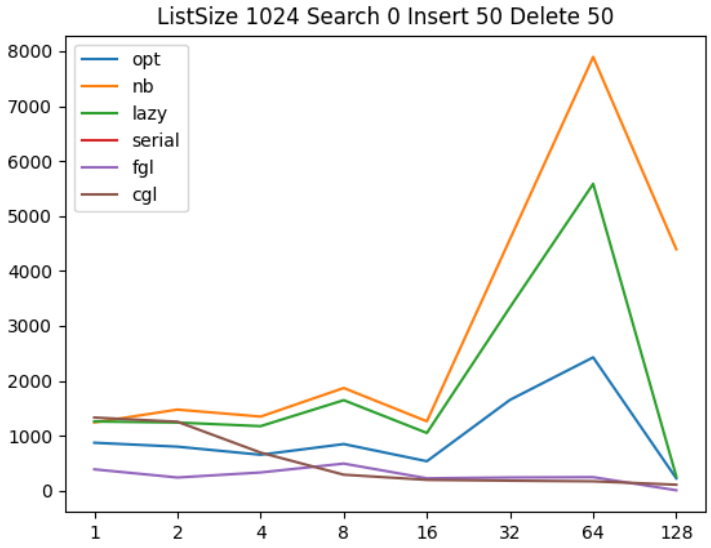


Παρατηρούμε πως το μεγαλύτερο speedup επιτυγχάνεται για 8 και 16 threadsγια όλα τα μεγέθη εισόδου. Το βέλτιστο speedup για τα μεγέθη 1024 και 2048 παρατηρείται για block size 32, ενώ για είσοδο 4096, για block size 64.

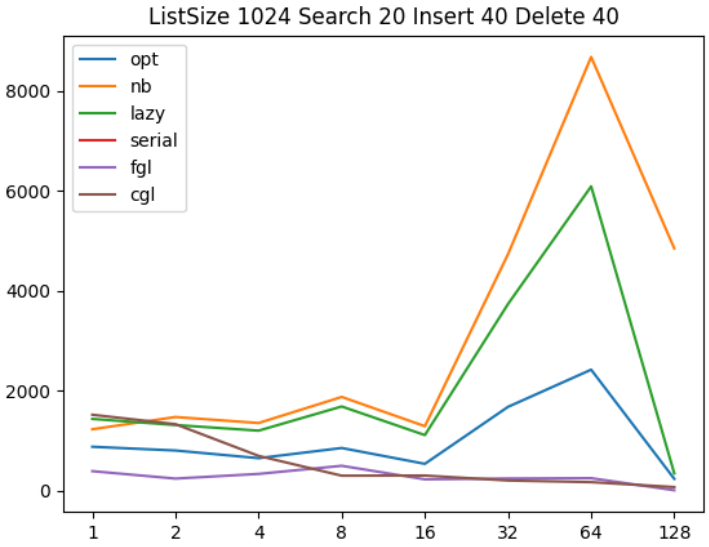
2.3 Ταυτόχρονες Δομές Δεδομένων

Διαγράμματα λίστας μεγέθους 1024 στοιχείων:

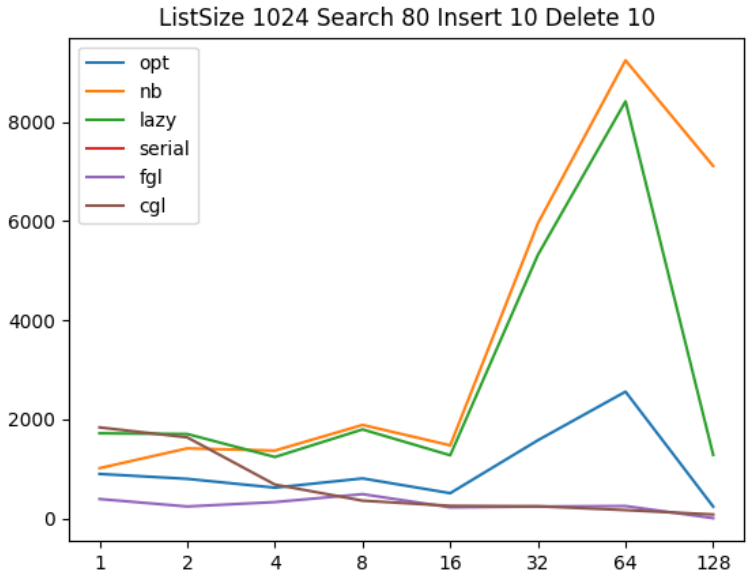
1. 0 Searches - 50 Inserts - 50 Deletes:



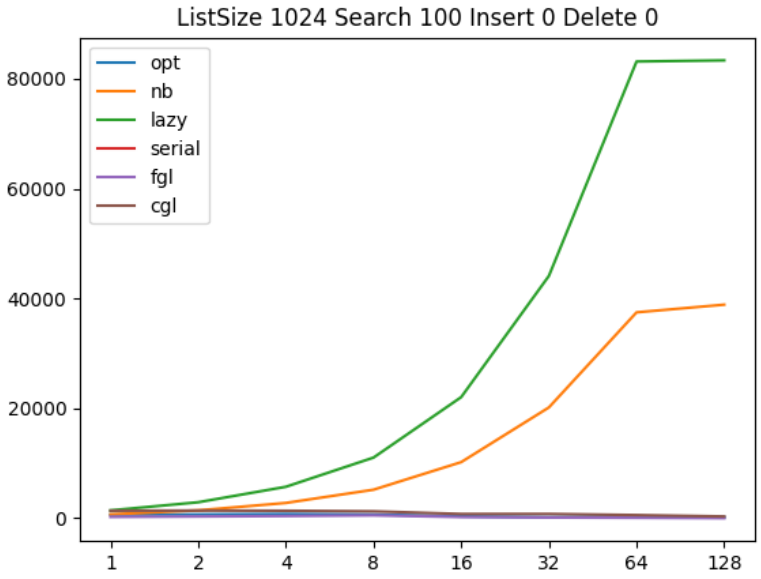
2. 20 Searches - 40 Inserts - 40 Deletes:



3. 80 Searches - 10 Inserts - 10 Deletes:

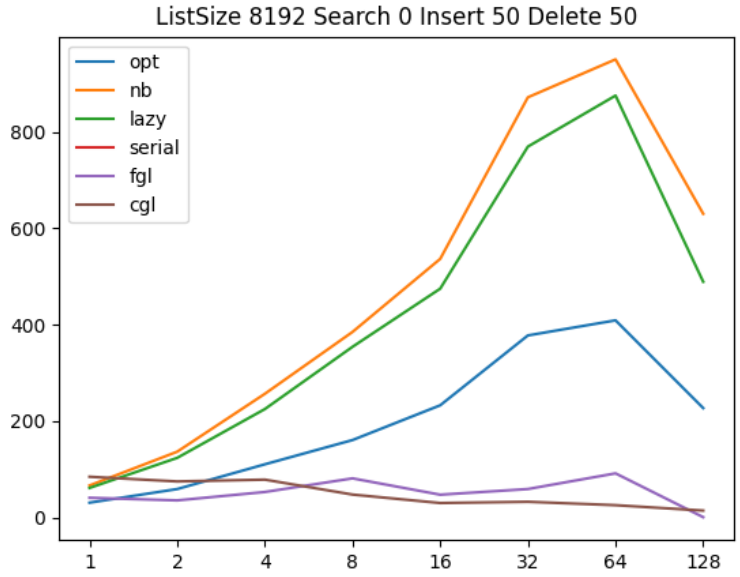


4. 100 Searches- 0 Inserts - 0 Deletes:

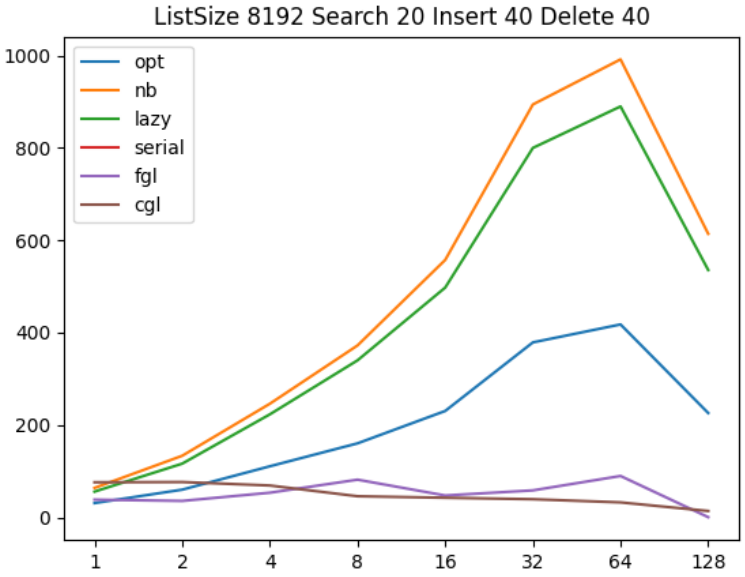


Διαγράμματα λίστας μεγέθους 8192 στοιχείων:

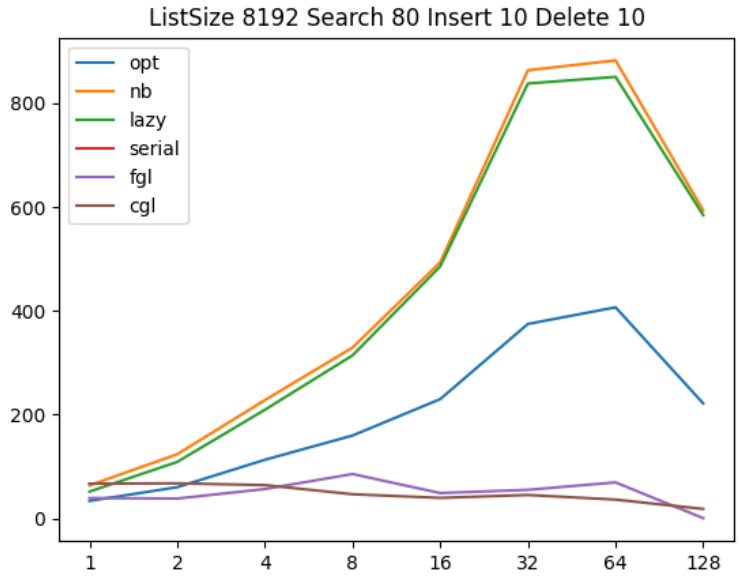
1. 0 Searches - 50 Inserts - 50 Deletes:



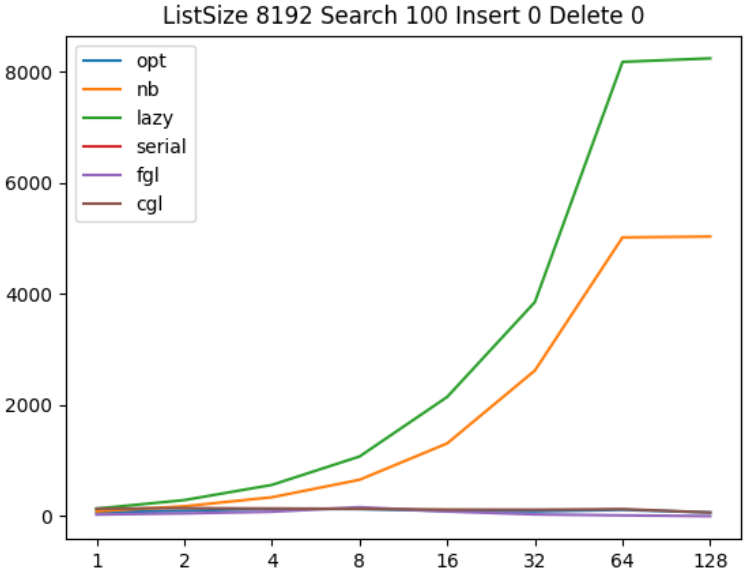
2. 20 Searches - 40 Inserts - 40 Deletes:



3. 80 Searches - 10 Inserts - 10 Deletes:



4. 100 Searches- 0 Inserts - 0 Deletes:



Coarse-grain Synchronization

Η μέθοδος αυτή αποτελεί την πιο απλή υλοποίηση συγχρονισμού για απλή συνδεδεμένη λίστα. Η μέθοδος coarse-grain δίνει ένα κοινό lock για όλη την λίστα, το οποίο διεκδικεί όποιο thread θέλει να κάνει remove(), add() ή contains(), δίνοντας την δυνατότητα μόνο σε μια μέθοδο/thread να βρίσκεται σε εκτέλεση τη δεδομένη στιγμή. Η πολυπλοκότητα αυτής της μεθόδου, όπως και το αν θα είναι starvation-free, εξαρτάται και από την υλοποίηση του lock που θα χρησιμοποιηθεί. Όμως σε περιπτώσεις πολλών διαφορετικών μεθόδων (high contention), ασχέτως από την υλοποίηση του lock, θα υπάρχει μεγάλος χρόνος αναμονής για το κάθε thread και άρα το performance του coarse-grain αλγορίθμου θα είναι χαμηλό.

Fine-grain Synchronization

Αποτελεί μια αρχική βελτίωση της μεθόδου coarse-grain. Η βασική ιδέα είναι αντί να κλειδώνει όλη η λίστα από ένα thread μέχρι να τελειώσει την μέθοδο που έχει αναλάβει (είτε είναι η add(), είτε η contains(), είτε η remove()), να χρησιμοποιούνται 2 κλειδώματα για συγκεκριμένους κόμβους. Πιο συγκεκριμένα το thread που θέλει να εκτελέσει μια μέθοδο, ξεκινάει απο την κεφαλή της λίστας, την οποία και κλειδώνει. Έπειτα κλειδώνει και τον δεύτερο κατά σειρά κόμβο. Στη συνέχεια ξεκλειδώνει την κεφαλή (προηγούμενος κόμβος) και κλειδώνει το επόμενο σε μια διαδικασία που λέγεται "hand-over-hand locking", μέχρι να συναντήσει το στοιχείο στο οποίο πάνω θέλει να εφαρμόσει μια από τις τρεις μεθόδους. Εφόσον το βρει (και άρα κρατώντας αυτό και το προηγούμενό του κλειδωμένα) εκτελεί την μέθοδο που του έχει ανατεθεί.
Αυτή η μέθοδος συγχρονισμού έχει αναμενόμενα καλύτερο performance σε σχέση με τον coarse-grain αλγόριθμο (και αντίστοιχα έχει starvation-free ιδιότητα) μιας η υλοποίηση φροντίζει κάθε φορά να κλειδώνονται δύο συγκεκριμένοι κόμβοι, σε αντίθεση με τον coarse-grain αλγόριθμο που κλείδωνε όλη τη λίστα. Ωστόσο, συγκριτικά με άλλους αλγορίθμους έχει κακή επίδοση λόγω του ότι περιλαμβάνει πολλά κλειδώματα και ξεκλειδώματα (τα οποία είναι ακριβά από άποψη χρόνου) και της απαγόρευσης της έννοιας της προσπέρασης. Πιο συγκεκριμένα, μέχρι να φτάσει ένα νήμα στον κόμβο-προορισμό, τα υπόλοιπα νοήματα που θέλουν να φτάσουν σε μεταγενέστερο κόμβο (π.χ. στο τέλος της λίστας), δεν επιτρέπεται να το προσπεράσουν και να συνεχίσουν να διατρέχουν την λίστα, έως ότου το αρχικό νήμα φτάσει στον κόμβο-προορισμό και τελειώσει την εκτέλεση του.

Optimistic Synchronization

Λόγω των περιορισμών ως προς την επίδοση της μεθόδου fine-grain, όπως αναφέρθηκαν παραπάνω, δημιουργήθηκε η optimistic μέθοδος, της οποίας η βασική ιδέα είναι να μην χρησιμοποιούνται καθόλου locks μέχρι το νήμα να βρει τον κόμβο-προορισμό. Αναλυτικότερα, η μέθοδος fine-grain είναι αρκετά ¨απαισιόδοξη¨, μιας και σε κάθε βήμα της υποθέτει ότι κάποιο άλλο thread έχει κάνει αλλαγές σε κόμβους που δραστηριοποιείται άλλος κόμβος. Ωστόσο, θεωρούμε πως τα απαισιόδοξα σενάρια είναι σπάνια και άρα προσπαθούμε να βελτιώσουμε την επίδοση της μέσης περίπτωσης, όπου διατρέχουμε την λίστα χωρίς κλειδώματα μέχρι να βρούμε τον κόμβο-στόχο. Εφόσον τον βρούμε κλειδώνουμε αυτόν και τον προηγούμενο του και τσεκάρουμε αν: α) ο προηγούμενος κόμβος είναι προσβάσιμος από την κεφαλή της λίστας (και άρα ξανατρέχουμε την λίστα από την αρχή) και β) ο προηγούμενος κόμβος δείχνει στον current κόμβο. Εαν και τα δύο ισχύουν τότε μπορεί το νήμα να εκτελέσει, σε διαφορετική περίπτωση ξεκλειδώνει τους κόμβους και ξαναπροσπαθεί από την αρχή.
Στην μέση περίπτωση, η μέθοδος αυτή λειτουργεί αποδοτικότερα από τις προηγούμενες, αφού στο traversal της λίστας δεν δημιουργείται contention και χρησιμοποιούνται τελικά λιγότερα locks. Ωστόσο, η μέθοδος αυτή λόγω του οτι διατρέχει την λίστα δύο φορές, ενδέχεται σε κάποιες περιπτώσεις να έχει χειρότερη απόδοση από το traversal της λίστας μια φορά με locks. Επίσης, αξίζει να σημείωθεί ότι ενώ τα κλειδώματα που χρησιμοποιούνται στην συγκεκριμένη περίπτωση μπορεί να είναι starvation-free, η μέθοδος συνολικά δεν είναι, μιας και μπορεί να υπάρξει περίπτωση όπου ένα thread περιμένει αν συνεχώς προστίθενται και αφαιρούνται νέοι κόμβοι.

Lazy Synchronization

Το κύριο μειονέκτημα της υλοποίησης Optimistic είναι ότι η contains() χρησιμοποιεί locks, πράγμα που δεν είναι επιθυμητό διότι η κλήση της contains() είναι πολύ πιο συχνή από αυτή των άλλων μεθόδων. Κύρια διαφοροποίηση της μεθόδου Lazy, είναι η προσθήκη ενός boolean πεδίου ονόματι marked, το οποίο δηλώνει αν ο κόμβος είναι εντός της λίστας. Πλέον λοιπόν η contains() αρκεί να ελέγξει την τιμή του boolean πεδίου για τον κόμβο για τον οποίο καλείται, και μετατρέπεται σε wait-free μέθοδο. Η μέθοδος add() ακολουθεί τα βασικά βήματα: διάσχυση, κλείδωμα, επικύρωση, προσθήκη, όπως και στην υλοποίηση Optimistic.
Η διαφοροποίηση των μεθόδων έγκειται στο ότι η επικύρωση γίνεται τώρα με χρήση του marked πεδίου, οπότε δεν χρειάζεται επιπλέον διάσχυση της λίστας. Η μέθοδος remove() ακολουθεί τα βασικά βήματα: διάσχυση, κλείδωμα, επικύρωση, αφαίρεση, όπως και στην υλοποίηση Optimistic. Και σε αυτή την περίπτωση η επικύρωση γίνεται με χρήση του marked πεδίου. Επιπλέον, η αφαίρεση αποτελείται από 2 τμήματα, την λογική και την πραγματική αφαίρεση. Η remove() θέτει το marked πεδίο ως false και μετά αλλάζει το πεδίο next του προηγούμενου κόμβου, αφαιρώντας πραγματικά τον επιθυμητό κόμβο. Ο διαχωρισμός της διαδικασίας αφαίρεσης σε 2 στάδια (με Lazy τρόπο), δίνει την δυνατότητα για επιτάχυνση της αφαίρεσης, ειδικά σε περιπτώσεις μεγάλων συνόλων αφαιρέσεων, διότι μπορούν να

γίνουν οι ταχύτερες λογικές αφαιρέσεις πρώτα, και οι πιο αργές επεμβατικές για τη δομή, φυσικές αφαιρέσεις, σε μεταγενέστερο χρόνο, όταν υπάρχει μικρότερη διαμάχη για την φυσική αλλαγή της λίστας. Κύριο μειονέκτημα της υλοποίησης Lazy είναι ότι οι μέθοδοι remove() και add() δεν είναι starvation-free και threads μπορούν να καθυστερήσουν για αόριστο χρόνο.

Non-Blocking synchronization

Στόχος της υλοποίησης Non-Blocking είναι η εξάλειψη των locks. Αυτό συμβαίνει με σύμπτιξη των πεδίων marked και next (όπως αναφέρονται στην υλοποίηση Lazy), σε μία δομή η οποία μπορεί να προσπελαστεί ατομικά. Η μέθοδος contains() παραμένει η ίδια με την Lazy υλοποίηση, διασχίζοντας όλους τους κόμβους ανεξάρτητα από το αν έχουν αφαιρεθεί λογικά ή όχι. Και η μέθοδος remove() ακολουθεί την ίδια λογική με διαχωρισμό φυσικής και λογικής αφαίρεσης, με την διαφορά ότι κατά την remove() γίνεται η λογική αφαίρεση και ύστερα δεν επιμένει για την ολοκλήρωση της φυσικής. Η υποχρέωση της φυσικής αφαίρεσης μοιράζεται μεταξύ των μεθόδων add() και remove(), οι οποίες καθώς καλούνται και διατρέχουν την λίστα για να φτάσουν στο στοιχείο που τις αφορά δεν διατρέχουν λογικά αφαιρεμένους κόμβους, αλλά όταν τους βρουν τους αφαιρούν πριν συνεχίσουν. Όπως είναι προφανές, η Non-Blocking υλοποίηση είναι lock-free.

Σχολιασμός και σύγκριση υλοποιήσεων

Λίστα 8192 στοιχείων:
Όπως προβλέπεται λόγω του τρόπου διαχείρισης κλειδωμάτων, στις υλοποιήσεις Coarse-Grain και Fine-Grain το speedup διατηρείται σε μικρές τιμές συγκριτικά με τις υπόλοιπες υλοποιήσεις, φτάνοντας τιμές κοντινές στο 1 για μεγάλο αριθμό threads. Για threads περισσότερα των 4 η Fine-Grain υλοποίηση είναι πιο αποδοτική, ενώ για 1 και 2 threads η Coarse-Grain παρουσιάζει speedup συγκρίσιμο με πιο περίπλοκες υλοποιήσεις. Η Fine-Grain υλοποίηση παρουσιάζει απρόσμενα peaks στα 8 και τα 64 threads, τα οποία εικάζουμε ότι οφείλονται στην εσωτερική αρχιτεκτονική του sandman.
Αμέσως καλύτερη συμπεριφορά, ξεπερνώντας το speedup των παραπάνω υλοποιήσεων κατά 100πλάσιες φορές, έχει η Optimistic υλοποίηση. Αυτό ισχύει σε όλες τις περιπτώσεις εκτός αυτής των 100 searches, όπου παρουσιάζει speedup ανάλογο των Fine-Grain και Coarse-Grain υλοποιήσεων, λόγω του ότι η μέθοδος contains() χρειάζεται locks, ενώ τρέχει τη συνάρτηση validate. Σε γενικά πλαίσια, δεύτερη αποδοτικότερη υλοποίηση είναι η Lazy, η οποία παρουσιάζει speedup τουλάχιστον 1,5 φορές μεγαλύτερο της υλοποίησης Optimistic σε όλες τις περιπτώσεις εκτός από την 100 searches-0 Inserts-0 Deletes.
Την καλύτερη απόδοση έχει η υλοποίηση Non-Blocking όπως ήταν αναμενόμενο, με τιμές speedup τάξης μεγέθους ανάλογης με την Lazy υλοποίηση. Η Non-Blocking υλοποίηση υστερεί μόνο στην περίπτωση των Searches όπου η κατασκευή των ατομικών δομών marked-next, επιφέρει καθυστερήσεις, παρόλο που η ίδια η
αναζήτηση λειτουργεί παρόμοια με την Lazy υλοποίηση.
Περιμέναμε τέτοια συμπεριφορά στην περίπτωση των 100 searches, διότι οι αναζητήσεις είναι η λειτουργία για την οποία ενδύκνειται η Lazy υλοποίηση. Παρατηρώντας και τα υπόλοιπα διαγράμματα, όταν οι αναζητήσεις είναι περισσότερες από τις άλλες λειτουργίες, η αποδοτικότητα της Lazy υλοποίησης πλησιάζει αυτήν της Non-Blocking, ενώ όσο οι αναζητήσεις μειώνονται και οι άλλες λειτουργίες αυξάνονται, τόσο απομακρύνεται και η Lazy από την Non-Blocking.
Στις υλοποιήσεις Optimistic, Lazy και Non-Blocking, η απόδοση παρουσιάζει μια αναμενόμενη ανοδική πορεία μέχρι τα 64 threads, όμως μειώνεται δραματικά στα 128 threads. Ωστόσο, κάτι τέτοιο είναι λογικό, διότι τότε υπάρχει καθυστέρηση λόγω ανάγκης διαμοιρασμού του υλικού (2 threads ανά επεξεργαστή).

Λίστα 1024 στοιχείων:
Η σχέση αποδοτικότητας μεταξύ των υλοποιήσεων είναι παρόμοια με την λίστα 8192 στοιχείων. Παρατηρούνται οι εξής ειδοποιοί διαφορές:

- Παρουσιάζεται μια απρόσμενη πτώση στο speedup στα 16 threads για όλες τις υλοποιήσεις εκτός της Coarse-Grain, η οποία θεωρούμε ότι οφείλεται στην εσωτερική αρχιτεκτονική του sandman.

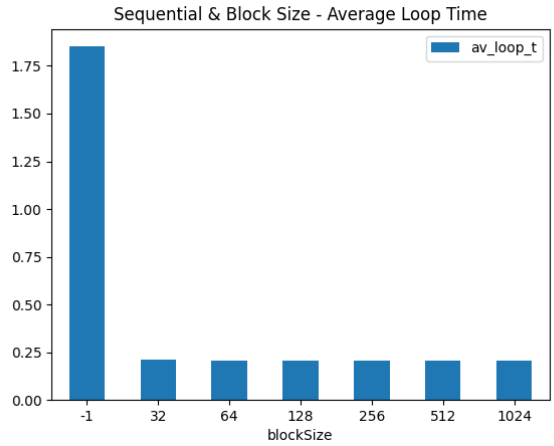
- Οι τάξεις μεγέθους speedup που υπολογίζονται είναι 100πλάσιες της λίστας 8192 στοιχείων παρόλο που ο αριθμός των λειτουργιών είναι ο ίδιος. Αυτό

συμβαίνει διότι όλες οι λειτουργείες πρέπει να διατρέξουν την λίστα από την αρχή ώστε να φτάσουν στο στοιχείο πάνω στο οποίο θέλουν να ενεργήσουν.

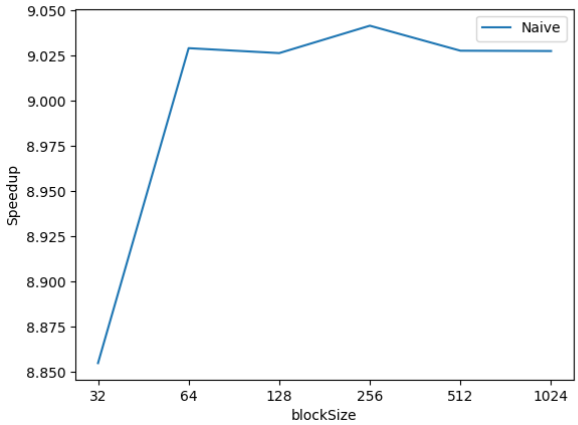
3 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε επεξεργαστές γραφικών

Naive Version

1. Είναι εμφανές από το παρακάτων διάγραμμα μέσου χρόνου εκτέλεσης ανά loop ότι η παράλληλη έκδοση είναι σημαντικά γρηγορότερη από την σειριακή. Πιο συγκεκριμένα, η παράλληλη έκδοση είναι 7 φορές γρηγορότερη από την σειριακή για όλα τα block sizes που δοκιμάστηκαν.



Σχήμα 1: Barplot διάγραμμα μέσου χρόνου ανά loop για σειριακή και naive έκδοση

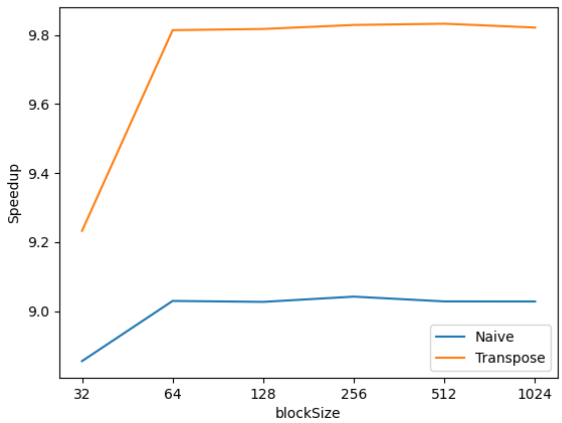


Σχήμα 2: Διάγραμμα Speedup για naive έκδοση

2. Παρόλο που δεν είναι προφανές στο διάγραμμα μέσου χρόνου ανά loop, στο διάγραμμα speedup παρατηρείται ότι οι διαφορετικές επιλογές block size επηρεάζουν την επίδοση του προγράμματος. Η χειρότερη επίδοση παρατηρείται για block size 32 threads με speedup 8.86. Για block sizes 64 έως 1024 threads το speedup κυμαίνεται σε τιμές κοντά στο 9, με ένα peak στην περίπτωση του block size 256 threads, με speedup 9.04.
Η GPU ομαδοποιεί προσβάσεις στην καθολική μνήμη στην ώστε να βελτιστοποιήσει τον συνολικό χρόνο προσβάσεων. Για κάθε block, ταυτόχρονες προσβάσεις μνήμης από διαφορετικά threads του block συνδυάζονται σε μία συνολική πρόσβαση μνήμης που εξυπηρετεί όλα αυτά τα threads. Οι προσβάσεις στην καθολική μνήμη γίνοται σε τμήματα συγκεκριμένου αριθμού bytes, στα οποία έχει οργανωθεί. Η συνένωση των αναφορών στην μνήμη εξαρτάται από την απόσταση των διευθύνσεων της φυσικής μνήμης που προσπελαύνουν τα threads.
Αυτός είναι και ο λόγος που παρατηρείται μεγάλη βελτίωση επίδοσης μεταξύ των block sizes 32 και 64 threads, καθώς στην πρώτη περίπτωση δεν πραγματοποιούνται αρκετές ταυτόχρονες προσπελάσεις μνήμης για να χρησιμοποιηθεί αποδοτικα ο μηχανισμός πρόσβασης στην καθολική μνήμη ενώ στην δεύτερη πραγματοποιούνται. Θεωρούμε ότι για block size μεγαλύτερο των 64 threads δεν υπάρχει δυνατότητα περαιτέρω βελτίωσης.

Transpose Version

1. Η έκδοση transpose είναι εμφανώς γρηγορότερη από την naive.



Σχήμα 3: Διάγραμμα Speedup για naive και transpose έκδοση

Για 32 threads block Size, η Transpose εμφανίζει speedup 9.23 έναντι του speedup 8.855 της naive. Για block size μεγαλύτερο του 64 παρουσιάζεται ακόμα μεγαλύτερη βελτίωση, με τιμές speedup που κυμαίνονται μεταξύ

9.81-9.83 έναντι των 9.01-9.04 της naive. Η βελτίωση της επίδοσης με την αύξηση του block Size φαίνεται να ακολουθεί παρόμοια συμπεριφορά με την ναιε έκδοση, με την μόνη διαφορά ότι η αύξηση του speedup κατά τη μετάβαση από τα 32 στα 64 threads είναι αρκετά μεγαλύτερη στην transpose. Αυτό οφείλεται στην αποδοτικότερη πρόσβαση στη μνήμη που επιτυγχάνει η transpose, σε συνδυασμό με τον μεγαλύτερο αριθμό ταυτόχρονων προσβάσεων μνήμης ανά block που προσφέρει το block size 64 threads.

2. Η διαφορά στην επίδοση μεταξύ των δύο εκδόσεων οφείλεται στον τρόπο πρόσβασης στην καθολική μνήμη από τον CUDA driver. Όπως προαναφέρθηκε, η GPU ομαδοποιεί ταυτόχρονες προσβάσεις μνήμης από διαφορετικά threads του ίδιου block, πραγματοποιώντας μία πρόσβαση στη μνήμη αντί πολλών ανεξάρτητων, στον βαθμό που επιτρέπει η απόσταση των διευθύνσεων μνήμης που προσπελαύνονται. Πιο συγκεκριμένα, ανάλογα με την υπολογιστική ικανότητα, το CUDA hardware μπορεί να ομαδοποιήσει προσβάσεις στη μνήμη που απέχουν συγκεκριμένη ευθυγραμμισμένη απόσταση bytes (π.χ. 128 byte segments για υπολογιστική ικανότητα μεγαλύτερη του 1.2). Οι αλλεπάλληλες προσβάσεις μνήμης στον πυρήνα της GPU καλούνται στην συνάρτηση εύρεσης κοντινότερου κέντρου *euclid_dist_2()* (ή *euclid_dist_transpose()* για την transpose έκδοση).
Ο πίνακας που προσπελάσεται είναι ο πίνακας clusters που περιέχει τις συντεταγμένες των κέντρων των clusters και ο πίνακας objects που περιέχει τις συντεταγμένες των αντικειμένων.
Στην περίπτωση της naive έκδοσης, και οι δύο πίνακες είναι μίας διάστασης και αποτελούνται από numClusters και numObjs ανάλογα σειριακά συντεταγμένες συστοιχίες μεγέθους numCoords. Η προσπέλαση του πίνακα γίνεται σειριακά, όπως είναι αποθηκευμένες οι συντεταγμένες των στοιχείων στην μνήμη. Αυτό σημαίνει πως, στην naive, οι ταυτόχρονες προσβάσεις μνήμης απέχουν τουλάχιστον $\#συντεταγμένων$ στοιχεία των πινάκων, δηλαδή $\#συντεταγμένων * \{bytes\ float\} = 4*\#συντεταγμένων$ bytes.
Οι διευθύνσεις που προσπελαύνονται δεν επιτρέπουν μία ολική πρόσβαση μνήμης ανά block λόγω της μεγάλης απόστασής τους, οπότε η συσκευή αναγκάζεται να κάνει πολλαπλλες προσβάσεις μνήμης, ενώ στην περίπτωση της transpose έκδοσης, οι δύο πίνακες αποτελούνται από numCoords σειριακά συντεταγμένες συστοιχίες numClusters ή numObjs στοιχείων ανάλογα. Η προσπέλαση του πίνακα γίνεται σειριακά, όμως οι ταυτόχρονες προσβάσεις μνήμης απέχουν 1 στοιχείο float, δηλαδή 4 bytes. Οι διευθύνσεις μνήμης που προσπελαύνονται ταυτόχρονα είναι κοντά στην φυσική μνήμη, οπότε το hardware μπορεί να συνδυάσει τις προσβάσεις στη μνήμη και να εξυπηρετήσει πολύ μεγαλύτερο αριθμό threads.

Ακολουθούν οι υλοποιήσεις των συναρτήσεων *euclid_dist_2()* και *euclid_dist_transpose()* όπου μπορεί να παρατηρηθεί η διαφορετική οργάνωση των πινάκων objects και clusters στην μνήμη.

```
__host__ __device__ inline static
float euclid_dist_2_transpose(int numCoords,
                    int    numObjs,
                    int    numClusters,
                    float *objects,  //[numCoords][numObjs]
                    float *clusters, //[numCoords][numClusters]
                    int    objectId,
                    int    clusterId)
{
    int i;
    float ans=0.0;

        /* TODO: Calculate the euclid_dist of elem=objectId of objects
        ↪  from elem=clusterId from clusters, but for column-base
        ↪  format!!! */
    for (i = 0; i < numCoords; i++)
        ans += (objects[objectId+i*numObjs] -
        ↪  clusters[clusterId+numClusters*i]) *
        ↪  (objects[objectId+i*numObjs] -
        ↪  clusters[clusterId+numClusters*i]);

    return(ans);
}
```

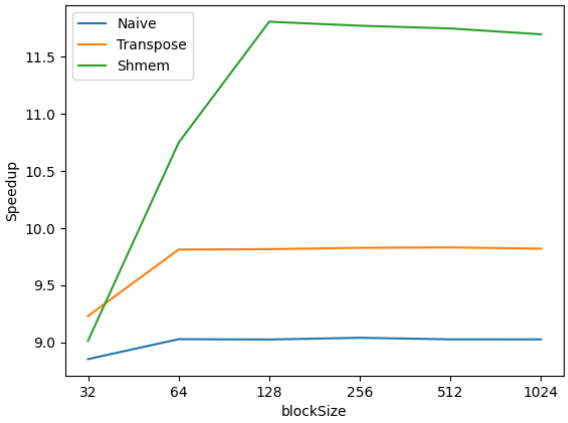
```
/* square of Euclid distance between two multi-dimensional points */
__host__ __device__ inline static
float euclid_dist_2(int    numCoords,
                    int    numObjs,
                    int    numClusters,
                    float *objects,     // [numObjs][numCoords]
                    float *clusters,    // [numClusters][numCoords]
                    int    objectId,
                    int    clusterId)
{
    int i;
    float ans=0.0;

    for(i=0; i<numCoords; i++)
        ans += (objects[objectId*numCoords+i] -
        ↪  clusters[clusterId*numCoords+i]) *
        ↪  (objects[objectId*numCoords+i] -
        ↪  clusters[clusterId*numCoords+i]);

    return(ans);
}
```

Shared Memory Version

1. Είναι προφανές ότι στην γενική περίπτωση η επίδοση της shared έκδοσης είναι πολύ καλύτερη της transpose και της naive.



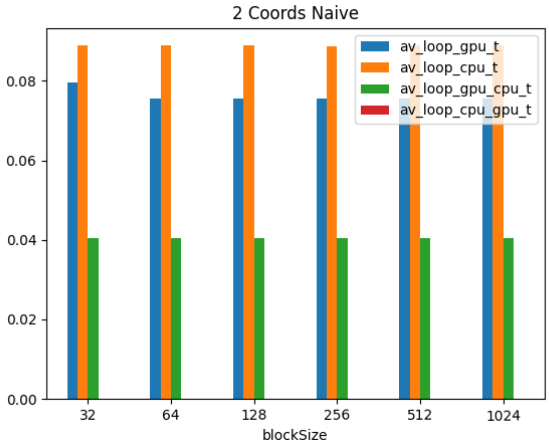
Σχήμα 4: Διάγραμμα Speedup για naive, transpose και shared memory έκδοση

Συγκεκριμένα, εμφανίζει speedup με peak 11.81.
Ωστόσο η συμπεριφορά της Shared Memory έκδοσης με την αύξηση του blockSize δεν ακολουθεί το ίδιο μοτίβο με τις προηγούμενες εκδόσεις. Για block size 32 threads το speedup της Shared Memory παίρνει τη μικρότερη τιμή του (9.02), μικρότερη και από αυτή της Transpose έκδοσης για το ίδιο block size. Για block size 64 παρουσιάζει δραματική αύξηση την οποία συνεχίζει με μικρότερο ρυθμό για block size 128. Τέλος, για blocksize μεγαλύτερα των 128 παρουσιάζεται σταθερή μείωση speedup με χαμηλό ρυθμό.
Στην έκδοση Shared Memory αξιοποιείται για την βελτίωση απόδοσης του k-means η κοινή μνήμη που μοιράζονται τα threads ενός block. Συγκεκριμένα το thread με id =1 σε κάθε block αντιγράφει στην κοινή μνήμη τον πίνακα clusters που περιέχει τις συντεταγμένες των κέντρων των clusters. Με αυτό τον τρόπο οι έλεγχοι της συνάρτησης *euclid_dist_2_transpose()*
πραγματοποιούνται γρηγορότερα καθώς πλέον τα δεδομένα είναι διαθέσιμα στην κοινή μνήμη, της οποίας η προσπέλαση είναι πολύ πιο γρήγορη από την global, λόγω του μεγαλύτερου bandwidth που προσφέρει. Επιπλέον, για την βελτίωση της επίδοσης χρησιμοποιείται και η αποδοτικότερη διαχείριση δεδομένων στην global μνήμη από την transpose έκδοση.
Είναι προφανές πως για block size 32 threads, τα threads που μοιράζονται την ίδια κοινή μνήμη δεν είναι αρκετά ώστε το άθροισμα χρόνου αντιγραφής του πίνακα Clusters με τον συνολικό χρόνο πρόσβασης στην κοινή μνήμη να

είναι μικρότερο του συνολικού χρόνου πρόσβασης στην καθολική μνήμη, αν ακολουθούσαμε την έκδοση transpose. Παρ' όλα αυτά με την αύξηση του block size, ο αριθμός των threads που έχει πρόσβαση στην ίδια κοινή μνήμη αυξάνεται, μειώνοντας τον συνολικό απαιτούμενο χρόνο για προσβαση στα στοιχεία του πίνακα Clusters και αυξάνοντας σημαντικά την απόδοση της GPU.

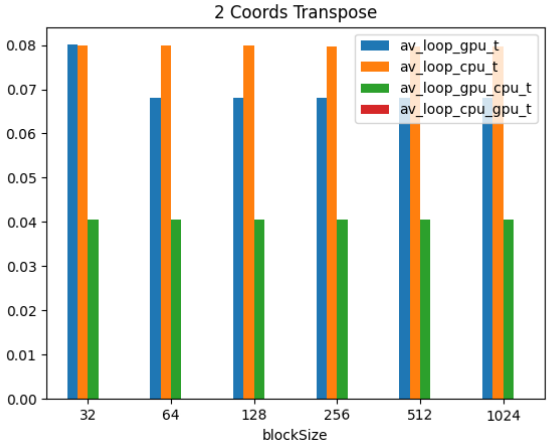
Σύγκριση Υλοποιήσεων

1. Είναι εμφανές ότι κατά βάση, στο πρόβλημα k-means, το μεγαλύτερο μέρος του χρόνου εκτέλεσης καταλαμβάνεται από τον χρόνο επεξεργασίας στην CPU.



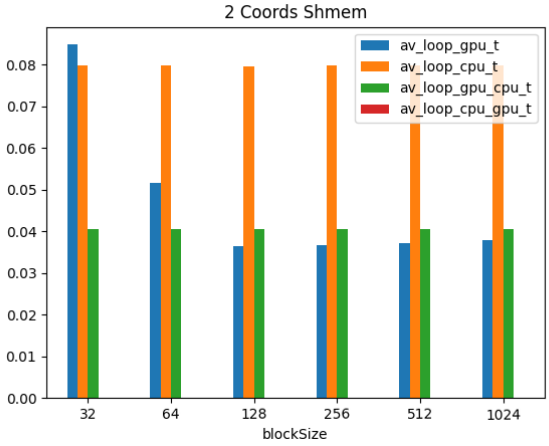
Σχήμα 5: Διάγραμμα μέσου χρόνου επεξεργασίας CPU, επεξεργασίας GPU, επικοινωνίας CPU-GPU και επικοινωνίας GPU-CPU για Naive έκδοση

.



Σχήμα 6: Διάγραμμα μέσου χρόνου επεξεργασίας CPU, επεξεργασίας GPU, επικοινωνίας CPU-GPU και επικοινωνίας GPU-CPU για Transpose έκδοση

.



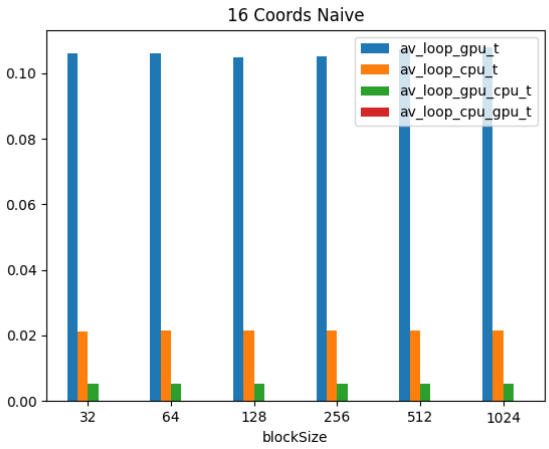
Σχήμα 7: Διάγραμμα μέσου χρόνου επεξεργασίας CPU, επεξεργασίας GPU, επικοινωνίας CPU-GPU και επικοινωνίας GPU-CPU για Shared Memory έκδοση

.

Οι περιπτώσεις του block size 32 threads, στις εκδόσεις shared memory και transpose παρεκκλίνουν από την συμπεριφορά αυτή, όμως και στις δύο εκδόσεις, οι μέσοι χρόνοι εκτέλεσης για μεγαλύτερα block sizes επιβεβαιώνουν την αρχική πρόταση. Αμέσως μεγαλύτερο τμήμα του συνολικού μέσου χρόνου εκτέλεσης καταλαμβάνουν οι πράξεις που λαμβάνουν χώρα

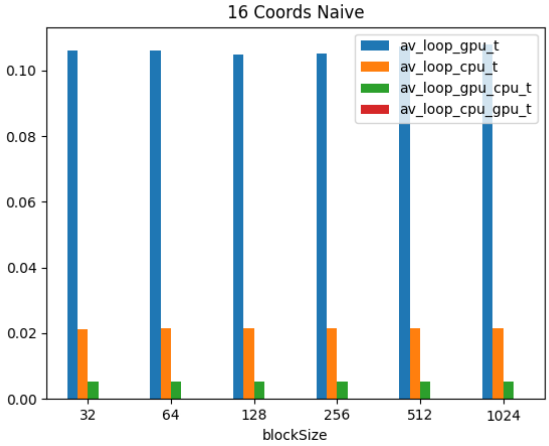
αποκλειστικά στην GPU, ενώ σχεδόν τον μισό από αυτόν τον χρόνο καταλαμβάνουν οι μεταφορές στοιχείων από την GPU στην CPU. Ο χρόνος που απαιτείται για την μεταφορά στοιχείων από την CPU στην GPU είναι αναλογικά πολύ μικρός, τόσο που δεν είναι καν εμφανής στα διαγράμματα σύγκρισης χρόνου.
Η συνολική επίδοση του iterative μέρους εμποδίζεται από το υπολογιστικό τμήμα που αναλαμβάνει η CPU, δηλαδή την ανανέωση των κέντρων των clusters. Ενώ με την αύξηση του block size η GPU αξιοποιείται καλύτερα και το πρόγραμμα είναι γρηγορότερο, ο χρόνος που απαιτεί η CPU παραμένει σταθερός και αρκετά μεγάλος, όπως παρατηρούμε στο διάγραμμα μέσων χρόνων εκτέλεσης ανά loop.

2. **Configuration Size, Coords, Clusters, Loops = 256,16,16,10**



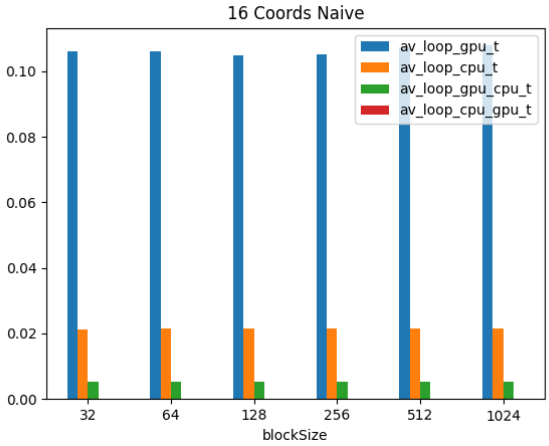
Σχήμα 8: Διάγραμμα μέσου χρόνου επεξεργασίας CPU, επεξεργασίας GPU, επικοινωνίας CPU-GPU και επικοινωνίας GPU-CPU για Naive έκδοση και νέο configuration

.



Σχήμα 9: Διάγραμμα μέσου χρόνου επεξεργασίας CPU, επεξεργασίας GPU, επικοινωνίας CPU-GPU και επικοινωνίας GPU-CPU για Transpose έκδοση και νέο configuration

.



Σχήμα 10: Διάγραμμα μέσου χρόνου επεξεργασίας CPU, επεξεργασίας GPU, επικοινωνίας CPU-GPU και επικοινωνίας GPU-CPU για Shared Memory έκδοση και νέο configuration

.

Στο νέο αυτό configuration το συνολικό μέγεθος δεδομένων εισόδου παραμένει ίδιο όμως οι συντεταγμένες που χρησιμοποιούνται για την περιγραφή κάθε αντικειμένου οκταπλασιάζονται. Συνεπώς ο αριθμός αντικειμένων

υπο-οκταπλασιάζεται . Υπάρχουν λιγότερα threads ανά block όμως περισσότερες πράξεις ανά τηρεαδ κατά την κλήση της *euclid_dist_2_transpose()* ή *euclid_dist_2()*, ανάλογα την υλοποίηση.
Συνολικά ο χρόνος εκτέλεσης για κάθε έκδοση μειώνεται. Και πάλι, το μεγαλύτερο μέρος του απαιτούμενου χρόνου καταλαμβάνεται από τις εργασίες στην CPU, για κάθε έκδοση και κάθε block size. Μάλιστα σε αυτή την περίπτωση ο χρόνος CPU είναι πολύ μεγαλύτερος των υπολοίπων. Τον αμέσως περισσότερο χρόνο απαιτούν οι εργασίες που
εκτελούνται αποκλειστικά στην GPU, με χρόνους συχνά μικρότερους από τους μισούς της αντίστοιχης έκδοσης για 2 συντεταγμένες ανά αντικείμενο. Αρκετά μικρότερος είναι και ο απαιτούμενος χρόνος μεταφοράς δεδομένων από την GPU προς την CPU, λόγω υπο-οκταπλασιασμού του πίνακα membership που μεταφέρεται, ενώ ο χρόνος μεταφοράς δεδομένων από CPU προς GPU παραμένει αμελητέος.
Αυτή η συμπεριφορά είναι προβλεπόμενη καθώς με το συγκεκριμένο configuration αξιοποιούνται καλύτερα οι ικανότητες της GPU για ταχείς αριθμητικές πράξεις.
Δεν θεωρούμε ότι η shared υλοποίηση είναι κατάλληλη για την επίλυση του kmeans για arbitrary configurations. Το μέγεθος της κοινής μνήμης ανά Steaming Multiprocessor (SM) είναι σταθερό και ανάλογο της GPU που χρησιμοποιείται. Αυξάνοντας τον αριθμό βλοςκς που καλείται να υποστηρίξει η GPU, αυξάνεται ο αριθμός blocks ανά SM, και συνεπώς το μέγεθος της διαθέσιμης κοινής μνήμης για κάθε ένα τέτοιο block. Υπάρχει κάποιο configuration, για το οποίο έχουμε μεγάλο αριθμό Objects, τέτοιο ώστε ο αριθμός blocks να προκαλεί σημαντική μείωση του διαθέσιμου χώρου κοινής μνήμης ανά block, και μεγάλο αριθμο Clusters, τέτοιο ώστε ο πίνακας κέντρων των Clusters να μην χωράει στην κοινή μνήμη.

4 Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

Λεπτομέρειες υλοποίησης

Jacobi

- Περνάμε στο κάθε process ένα block του αρχικού πίνακα το οποίο αντιγράφει το κάθε process στον τοπικό πίνακα, ξεκινώντας από τη θέση [1][1] ώστε να αφήσουμε μια κενή γραμμή και στήλη για τα δεδομένα που θα χρειαστεί να στείλουν τα γειτονικά processes. Για να το κάνουμε αυτό χρησιμοποιούμε custom datatypes του MPI.

- Ο διαμοιρασμός των δεδομένων από την διεργασία 0 γίνεται με

```
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset,
↪  global_block, &(u_current[1][1]), 1, local_block,
↪  0, CART_COMM);
```

- Οι buffers που χρησιμοποιούνται για την επικοινωνία των διεργασιών ορίζονται εκτός του for.

- Ορίζουμε 4 τιμές για κάθε block, 2 για το εύρος των στηλών (j_min, j_max) και 2 για το εύρος των γραμμών i_min, i_max, πάνω στις οποίες θα κάνουμε τους υπολογισμούς της εξίσωσης θερμότητας, ανάλογα με την θέση του block στον αρχικό πίνακα. Για να υπολογίσουμε τις τιμές αυτές λαμβάνουμε υπόψιν αν χρησιμοποιούμε padding και αν βρισκόμαστε σε συνοριακό block.

- Η αντιγραφή των δεδομένων που θα χρειαστεί να στείλει η κάθε διεργασία στους αντίστοιχους buffers (ανάλογα με το που θα χρειαστεί να στείλει τα δεδομένα βάση των καρτεσιανών συντεταγμένων) γίνεται με τον εξής τρόπο:

```
for (i = 0; i < local[0]; i++) {
    right_send[i] = u_current[i+1][local[1]];
    left_send[i] = u_current[i+1][1];
}

for (i = 0; i < local[1]; i++) {
    up_send[i] = u_current[1][i+1];
    down_send[i] = u_current[local[0]][i+1];
}
```

- Για την επικοινωνία των διεργασιών χρησιμοποιούμε τα MPI_Isend και MPI_Irecv, όπου στέλνουμε στα γειτονικά blocks (στην περίπτωση που υπάρχουν) τους buffers. Επίσης ορίζουμε έναν counter, ώστε το κάθε block να γνωρίζει πόσα send requests και πόσα receive requests να περιμένει, εφόσον στο τέλος χρησιμοποιούμε *Waitall* .

```
if (north != -1) {
//  Send north
    MPI_Isend(up_send,local[1],MPI_DOUBLE,north,north,
    CART_COMM, &send_requests[counter]);
    MPI_Irecv(up_receive,local[1],MPI_DOUBLE,north,rank,
    CART_COMM, &receive_requests[counter]);
    counter++;
}
.
.
(Same for east, west, south)
MPI_Waitall(counter,receive_requests,MPI_STATUSES_IGNORE);
MPI_Waitall(counter,send_requests,MPI_STATUSES_IGNORE);
```

- Έπειτα κάνουμε την αντιγραφή των δεδομένων από τους buffers στους τοπικούς πίνακες και μετά κάνουμε τον υπολογισμό

- Στην περίπτωση που κάνουμε έλεγχο σύγκλισης πρώτα κάνουμε τον έλεγχο με την συνάρτηση *converge()*, όπως μας δίνεται στο utils.h και έπειτα χρήσιμοποιούμε *Allreduce* των τοπικών μεταβλητών converge σε μια global μεταβλητή global_converge, με τη χρήση του MPI_MIN operation, όπου αν όλα τα processes επιστρέψουν 1 τότε έχουμε σύγκλιση, αλλιώς αν έστω και ένα process επιστρέψει 0, τότε δεν έχουν συγκλίνει όλα τα processes, οπότε και συνεχίζουμε να τρέχουμε το for loop.

```
if (t%C==0) {
    gettimeofday(&tss,NULL);
    //************TODO*************//
    /*Test convergence*/
    // Local conv check, converged = 1 if conv else
    ↪  converged = 0
    converged = converge(u_previous, u_current, i_min,
    ↪  i_max-1, j_min, j_max-1);
    MPI_Allreduce(&converged, &global_converged, 1,
    ↪  MPI_INT, MPI_MIN, CART_COMM);
    gettimeofday(&tsf,NULL);
    tconv+=(tsf.tv_sec-tss.tv_sec)+
    (tsf.tv_usec-tss.tv_usec)*0.000001;
}
```

- Αφού τελειώσουμε τους υπολογισμούς, συγκεντρώνουμε όλα τα δεδομένα των τοπικών blocks στον αρχικό global πίνακα U.

```
MPI_Gatherv(&u_current[1][1], 1, local_block, &U[0][0],
↪  scattercounts, scatteroffset, global_block, 0,
↪  CART_COMM);
```

Gauss-Seidel

Στην υλοποίηση Gauss-Seidel η λογική παραμένει η ίδια με την Jacobi με μόνη διαφορά την επικοινωνία των processes μέσα στο for loop. Αυτό συμβαίνει γιατί τώρα χρησιμοποιούμε για τους υπολογισμούς του κάθε σημείου δεδομένα από προηγούμενη και τωρινή χρονική στιγμή. Πιο συγκεκριμένα, ο κώδικας είναι παρόμοιος με αυτόν από την υλοποίηση του Jacobi, μόνο που εδώ το κάθε block πριν τον υπολογισμό στέλνει δεδομένα στα πάνω και αριστερά του block (εφόσον αυτά υπάρχουν) και περιμένει να πάρει από τα δεξιά και τα κάτω blocks. Έπειτα ενημερώνει το κάθε block τους πίνακες u_previous και u_current και περιμένει να λάβει τα δεδομένα από τα πάνω και τα αριστερά blocks. Στη συνέχεια κάνει τους υπολογισμούς για το κάθε στοιχείο και έπειτα στέλνει με τη σειρά του τα δεδομένα κάτω και δεξιά της νέας χρονικής στιγμής (μιας και έχει ήδη γίνει ο υπολογισμός) για να τα χρησιμοποιήσουν τα γειτονικά blocks για τους δικούς τους υπολογισμούς.

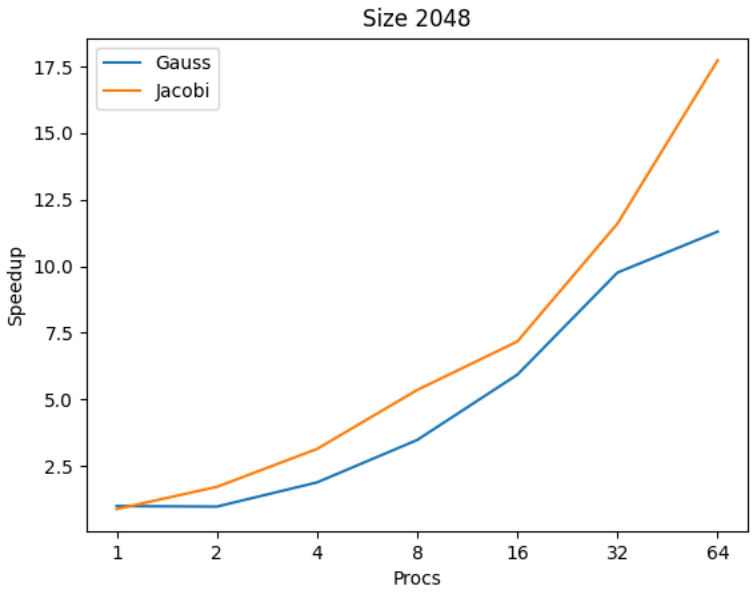
Μετρήσεις με έλεγχο σύγκλισης

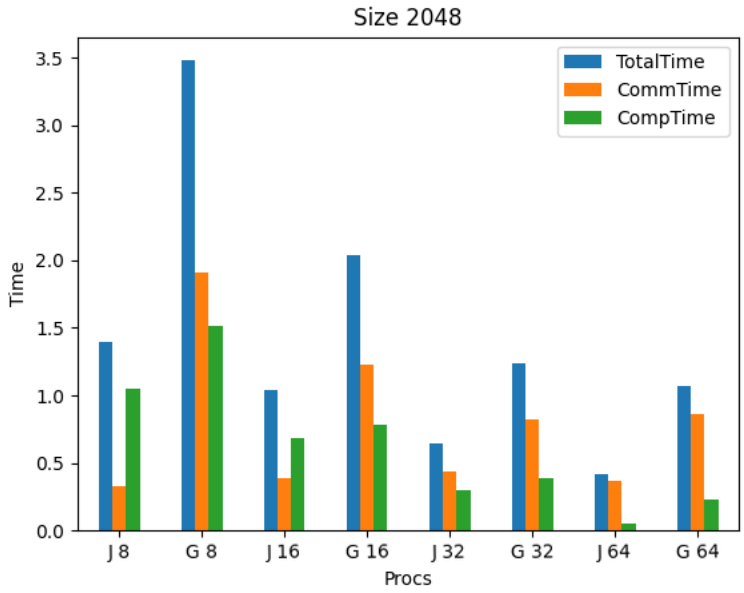
Πραγματοποιώντας μετρήσεις για πίνακα μεγέθους 1024×1024 με 64 MPI διεργασίες παρατηρούμε ότι η μέθοδος Jacobi συγκλίνει μετά από 798200 επαναλήψεις, καταναλώνοντας 221 δευτερόλεπτα, ενώ η Gauss-Seidel συγκλίνει ύστερα από 3200 επαναλήψεις και 2.37 δευτερόλεπτα.
Βλέπουμε πως η μέθοδος Gauss-Seidel συγκλίνει περίπου 100 φορές πιο γρήγορα, παρόλο που για δεδομένο αριθμό επαναλήψεων ενδεχομένως να είναι πιο αργή. Για τον λόγο αυτό, για την επίλυση του προβλήματος σε ένα σύστημα κατανεμημένης μνήμης θα επιλέγαμε την μέθοδο Gauss-Seidel, καθώς μας ενδιαφέρει ο πίνακας κατά την σύγκλιση και όχι ύστερα από κάποιον προκαθορισθμένο αριθμό επαναλήψεων.

Μετρήσεις χωρίς έλεγχο σύγκλισης

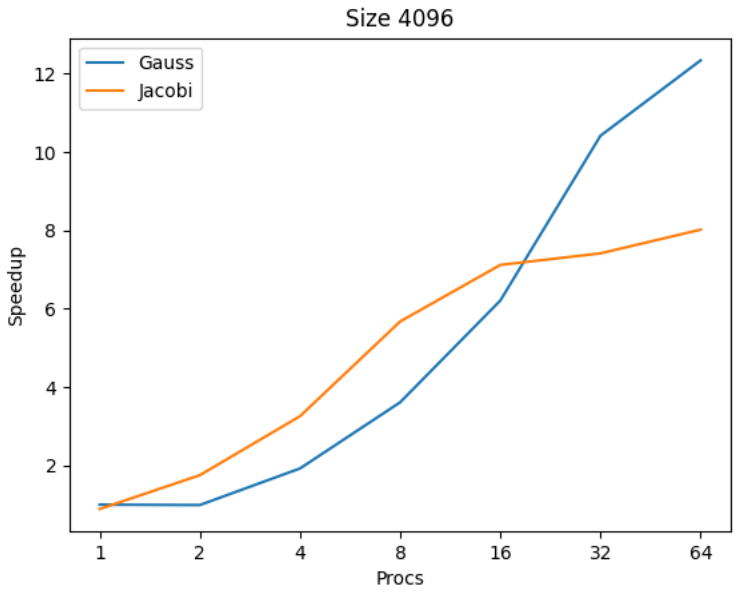
Πραγματοποιήσαμε μετρήσεις για 256 επαναλήψεις, 1, 2, 4, 8, 16, 32 και 64 MPI διεργασίες και μεγέθη πίνακα $2048 \times 2048, 4096 \times 4096, 6144 \times 6144$
Παρακάτω παραθέτουμε τα speedup και barplot διαγράμματα οργανωμένα ανά μέγεθος πίανκα:

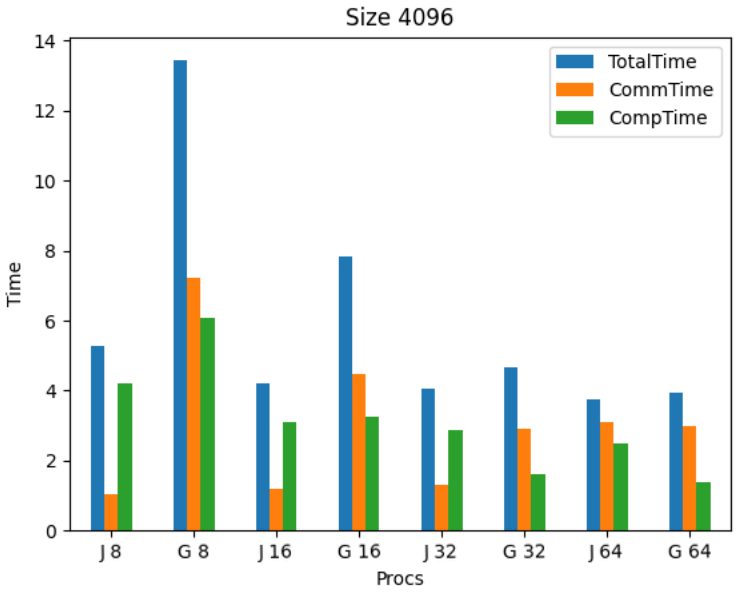
1. Grid size 2048×2048



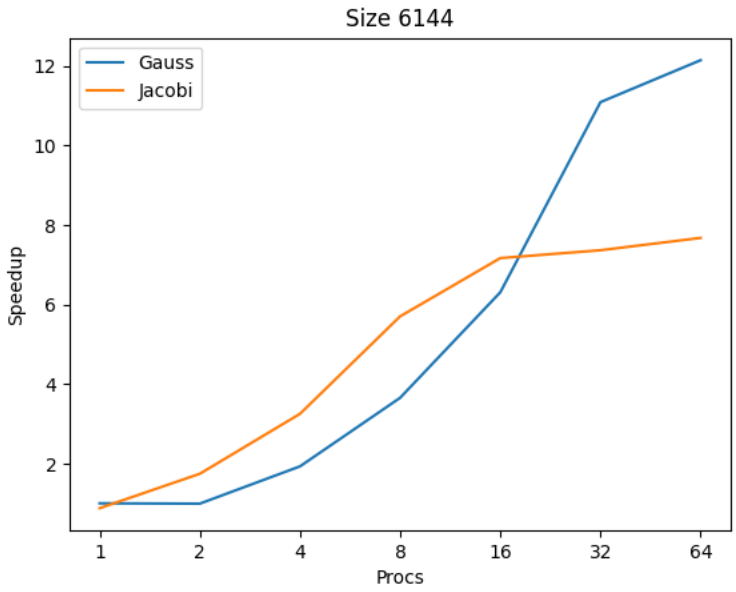


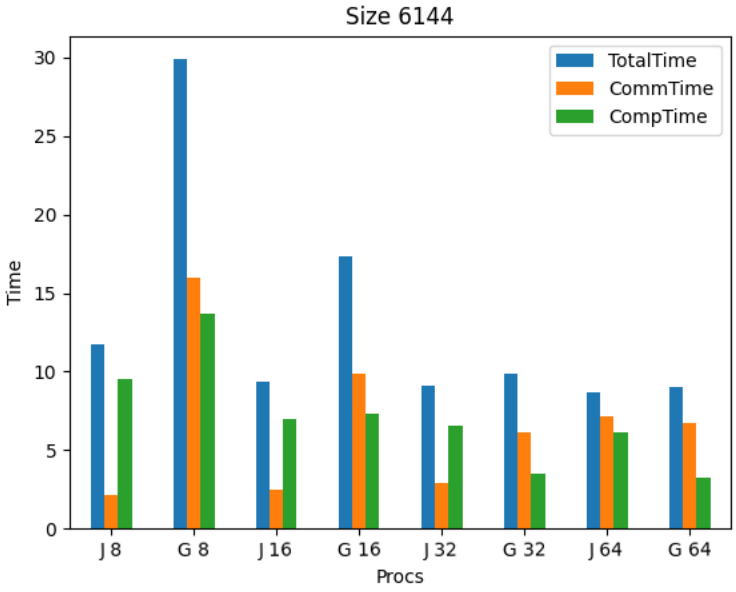
2. Grid size 4096×4096





3. Grid size 6144×6144





Παρατηρήσεις

- Σε γενικές γραμμές ο χρόνος επικοινωνίας είναι μεγαλύτερος για την μέθοδο Gauss-Seidel, όπως είναι αναμενόμενο καθώς υπάρχουν 2 σημεία επικοινωνίας διεργασιών (πριν και μετά τον υπολογισμό για δεδομένο χρόνο t)
- Η μέθοδος Jacobi είναι πάντα το ίδιο γρήγορη ή γρηγορότερη από την Gauss-Seidel για δεδομένο Configuration (αριθμός επαναλήψεων, μέγεθος πίνακα, MPI διεργασίες). Αυτό είναι ανεμενόμενο καθώς το μεγάλο πλεονέκτημα της Gauss-Seidel είναι ο ρυθμός σύγκλισης, όταν όμως έχουμε ίδιο αριθμό επαναλήψεων η Jacobi χρησιμοποιεί απλούστερο σχήμα επικοινωνίας και υπολογισμών που ευνοεί την παράλληλη εκτέλεση.
- Αναφορικά με το speedup, για κάθε μέθοδο ξεχωριστά παρατηρούμε πως είναι ικανοποιητικό με εξαίρεση ίσως την Jacobi για πίνακες 4096×4096 και 6144×6144 όπου για 16,32,64 διεργασίες το speedup μεταβάλλεται ελάχιστα. Μεταξύ των μεθόδων η σύγκριση speedup δεν είναι ουσιώδης, καθώς εξαρτάται από την απόδοση της σειριακής εκδοχής τους που παρουσιάζει μεγάλη διαφορά (ενδεικτικά για πίνακα 6144×6144 η σειριακή έκδοση της Jacobi διαρκεί 67 δευτερόλεπτα ενώ η αντίστοιχη της Gauss-Seidel 109)