ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Συστήματα Παράλληλης Επεξεργασίας

Εργαστηριακές Ασκήσεις

Ιωάννης Ρέκκας
03119049

Αναστάσιος Στέφανος Αναγνώστου
03119051

Γεώργιος Αναστασίου
03119112

4 Φεβρουαρίου 2024

Περιεχόμενα

Μέρος I
Άσκηση 1

1 Συλλογή Δεδομένων

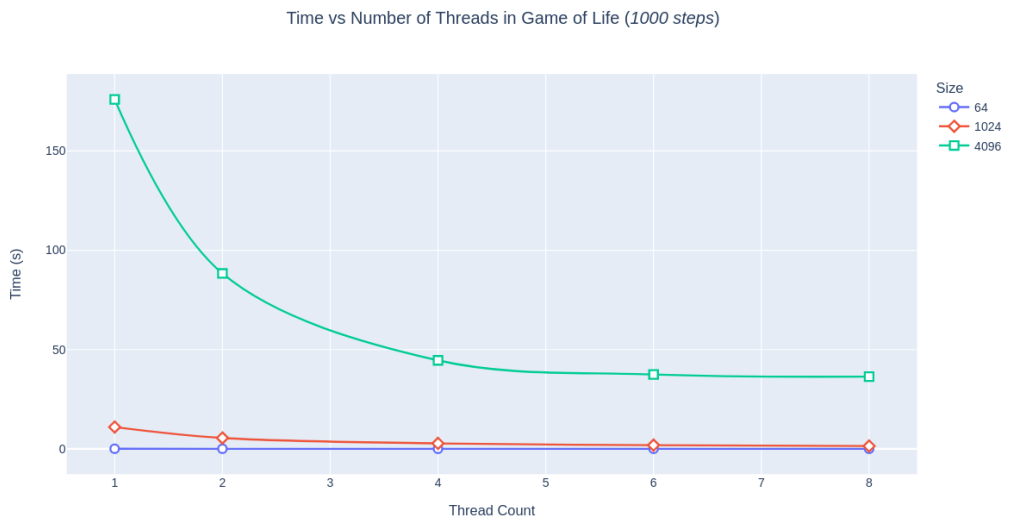
Αρχικά, το πρόγραμμα παραλληλοποιήθηκε χρησιμοποιώντας την κατάλληλη μακροεντολή του OpenMP, όπως φαίνεται στον κώδικα 1 (βλ. Παράρτημα). Για την συλλογή δεδομένων δεσμεύτηκε ένας υπολογιστικός κόμβος με οκτώ πυρήνες και εκτελέστηκε το script 4. Το εκτελέσιμο δημιουργήθηκε με την μεταγλώττιση όπως φαίνεται στο Makefile 3. Τα δεδομένα εξόδου φαίνονται παρακάτω.

```
Thread Count = 1
GameOfLife: Size 64 Steps 1000 Time 0.023253
GameOfLife: Size 1024 Steps 1000 Time 10.968962
GameOfLife: Size 4096 Steps 1000 Time 175.976354
Thread Count = 2
GameOfLife: Size 64 Steps 1000 Time 0.013578
GameOfLife: Size 1024 Steps 1000 Time 5.457094
GameOfLife: Size 4096 Steps 1000 Time 88.317703
Thread Count = 4
GameOfLife: Size 64 Steps 1000 Time 0.010049
GameOfLife: Size 1024 Steps 1000 Time 2.723685
GameOfLife: Size 4096 Steps 1000 Time 44.545233
Thread Count = 6
GameOfLife: Size 64 Steps 1000 Time 0.009147
GameOfLife: Size 1024 Steps 1000 Time 1.833331
GameOfLife: Size 4096 Steps 1000 Time 37.423192
Thread Count = 8
GameOfLife: Size 64 Steps 1000 Time 0.009746
GameOfLife: Size 1024 Steps 1000 Time 1.376557
GameOfLife: Size 4096 Steps 1000 Time 36.365997
```

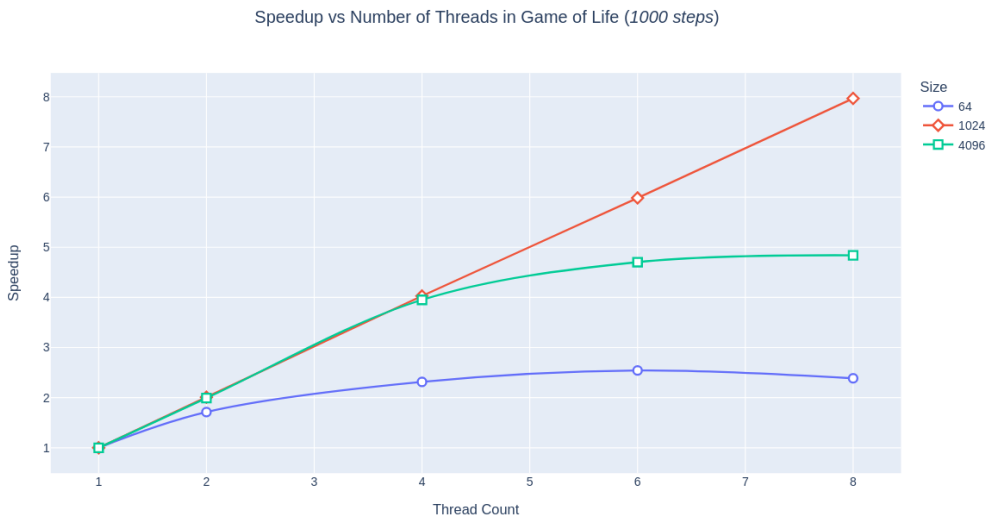
Σχήμα 1: Δεδομένα Εξόδου Game Of Life

2 Σύγκριση Μετρήσεων

Οι μετρήσεις συνοψίζονται στα παρακάτω γραφήματα.



(α′) Χρόνος εκτέλεσης του Game Of Life ως προς το πλήθος των διεργασιών για διάφορα μεγέθη του παιχνιδιού.



(β′) Επιτάχυνση του Game Of Life για διάφορα μεγέθη του παιχνιδιού.

Στο γράφημα 2α′ απεικονίζονται τρεις καμπύλες, μία για κάθε μέγεθος της προσομοίωσης Game of Life, οι οποίες αποτυπώνουν τον χρόνο εκτέλεσης της προσομοίωσης για διάφορα πλήθη νημάτων. Στο γράφημα 2β′ απεικονίζονται τρεις καμπύλες, οι οποίες αποτυπώνουν την επιτάχυνση (speedup) για καθένα από τα μεγέθη της προσομοίωσης.

3 Ερμηνεία Αποτελεσμάτων

Αρχικά, φαίνεται ότι τόσο το 64 όσο και το 4096 δεν κλιμακώνουν.

Στο μέγεθος προσομοίωσης 64×64 η εκτέλεση είναι τόσο γρήγορη ώστε η δημιουργία νημάτων και μόνο να προκαλεί τόση καθυστέρηση ώστε το πρόγραμμα να μην κλιμακώνει. Ήδη από τα 2 νήματα και πάνω η κλιμάκωση δεν είναι ιδανική. Σημειώνεται επιτάχυνση, απλώς όχι τόση όση αναμένεται από τον παραλληλισμό.

Στο μέγεθος προσομοίωσης 4096×4096 η εκτέλεση είναι χρονοβόρα, άρα η δημιουργία νημάτων δεν προκαλεί συγκρίσιμη καθυστέρηση. Προκαλείται, όμως, συμφόρηση στον δίαυλο επειδή πλέον δεν χωράνε όλα τα δεδομένα στη cache και έτσι τα νήματα αιτούνται δεδομένα συχνότερα από την κύρια μνήμη. Η ύπαρξη του διαύλου σειριοποιεί ένα κομμάτι του προγράμματος αναγκάζοντας κάποια νήματα να περιμένουν άλλα για να πάρουν τα δικά τους δεδομένα. Αυτή η συμπεριφορά βέβαια εκδηλώνεται στην δημιουργία 6 και περισσότερων νημάτων. Μέχρι και τα 4 νήματα η κλιμάκωση είναι σχεδόν ιδανική και ο δίαυλος μπορεί να ανταπεξέλθει στις αιτήσεις.

Αντίθετα στο μέγεθος 1024×1024 παρατηρούμε ιδανικό γραμμικό speedup για όλα τα πλήθη νημάτων. Εκεί οι παράμετροι εναρμονίζονται έτσι ώστε να μην υπάρχει συμφόρηση στον διαυλο και η κατανομή εργασίας στα νήματα είναι τέλεια.

4 Παράρτημα

Listing 1: Parallelized Game Of Life

```
...
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for private(nbrs, i, j) shared(previous, current)
    for ( i = 1 ; i < N-1 ; i++ )
        for ( j = 1 ; j < N-1 ; j++ ) {
            nbrs = previous[i+1][j+1] + previous[i+1][j] + previous[i+1][j-1] \
                 + previous[i][j-1] + previous[i][j+1] \
                 + previous[i-1][j-1] + previous[i-1][j] + previous[i-1][j+1];
            if ( nbrs == 3 || ( previous[i][j]+nbrs ==3 ) )
                current[i][j]=1;
            else
                current[i][j]=0;
        }
        #ifdef OUTPUT
        print_to_pgm(current, N, t+1);
        #endif
        //Swap current array with previous array
        swap=current;
        current=previous;
        previous=swap;
}
...
```

Listing 2: Bash Script to build Game Of Life

```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N make_omp_gol

## Output and error files
#PBS -o make_omp_gol.out
#PBS -e make_omp_gol.err

## How many machines should we get?
#PBS -l nodes=1:ppn=1

##How long should the job run for?
#PBS -l walltime=00:01:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab30/a1
make
```

Listing 3: Makefile

```
all: Game_Of_Life

Game_Of_Life: Game_Of_Life.c
        gcc -O3 -fopenmp -o Game_Of_Life Game_Of_Life.c

clean:
        rm Game_Of_Life
```

Listing 4: Bash Script to Gather Measurements

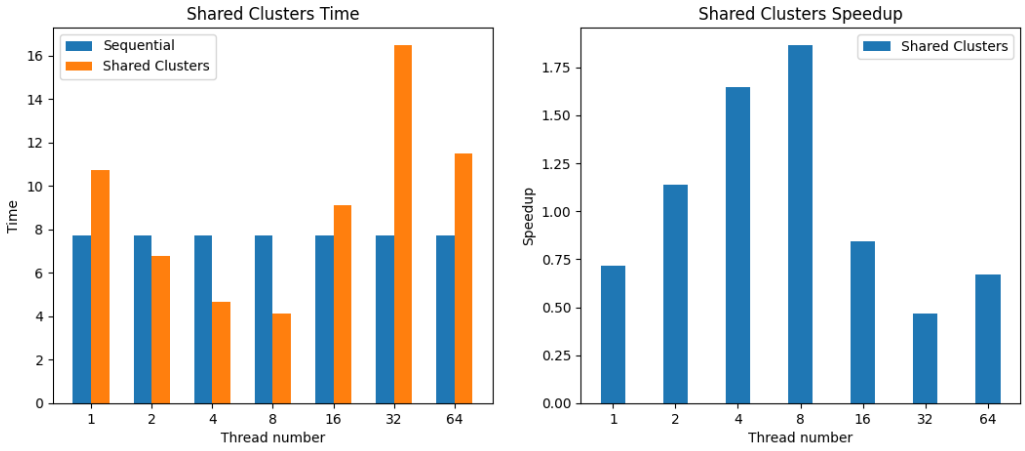
```
#!/bin/bash

## Give the Job a descriptive name
#PBS -N run_gol

## Output and error files
#PBS -o .run.out
#PBS -e .run.err

## How many machines should we get?
#PBS -l nodes=1:ppn=8

##How long should the job run for?
#PBS -l walltime=00:10:00

## Start
## Run make in the src folder (modify properly)

module load openmp
cd /home/parallel/parlab30/a1

threads=(1 2 4 6 8)
sizes=(64 1024 4096)
speed=1000

for thread in "${threads[@]}"
do
    export OMP_NUM_THREADS="$thread"
    for size in "${sizes[@]}"
    do
        ./Game_Of_Life "$size" "$speed"
    done
done
```

Μέρος II
Άσκηση 2

5 K-MEANS

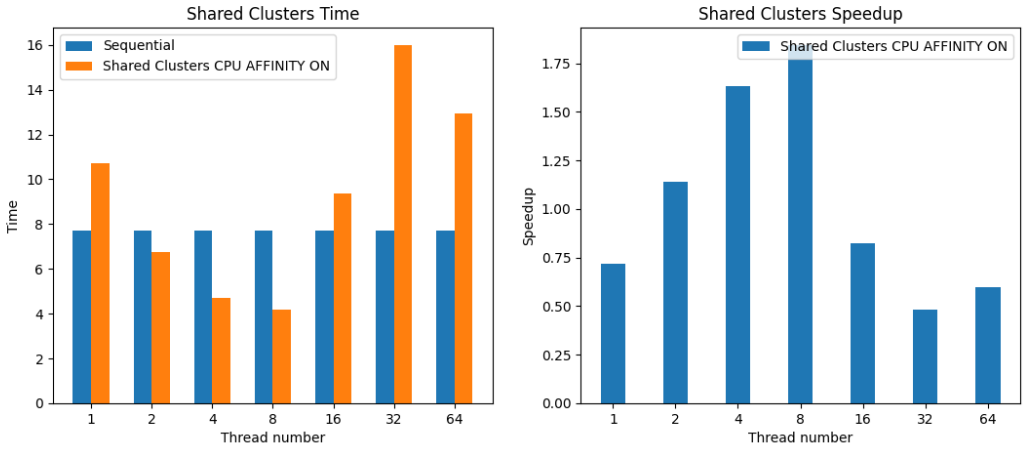
5.1 shared clusters

Ο κώδικας παραλληλοποιήθηκε προσθέτοντας τα κατάλληλα OMP compiler directives στα παραλληλοποιήσιμα σημεία του προγράμματος (βλέπε 5). Οι μετρήσεις των πειραμάτων παρουσιάζονται στα γραφήματα 3. Παρατηρείται ότι, σε κάθε περίπτωση, ανεξαρτήτως πλήθους νημάτων, η παράλληλη εκδοχή του αλγορίθμου είναι πιο αργή από την σειριακή. Αυτό οφείλεται στο γεγονός ότι η απόπειρα παραλληλοποίησης ήταν αφελής. Συγκεκριμένα, όλα τα νήματα μοιράζονται τους πίνακες με τα δεδομένα των clusters με αποτέλεσμα να σπαταλάται πολύς υπολογιστικός χρόνος συγχρονίζοντας την πρόσβαση σε αυτά. Αξιοσημειώτο είναι ότι ακόμα και για ένα μόνο νήμα, η παράλληλη εκδοχή είναι βραδύτερη. Αυτό οφείλεται στην χρήση ατομικών εντολών για συγχρονισμό, οι οποίες κλειδώνουν τον δίαυλο και επεμβαίνουν με τις λειτουργίες επιτάχυνσης του επεξεργαστή. Στον σειριακό αλγόριθμο απουσιάζουν οι ατομικές εντολές, γιατί η ατομικότητα είναι εξασφαλίσμενη εκ φύσεως.



Σχήμα 3: Χρόνος Εκτέλεσης και Speedup του Naive Kmeans Shared Clusters

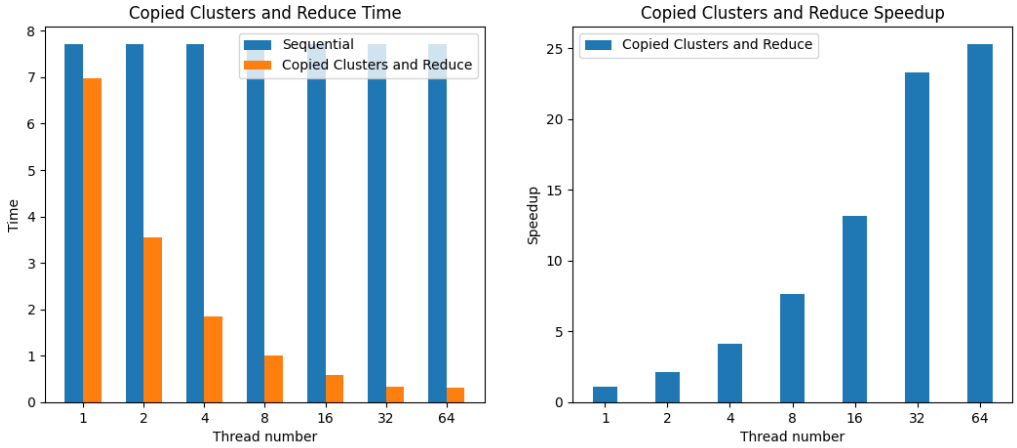
Αν αρχικοποιηθεί καταλλήλως η μεταβλητή περιβάλλοντος GOMP_CPU_AFFINITY, τότε τα νήματα παραμένουν σε έναν επεξεργαστή κατά την διάρκεια ζωής τους. Αυτό εξασφαλίζει ότι δεν θα μετακινηθούν σε άλλον επεξεργαστικό πυρήνα και ότι θα διατηρήσουν την τοπικότητα των δεδομένων τους.



Σχήμα 4: Speedup του Naive Kmeans Shared Clusters CPU Affinity on

Φαίνεται ότι δεν έχει μεγάλη επίδραση στον χρόνο εκτέλεσης, πράγμα αναμενόμενο, αφού το πρόβλημα ήταν διαφορετικό, δεν σχετιζόταν με την εναλλαγή των νημάτων στους πυρήνες.

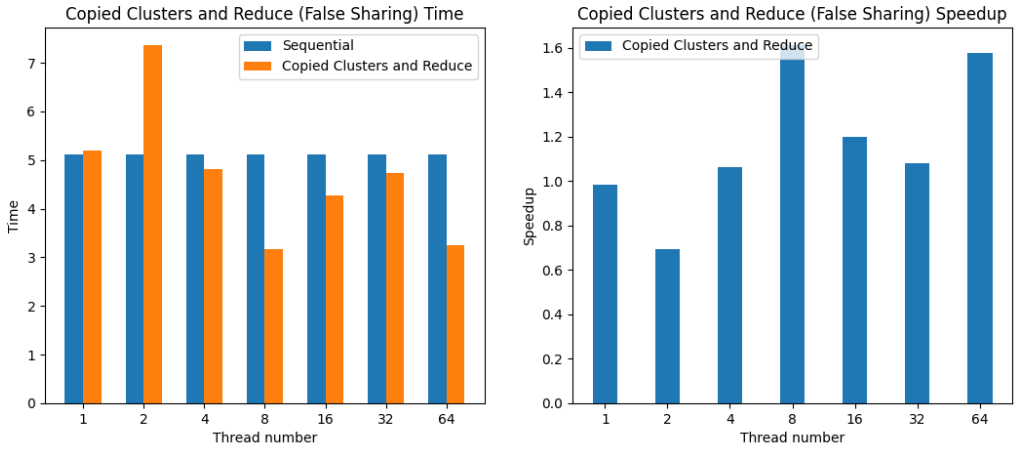
5.2 copied clusters and reduce



Σχήμα 5: Speedup του Kmeans Copy Reduce

Φαίνεται ότι επιτυγχάνεται πολύ καλύτερη κλιμάκωση και σχεδόν τέλεια γραμμικό speedup. Σε αυτήν την υλοποίηση, κάθε νήμα έχει το δικό του αντίγραφο των clusters και αυτοί ενημερώνονται στο τέλος κάθε επανάληψης. Αυτό έχει ως αποτέλεσμα να μην υπάρχει ανταγωνισμός για τους πόρους, ελαττώνοντας κατά πολύ τον χρόνο εκτέλεσης. Υπάρχει μόνο ένα μικρό overhead για τον συγχρονισμό των νημάτων στο τέλος καθεμίας επανάληψης. Σημειώνεται, ότι για την καλή επίδοση οφείλεται και το CPU_AFFINITY, το οποίο περιορίζει τις εναλλαγές των νημάτων στους επεξεργαστές και διατηρεί την τοπικότητα των δεδομένων.

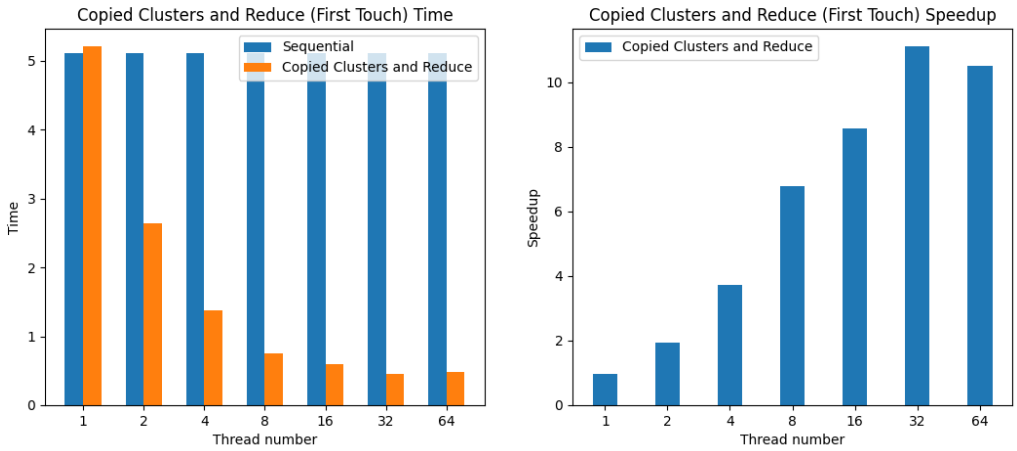
Κάνοντας getconf -a — grep CACHE στον scirouter βρίσκεται ότι κάθε επίπεδο κρυφής μνήμης έχει cache line size = 64 Byte. Στην περίπτωση configuration {256, 1, 4, 10}, το κέντρο καθενός cluster είναι ένα σημείο αναπαριστάμενο από έναν double = 8 byte. Αυτό σημαίνει ότι μία γραμμή της κρυφής μνήμης χωράει δύο



Σχήμα 6: Speedup του Naive Kmeans Shared Clusters

αντίγραφα των πινάκων local newClusters. Αυτό έχει ως αποτέλεσμα να μην απομονώνονται όντως τα δεδομένα κάθε νήματος από αυτά των άλλων, αλλά να τοποθετούνται στην ίδια γραμμή της κρυφής μνήμης, αντίγραφα της γραμμής να αποθηκεύονται στις κρυφές μνήμες πολλών πυρήνων και να ενεργοποιείται το πρωτόκολο συνάφειας, χωρίς όντως να υπάρχει διαμοιρασμός δεδομένων. Αυτό προκαλεί κίνηση στον δίαυλο και μεγάλη καθυστέρηση.

Μια λύση είναι να εκμεταλλευτεί κανείς την πολιτική first touch του Linux 6.

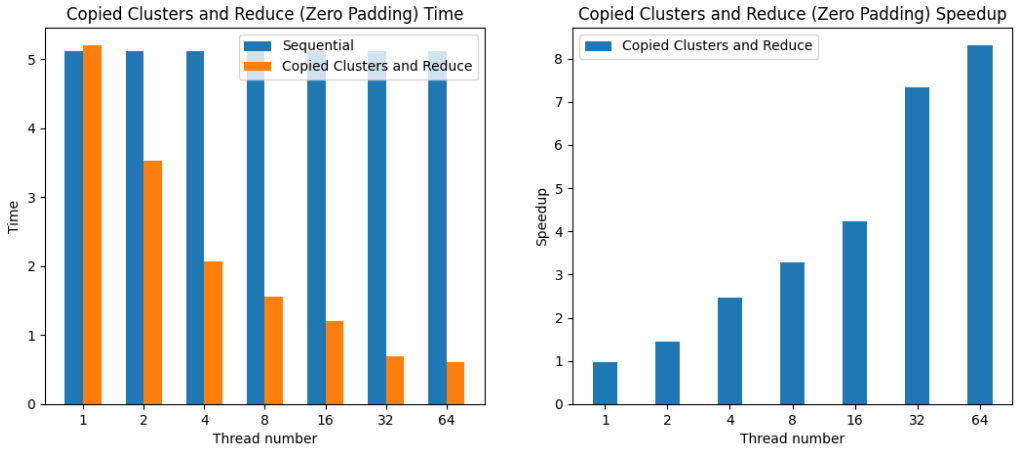


Σχήμα 7: Speedup του Kmeans Copy Reduce First Touch

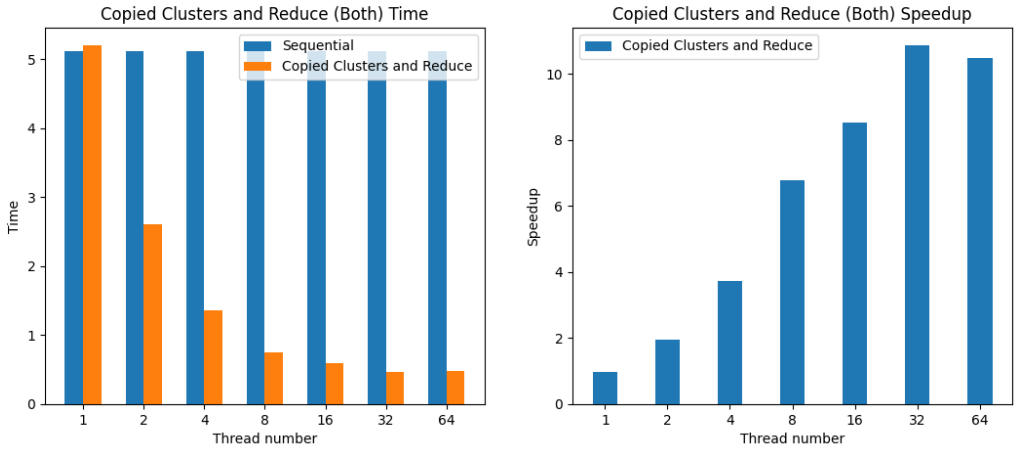
Μία άλλη λύση είναι να δεσμεύσει κανείς περισσότερο χώρο από τον απαραίτητο για κάθε νήμα, ουσιαστικά παρεμβάλλοντας μηδενικά στην γραμμή της κρυφής μνήμης και αποφεύγοντας έτσι το false sharing 7.

Φυσικά μπορεί κανείς να δοκιμάσει και τις δύο μεθόδους ταυτοχρόνως.

Φαίνεται τελικά ότι η μέθοδος Zero Padding κλιμακώνει καλύτερα.



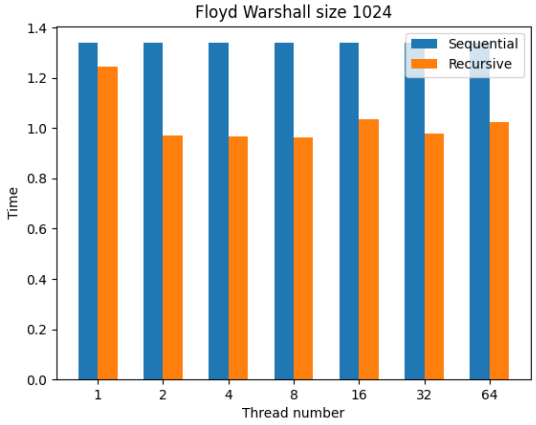
Σχήμα 8: Speedup του Kmeans Copy Reduce Zero Padding



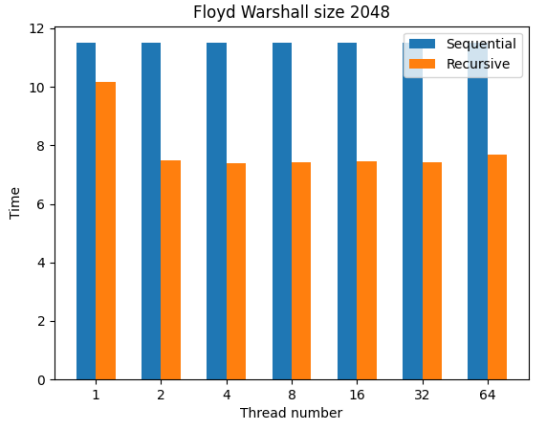
Σχήμα 9: Speedup του Kmeans Copy Reduce Both

6 Floyd-Warshall

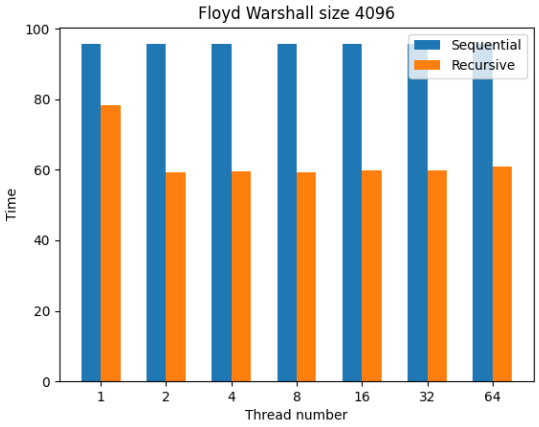
Η αναδρομική υλοποίηση 8 του αλγορίθμου Floyd - Warshall, παρότι έχει μία αρχική βελτίωση στον χρόνο εκτέλεσης, δεν κλιμακώνει ικανοποιητικά. Αυτό φυσικά είναι αναμενόμενο, γιατί δεν παρουσιάζει πολλή παραλληλία στον task graph. Από τις 8 (αναδρομικές) κλήσεις της συνάρτησης αναδρομής, μόνο 4 είναι ανά δύο παράλληλες μεταξύ τους, δεν εξαρτώνται δηλαδή τα δεδομένα τους από την άλλη κλήση. Επομένως, σε αυτήν την περίπτωση, γίνεται η παραλληλοποίηση του ενός από τα δύο tasks για δύο συνολικά κλήσεις από τις 8, και γίνεται χρήση ενός μόνο extra thread αντί όλων όσων ενδεχομένως δημιουργήθηκαν. Έτσι, για εκτέλεση του recursive με 2 threads έναντι ενός, ο χρόνος εκτέλεσης μειώνεται κατά 25% πλην το κόστος δημιουργίας ενός thread που είναι συγκριτικά αμελητέο, αφού γίνονται τα 8 tasks της αναδρομής σε χρόνο 6 tasks. Από αυτό το σημείο και έπειτα, τα extra threads δεν αποσκοπούν σε περεταίρω μείωση του χρόνου, αλλά καθυστερούν την εκτέλεση κατά όσο χρόνο παίρνει η δημιουργία τους.



(α′) Floyd Warshall αναδρομικό 1024



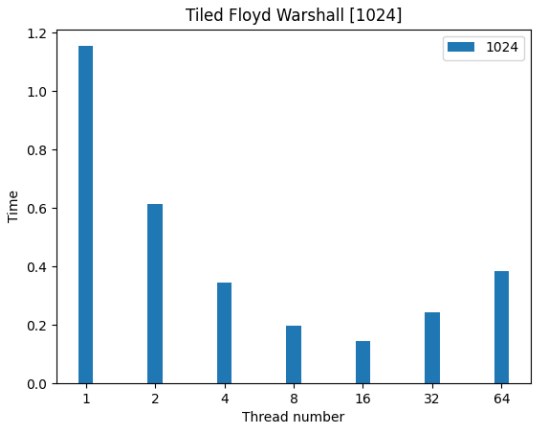
(β′) Floyd Warshall αναδρομικό 2048



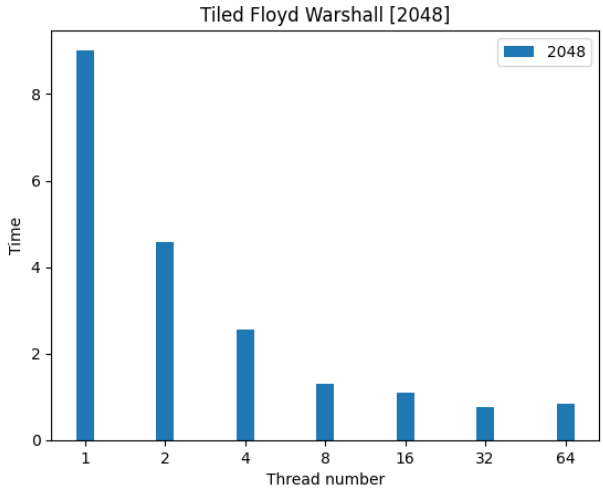
(γ′) Floyd Warshall αναδρομικό 4096

Η δε tiled 9 εκδοχή του αλγορίθμου παρουσιάζει περισσότερο παραλληλισμό και αυτό πράγματι αποτυπώνεται στα διαγράμματα χρόνου εκτέλεσης, όπου είναι προφανές ότι η κλιμάκωση είναι ικανοποιητική. Σημειώνεται ότι η κλιμάκωση στα μικρά μεγέθη χαλάει νωρίτερα από τα μεγάλα. Αυτό έχει σχέση με το μέγεθος του προβλήματος ως προς το κόστος δημιουργίας περισσοτέρων threads. Η κλιμάκωση, δηλαδή, 'χαλάει' εκεί όπου ο χρόνος δημιουργίας των threads γίνεται μεγαλύτερος από αυτόν που 'γλιτώνεται' με την παραλληλοποίηση. Ο χρόνος που γλιτώνεται με την παραλληλοποίηση αλλάζει με το μέγεθος του προβλήματος ενώ ο χρόνος δημιουργίας

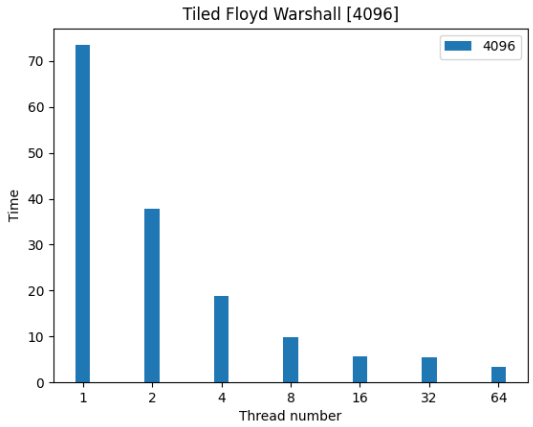
δεδομένου αριθμού threads παραμένει σταθερός, πράγμα που οδηγεί σε διαφορετικό breakthrough point ανάλογα με το μέγεθος του προβλήματος. Συγκεκριμένα, στα 1024 το breakthrough point είναι στα 16 threads, στα 2048 κάπου ανάμεσα στα 32-64 threads ενώ στα 4096 είναι από τα 64 threads και μετά.



(α′) Floyd Warshall tiled 1024



(β′) Floyd Warshall tiled 2048



(γ′) Floyd Warshall tiled 4096

7 Παράρτημα

Listing 5: Naive Kmeans Shared Clusters

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

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

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

    // update new cluster centers : sum of objects located within
    /*
     * TODO: protect update on shared "newClusterSize" array
     */
    #pragma omp atomic
    newClusterSize[index]++;
    for (j=0; j<numCoords; j++)
        /*
         * TODO: protect update on shared "newClusters" array
         */
        #pragma omp atomic
        newClusters[index*numCoords + j] += objects[i*numCoords + j];
}
// average the sum and replace old cluster centers with newClusters
for (i=0; i<numClusters; i++) {
    if (newClusterSize[i] > 0) {
        for (j=0; j<numCoords; j++) {
            clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i];
        }
    }
}
...
```

Listing 6: Kmeans First Touch

```
...
#pragma omp parallel for private(k)
for (k=0; k<nthreads; k++)
{
        local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters, sizeof
    (**local_newClusterSize));
        local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords,
    sizeof(**local_newClusters));
}
...
```

Listing 7: Kmeans Zero Padding

```
...
#define CACHE_LINE_SIZE 64
#define PADDING_DOUBLE (CACHE_LINE_SIZE - sizeof(double))

// Each thread calculates new centers using a private space. After that, thread 0 does an array reduction on them.
int * local_newClusterSize[nthreads];  // [nthreads][numClusters]
double * local_newClusters[nthreads];    // [nthreads][numClusters][numCoords]

for (k=0; k<nthreads; k++)
{
        local_newClusterSize[k] = (typeof(*local_newClusterSize)) calloc(numClusters, sizeof(**local_newClusterSize));
        local_newClusters[k] = (typeof(*local_newClusters)) calloc(numClusters * numCoords + numClusters * PADDING_DOUBLE, sizeof(**local_newClusters));
}
...
```

Listing 8: FW Recursive

```
...
else {  // This is called as FW_SR(A, A, A) so A, B, C are the same array.
    #pragma omp parallel
    {
        #pragma omp single
        {
            FW_SR(A,arow, acol,B,brow, bcol,C,crow, ccol, myN/2, bsize); // A00, B00, C00

            #pragma omp task
            FW_SR(A,arow, acol+myN/2,B,brow, bcol,C,crow, ccol+myN/2, myN/2, bsize); // A01,
     B00, C01
            #pragma omp task if(0)
            {
                    FW_SR(A,arow+myN/2, acol,B,brow+myN/2, bcol,C,crow, ccol, myN/2, bsize);
     // A10, B10, C00
            }
            #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); // A11, B10, C01


            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); // A11, B11, C11

            #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); // A10, B11, C10
            #pragma omp task if(0)
            {
                    FW_SR(A,arow, acol+myN/2,B,brow, bcol+myN/2,C,crow+myN/2, ccol+myN/2,
    myN/2, bsize); // A01, B01, C11
            }
            #pragma omp taskwait
            FW_SR(A,arow, acol,B,brow, bcol+myN/2,C,crow+myN/2, ccol, myN/2, bsize); // A00,
     B01, C10
        }
    }
}
...
```

Listing 9: FW Tiled

```
...
for(k=0;k<N;k+=B){
        FW(A,k,k,k,B); //panw aristeri gwnia

        # pragma omp parallel for private(i)
        for(i=0; i<k; i+=B)
                FW(A,k,i,k,B);

        # pragma omp parallel for private(i)
        for(i=k+B; i<N; i+=B)
                FW(A,k,i,k,B);

        # pragma omp parallel for private(j)
        for(j=0; j<k; j+=B)
                FW(A,k,k,j,B);

        # pragma omp parallel for private(j)
        for(j=k+B; j<N; j+=B)
                FW(A,k,k,j,B);

        # pragma omp parallel for private(i, j)
        for(i=0; i<k; i+=B)
                for(j=0; j<k; j+=B)
                        FW(A,k,i,j,B);

        # pragma omp parallel for private(i, j)
        for(i=0; i<k; i+=B)
                for(j=k+B; j<N; j+=B)
                        FW(A,k,i,j,B);

        # pragma omp parallel for private(i, j)
        for(i=k+B; i<N; i+=B)
                for(j=0; j<k; j+=B)
                        FW(A,k,i,j,B);

        # pragma omp parallel for private(i, j)
        for(i=k+B; i<N; i+=B)
                for(j=k+B; j<N; j+=B)
                        FW(A,k,i,j,B);
}
...
```

Μέρος III
Άσκηση 3

8 K-MEANS

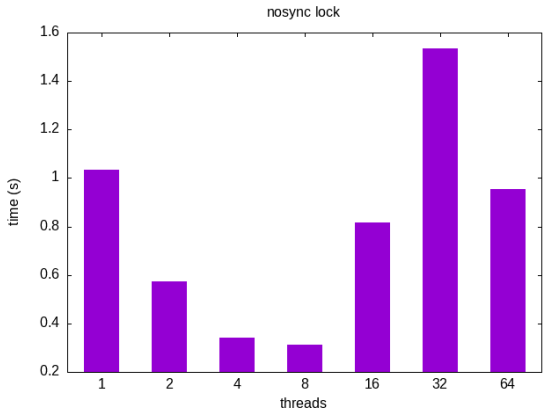
8.1 Αμοιβαίος Αποκλεισμός - Κλειδώματα

Αρχικά, ποιοτικά όλες οι υλοποιήσεις, εκτός από τις υλοποίησεις array lock και clh lock, παρουριάζουν την ίδια συμπεριφορά. Δηλαδή, διατηρούν την επιτάχυνση λόγω παραλληλισμού στην καλύτερη περίπτωση μέχρι τα 4 νήματα και επιδεικνύουν μεγάλη καθυστέρηση για 8 νήματα και άνω. Οι διαφόρες τους παρατηρούνται κυρίως στις απόλυτες μετρήσεις. Αυτή η συμπεριφορά οφείλεται στο γεγονός ότι τα κλειδώματα προσφέρονται κυρίως για τυχαίες προσβάσεις σε κοινά δεδομένα, με σκοπό να διατηρηθεί η ορθότητα. Εν προκειμένω, πρόκειται για ένα παράλληλο πρόγραμμα κατά την εκτέλεση του οποίου όλα τα νήματα ζητούν ταυτόχρονα πρόσβαση στα κοινά δεδομένα, με αποτελέσμα να συνοστίζονται και ουσιαστικά η εκτέλεση να σειριοποιείται.

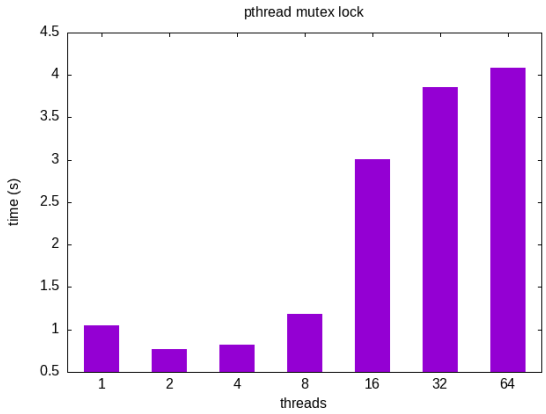
Συγκεκριμένα, το κλείδωμα με pthread mutex lock 12β′ αποδίδει ήδη χειρότερα στην εκτέλεση με 8 νήματα και παρουσιάζει απότομη αύξηση του χρόνου εκτέλεσης για περισσότερα νήματα, φτάνοντας στην χειρότερη περίπτωση στα 4 δευτερόλεπτα. Το δε pthread spin lock 12γ′ παρουσιάζει την ίδια συμπεριφορά αλλά είναι ακόμα βραδύτερο στις εκτελέσεις με περισσότερα από 16 νήματα. Αυτό οφείλεται στο γεγονός ότι τα mutex locks κοιμίζουν όσα νήματα δεν καταφέρουν να αποκτήσουν πρόσβαση στον κλειδωμένο πόρο, αποδεσμεύοντας υπολογιστικούς πόρους και επιτρέποντας σε άλλα νήματα να εκτελεστούν, ενώ τα spin locks δεσμεύουν τους υπολογιστικούς πόρους. Αυτός ο αριθμός νημάτων δεν είναι τυχαίος, καθώς το σύστημα στο οποίο εκτελείται το πρόγραμμα έχει 4 κόμβους 4 πυρήνων έκαστος με υποστήριξη έως και 2 νημάτων εκτέλεσης, δηλαδή στην καλύτερη περίπτωση 32 νήματα εκτέλεσης, αν όλα τα νήματα δεν εκτελούν ακριβώς την ίδια διεργασία, κάτι που δεν συμβαίνει εδώ.

Τα κλειδώματα test and set 12δ′, test and test and set 12ε′ παρουσιάζουν την ίδια συμπεριφορά. Αυτά, επειδή είναι κλειδώματα σε επίπεδο υλικού, είναι ακόμα βραδύτερα από τα προηγούμενα κλειδώματα τα οποία ήταν σε επίπεδο λογισμικού. Αυτό συμβαίνει διότι τέτοιου είδους κλειδώματα συνήθως προσφέρονται για σύντομα χρονικά διαστήματα, δηλαδή σύντομα critical sections, αλλά η παρούσα εφαρμογή κλειδώνει ένα σχετικά μεγάλο critical section. Επίσης, παρατηρείται ότι το κλείδωμα test and test and set παρουσιάζει μεγαλύτερη ταχύτητα από το test and set. Αυτό συμβαίνει γιατί τα κλειδώματα test and set έχουν το μειονέκτημα ότι ενεργοποιούν το πρωτόκολλο συνάφειας μνήμης όταν δύο ή παραπάνω πυρήνες εξετάζουν κοινά δεδομένα, στέλνωντας bus read x σε κάθε απόπειρα να πάρουν το κλείδωμα, με αποτέλεσμα να απασχολείται ο δίαυλος και να επιβαρύνεται το σύστημα. Αντιθέτως, το test and test and set το αποφεύγει αυτό γιατί διαβάζει πρώτα το κλείδωμα μόνο για ανάγνωση bus read, και αν το πάρει, τότε στέλνει bus read x.

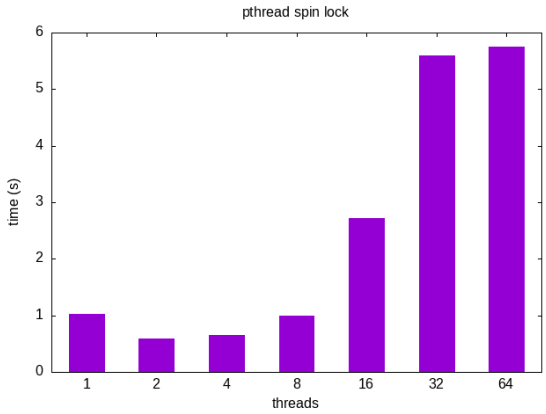
Τα array lock 12ϛ′ και clh lock 12ζ′ αποδίδουν πολύ καλύτερα από τα προηγούμενα κλειδώματα. Αφενός επιτρέπουν την επιτάχυνση του προγράμματος μέχρι και με 8 νήματα, αφετέρου παραμένουν πολύ ταχύτερα από τα άλλα κλειδώματα όταν τα νήματα είναι τόσα ώστε η εκτέλεση να είναι βραδύτερη από την σειριακή (> 16 νήματα). Αμφότερα καταφέρνουν την καλύτερη επίδοση επειδή, αντί να ανταγωνίζονται όλα τα νήματα για ένα κλείδωμα το οποίο βρίσκεται σε μία συγκεκριμένη θέση μνήμης, συντονίζονται πάνω σε πολλαπλά κλειδώματα σε διαφορετικές θέσεις μνήμης. Συγκεκριμένα, το array lock λειτουργεί ορίζοντας έναν boolean πίνακα μεγέθους τόσου όσο το πλήθος των νημάτων, κάθε θέση του οποίου δρα ως κλείδωμα για το αντίστοιχο νήμα. Κάθε νήμα καταφέρνει να πάρει το κλείδωμα και να εκτελέσει τον κρίσιμο κώδικα μόνον αν η αντίστοιχη θέση του πίνακα έχει τιμή true. Κατά την έξοδο από τον κρίσιμο κώδικα, θέτει το κλείδωμά του false και θέτει το κλείδωμα του επόμενου νήματος true. Το clh lock είναι λίγο ταχύτερο από το array lock καθώς αποτελεί βελτίωση σε αυτό, γιατί προκαλεί λιγότερη συμφόρηση στον δίαυλο και απαιτεί λιγότερη μνήμη.



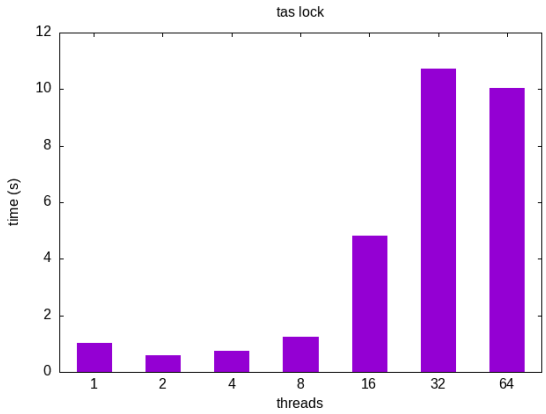
(α′) no sync lock



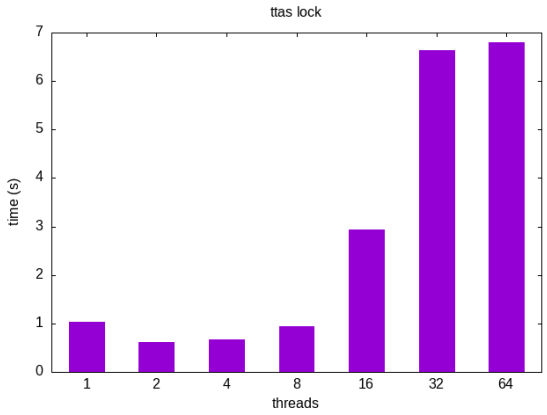
(β′) pthread mutex lock



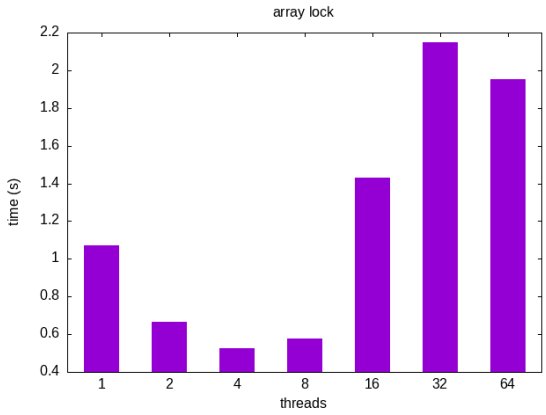
(γ′) pthread spin lock



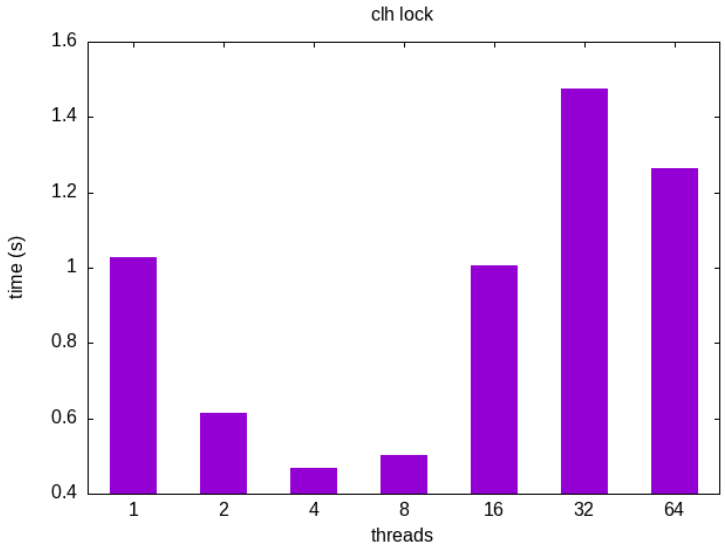
(δ′) test and set lock



(ε′) test and test and set lock



(ϛ′) array lock



(ζ′) clh lock

8.2 Ταυτόχρονες Δομές Δεδομένων

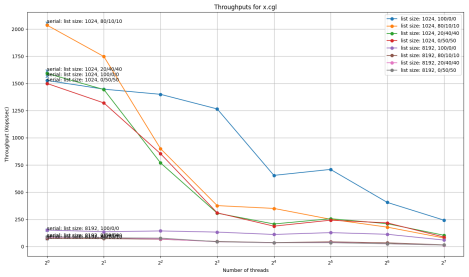
Η υλοποίηση 13α΄ με coarse-grain locking παρουσιάζει φθίνον throughput καθώς αυξάνονται τα νήματα εκτέλεσης. Αυτό είναι αναμενόμενο γιατί η δομή μπλοκάρεται για οποιαδήποτε λειτουργία σε αυτήν. Χαρακτηριστικό είναι ότι ακόμα και στην περίπτωση 100% αναζητήσεων, η επίδοση μειώνεται. Το φαινόμενο εντείνεται περισσότερο όταν εμπλέκονται λειτουργίες πέρα από την αναζήτηση, οι οποίες κρατάνε το κλείδωμα για περισσότερη ώρα. Αυτό το φαινόμενο, όμως, δεν είναι τόσο έντονο στις λίστες μεγαλύτερου μεγέθους, γιατί η αναζήτηση είναι πολύ χρονοβόρα ούτως ή άλλως και η παραπάνω διάρκεια για την επεξεργασία κελιών της λίστας είναι αμελητέα.

Η υλοποίηση με fine grain locking 13β΄ έχει χειρότερο throughput από το coarse πιθανόν λόγω overhead των πολλών επιπλέον locks. Αυτό φαίνεται σίγουρα στην περίπτωση του ενός thread που είναι εξαρχής πιο χαμηλό (το throughput) από την αντίστοιχη σειριακή εκδοχή χωρίς lock (σημειωμένη με ΄x΄ στο διάγραμμα). Παρ΄όλάυτά βλέπουμε ότι δεν σημειώνει αντίστοιχη πτώση κατά την άυξηση των thread και μάλιστα σχεδόν μένει σταθερό.

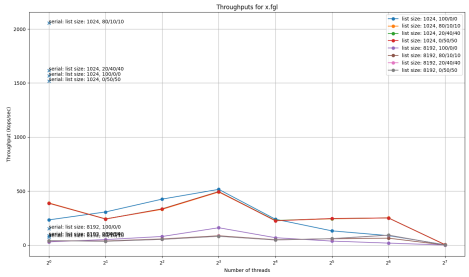
Η υλοποίηση με optimistic 13γ΄ παρατηρούμε γενικά καλύτερη κλιμάκουση του throughput γιατί πλέον δεν κλειδώνεται κάθε κόμβος για τις διεργασίες που θέλει να κάνει το νήμα. Αντί αυτού, κλειδώνεται μόνο ο κόμβος στον οποίον χρειάζεται να γίνει κάποια διεργασία. Έτσι, εξαλείφεται ένα μεγάλο κομμάτι του overhead που προκαλούταν από το συνεχές κλείδωμα και ξεκλείδωμα όλων των κόμβων. Στα 128 threads παρατηρείται μια μεγάλη πτώση του throughput, κάτι που πιθανόν οφείλεται στο ότι πολλά threads θέλουν να κάνουν πολλές διεργασίες ταυτόχρονα και καθυστερεί το ένα το άλλο.

Η υλοποίηση 13δ΄ με lazy locking έχει καλή κλιμάκωση στο throughput, ιδιαίτερα στην περίπτωση που κάνουμε μόνο read. Αυτό οφείλεται στο ότι στην περίπτωση lazy δεν κλειδώνει καθόλου η read, και διασχίζει την λίστα χωρίς καθόλου κλειδώματα σε αντίθεση με τις προηγούμενες περιπτώσεις. Επίσης, παρατηρούμε καλή κλιμάκωση και στις περιπτώσεις. Επίσης, πλέον, οι υπόλοιπες υλοποιήσεις κλιμακώνουν επειδή η validate δεν διατρέχει όλη τη λίστα, που αποτελεί βελτίωση έναντι της optimistic. Παρατηρούμε ότι η κλιμάκωση ΄σταματάει΄ κάπου στα 128 threads, το οποίο πιθανόν οφείλεται στο ότι το hyperthreading είναι ιδανικό για νήματα εκτέλεσης διαφορετικών λειτουργιών, κάτι που για 2 threads ανά core μπορεί να καταφέρνουν όντως να μοιραστούν τους πόρους ενός επεξεργαστή, αλλά για τα 4 threads είναι κάτι πιο δύσκολο και μάλλον δεν τα καταφέρνουν, οπότε η αύξηση του throughput είναι μικρότερη.

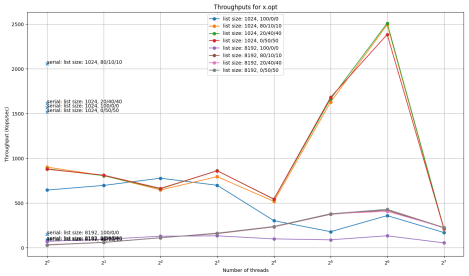
Η υλοποίηση 13ε΄ με non-blocking έχει καλύτερη κλιμάκωση στο throughput από το lazy, εκτός από τις περιπτώσεις που κάνουμε μόνο read. Αυτό λογικά οφείλεται στο μεγάλο overhead που απαιτείται για να ΄χτίσουμε΄ αυτή τη δομή, και η read δεν έχει διαφορετική υλοποίηση από την αντίστοιχη read του lazy, απλώς έχει μεγαλύτερο overhead. Όσο για τις άλλες περιπτώσεις, δηλαδή add και remove, η υλοποίηση τους δεν περιλαμβάνει blocking (έναντι της lazy), και αντ΄ αυτού χρησιμοποιείται η find η οποία για μεγάλο αριθμό threads είναι πιο αποδοτική, σε σύγκριση με το να μπλοκάρει την λίστα για τα υπόλοιπα threads, καθώς όταν είναι πολλά έχουν περισσότερες πιθανότητες να ΄κολλήσουν΄ σε εκείνο το σημείο. Μπορεί, δηλαδή, να είναι λιγότερο αποδοτικές οι add και remove για ένα μεμονωμένο thread, άλλα για το σύνολο των threads, αν αυτά είναι αρκετά, να είναι καλύτερες. Συγκεκριμένα, παρατηρούμε από τα δεδομένα ότι μέχρι και τα 16 threads η lazy είναι πιο αποδοτική, στα 32 και 64 threads η lazy και η non-blocking έχουν παρόμοια απόδοση, ενώ στα 128 threads η non-blocking είναι πιο γρήγορη.



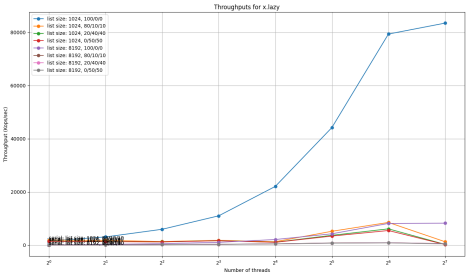
(α′) coarse-grain locking



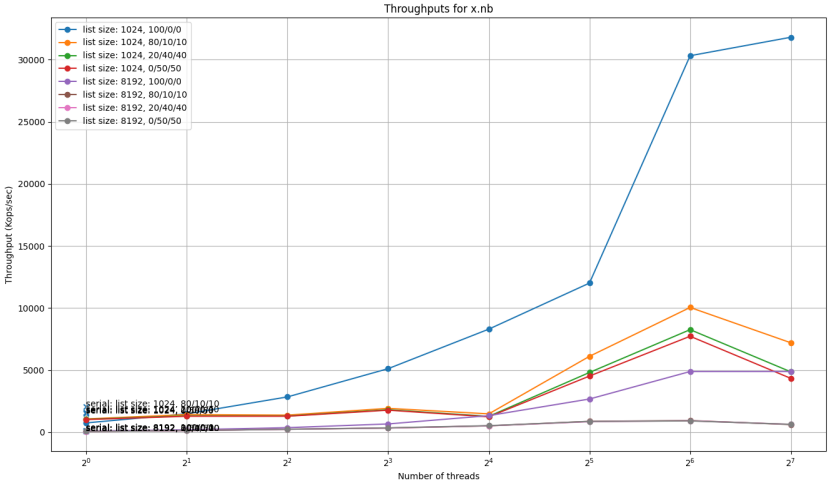
(β′) fine-grain locking



(γ′) optimistic locking



(δ′) lazy locking



(ε′) non-blocking locking

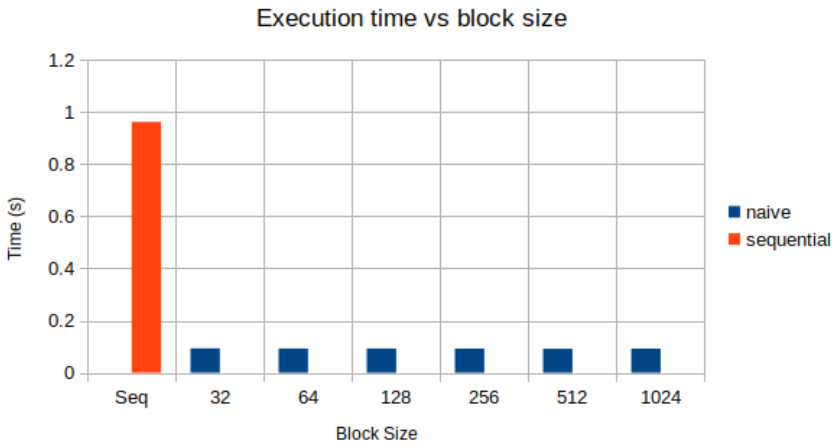
	contains	add	remove	validate
Coarse-grain	1 lock	1 lock	1 lock	no
Fine-grain	hand-over-hand locking	hand-over-hand locking	hand-over-hand locking	no
Optimistic	local locking	local locking	local locking	διάσχιση από την αρχή
Lazy	no locking	local locking	local locking	local checks
Non-blocking	wait-free	lock-free (with retries)	lock-free (with retries)	no

(ϝ΄) Πηγή σημειώσεων μαθήματος

Η σύγκριση της λειτουργίας των υλοποιήσεων περιγράφεται και πολύ σύντομα από τον παραπάνω πίνακα:

Μέρος IV
Άσκηση 4

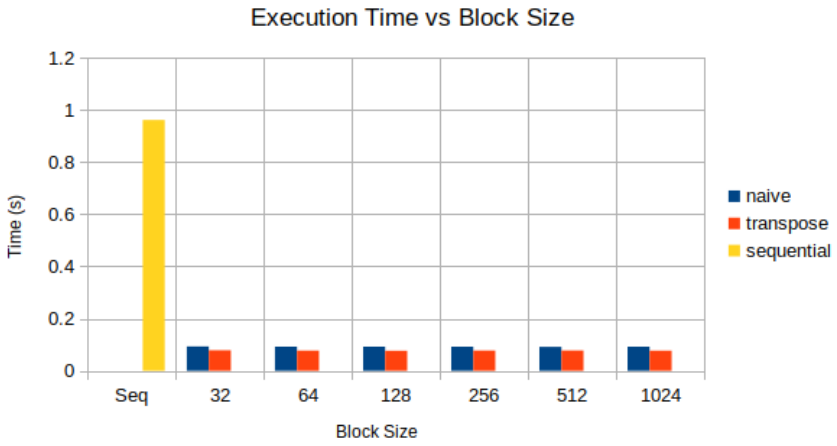
9 Naive



Σχήμα 14: Χρόνος εκτέλεσης

Η παράλληλη υλοποιήση προφανώς είναι πιο γρήγορη από την σειριακή, και αποφορτώνοντας το ’βαρύ’ υπολογιστικό κομμάτι στην GPU παρατηρούμε επιτάχυνση κοντά σε μία τάξη μεγέθους, δηλαδή 10 φορές. Όσον αφορά το μέγεθος των blocks, παρατηρούμε μία σχετική σταθερότητα ανά το block size. Γενικά, η επίδοση ανά το block size επηρεάζεται κυρίως μέσω του occupancy. Επομένως, είτε το occupancy είναι ίδιο και ανεξάρτητο του block size, είναι, δηλαδή, πιθανώς 100%, είτε ο χρόνος που χρειάζεται το πρόγραμμα για να τρέξει αποτελείται κυρίως από χρόνο επικοινωνίας με την GPU, και η GPU εκτελεί το προγραμματιστικό κομμάτι σχετικά άμεσα.

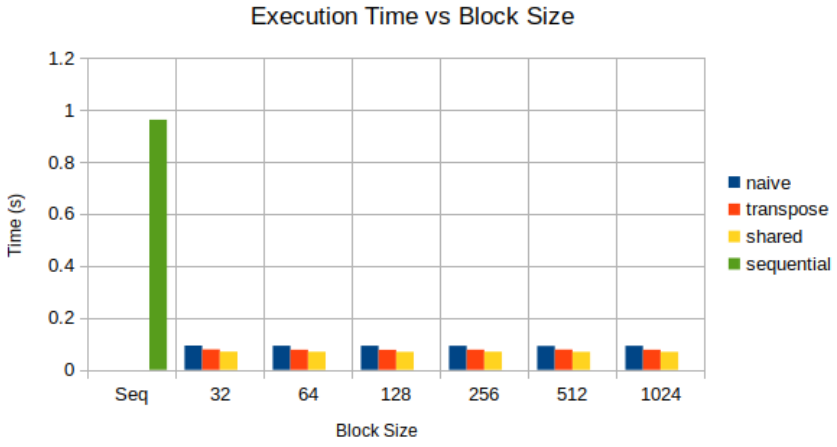
10 Transpose



Σχήμα 15: Χρόνος εκτέλεσης

Η υλοποίηση με transpose αποφαίνεται πάλι να είναι πιο γρήγορη από naive αλλά σχετικά ανεξάρτητη από block size. Αυτό πιθανόν να οφείλεται στο πως γίνεται καλύτερος διαμοιρασμός μνήμης με το transpose και τα δεδομένα που χρειάζεται κάθε warp είναι συνεχόμενα, κάτι που πιθανώς να μη συμβαίνει στο naive approach. Αυτό το γνωρίζουμε επίσης επειδή το transpose δεν αλλάζει την υλοποίηση ή την πολυπλοκότητα της υλοποίηση αυτής καθάυτής, αλλά το πως θα έχει η GPU πρόσβαση στα δεδομένα.

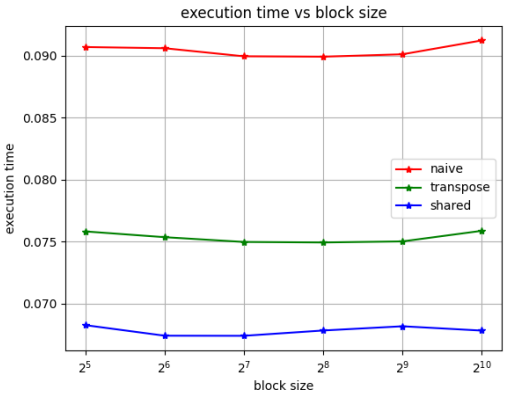
11 Shared



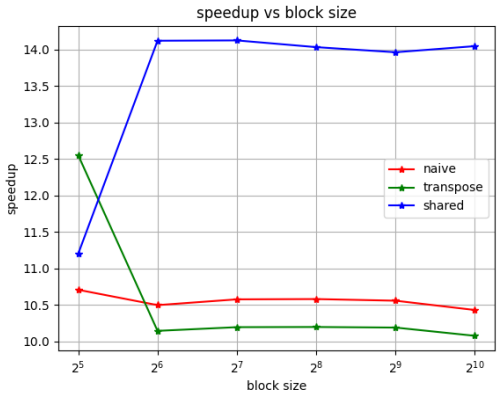
Σχήμα 16: Χρόνος εκτέλεσης

Τέλος, η υλοποίηση με shared memory αποφαίνεται να είναι ακόμα ένα βήμα παραπάνω, από άποψη απόδοσης, έναντι του transpose, και επίσης, πάλι, φαίνεται να είναι σχετικά ανεξάρτητη από το (thread) block size.

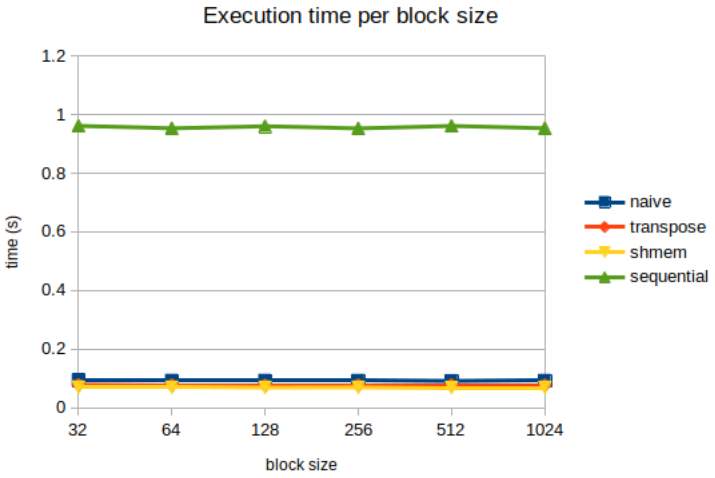
Αυτό, πάλι, οφείλεται στον πολύ μικρό χρόνο της υλοποίησης που ¨καταναλώνεται' σε υπολογισμούς και μάλλον καταναλώνεται σε μεταφορά δεδομένων, το οποίο η shared memory ¨προσπαθεί' να βελτιώσει, δίνοντας τη δυνατότητα σε threads να ενημερώνουν κοινή μνήμη. Πάλι, δηλαδή, βλέπουμε ότι δεν επηρεάζεται ο χρόνος εκτέλεσης τόσο από το block size επειδή αποτελεί πολύ μικρότερο μέρος του χρόνου εκτέλεσης σε σχέση με τον χρόνο επικοινωνίας και ενδεχομένως setup.



(α′) Χρόνος εκτέλεσης



(β′) speedup

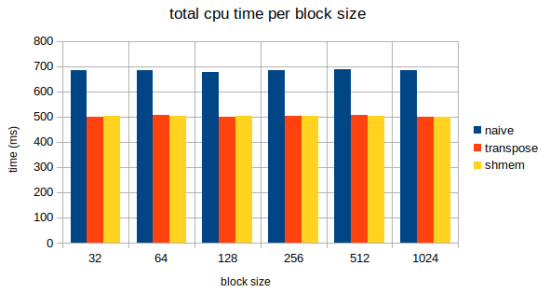


(γ′) Χρόνος εκτέλεσης σε σύγκριση με serial

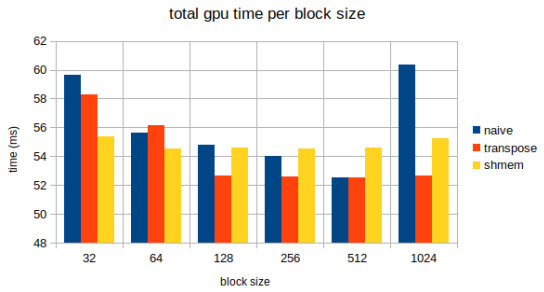
12 Bottleneck Analysis

12.1 2 Coordinates:

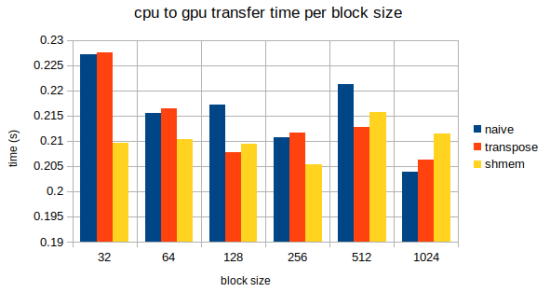
Θα προστεθεί σχολιασμός



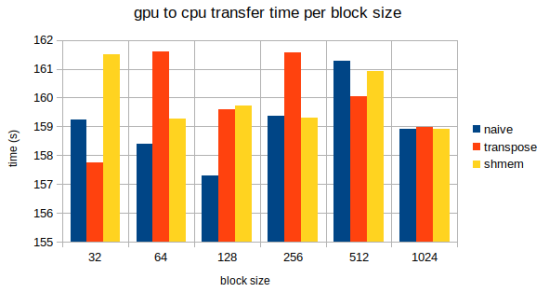
(α′) Χρόνος εκτέλεσης cpu ανά το block size



(β′) Συνολικός Χρόνος υπολογισμών σε GPU



(γ′) Χρόνος μεταφοράς δεδομένων από CPU σε GPU

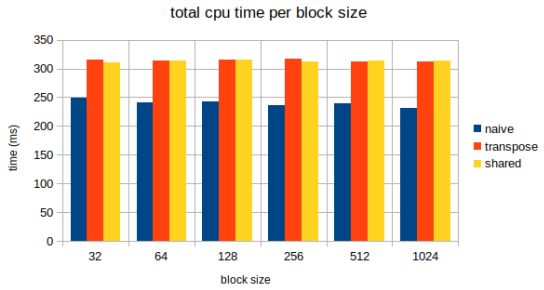


(δ′) Χρόνος μεταφοράς δεδομένων από GPU σε CPU

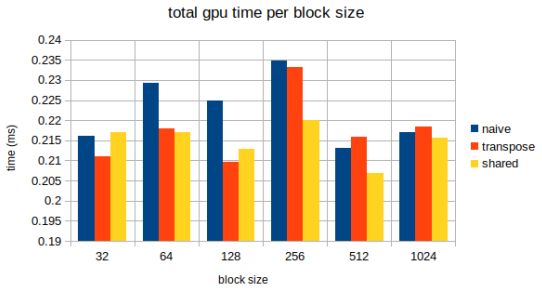
Σχήμα 18: configuration with coordinate dim = 2

12.2 16 coordinates:

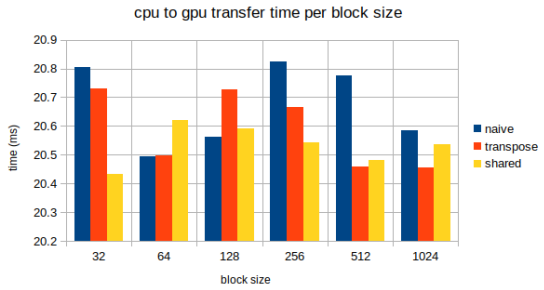
Θα προστεθεί σχολιασμός



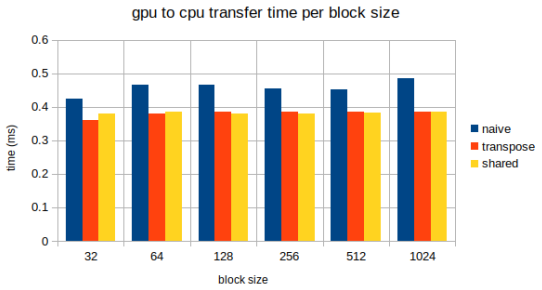
(α′) Συνολικός Χρόνος υπολογισμών σε CPU



(β′) Συνολικός Χρόνος υπολογισμών σε GPU



(γ′) Χρόνος μεταφοράς δεδομένων από CPU σε GPU



(δ′) Χρόνος μεταφοράς δεδομένων από GPU σε CPU

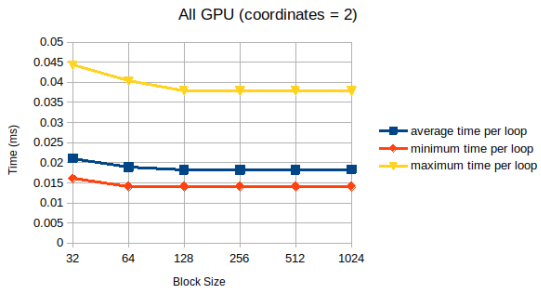
Σχήμα 19: configuration with coordinate dim = 16

13 Bonus: Full Offload (All-GPU) version

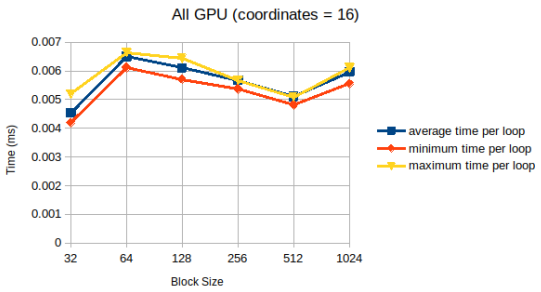
Για την bonus άσκηση 'ALL GPU', παρατηρούμε ότι για περισσότερες συντεταγμένες, εφόσον το πλήθος δεδομένων παραμένει ίδιο (άρα μειώνεται ο αριθμός των αντικειμένων κατά $16/2 = 8$, είναι πιο γρήγορη η εκτέλεση, καθώς οι περισσότερες συντεταγμένες παραλληλοποιούνται πιο εύκολα χωρίς κανένα κόστος επικοινωνίας, ενώ τα περισσότερα δεδομένα πρέπει για την δημιουργία των clusters να επικοινωνούν κιόλας μεταξύ τους.

Η υλοποίηση του update$_c$*entroids* έγινε εν μέρει στην find$_n$*earest*$_c$*luster* όπου εκεί, αμέσως μετά την εύρεση του κοντινότερου cluster γίνεται με χρήση ατομικών πράξεων στην ουσία το reduce(+) των object της κλάσης καθώς και η εύρεση μεγέθους της. Στην συνέχεια, στην update$_c$*entroids* κάθε νήμα αναλαμβάνει ένα cluster και μία διάσταση αυτού και διαιρεί διά τον αριθμό των στοιχείων στην κλάση ώστε να υπολογιστεί ο μέσος όρος. Θα ακολουθήσει το παράρτημα κώδικα

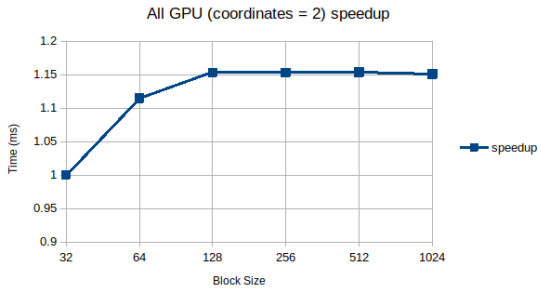
Όσον αφορά το block size, για τις δύο συντεταγμένες, σύμφωνα με τα διαγράμματα, 20α΄, παρατηρούμε



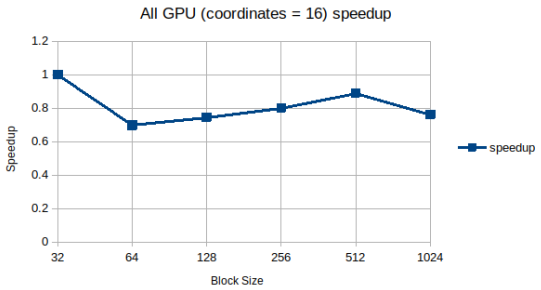
(α΄) Χρόνος εκτέλεσης all GPU, 2 συντεταγμένες



(β΄) Χρόνος εκτέλεσης all GPU, 16 συντεταγμένες



(γ΄) All GPU Speedup, 2 συντεταγμένες



(δ΄) All GPU Speedup, 16 συντεταγμένες

14 Παράρτημα

Skip του κώδικα που ακολουθεί: *Skip Code*

Listing 10: Kmeans GPU Naive

```
__device__ int get_tid(){
  return blockIdx.x * blockDim.x + threadIdx.x; /* TODO: copy me from naive version... */
}

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

  /* TODO: Calculate the euclid_dist of elem=objectId of objects from elem=clusterId from
    clusters*/
    // was empty
    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);
}

__global__ static
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters,
                          double *objects,           //  [numObjs][numCoords]
                          double *deviceClusters,    //  [numClusters][numCoords]
                          int *deviceMembership,          //  [numObjs]
                          double *devdelta)
{

  /* Get the global ID of the thread. */
    int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
    if (tid < numObjs) { // was 1
        int   index, i;
        double dist, min_dist;

        /* find the cluster id that has min distance to object */
        index = 0;
        /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
        min_dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters,
    tid, 0); // was empty

        for (i=1; i<numClusters; i++) {
            /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
            dist = euclid_dist_2(numCoords, numObjs, numClusters, objects, deviceClusters,
    tid, i); // was empty

            /* no need square root */
            if (dist < min_dist) { /* find the min and its array index */
                min_dist = dist;
```

```
57                  index    = i;
58              }
59          }
60
61          if (deviceMembership[tid] != index) {
62            /* TODO: Maybe something is missing here... is this write safe? */
63              atomicAdd(devdelta, 1.0); // was (*devdelta)+= 1.0;
64          }
65
66          /* assign the deviceMembership to object objectId */
67          deviceMembership[tid] = index;
68      }
69  }
70
71
72      const unsigned int numThreadsPerClusterBlock = (numObjs > blockSize)? blockSize: numObjs
        ;
73      /* TODO: Calculate Grid size, e.g. number of blocks. */
74      const unsigned int numClusterBlocks = (numObjs + numThreadsPerClusterBlock - 1) /
        numThreadsPerClusterBlock; // was -1
75      const unsigned int clusterBlockSharedDataSize = 0;
76
77
78      do {
79          timing_internal = wtime();
80
81      /* GPU part: calculate new memberships */
82          #ifdef TIMER_ANALYSIS
83          time_start = wtime();
84          #endif
85
86          /* TODO: Copy clusters to deviceClusters
87          checkCuda(cudaMemcpy(...)); */
88          checkCuda(cudaMemcpy(deviceClusters, clusters,
89                numClusters*numCoords*sizeof(double), cudaMemcpyHostToDevice));
90
91          checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
92
93          #ifdef TIMER_ANALYSIS
94          TIME(cpu_gpu_time);
95          #endif
96
97          find_nearest_cluster
98              <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
99              (numCoords, numObjs, numClusters,
100              deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
101
102         cudaDeviceSynchronize(); checkLastCudaError();
103
104         #ifdef TIMER_ANALYSIS
105         TIME(gpu_time);
106         #endif
107
108     /* TODO: Copy deviceMembership to membership
109         checkCuda(cudaMemcpy(...)); */
110         checkCuda(cudaMemcpy(membership, deviceMembership,
111               numObjs*sizeof(int), cudaMemcpyDeviceToHost));
112
113       /* TODO: Copy dev_delta_ptr to &delta
114         checkCuda(cudaMemcpy(...)); */
115         checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
116               sizeof(double), cudaMemcpyDeviceToHost));
117
118         #ifdef TIMER_ANALYSIS
119         TIME(gpu_cpu_time);
```

```
120         #endif
121
122     /* CPU part: Update cluster centers*/
123
124         for (i=0; i<numObjs; i++) {
125             /* find the array index of nestest cluster center */
126             index = membership[i];
127
128             /* update new cluster centers : sum of objects located within */
129             newClusterSize[index]++;
130             for (j=0; j<numCoords; j++)
131                 newClusters[index][j] += objects[i*numCoords + j];
132         }
133
134         /* average the sum and replace old cluster centers with newClusters */
135         for (i=0; i<numClusters; i++) {
136             for (j=0; j<numCoords; j++) {
137                 if (newClusterSize[i] > 0)
138                     clusters[i*numCoords + j] = newClusters[i][j] / newClusterSize[i];
139                 newClusters[i][j] = 0.0;   /* set back to 0 */
140             }
141             newClusterSize[i] = 0;   /* set back to 0 */
142         }
143
144         delta /= numObjs;
145         //printf("delta is %f - ", delta);
146         loop++;
147         ...
148     } while (delta > threshold && loop < loop_threshold);
```

Listing 11: Kmeans GPU Transpose

```
__device__ int get_tid(){
  return blockIdx.x * blockDim.x + threadIdx.x; /* TODO: copy me from naive version... */
}

/* square of Euclid distance between two multi-dimensional points using column-base format
    */
  __host__ __device__ inline static
double euclid_dist_2_transpose(int numCoords,
    int    numObjs,
    int    numClusters,
    double *objects,     // [numCoords][numObjs]
    double *clusters,    // [numCoords][numClusters]
    int    objectId,
    int    clusterId)
{
  int i;
  double 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[i*numObjs + objectId] - clusters[i*numClusters + clusterId]) *
      (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]);

  return(ans);
}

  __global__ static
void find_nearest_cluster(int numCoords,
    int numObjs,
    int numClusters,
    double *objects,           //  [numCoords][numObjs]
    double *deviceClusters,    //  [numCoords][numClusters]
    int *membership,          //  [numObjs]
    double *devdelta)
{
  /* TODO: copy me from naive version... */

  /* Get the global ID of the thread. */
  int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
  if (tid < numObjs) { // was 1
    int   index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
    deviceClusters, tid, 0); // was empty

    for (i=1; i<numClusters; i++) {
      /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
      dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
    deviceClusters, tid, i); // was empty

      /* no need square root */
      if (dist < min_dist) { /* find the min and its array index */
        min_dist = dist;
        index    = i;
      }
    }
```

```
60
61      if (membership[tid] != index) {
62        /* TODO: Maybe something is missing here... is this write safe? */
63        atomicAdd(devdelta, 1.0); // was (*devdelta)+= 1.0;
64      }
65
66      /* assign the deviceMembership to object objectId */
67      membership[tid] = index;
68    }
69  }
70  ...
71  /* TODO: Transpose dims */
72  double  **dimObjects = NULL; //calloc_2d(...) -> [numCoords][numObjs]
73  double  **dimClusters = NULL;  //calloc_2d(...) -> [numCoords][numClusters]
74  double  **newClusters = NULL;  //calloc_2d(...) -> [numCoords][numClusters]
75  dimObjects  = (double**) calloc_2d(numCoords, numObjs, sizeof(double));
76  dimClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
77  newClusters = (double**) calloc_2d(numCoords, numClusters, sizeof(double));
78  ...
79
80  //  TODO: Copy objects given in [numObjs][numCoords] layout to new
81  //  [numCoords][numObjs] layout
82  for (i = 0; i < numCoords; i++) {
83    for (j = 0; j < numObjs; j++) {
84      dimObjects[i][j] = objects[j*numCoords + i];
85    }
86  }
87  ...
88  do {
89    timing_internal = wtime();
90
91    /* GPU part: calculate new memberships */
92
93  #ifdef TIMER_ANALYSIS
94    time_start = wtime();
95  #endif
96
97    /* TODO: Copy clusters to deviceClusters
98       checkCuda(cudaMemcpy(...)); */
99    checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
100         numClusters*numCoords*sizeof(double), cudaMemcpyHostToDevice));
101
102   checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
103
104 #ifdef TIMER_ANALYSIS
105   TIME(cpu_gpu_time);
106 #endif
107
108   find_nearest_cluster
109     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
110     (numCoords, numObjs, numClusters,
111      deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
112
113   cudaDeviceSynchronize(); checkLastCudaError();
114
115 #ifdef TIMER_ANALYSIS
116   TIME(gpu_time);
117 #endif
118
119   /* TODO: Copy deviceMembership to membership
120      checkCuda(cudaMemcpy(...)); */
121   checkCuda(cudaMemcpy(membership, deviceMembership,
122         numObjs*sizeof(int), cudaMemcpyDeviceToHost));
123
124   /* TODO: Copy dev_delta_ptr to &delta
```

```
125       checkCuda(cudaMemcpy(...)); */
126     checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
127           sizeof(double), cudaMemcpyDeviceToHost));
128 
129 #ifdef TIMER_ANALYSIS
130     TIME(gpu_cpu_time);
131 #endif
132 
133     /* CPU part: Update cluster centers*/
134 
135     for (i=0; i<numObjs; i++) {
136       /* find the array index of nestest cluster center */
137       index = membership[i];
138 
139       /* update new cluster centers : sum of objects located within */
140       newClusterSize[index]++;
141       for (j=0; j<numCoords; j++)
142         newClusters[j][index] += objects[i*numCoords + j];
143     }
144 
145     /* average the sum and replace old cluster centers with newClusters */
146     for (i=0; i<numClusters; i++) {
147       for (j=0; j<numCoords; j++) {
148         if (newClusterSize[i] > 0)
149           dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
150         newClusters[j][i] = 0.0;   /* set back to 0 */
151       }
152       newClusterSize[i] = 0;   /* set back to 0 */
153     }
154 
155     delta /= numObjs;
156     //printf("delta is %f - ", delta);
157     loop++;
158     //printf("completed loop %d\n", loop);
159     ...
160 } while (delta > threshold && loop < loop_threshold);
161 
162 /*TODO: Update clusters using dimClusters. Be carefull of layout!!! clusters[numClusters][
        numCoords] vs dimClusters[numCoords][numClusters] */
163 for (i = 0; i < numCoords; i++) {
164     for (j = 0; j < numClusters; j++) {
165       clusters[j*numCoords + i] = dimClusters[i][j];
166     }
167 }
```

Listing 12: Kmeans GPU Shared

```
__device__ int get_tid(){
  return blockIdx.x * blockDim.x + threadIdx.x; /* TODO: copy me from naive version... */
}

/* square of Euclid distance between two multi-dimensional points using column-base format
    */
  __host__ __device__ inline static
double euclid_dist_2_transpose(int numCoords,
    int    numObjs,
    int    numClusters,
    double *objects,     // [numCoords][numObjs]
    double *clusters,    // [numCoords][numClusters]
    int    objectId,
    int    clusterId)
{
  int i;
  double ans=0.0;

  /* TODO: Copy me from transpose version*/

  /* 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[i*numObjs + objectId] - clusters[i*numClusters + clusterId]) *
      (objects[i*numObjs + objectId] - clusters[i*numClusters + clusterId]);

  return(ans);
}
  __global__ static
void find_nearest_cluster(int numCoords,
    int numObjs,
    int numClusters,
    double *objects,           //  [numCoords][numObjs]
    double *deviceClusters,    //  [numCoords][numClusters]
    int *deviceMembership,          //  [numObjs]
    double *devdelta)
{
  extern __shared__ double shmemClusters[];

  /* TODO: Copy deviceClusters to shmemClusters so they can be accessed faster.
BEWARE: Make sure operations is complete before any thread continues... */
  for (int i = 0; i < numClusters; i++) {
    for (int j = 0; j < numCoords; j++) {
      shmemClusters[j * numClusters + i] = deviceClusters[j * numClusters + i];
    }
  }

  __syncthreads();

  /* Get the global ID of the thread. */
  int tid = get_tid();

  /* TODO: Maybe something is missing here... should all threads run this? */
  if (tid < numObjs) { // was 1
    int   index, i;
    double dist, min_dist;

    /* find the cluster id that has min distance to object */
    index = 0;
    /* TODO: call min_dist = euclid_dist_2(...) with correct objectId/clusterId */
    min_dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects,
    shmemClusters, tid, 0); // was empty

```

```
61      for (i=1; i<numClusters; i++) {
62        /* TODO: call dist = euclid_dist_2(...) with correct objectId/clusterId */
63        dist = euclid_dist_2_transpose(numCoords, numObjs, numClusters, objects, shmemClusters
      , tid, i); // was empty
64
65        /* no need square root */
66        if (dist < min_dist) { /* find the min and its array index */
67          min_dist = dist;
68          index    = i;
69        }
70      }
71
72      if (deviceMembership[tid] != index) {
73        /* TODO: Maybe something is missing here... is this write safe? */
74        atomicAdd(devdelta, 1.0); // was (*devdelta)+= 1.0;
75      }
76
77      /* assign the deviceMembership to object objectId */
78      deviceMembership[tid] = index;
79    }
80  }
81  ...
82  /*  Define the shared memory needed per block.
83    - BEWARE: We can overrun our shared memory here if there are too many
84    clusters or too many coordinates!
85    - This can lead to occupancy problems or even inability to run.
86    - Your exercise implementation is not requested to account for that (e.g. always assume
      deviceClusters fit in shmemClusters */
87  const unsigned int clusterBlockSharedDataSize = numClusters*numCoords*sizeof(double);
88  ...
89  do {
90    timing_internal = wtime();
91    /* GPU part: calculate new memberships */
92  #ifdef TIMER_ANALYSIS
93    time_start = wtime();
94  #endif
95    /* TODO: Copy clusters to deviceClusters
96       checkCuda(cudaMemcpy(...)); */
97    checkCuda(cudaMemcpy(deviceClusters, dimClusters[0],
98          clusterBlockSharedDataSize, cudaMemcpyHostToDevice));
99
100   checkCuda(cudaMemset(dev_delta_ptr, 0, sizeof(double)));
101 #ifdef TIMER_ANALYSIS
102   TIME(cpu_gpu_time);
103 #endif
104   //printf("Launching find_nearest_cluster Kernel with grid_size = %d, block_size = %d,
      shared_mem = %d KB\n", numClusterBlocks, numThreadsPerClusterBlock,
      clusterBlockSharedDataSize/1000);
105   find_nearest_cluster
106     <<< numClusterBlocks, numThreadsPerClusterBlock, clusterBlockSharedDataSize >>>
107     (numCoords, numObjs, numClusters,
108      deviceObjects, deviceClusters, deviceMembership, dev_delta_ptr);
109
110   cudaDeviceSynchronize(); checkLastCudaError();
111   //printf("Kernels complete for itter %d, updating data in CPU\n", loop);
112 #ifdef TIMER_ANALYSIS
113   TIME(gpu_time);
114 #endif
115   /* TODO: Copy deviceMembership to membership
116      checkCuda(cudaMemcpy(...)); */
117   checkCuda(cudaMemcpy(membership, deviceMembership,
118         numObjs*sizeof(int), cudaMemcpyDeviceToHost));
119
120   /* TODO: Copy dev_delta_ptr to &delta
121      checkCuda(cudaMemcpy(...)); */
```

```
122   checkCuda(cudaMemcpy(&delta, dev_delta_ptr,
123         sizeof(double), cudaMemcpyDeviceToHost));
124 #ifdef TIMER_ANALYSIS
125   TIME(gpu_cpu_time);
126 #endif
127   /* CPU part: Update cluster centers*/
128   for (i=0; i<numObjs; i++) {
129     /* find the array index of nestest cluster center */
130     index = membership[i];
131
132     /* update new cluster centers : sum of objects located within */
133     newClusterSize[index]++;
134     for (j=0; j<numCoords; j++)
135       newClusters[j][index] += objects[i*numCoords + j];
136   }
137   /* average the sum and replace old cluster centers with newClusters */
138   for (i=0; i<numClusters; i++) {
139     for (j=0; j<numCoords; j++) {
140       if (newClusterSize[i] > 0)
141         dimClusters[j][i] = newClusters[j][i] / newClusterSize[i];
142       newClusters[j][i] = 0.0;   /* set back to 0 */
143     }
144     newClusterSize[i] = 0;   /* set back to 0 */
145   }
146   delta /= numObjs;
147   loop++;
148   ...
149 } while (delta > threshold && loop < loop_threshold);
150
151 /*TODO: Update clusters using dimClusters. Be carefull of layout!!! clusters[numClusters][
      numCoords] vs dimClusters[numCoords][numClusters] */
152 for (i = 0; i < numCoords; i++) {
153   for (j = 0; j < numClusters; j++) {
154     clusters[j*numCoords + i] = dimClusters[i][j];
155   }
156 }
```

Listing 13: Kmeans All GPU

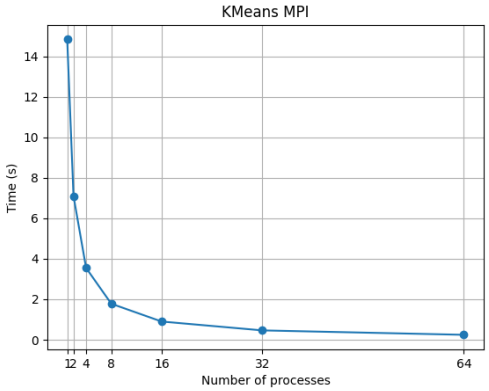
```
__global__ static
void find_nearest_cluster(int numCoords,
                          int numObjs,
                          int numClusters,
                          double *deviceobjects,           //  [numCoords][numObjs]
/*
                         TODO: If you choose to do (some of) the new centroid calculation here, you will need some extra parameters here (from "update_centroids").
*/
                          int *devicenewClusterSize,           //  [numClusters]
                          double *devicenewClusters,    //  [numCoords][numClusters]
//added above two
                          double *deviceClusters,    //  [numCoords][numClusters]
                          int *deviceMembership,           //  [numObjs]
                          double *devdelta)
{
     extern __shared__ double shmemClusters[];

  /* TODO: copy me from shared version... */
  . . .
      /* TODO: additional steps for calculating new centroids in GPU? */
        atomicAdd(&devicenewClusterSize[index], 1);
        for (i = 0; i < numCoords; ++i)
            atomicAdd(&devicenewClusters[i*numClusters + index], deviceobjects[i*numObjs + tid]);
    }
}

__global__ static
void update_centroids(int numCoords,
                          int numClusters,
                          int *devicenewClusterSize,           //  [numClusters]
                          double *devicenewClusters,    //  [numCoords][numClusters]
                          double *deviceClusters)    //  [numCoords][numClusters])
{

    /* TODO: additional steps for calculating new centroids in GPU? */
//was empty
    const int tid = get_tid();
    if (tid >= numClusters*numCoords) return;
    int cluster = tid % numClusters; // tid = coord*numClusters + cluster, which makes access bellow fast af
    if (devicenewClusterSize[cluster] > 0)
        deviceClusters[tid] = devicenewClusters[tid]/devicenewClusterSize[cluster];
    devicenewClusters[tid] = 0.0;
    // apparently synchronizing here doesn't change the results (also each thread does it lol, could add if (coord == 0))
    devicenewClusterSize[cluster] = 0;
}
```

Μέρος V
Άσκηση 5

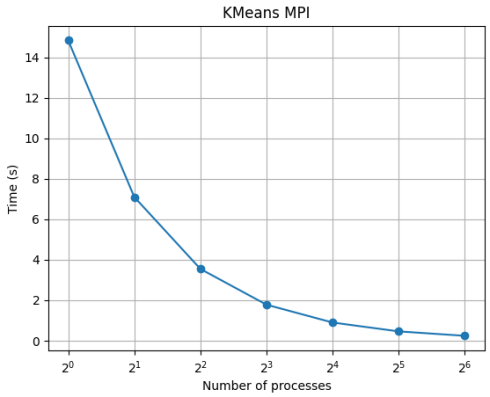
15 K-MEANS

15.1 Υλοποίηση MPI

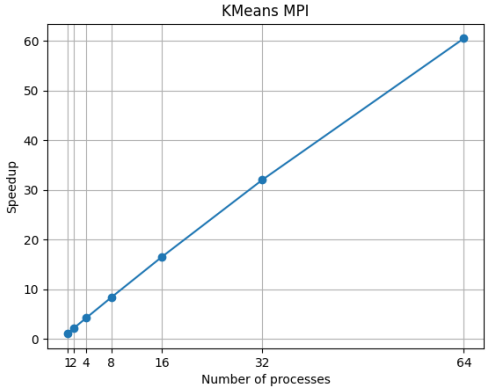
Φαίνεται από τα διαγράμματα ότι η κλιμάκωση του αλγορίθμου είναι πολύ ικανοποιητική, καθώς επιτυγχάνεται σχεδόν ιδανικό speedup.



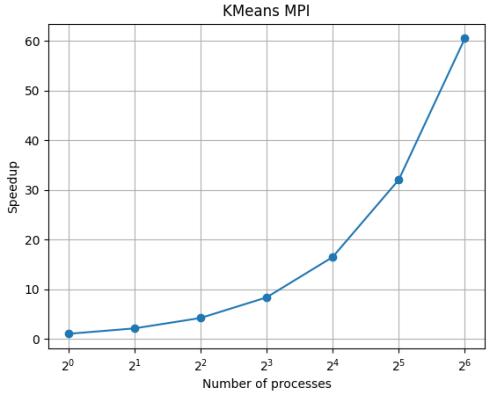
(α′) Χρόνος εκτέλεσης, σύγκριση χωρίς σειριακή εκτέλεση



(β′) Χρόνος εκτέλεσης (λογαριθμική κλίμακα)



(γ′) Speedup



(δ′) Speedup (λογαριθμική κλίμακα)

15.2 BONUS ερώτημα: Σύγκριση με OpenMP

Αγνοώντας τις συγκεκριμένες τιμές των μετρήσεων, φαίνεται ότι η υλοποίηση με MPI, δηλαδή προγραμματίζοντας σε μοντέλο κατανεμημένης μνήμης, κλιμακώνει καλύτερα και δεν δείχνει τάση αποκλιμάκωσης σε μεγαλύτερα πλήθη διεργασιών και δεδομένων, όπως κάνει η υλοποίηση με OpenMP. Ενδεικτικά είναι τα διαγράμματα 8 και 21δ′.

16 Διάδοση Θερμότητας σε δύο διαστάσεις

16.1 Μετρήσεις με έλεγχο σύγκλισης

Στο σχήμα 22 φαίνονται τα δεδομένα από την εκτέλεση του παράλληλου αλγορίθμου Jacobi για πλέγμα διαστάσεων 1024×1024 και 64 διεργασίες. Φαίνεται ότι το μεγαλύτερο μέρος του χρόνου καταναλώνεται στην επικοινωνία μεταξύ διεργασιών. Αυτό οφείλεται κυρίως στον έλεγχο σύγκλισης, τον οποίον επιτελεί καθεμία διεργασία ξεχωριστά για το τμήμα του πλέγματος για το οποίο ευθύνεται, και στην χρήση της κλήσης MPI_Allreduce για τον συνδυασμό των αποτελεσμάτων και την συνέχεια της εκτέλεσης.

Σημειώνεται επίσης ότι οι απαιτούμενες επαναλήψεις για την σύγκλιση είναι στην τάξη του εκατομμυρίου και ότι η θερμοκρασία στο κέντρο της επιφάνειας είναι μη μηδενική.

```
Jacobi X 1024 Y 1024 Px 8 Py 8 Iter 798201
ComputationTime 39.826432 MessagingTime 179.486283 TotalTime 222.866609 midpoint 5.431022
```

Σχήμα 22: Δεδομένα Εξόδου Jacobi 1024×1024

16.2 Μετρήσεις χωρίς έλεγχο σύγκλισης

Εδώ σημειώνεται ότι, όπως φαίνεται στο σχήμα 23 η θερμοκρασία στο κέντρο της επιφάνειας προκύπτει μηδενική, κάτι το οποίο σημαίνει ότι ο αλγόριθμος μάλλον δεν έχει συγκλίνει και διακόπηκε η εκτέλεσή του νωρίς. Αυτό είναι αναμενόμενο, καθώς, όπως σημειώθηκε προηγουμένως, οι απαιτούμενες επαναλήψεις για σύγκλιση είναι στην τάξη του εκατομμυρίου και το όριο εν προκειμένω είναι μόλις 256 επαναλήψεις.

```
Jacobi X 2048 Y 2048 Px 1 Py 1 Iter 256 CompTime 8.417019 MessTime 0.000289 Total 8.417435 midpoint 0.000
Jacobi X 2048 Y 2048 Px 2 Py 1 Iter 256 CompTime 4.212652 MessTime 0.107972 Total 4.319512 midpoint 0.000
Jacobi X 2048 Y 2048 Px 2 Py 2 Iter 256 CompTime 2.111995 MessTime 0.251033 Total 2.360420 midpoint 0.000
Jacobi X 2048 Y 2048 Px 4 Py 2 Iter 256 CompTime 1.052504 MessTime 0.336826 Total 1.384202 midpoint 0.000
Jacobi X 2048 Y 2048 Px 4 Py 4 Iter 256 CompTime 0.663224 MessTime 0.364288 Total 1.012555 midpoint 0.000
Jacobi X 2048 Y 2048 Px 8 Py 4 Iter 256 CompTime 0.290805 MessTime 0.373848 Total 0.628165 midpoint 0.000
Jacobi X 2048 Y 2048 Px 8 Py 8 Iter 256 CompTime 0.053300 MessTime 0.335123 Total 0.385897 midpoint 0.000
Jacobi X 4096 Y 4096 Px 1 Py 1 Iter 256 CompTime 33.634679 MessTime 0.000419 Total 33.635258 midpoint 0.000
Jacobi X 4096 Y 4096 Px 2 Py 1 Iter 256 CompTime 16.832444 MessTime 0.152904 Total 16.985511 midpoint 0.000
Jacobi X 4096 Y 4096 Px 2 Py 2 Iter 256 CompTime 8.436994 MessTime 0.731161 Total 9.168280 midpoint 0.000
Jacobi X 4096 Y 4096 Px 4 Py 2 Iter 256 CompTime 4.224044 MessTime 1.028663 Total 5.246204 midpoint 0.000
Jacobi X 4096 Y 4096 Px 4 Py 4 Iter 256 CompTime 3.039436 MessTime 1.134723 Total 4.155380 midpoint 0.000
Jacobi X 4096 Y 4096 Px 8 Py 4 Iter 256 CompTime 2.785139 MessTime 1.475225 Total 3.954300 midpoint 0.000
Jacobi X 4096 Y 4096 Px 8 Py 8 Iter 256 CompTime 1.979021 MessTime 2.434826 Total 3.208476 midpoint 0.000
Jacobi X 6144 Y 6144 Px 1 Py 1 Iter 256 CompTime 75.709465 MessTime 0.000452 Total 75.710063 midpoint 0.000
Jacobi X 6144 Y 6144 Px 2 Py 1 Iter 256 CompTime 37.881797 MessTime 0.216098 Total 38.083830 midpoint 0.000
Jacobi X 6144 Y 6144 Px 2 Py 2 Iter 256 CompTime 18.976311 MessTime 1.516096 Total 20.477842 midpoint 0.000
Jacobi X 6144 Y 6144 Px 4 Py 2 Iter 256 CompTime 9.496531 MessTime 2.161902 Total 11.645876 midpoint 0.000
Jacobi X 6144 Y 6144 Px 4 Py 4 Iter 256 CompTime 6.850163 MessTime 2.372009 Total 9.190001 midpoint 0.000
Jacobi X 6144 Y 6144 Px 8 Py 4 Iter 256 CompTime 6.443347 MessTime 3.461622 Total 8.937407 midpoint 0.000
Jacobi X 6144 Y 6144 Px 8 Py 8 Iter 256 CompTime 4.797882 MessTime 5.542366 Total 7.580173 midpoint 0.000
```

Σχήμα 23: Δεδομένα Εξόδου Jacobi για διάφορες διαστάσεις επιφάνειες.

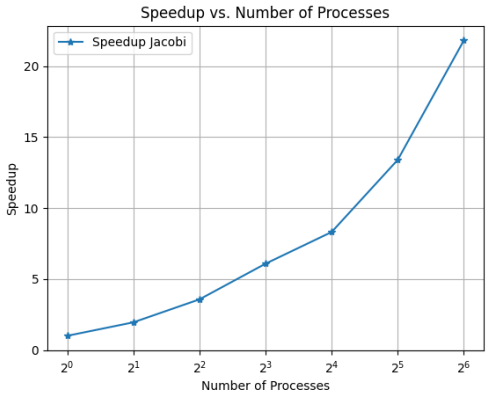
Στον πίνακα μεγέθους 2048×2048 παρατηρούμε ικανοποιητική κλιμάκωση του χρόνου εκτέλεσης και πολύ καλή κλιμάκωση στον χρόνο υπολογισμού, με τον χρόνο επικοινωνίας να μένει περίπου σταθερός (βλ. 24α΄). Ιδιαίτερα παρατηρούμε ότι στα 64 τηρεαδς έχουμε μείωση του χρόνου κατά περισσότερο από το διπλάσιο σε σχέση με τα 32 threads (25α΄). Αυτό μπορεί να οφείλεται στο ότι αφού είναι μικρός ο πίνακας πιθανόν να έχει μοιραστεί

τέλεια στις cache των επεξεργαστών. Πέρα από το computational time, ο υπόλοιπος μη-υπολογιστικός χρόνος οφείλεται σε overhead, setup, επικοινωνία μεταξύ διεργασιών κλπ, που είναι σχετικά σταθερά για τον πίνακα μεγέθους 2048 λόγω του μικρού σχετικά φόρτου δεδομένων.

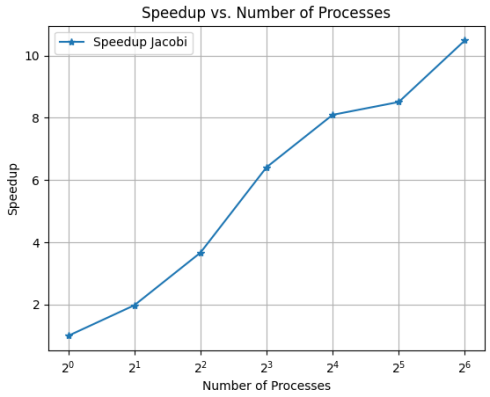
Στον πίνακα μεγέθους 4096×4096, παρατηρούμε τέλεια κλιμάκωση της υπολογιστικής ταχύτητας μέχρι και τα 8 processes, βλέπε 24β΄, και πως συνεχίζει η κλιμάκωση, αν και όχι ιδανική, μέχρι και τα 64 processes για τα οποία έχουμε δεδομένα. Η μη ιδανική φύση της κλιμάκωσης πιθανόν να έχει σχέση με το γεγονός ότι το μέγεθος της cache πιθανόν να μην εξυπηρετεί το μέγεθος του -πλέον- προβλήματος. Αυτό φαίνεται και στην αύξηση του κόστους επικοινωνίας ανάλογα με τον αριθμό διεργασιών, όπως φαίνεται στο 25β΄, που έχει σχέση με το ότι οι πλέον πολλές διεργασίες έχουν μεγάλο κόστος επικοινωνίας ακριβώς επειδή δεν χωράνε στην cache, κάτι που περιπλέκει την επικοινωνία με τον αύξοντα αριθμό τους.

Τελείως αντίστοιχα με το 4096, για πλέγμα 6144×6144 προκύπτει το ίδιο πρόβλημα, δηλαδή το hardware/setup δεν εξυπηρετούν εξίσου αυτό το μέγεθος πίνακα, με αποτέλεσμα όταν αυξηθεί ο αριθμός διεργασιών η επικοινωνία να γίνει αρκετά πιο δύσκολη και να κοστίσει.

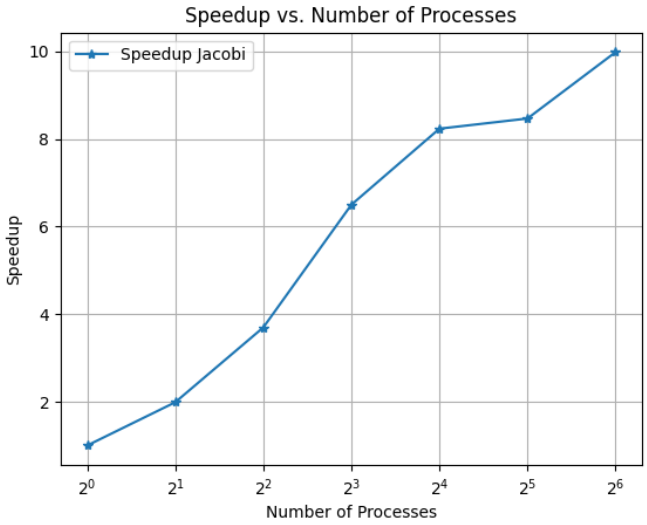
Σημειώνεται ότι, αν και οι συμπεριφορές, ειδικά των προβλημάτων μεγέθους 4096 και 6144, μοιάζουν, κατά απόλυτη τιμή ο χρόνος επικοινωνίας πράγματι πολλαπλασιάζεται, ξεκινόντας από περίπου μισό δευτερόλεπτο στο πιο μικρό πρόβλημα και καταλήγοντας κοντά στα 6 δευτερόλεπτα στο πιο μεγάλο.



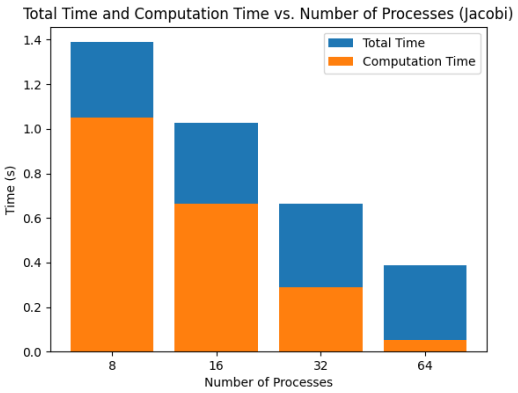
(α΄) speedup 2048×2048



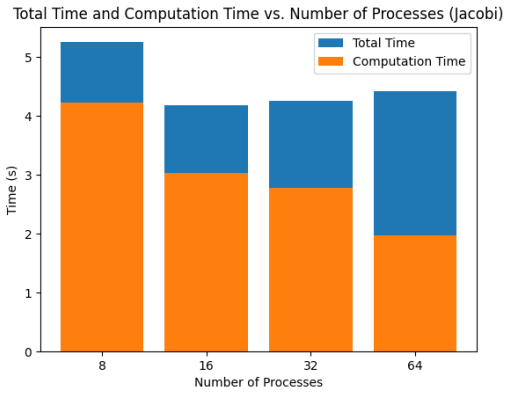
(β΄) speedup 4096×4096



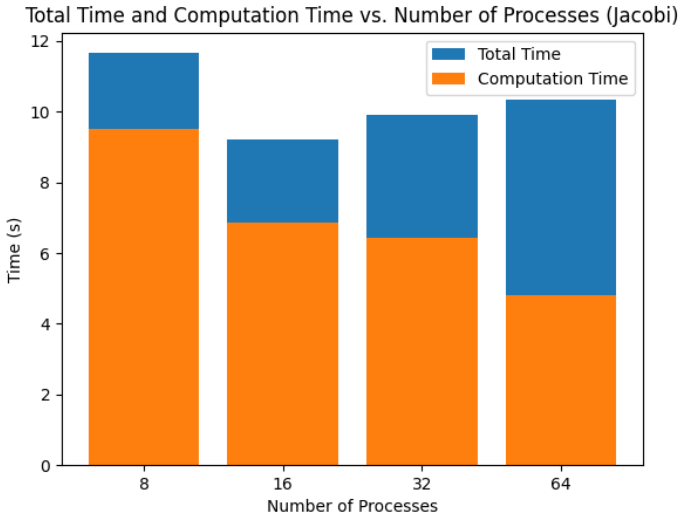
(γ΄) speedup 6144×6144



(α′) speedup 2048×2048



(β′) speedup 4096×4096



(γ′) speedup 6144×6144

17 Παράρτημα

Listing 14: Kmeans MPI

```
...
do {
    // before each loop, set cluster data to 0
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++)
            rank_newClusters[i*numCoords + j] = 0.0;
        rank_newClusterSize[i] = 0;
    }
    rank_delta = 0.0;
    for (i=0; i<numObjs; i++) {
        // find the array index of nearest cluster center
        index=find_nearest_cluster(numClusters,numCoords,&objects[i*numCoords],clusters);
        // if membership changes, increase rank_delta by 1
        if (membership[i] != index)
            rank_delta += 1.0;
        // assign the membership to object i
        membership[i] = index;
        // update new cluster centers : sum of objects located within
        rank_newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            rank_newClusters[index*numCoords + j] += objects[i*numCoords + j];
    }
    /*
     * TODO: Perform reduction of cluster data (rank_newClusters, rank_newClusterSize) from
    local arrays to shared.
     */
    MPI_Allreduce(rank_newClusters, newClusters, numClusters*numCoords, MPI_DOUBLE, MPI_SUM,
     MPI_COMM_WORLD);
    MPI_Allreduce(rank_newClusterSize, newClusterSize, numClusters, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);
    // average the sum and replace old cluster centers with newClusters
    for (i=0; i<numClusters; i++) {
        if (newClusterSize[i] > 0) {
            for (j=0; j<numCoords; j++) {
                clusters[i*numCoords + j] = newClusters[i*numCoords + j] / newClusterSize[i
    ];
            }
        }
    }
    /*
     * TODO: Perform reduction from rank_delta variable to delta variable, that will be used
     for convergence check.
     */
    MPI_Allreduce(&rank_delta, &delta, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    // Get fraction of objects whose membership changed during this loop. This is used as a
    convergence criterion.
    delta /= numObjs;
    loop++;
} while (delta > threshold && loop < loop_threshold);
...
```

Listing 15: Jacobi MPI setup

```
...
//-----Ensure that u_current and u_previous are both initialized-----//
init2d(u_current,local[0]+2,local[1]+2);
init2d(u_previous,local[0]+2,local[1]+2);
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block, &(u_previous[1][1]), 1,
     local_block, 0, MPI_COMM_WORLD);
MPI_Scatterv(&(U[0][0]), scattercounts, scatteroffset, global_block, &(u_current[1][1]), 1,
     local_block, 0, MPI_COMM_WORLD);

if (rank==0) free2d(U);
MPI_Datatype row;
MPI_Type_contiguous(local[1], MPI_DOUBLE, &dummy);
MPI_Type_create_resized(dummy, 0, sizeof(double), &row);
MPI_Type_commit(&row);
MPI_Datatype column;
MPI_Type_vector(local[0], 1, local[1]+2, MPI_DOUBLE, &dummy);
MPI_Type_create_resized(dummy, 0, sizeof(double), &column);
MPI_Type_commit(&column);

//-----Find the 4 neighbors with which a process exchanges messages-----//
int north, south, east, west;

// the processes "in the middle" of the grid have 4 neighbors
// the processes on the edges have 3 neighbors
// the processes on the corners have 2 neighbors

MPI_Cart_shift(CART_COMM, 0, 1, &north, &south);
MPI_Cart_shift(CART_COMM, 1, 1, &west, &east);
...
```

Listing 16: Jacobi MPI ranges

```
...
//---Define the iteration ranges per process-------//
int i_min,i_max,j_min,j_max;
/*Three types of ranges:
-internal processes
-boundary processes
-boundary processes and padded global array
*/
// these are the ranges that will be used later for the double for loop
i_min = (north == MPI_PROC_NULL) ? 2 : 1;
j_min = (west == MPI_PROC_NULL) ? 2 : 1;
i_max = (south == MPI_PROC_NULL) ? local[0] - (global_padded[0] - global[0] + 1): local[0];
j_max = (east == MPI_PROC_NULL) ? local[1]- (global_padded[1] - global[1] + 1): local[1];
// because of the ghost cells, a 'normal' 'internal' process would start from its
// first row and column and end just before its last row and column
// if the process is on some edge, then the boundaries are shifted by one row (+) or column
    (-)
...
```

Listing 17: Jacobi MPI core

```
//-----Computational core-----//
gettimeofday(&tts, NULL);
#ifdef TEST_CONV
for (t=0;t<T && !global_converged;t++) {
#endif
#ifndef TEST_CONV
#undef T
#define T 256
for (t=0;t<T;t++) {
#endif
```

```
11      swap = u_previous;
12      u_previous = u_current;
13      u_current = swap;
14      /*Compute and Communicate*/
15      int count = 0;
16      MPI_Request req[8];
17      MPI_Status stat[8];
18      gettimeofday(&tms,NULL);
19      if (north != MPI_PROC_NULL)
20      {
21          MPI_Isend(&u_previous[1][1], 1, row, north, 0, MPI_COMM_WORLD, &req[count++]);
22          MPI_Irecv(&u_previous[0][1], 1, row, north, 0, MPI_COMM_WORLD, &req[count++]);
23      }
24      if (south != MPI_PROC_NULL)
25      {
26          MPI_Isend(&u_previous[local[0]][1], 1, row,south, 0, MPI_COMM_WORLD, &req[count++]);
27          MPI_Irecv(&u_previous[local[0]+1][1],1,row,south, 0, MPI_COMM_WORLD, &req[count++]);
28      }
29      if (east != MPI_PROC_NULL)
30      {
31          MPI_Isend(&u_previous[1][local[1]], 1,column,east,0,MPI_COMM_WORLD, &req[count++]);
32          MPI_Irecv(&u_previous[1][local[1]+1],1,column,east,0,MPI_COMM_WORLD, &req[count++]);
33      }
34      if (west != MPI_PROC_NULL)
35      {
36          MPI_Isend(&u_previous[1][1], 1, column, west, 0, MPI_COMM_WORLD, &req[count++]);
37          MPI_Irecv(&u_previous[1][0], 1, column, west, 0, MPI_COMM_WORLD, &req[count++]);
38      }
39      // Isend / Irecv do not block.
40      MPI_Waitall(count, req, stat);
41      gettimeofday(&tmf,NULL);
42      tmsg += (tmf.tv_sec-tms.tv_sec)+(tmf.tv_usec-tms.tv_usec)*0.000001;
43      gettimeofday(&tcs,NULL);
44      for (i = i_min; i <= i_max; i++) {
45          for (j = j_min; j <= j_max; j++) {
46              u_current[i][j] = 0.25 * (u_previous[i-1][j] + u_previous[i+1][j]
47                                      +  u_previous[i][j-1] + u_previous[i][j+1]);
48          }
49      }
50      gettimeofday(&tcf,NULL);
51      tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
52      /*Add appropriate timers for computation*/
53  #ifdef TEST_CONV
54      if (t%C==0) {
55      /*Test convergence*/
56          converged = converge(u_current, u_previous, i_min, i_max, j_min, j_max);
57          MPI_Allreduce(&converged, &global_converged, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
58      }
59  #endif
60  }
```