

Συστήματα Παράλληλης Επεξεργασίας

1η Εργαστηριακή Άσκηση

Αρετή Μέη εΙ20052

Κωνσταντίνος Παπαδόπουλος εΙ20152

Ελισάβετ Παπαδοπούλου εΙ20190

19 Οκτωβρίου 2024



Η παράλληλη Υλοποίηση

Στον κώδικα που μας δόθηκε κληθήκαμε να το υλοποιήσουμε παράλληλα. Αυτό έγινε με το OPENMP, και την βιβλιοθήκη “omp.h”. Η παραλληλοποίηση του προγράμματος έγινε στη λογική, ότι το κάθε thread, θα αναλάβει μια σειρά του ζητούμενου ταμπλό.

```
konstantinosp. — parlab29@scirouter: ~/a1 — ssh parlab29@orion.cslab.ece.ntua.gr — 114x38
}
/*Allocate and initialize matrices*/
current = allocate_array(N);           //allocate array for current time step
previous = allocate_array(N);          //allocate array for previous time step
init_random(previous, current, N);     //initialize previous array with pattern
#ifndef OUTPUT
print_to_pgm(previous, N, 0);
#endif
/*Game of Life*/
gettimeofday(&ts,NULL);
for ( t = 0 ; t < T ; t++ ) {
    #pragma omp parallel for private(nbrs)
    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;
}

```

Με την εντολή `#pragma omp parallel for private(nbrs)`, παραλληλοποιούμε το for loop, χωρίζεται δηλαδή η κάθε επανάληψη του βρόχου, μεταξύ των thread. Το μόνο πρόβλημα είναι ότι by default οι παράλληλοι πυρήνες βλέπουν ως shared τις μεταβλητές που ορίστηκαν εκτός του scope της παραλληλοποίησης (εδώ οι μεταβλητές που ορίστηκαν πριν την 2η for loop είναι current, previous, nbrs). Επειδή η μεταβλητή nbrs (η οποία είναι μοναδική για κάθε στοιχείο του ταμπλού), είναι by default shared, έπρεπε να την κάνουμε private, ώστε να είναι διαφορετική σε κάθε πυρήνα, ώστε να μην έχουμε λογικό σφάλμα στο πρόγραμμα. Οι υπόλοιπες μεταβλητές είναι by default ορισμένες όπως μας βολεύει.

Scripts για Υποβολή

Για την υποβολή χρησιμοποιήσαμε τα 2 scripts που δόθηκαν στο παράδειγμα, με τις κατάλληλες τροποποιήσεις.

```
#!/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=00:10:00

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

module load openmp
cd /home/parallel/parlab29/a1
make
```

To make_on_queue.sh είναι υπέυθυνο ώστε να φτάξει τα κατάλληλα εκτελέσιμα.

```
#!/bin/bash

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

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

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

## How long should the job run for?
#PBS -l walltime=02:00:00

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

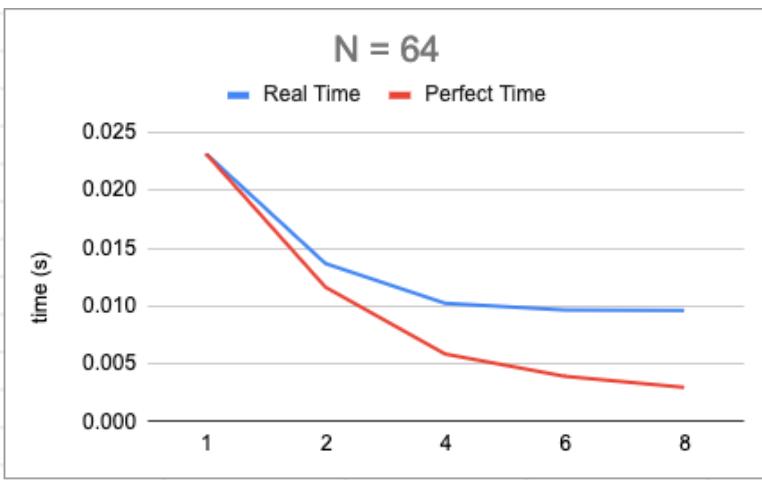
module load openmp
cd /home/parallel/parlab29/a1

for i in 1 2 4 6 8
do
  for j in 64 1024 4096
  do
    export OMP_NUM_THREADS=$i
    # Create separate output and error files for each combination of i and j
    ./Game_Of_Life $j 1000 > "run_omp_GOL_${i}_${j}.out" 2> "run_omp_GOL_${i}_${j}.err"
  done
done
```

Ενώ το run_on_queue.sh είναι υπεύθυνο να τρέξει τα εκτελέσιμα με κατάλληλα ορίσματα (το μέγεθος του ταμπλό, τα threads τις εποχές κτλ)

Αποτελέσματα

Cores	Χρόνος Εκτέλεσης (s)	Ιδανική Επιτάχυση (s)
1	0.023088	0.023088
2	0.01359	0.011544
4	0.010154	0.005772
6	0.009571	0.003848
8	0.009536	0.002886
For 64x64		
1	10.969189	10.969189
2	5.457674	5.4845945
4	2.723307	2.74229725
6	1.831874	1.828198167
8	1.379513	1.371148625
For 1024x1024		
1	175.905239	175.9085239
2	88.255485	87.95426195
4	45.448324	43.97713098
6	42.038691	29.31808732
8	41.420011	21.98856549
For 4096x4096		

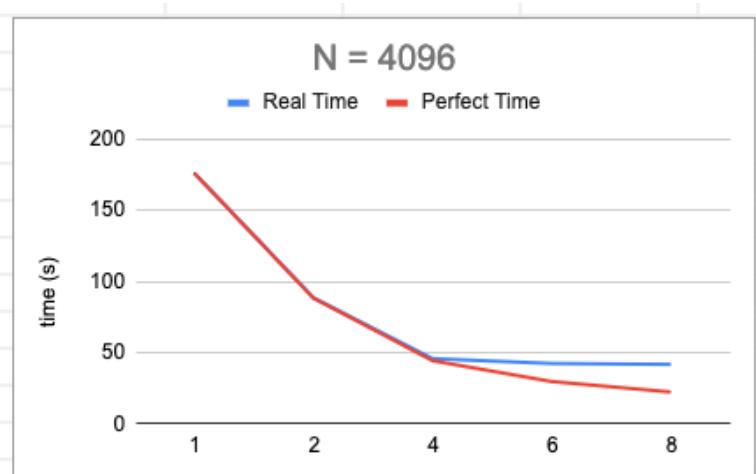


κάθε thread είναι μικρό σε σχέση με τους χρόνους παραλληλοποίησης (δημιουργία threads etc) και συγχρονισμού- επικοινωνίας μεταξύ thread.

Όπως φαίνεται από το διάγραμμα για μικρό μέγεθος ταμπλό, δεν έχουμε επιτάχυνση για πάνω από 4 πυρήνες. Αυτό οφείλεται στο ότι το ταμπλό είναι πολύ μικρό και ο φόρτος εργασίας



Για ταμπλό μεσαίου μεγέθους βλέπουμε σχεδόν ιδανική παραλληλοποίηση, για τον αριθμό των πυρήνων που δοκιμάσαμε.



Τέλος για μεγάλο ταμπλό, βλέπουμε ότι πάλι δεν κλιμακώνει για πυρήνες πάνω από 4. Αυτό πιθανά οφείλεται στο μέγεθος του ταμπλό το οποίο είναι $4096 \times 4096 \times 4 = 128$ MBytes. Συνεπώς δεν χωράει όλο το matrix στην cache και

αυτό δημιουργεί συμφόρηση στο διάδρομο μνήμης (Amdahl's law).

Συστήματα Παράλληλης Επεξεργασίας

2.Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κοινής μνήμης

Αρετή Μέη	——	A.Μ.: 03120062
Κωνσταντίνος Παπαδόπουλος	——	A.Μ.: 03120152
Ελισάβετ Παπαδοπούλου	——	A.Μ.: 03120190

2.1 Παραλληλοποίηση και βελτιστοποίηση του αλγορίθμου K-means

Στο συγκεκριμένο ερώτημα πραγματοποιήσαμε την παραλληλοποίηση του αλγορίθμου **K-means**, σε δύο διαφορετικές εκδόσεις του:

1η Έκδοση: Shared Clusters

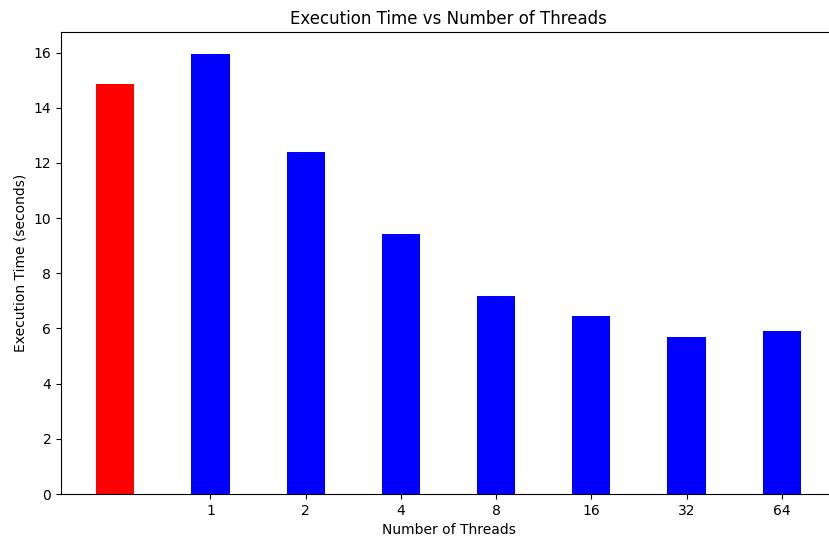
Για την πρώτη έκδοση, υλοποιήσαμε μια βασική "naive" παραλληλοποίηση του **σειριακού αλγορίθμου K-means**, προσθέτοντας τις κατάλληλες εντολές στον υπάρχων σκελετό και φροντίζοντας για τον ορθό διαμοιρασμό των δεδομένων. Συγκεκριμένα, πραγματοποιήσαμε την παραλληλοποίηση του *for-loop* που αναλαμβάνει την ανάθεση των σημείων στα clusters.

Η μεταβλητή **delta**, η οποία αρχικοποιείται στο μηδέν για κάθε επανάληψη του βρόχου while, αυξάνεται στον αριθμό των σημείων των οποίων το membership μεταποιήθηκε (ταυτίζεται δηλαδή με τον αριθμό των σημείων που άλλαξαν cluster). Είναι λοιπόν shared μεταξύ των threads, επομένως ελοχεύει ο κινδυνος race condition σε περιπτωση που περισσότερα από ένα νήματα πάνε να την αυξήσουν ταυτόχρονα. Αρχικά, επιλέξαμε να χρησιμοποιήσουμε το omp atomic directive, προκειμένου το access στο delta να γίνεται ατομικά, παρατηρήσαμε ωστόσο κακή επίδοση, ακριβώς επειδή το atomic access σταματούσε τον κώδικα όλως των νημάτων που προσπαθούσαν να μεταβάλλουν το delta, χωρίς αυτό να είναι απαραότητο, και εισάγονταν άχρηστες καθυστερησεις. Καταλήξαμε λοιπόν στη χρήση του clause **reduction(+:delta)**, επιτρέποντας σε όλα τα νήματα που εκτελούνται παράλληλα να δημιουργούν ένα τοπικό αντίγραφο της μεταβλητής delta, και να αυξάνουν το κάθε thread τη δική του μεταβλητή, και με το πέρας της παράλληλης εκτέλεσης του βρόχου for, οι επιμέρους τιμές delta από κάθε νήμα να συνενώνονται αθροιστικά, αποτελεσματικά και χωρίς περιττές συγχρούσεις, ή τις καθυστερήσεις που θα εισήγαγε η χρήση του atomic.

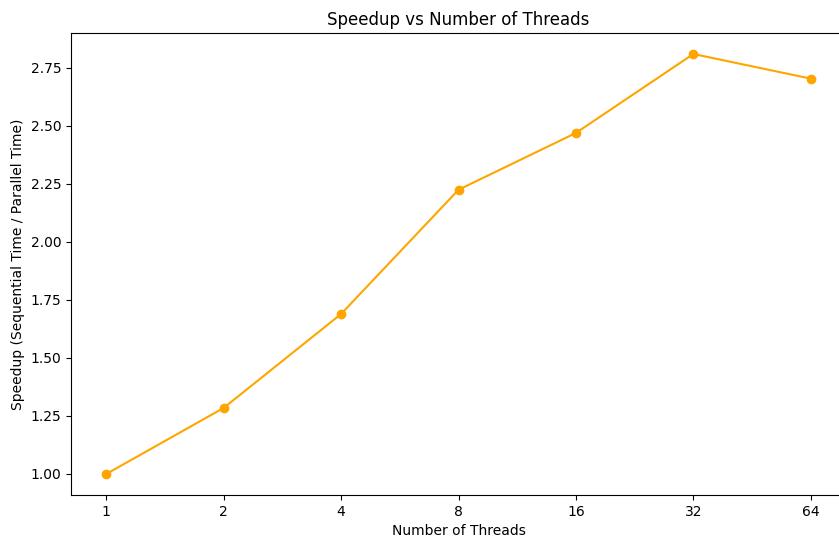
Ταυτόχρονα, χρησιμοποιούμε ατομικές προσβάσεις για τις εγγραφές στα κοινά δεδομένα (δηλαδή τους πίνακες (`newClusters[]`, `newClusterSize[]`), διασφαλίζοντας την ασφαλή πρόσβαση και αποφεύγοντας race conditions. Ακριβώς όμως επειδή χρησιμοποιούμε atomic, περιμένουμε να μην έχουμε ακραία καλούς χρόνους, λόγω της ανάγκης "κλειδώματος" για την επικαιροποίηση των τιμών του πίνακα.

Παρακάτω παρατίθενται τα διαγράμματα **χρόνου εκτέλεσης** και **Speedup** για threads = {1, 2, 4, 8, 16, 32, 64} και configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}.

- **Time Execution Barplot**



- **Speedup plot**



Όπως φαίνεται παραπάνω, η παραλληλοποίηση βελτιώνει σημαντικά την εκτέλεση του αλγορίθμου. Όπως φαίνεται στο γράφημα, μέχρι τους 8 πυρήνες η επιτάχυνση αυξάνεται σχεδόν

γραμμικά. Ωστόσο, μετά από αυτό το σημείο, η αποδοτικότητα της παραλληλοποίησης μειώνεται λόγω του πρόσθετου φόρτου συγχρονισμού και της επικοινωνίας μεταξύ των νημάτων. Επιπλέον, η υπερφόρτωση της CPU και η περιορισμένη απόδοση της μνήμης μπορούν να προκαλέσουν συμφόρηση, μειώνοντας το πραγματικό όφελος από την αύξηση των νημάτων.

Στην συνέχεια, δοκιμάζουμε να εκτελέσουμε τον αλγόριθμο ξανά, με την ίδια παραλληλοποίηση, αλλά τη χρήση της μεταβλητής περιβάλλοντος **GOMP_CPU_AFFINITY**. Η μεταβλητή αυτή χρησιμοποιείται για να ελέγξει την δέσμευση των threads σε συγκεκριμένους επεξεργαστές, καθορίζει δηλαδή σε ποιον επεξεργαστή μπορεί να εκτελεστεί κάθε thread κατά την εκτέλεση του παράλληλου προγράμματος. Ακριβώς, λοιπόν επειδή κάθε νήμα προσδένεται σε ένα συγκεκριμένο core και αρχίζει και ολοκληρώνει την εκτέλεση του σε αυτό, βελτιώνεται αρκετά το locality μας, αφού τα νήματα δεν θα πηγαίνονται μεταξύ των πυρήνων και δεν θα παραστεί ανάγκη για αντιγραφή των cache από το ένα σημείο στο άλλο.

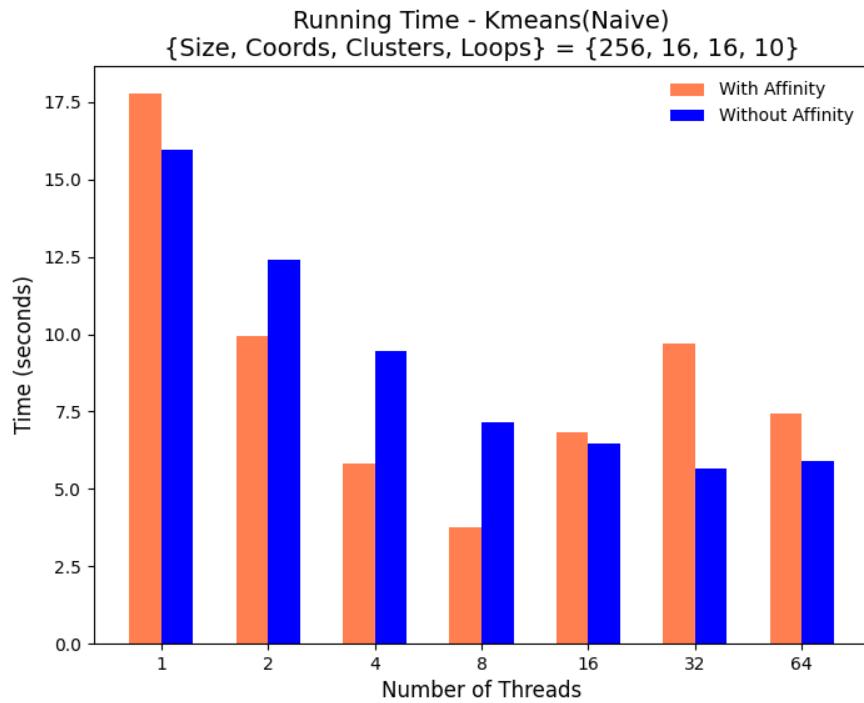
Γνωρίζουμε ότι ο επεξεργαστής που χρησιμοποιούμε είναι ο Intel Xeon E5-4620, ο οποίος διαθέτει σε κάθε node του 8 physical cores, και κάθε ένας πυρήνας μπορεί να τρέξει έως και 2 threads. Εμείς λοιπόν θέσαμε την μεταβλητή περιβάλλοντος **GOMP_CPU_AFFINITY="0-63"**, για τα 64 νήματά μας, και παρατηρήσαμε πως το scaling ήταν κακό για αριθμό πυρήνων από 16 και πάνω. Εικάζουμε ότι αυτό συμβαίνει λόγω του ότι με τη χρήση του range 0-63 και για νήματα πάνω από 16, τα νήματα δεν θα παραμείνουν σε ένα μόνο node, αλλά θα διαμοιραστούν και στα 4 διαθέσιμα, με αποτέλεσμα να παρουσιάζονται θέματα memory latency, εάν κάποιο νήμα χρειαστεί μνήμη η οποία είναι attached σε κάποιο άλλο node. Εμείς θέλουμε να εκμεταλλευτούμε την ίδιότητα του hyperthreading, και ουσιαστικά αφού γεμίσουν τα cores 0-7 με τα πρώτα 8 νήματα, να συνέχισουμε τοποθετώντας τα επόμενα 8 νήματα πάλι στο ίδιο NUMA node, κυκλικά δηλαδή στα cores 32-39 γεμίζοντας τα logical cores κάθε φυσικού πυρήνα. Για κάθε NUMA node θα ισχύει:

- NUMA node 0: 0-7, 32-39
- NUMA node 0: 8-15, 40-47
- NUMA node 0: 16-23, 48-55
- NUMA node 0: 24-31, 56-63

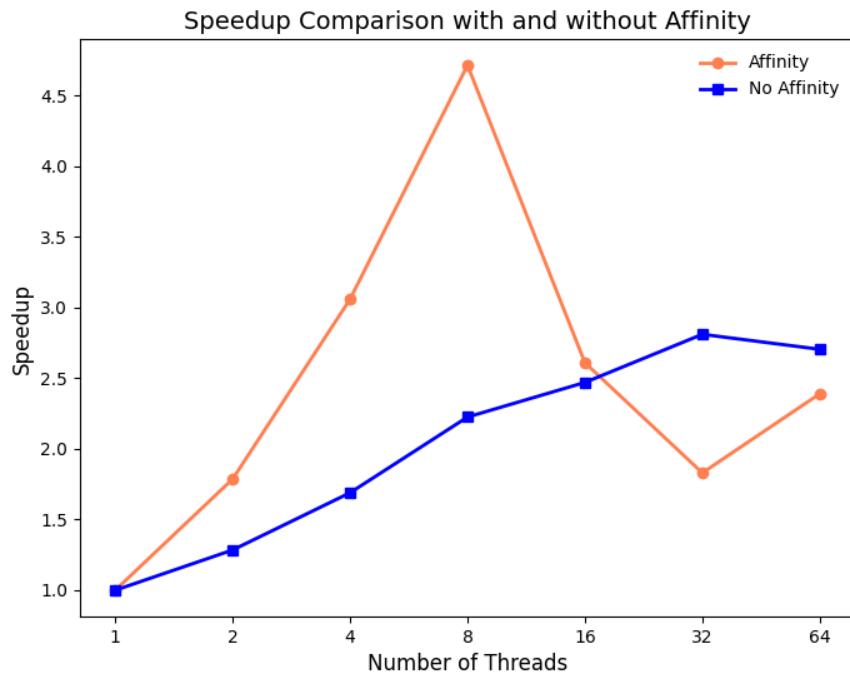
Εικάζουμε ότι θέτοντας την μεταβλητή περιβάλλοντος **GOMP_CPU_AFFINITY="0-7 32-39"**, το scaling και οι χρονοί θα βελτιώνονται αρκετά.

Παρακάτω παρατίθενται ξανά τα διαγράμματα **χρόνου εκτέλεσης** και **Speedup** για threads = { 1, 2, 4, 8, 16, 32, 64} και configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}.

- Time Execution Barplot



- Speedup plot



Η αύξηση των threads βελτιώνει την απόδοση λόγω καλύτερης αξιοποίησης των διαθέσιμων πυρήνων, όμως μετά από 8 threads παρατηρείται μείωση στην κλιμάκωση. Αυτό συμβαίνει επειδή τα πρώτα 8 threads εκτελούνται σε έναν NUMA node, ενώ τα υπόλοιπα μεταφέρονται

σε δεύτερο node, αυξάνοντας το κόστος προσπέλασης στη μνήμη. Επιπλέον, η κατανομή των threads δεν εκμεταλλεύεται σωστά το hyperthreading, με αποτέλεσμα μεγαλύτερο overhead. Στα 32 threads η απόδοση επιδεινώνεται ακόμη περισσότερο, καθώς έχουν δεσμευτεί όλοι οι πυρήνες από διαφορετικά nodes, επιβαρύνοντας την επικοινωνία και μειώνοντας την τοπικότητα της μνήμης.

Η αύξηση του αριθμού των threads βελτιώνει σημαντικά την απόδοση λόγω της καλύτερης αξιοποίησης των διαθέσιμων πυρήνων. Ωστόσο, παρατηρείται ότι μετά τα 8 threads η κλιμάκωση χειροτερεύει. Αυτό οφείλεται στη δομή του συστήματος NUMA (Non-Uniform Memory Access). Τα threads 0-7 εκτελούνται στον πρώτο NUMA node, όπου η πρόσβαση στη μνήμη είναι γρήγορη και τοπική. Όμως, από το thread 8 και μετά, τα νέα threads εκτελούνται σε δεύτερο NUMA node, με αποτέλεσμα να αυξάνεται το κόστος πρόσβασης στη μνήμη, αφού οι διεργασίες πρέπει να ανταλλάσσουν δεδομένα μεταξύ των nodes, επιβαρύνοντας το memory bandwidth.

Επιπλέον, η κατανομή των threads δεν εκμεταλλεύεται σωστά το hyperthreading. Τα threads δεν τοποθετούνται στρατηγικά ώστε να αξιοποιούν πλήρως τους φυσικούς πυρήνες πριν χρησιμοποιηθούν τα λογικά (SMT) threads, γεγονός που οδηγεί σε suboptimal εκτέλεση. Αυτό προκαλεί επιπλέον overhead και μειώνει το efficiency της πολυνηματικής εκτέλεσης.

Η κατάσταση χειροτερεύει ακόμα περισσότερο στα 32 threads, καθώς πλέον όλα τα NUMA nodes έχουν ενεργοποιηθεί πλήρως, και η επικοινωνία μεταξύ τους γίνεται ακόμα πιο δαπανηρή. Τα threads που βρίσκονται σε διαφορετικά nodes χρειάζονται περισσότερη επικοινωνία για την ανταλλαγή δεδομένων, κάτι που αυξάνει σημαντικά το latency και μειώνει την αποτελεσματικότητα της cache. Έτσι, ενώ η αύξηση των threads σε πρώτη φάση βελτιώνει την απόδοση, μετά από ένα σημείο το κόστος του memory access και του synchronization οδηγεί σε μειωμένη αποδοτικότητα.

```

1 or i in 1 2 4 8 16 32 64; do
2     export OMP\_NUM\_THREADS="$i"
3     export GOMP_CPU_AFFINITY="0-7 32-39"
4     # Bind threads to CPUs on NUMA nodes
5     if [ "$i" -le 8 ]; then
6         NUMA_OPTIONS="--cpunodebind=0 --membind=0"
7     elif [ "$i" -le 16 ]; then
8         NUMA_OPTIONS="--cpunodebind=0,1 --membind=0,1"
9     elif [ "$i" -le 32 ]; then
10        NUMA_OPTIONS="--cpunodebind=0,1,2,3 --membind=0,1,2,3"
11    else
12        NUMA_OPTIONS="--interleave=all"
13    fi
14
15    # Execute for different matrix and block sizes
16    for j in 4096; do
17        for b in 64 128; do
18            numactl $NUMA_OPTIONS ./fw_tiled_tasks "$j" "$b"
19        done
20    done
21
22 done

```

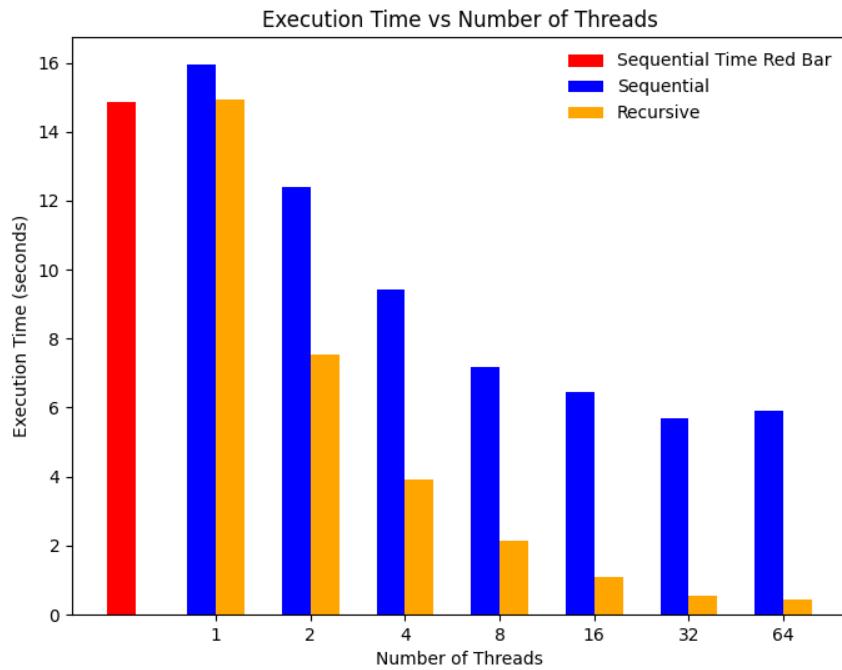
2η Έκδοση: Copied Clusters and Reduce

Στο ερώτημα αυτό, ασχοληθήκαμε με την παραλληλοποίηση μιας δεύτερης, αναδρομικής εκδοχής του αλγορίθμου K-means. Και σε αυτή την περίπτωση, για την μεταβλητή **delta** χρησιμοποιήσαμε το clause **reduction(+:delta)**, προκειμένου να διασφαλίσουμε τον σωστό υπολογισμό της, καθώς επίσης και να περιορίσουμε όσο μπορούμε την καθυστέρηση που θα προκαλούσε η ατομική πρόσβαση σε αυτήν.

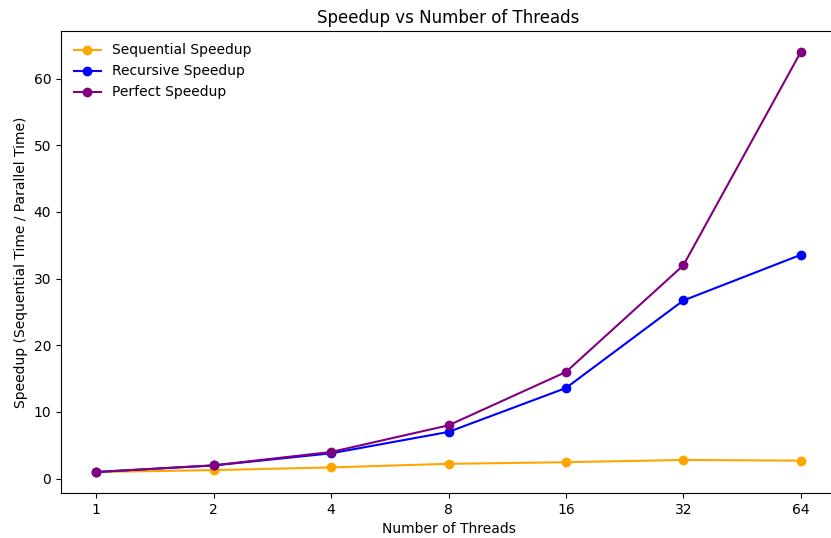
Σε αυτή την εκδοχή του αλγορίθμου ωστόσο, δεν καλούμαστε πλέον να αντιμετωπίσουμε την ταυτόχρονη εγγραφή στα δεδομένα μας, καθώς, αν και οι πίνακές μας είναι διαμοιραζόμενοι ανάμεσα στα νήματα, πλέον οι εγγραφές σε αυτούς πραγματοποιούνται σε διαφορετικό σημείο για κάθε νήμα. Στόχος μας λοιπόν πλέον, είναι η σωστή αποθήκευση των δεδομένων στους τελικούς πίνακες, μετά την λήξη της παράλληλης περιοχής. Σε αυτό το σημείο λοιπόν, αλλάζουμε τη δομή των πινάκων που είχαμε στο προηγούμενο ερώτημα, και τους προσθέτουμε μια επιπλέον διάσταση: τη διάσταση των νημάτων. Έτσι ο πίνακας **local_newClusters[]** γίνεται 3-διάστατος, και κάθε νήμα ουσιαστικά αντιστοιχίζεται σε κάθε επίπεδο που ορίζεται από το πρώτο index του πίνακα, και αντίστοιχα ο **local_newClusterSize[]** γίνεται διδιάστατος, και κάθε νήμα αντιστοιχίζεται σε κάθε γραμμή του. Για τον λόγο αυτό, πραγματοποιούμε Reduction από τους πίνακες **local_newClusters[], local_newClusterSize[]** στους τελικούς **newClusters[], newClusterSize[]**, προσθέτοντας μεταξύ τους τους υπολογισμούς στους οποίους κατέληξε το κάθε νήμα και βγάζοντας τον μέσο όρο των ανθροισμάτων από το master thread, προκειμένου να προκύψουν τελικά τα νέα κέντρα Clusters για εκείνη την επανάληψη του βρόχου.

Παρακάτω παρατίθενται τα διαγράμματα **χρόνου εκτέλεσης** και **Speedup** για threads = { 1, 2, 4, 8, 16, 32, 64} και configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}.

- Time Execution Barplot



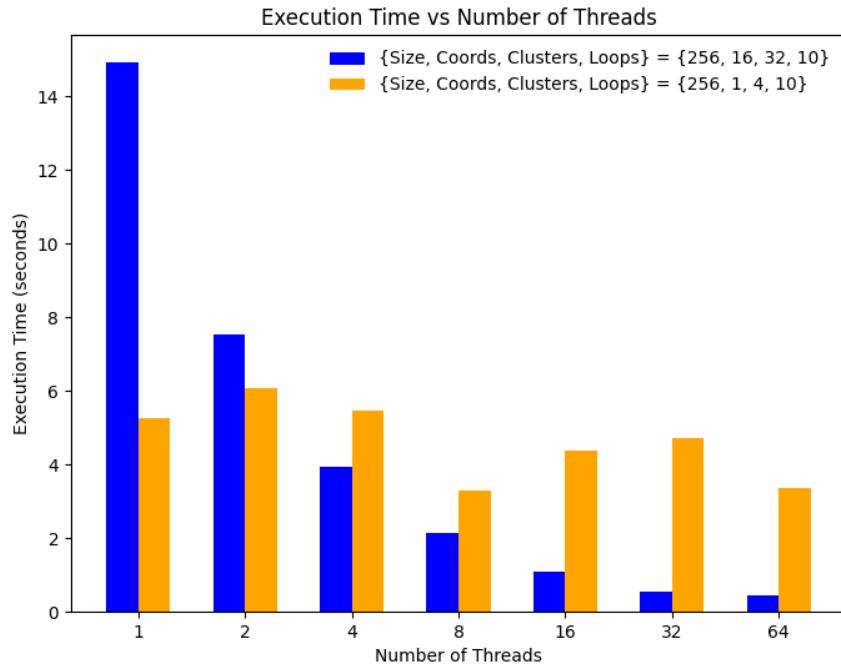
- Speedup plot



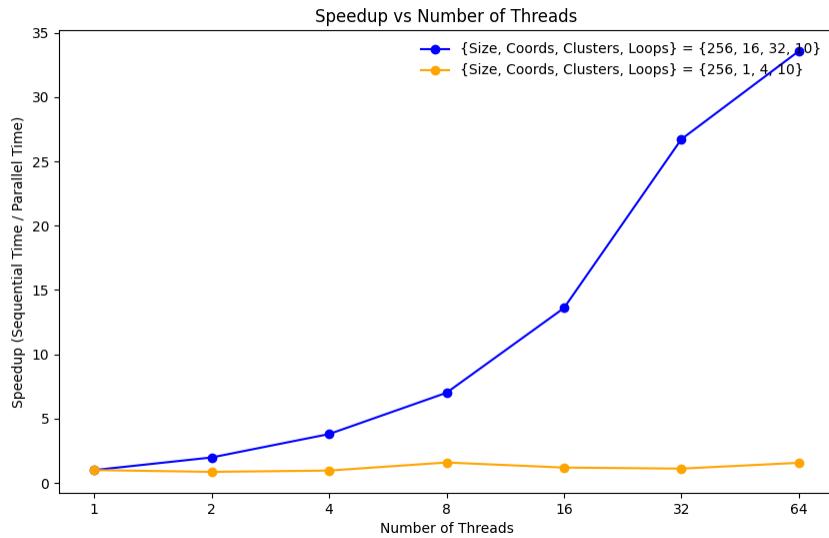
Όπως φαίνεται παραπάνω, η παραλληλοποίηση της δεύτερης έκδοσης του αλγορίθμου επιφέρει πολύ καλύτερες επιδόσεις, τόσο στον χρόνο εκτέλεσης αλλά και στο Speedup του, πλησιάζοντας αυτές τις ιδανικής περίπτωσης. Πράγματι, πλέον η παραλληλοποίηση δεν απαιτεί ούτε την αναμονή των νημάτων για την εγγραφή σε κοινά δεδομένα, ούτε εμπεριέχει κάποια εξάρτηση μεταξύ τους στην παράλληλη περιοχή, δηλαδή το *for-loop* που θεωρήσαμε και στο προηγούμενο ερώτημα.

Στην συνέχεια, διαφοροποιούμε το αρχικό configuration του προγράμματός μας για την αναδρομική λύση από $\{\text{Size}, \text{Coords}, \text{Clusters}, \text{Loops}\} = \{256, 16, 32, 10\}$ στο $\{\text{Size}, \text{Coords}, \text{Clusters}, \text{Loops}\} = \{256, 1, 4, 10\}$, και ξανατρέχουμε το πρόγραμμα. Προκύπτουν τα παρακάτω:

- Time Execution Barplot



- Speedup plot



Όπως φαίνεται, το νέο configuration δεν είναι καθόλου ευνοϊκό για την επίδοση του προγράμματός μας. Προκειμένου να εξηγήσουμε αυτή τη συμπεριφορά, αλλά και να βελτιώσουμε την επίδοσή μας, αναλύουμε αρχικά τις παρακάτω έννοιες:

1. First-touch Policy: Πρόκειται για μια στρατηγική memory allocation, σύμφωνα με την οποία τα δεδομένα κατανείμονται στην πρώτη επεξεργαστική μονάδα που τα επεξεργάζεται. Επομένως, στην περίπτωση μας η πρώτη επεξεργασία γίνεται μέσω της `calloc` η οποία κάνει τόσο το allocation και δεσμεύει την εικονική μνήμη, όσο και την πρώτη αρχικοποίηση σε 0 των δεδομένων, ώστε να έρθουν πιο κοντά στο NUMA node του νηματος που τα επεξεργάζεται, περιορίζοντας έτσι το memory latency.

Παρατηρώντας το configuration το οποίο καλούμαστε να συγχρίνουμε με το αρχικό, εύκολα βλέπουμε πως ο αριθμός, τόσο των clusters αλλά και των σημείων, έχει μειωθεί πολύ. Έτσι, μπορούμε να υπονθέσουμε πως η αρχικοποίηση της `calloc` αποθηκεύει στο ίδιο cache line δεδομένα τα οποία επεξεργάζονται διαφορετικά νήματα, με αποτέλεσμα με κάθε modification του cache line από ένα νήμα, η cache line να γίνεται dirty και να καθίσταται invalidated στις caches των υπολοίπων νημάτων, οδηγούμαστε δηλαδή στο φαινόμενο του false sharing:

2. False-sharing: Πρόκειται για την μείωση της απόδοσης ενός προγράμματος, που προκύπτει όταν πολλοί επεξεργασατές μεταβάλλουν συνεχώς δεδομένα που περιέχονται στην ίδια cache line.

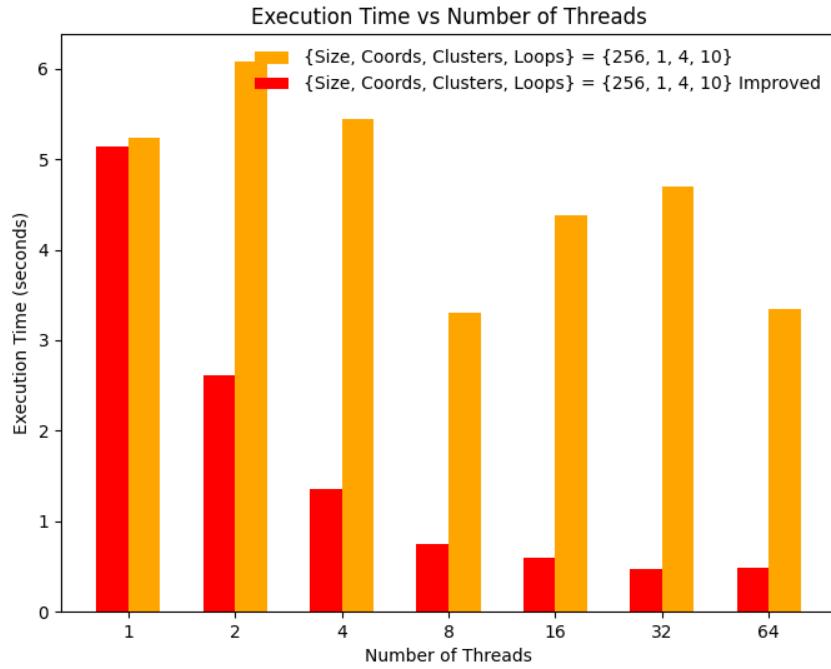
Προκειμένου λοιπόν να βελτιώσουμε την απόδοση του προγράμματος με αυτό το configuration, επιλέγουμε να εξαλείψουμε το φαινόμενο false-sharing χρησιμοποιώντας το first-touch. Συγκεκριμένα, αρχικοποιούμε πλέον τα δεδομένα μας παράλληλα, εξασφαλίζοντας local allocation. Έτσι, κάθε νήμα αγγίζει με την παράλληλη αρχικοποίηση μόνο τα δεδομένα τα οποία θα χρειαστεί να κάνει update, και έτσι αποφεύγεται ο διαμοιρασμός των cache lines μεταξύ νημάτων, και επομένως NUMA nodes οι οποίες είναι shared μεταξύ των νημάτων, αποφέυγοντας το false sharing.

NUMA aware allocation

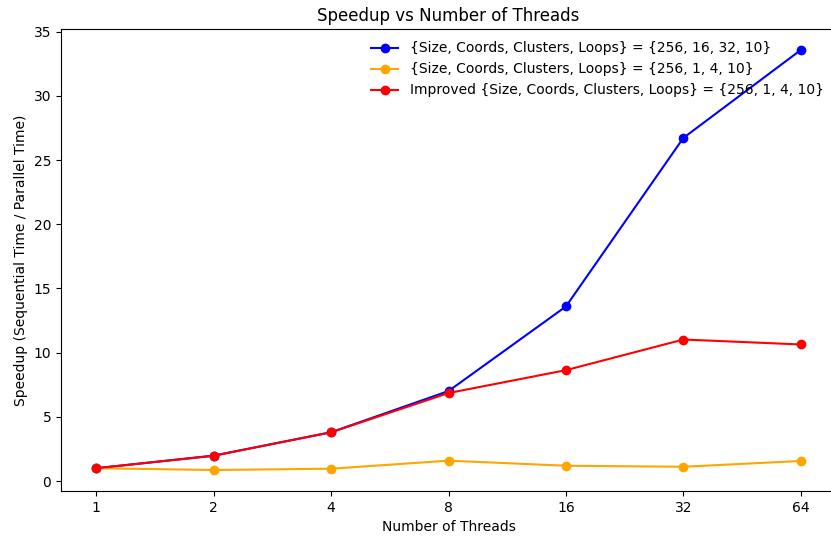
Στο συγκεκριμένο ερώτημα, μας ζητείται να αξιοποιήσουμε τη λογική του first touch, έχοντας υπόψιν τα NUMA χαρακτηριστικά του μηχανήματος και το διαμοιρασμό των πίνακα objects στα memory nodes. Θα χρησιμοποιήσουμε λοιπόν και για την αρχικοποίηση του πίνακα objects το first touch policy, προκειμένου να να διαμοιράσουμε τα δεδομένα στα τοπικά NUMA nodes και τα δεδομένα του πίνακα θα δεσμευθούν στο node που είναι κοντά στον επεξεργαστή που κάνει την πρώτη προσπέλαση, μειώνοντας τον overhead από προσβάσεις σε απομακρυσμένη μνήμη. Το κύριο bottleneck εδώ είναι η πρόσβαση στη μνήμη, ειδικά αν τα δεδομένα κατανεμηθούν σε απομακρυσμένους nodes.

Με βάση όλα αυτά, προκύπτουν τα παρακάτω:

- **Time Execution Barplot**



- Speedup plot



2.2 Παραλληλοποίηση του αλγορίθμου Floyd-Warshall

2η Έκδοση: Recursive algorithm

Σε αυτό το σημείο της εργασίας πραγματοποιήσαμε μια παραλληλοποίηση του αναδρομικού αλγορίθμου Floyd-Warshall. Συγκεκριμένα, για την παραλληλοποίηση χρησιμοποιήθηκε το εργαλείο OpenMP tasks, με το οποίο ανεξάρτητες μεταξύ τους δουλειές ανατέθηκαν ως δι-

αφορετικά tasks για να πραγματοποιηθούν παράλληλα.

FWR (A₀₀, B₀₀, C₀₀);

1

FWR (A₀₁, B₀₀, C₀₁);

2

FWR (A₁₀, B₁₀, C₀₀);

FWR (A₁₁, B₁₀, C₀₁);

3

FWR (A₁₁, B₁₀, C₀₁);

4

FWR (A₁₀, B₁₀, C₀₀);

5

FWR (A₀₁, B₀₀, C₀₁);

FWR (A₀₀, B₀₀, C₀₀);

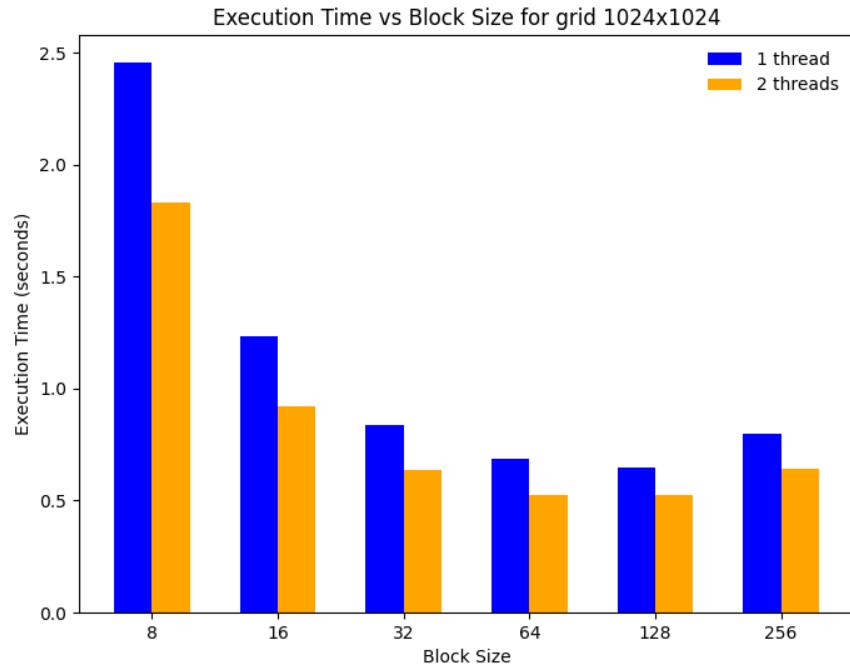
6

Όπως φαίνεται και στην εικόνα παραπάνω, υπήρχε στην πραγματικότητα πολύ μικρός χώρος παραλληλοποίησης του συγκεκριμένου αλγορίθμου, καθώς στην πλειοψηφία τους οι εντολές ήταν εξαρτημένες ηγ μία από την άλλη, και άρα δεν μπορούσαν να πραγματοποιηθούν παράλληλα. Τελικά, η παραλληλοποίηση έγινε στα jobs 2-3, 5-6.

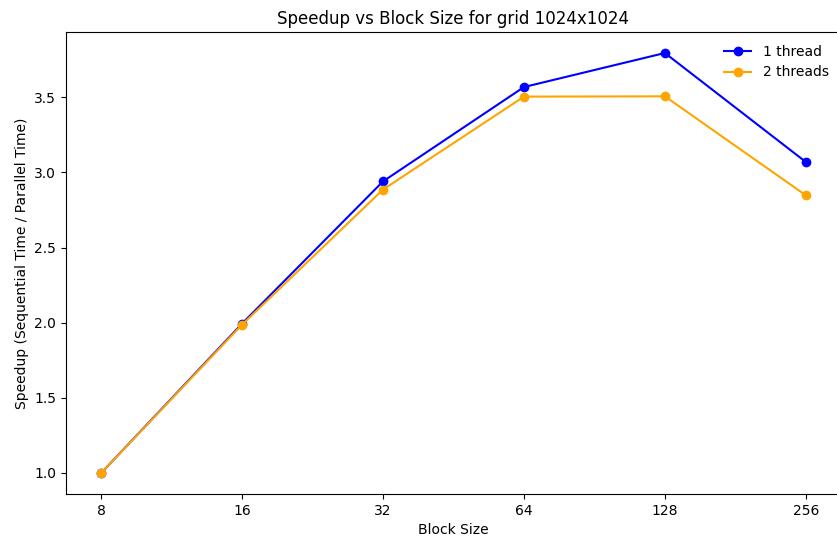
Παρακάτω παρατίθενται τα διαγράμματα **χρόνου εκτέλεσης** και **Speedup** για threads = {1, 2, 4, 8, 16, 32, 64} και configuration {Size, Coords, Clusters, Loops} = {256, 16, 32, 10}, στα παρακάτω διαφορετικά μεγέθη πινάκων:

1024x1024

- Time Execution Barplot

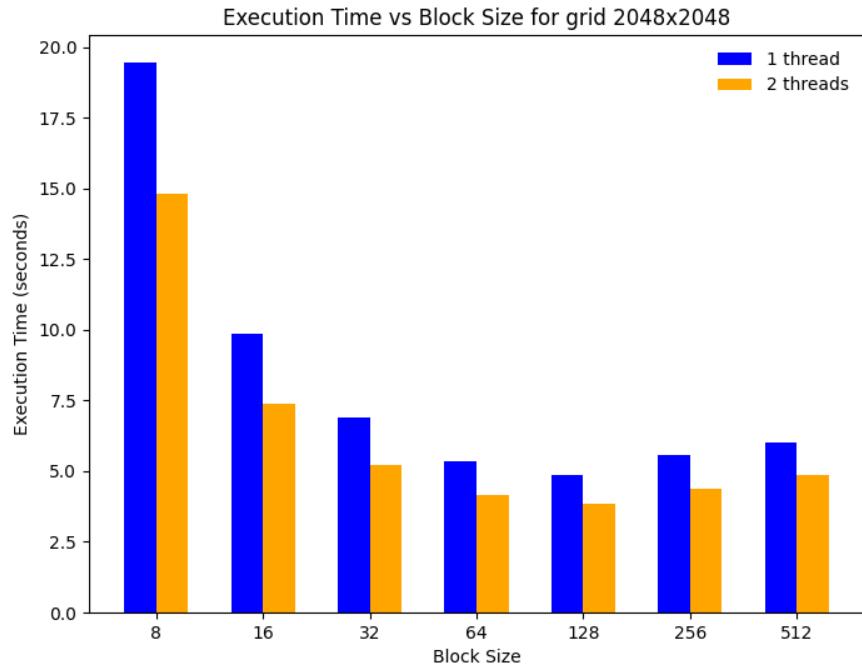


- Speedup plot

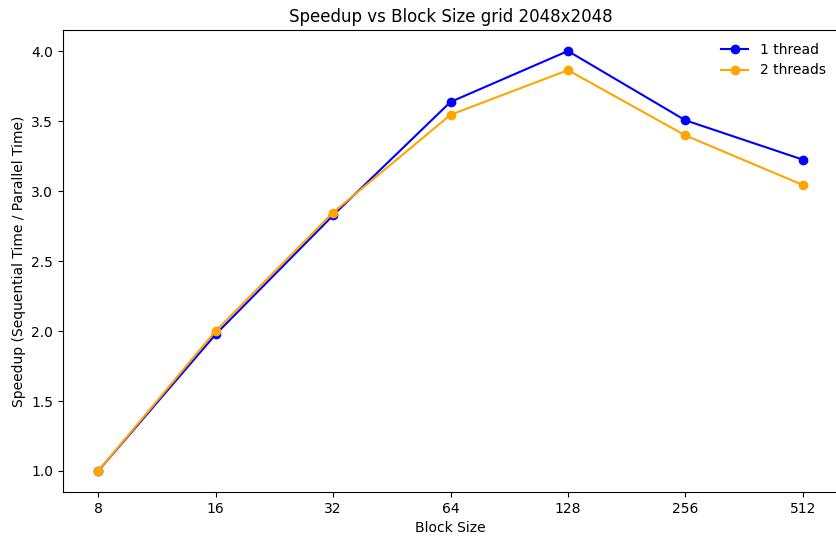


2048x2048

- Time Execution Barplot

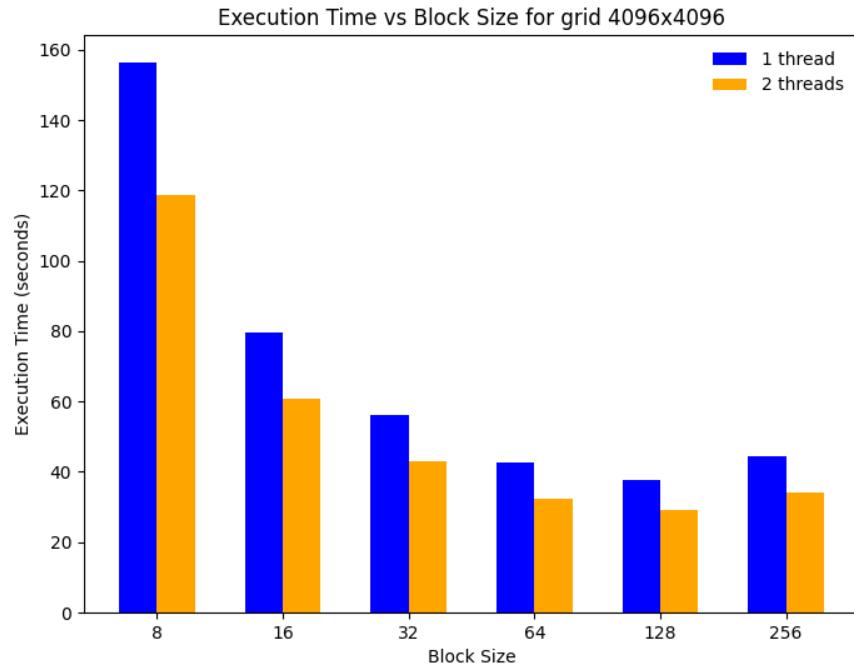


- Speedup plot

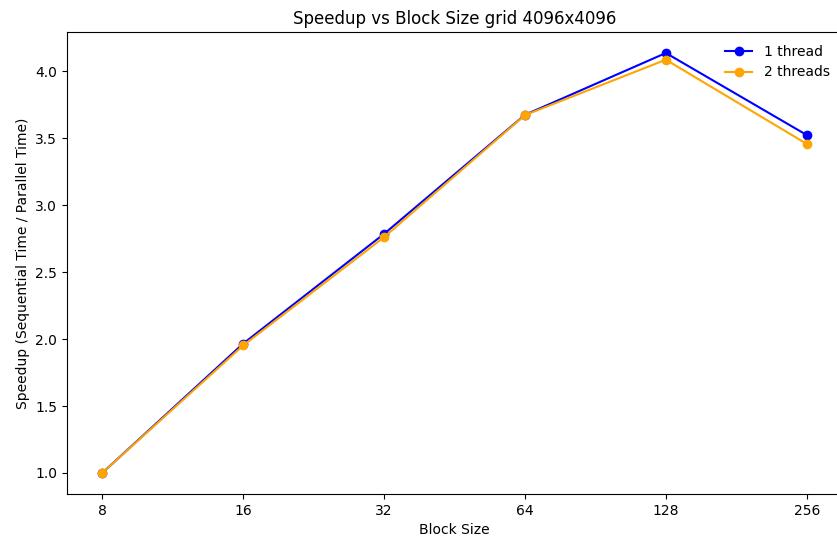


4096x4096

- Time Execution Barplot



- Speedup plot



Στο συγκεκριμένο ερώτημα, η απόδοση των προγραμμάτων εξαρτάται από 2 παράγοντες:

1. Το block size να είναι σχετικά μικρό και να χωράει στη cache, ούτως ώστε να εξασφαλίζεται η βέλτιστη χρήση της cache με βάση το block size και να αποφέυγονται οι πολλαπλές προσβάσεις στη μνήμη

2. Το block size δεν πρέπει όμως να έιναι πολύ μικρό γιατί τότε το overhead των αναδρομικών κλήσεων θα υπερβαίνει το πλεονέκτημα του dividing της δουλειάς που έχουμε να κάνουμε σε blocks.

Με βάση τα παρακάτω, μπορούμε να δούμε ότι το καλύτερο block size μας είναι το 128.

2η Έκδοση: Tiled algorithm

Στο συγκεκριμένο ερώτημα δοκιμάσαμε 2 διαφορετικές προσεγγίσεις: τόσο την παραλληλοποίηση των for loops, όσο και τη χρήση tasks. Τα αποτελέσματα και με τις δύο μεθόδους ήταν τα σωστά, ωστόσο η χρήση των tasks ήταν πολύ χειρότερη, λόγω του overhead που εισάγουν για συγχρονισμό των tasks, αφού μετά την χλήση του FW για το στοιχείο pivot της διαγωνίου, τα loop που καλόύν τον FW τόσο για τη σειρά όσο και για τη στήλη του εκάστοτε στοιχείου επεξεργάζονται τα αντίστοιχα tiles (εξού και τα dependencies out). Στη συνέχεια βλέπουμε ότι για τα for loops τα οποία επεξεργάζονται τα κομμάτια του πίνακα που βρίσκονται πάνω δεξιά, πάνω αριστερά, κάτω δεξιά και κάτω αριστερά, χρειάζονται τα αποτελέσματα των πάνω for loops (εξού και το dependency in). Ο κώδικας των αναφερόμενων loops για την αρχική προσέγγιση με tasks:

```
1 int main(int argc, char **argv)
2 {
3     int **A;
4     int i,j,k;
5     struct timeval t1, t2;
6     double time;
7     int B=64;
8     int N=1024;
9
10    if (argc != 3){
11        fprintf(stdout, "Usage %s N B\n", argv[0]);
12        exit(0);
13    }
14
15    N=atoi(argv[1]);
16    B=atoi(argv[2]);
17
18    A=(int **)malloc(N*sizeof(int *));
19    for(i=0; i<N; i++) A[i]=(int *)malloc(N*sizeof(int));
20
21    graph_init_random(A,-1,N,128*N);
22
23    gettimeofday(&t1,0);
24
25    for (k = 0; k < N; k += B) {
26        FW(A, k, k, k, B); //1
27
28        #pragma omp parallel
29        {
30            #pragma omp for
31            for (j=0; j<B; j++)
32                for (i=0; i<N; i++)
33                    A[i][j] = 0;
34        }
35    }
36
37    gettimeofday(&t2,0);
38    time = (double)(t2.tv_sec - t1.tv_sec) + (double)(t2.tv_usec - t1.tv_usec)/1000000.0;
39    printf("Time taken : %f\n", time);
40}
```

```

31         #pragma omp for nowait //2, apo 0 ews correct tiles   antia,
32         ,
33         edw
34         for (i = 0; i < k; i += B)
35             FW(A, k, i, k, B);
36
37         #pragma omp for nowait //2, apo current tile ws telos pinaka
38         for (i = k + B; i < N; i += B)
39             FW(A, k, i, k, B);
40
41         #pragma omp for nowait //2 sthles
42         for (j = 0; j < k; j += B)
43             FW(A, k, k, j, B);
44
45         #pragma omp for nowait //2 sthles
46         for (j = k + B; j < N; j += B)
47             FW(A, k, k, j, B);
48
49
50         #pragma omp barrier
51
52
53         #pragma omp for collapse(2) nowait
54
55         for (i = 0; i < k; i += B) {
56
57             for (j = 0; j < k; j += B) {
58                 FW(A, k, i, j, B);
59             }
60         } //NW
61
62
63         #pragma omp for collapse(2) nowait
64             for (i = 0; i < k; i += B) {
65                 for (j = k + B; j < N; j += B) {
66                     FW(A, k, i, j, B);
67                 }
68             } //NE
69
70
71         #pragma omp for collapse(2) nowait
72             for (i = k + B; i < N; i += B) {
73                 for (j = 0; j < k; j += B) {
74                     FW(A, k, i, j, B);
75                 }
76             } //SW
77
78         #pragma omp for collapse(2) nowait
79             for (i = k + B; i < N; i += B) {
80                 for (j = k + B; j < N; j += B) {

```

```

79             FW(A, k, i, j, B);
80         }
81     }
82 }
83 }
84 }
85 }
86 gettimeofday(&t2,0);
87
88 time=(double)((t2.tv_sec-t1.tv_sec)*1000000+t2.tv_usec-t1.tv_usec)
89 /1000000;
90 printf("FW_TILED,\%d,\%d,\%.4f\n", N,B,time);
91 for(i=452; i<500; i++){
92     fprintf(stdout,"%d\t", A[i][453]);
93     if (i%4==0) fprintf(stdout,"\n");
94 }
95 /*
96 for(i=0; i<N; i++)
97     for(j=0; j<N; j++) fprintf(stdout,"%d\n", A[i][j]);
98 */
99
100 return 0;
}

```

Listing 1: Floyd-Warshall Tiled Parallel Implementation

Από την άλλη, η παραλληλοποίηση του επαναληπτικού βρόχου είναι πολύ πιο απλή και γρήγορη. Παρατηρούμε πως μετά από την αρχική κλήση του FW για το στοιχείο της διαγωνίου σε κάθε iteration του εξωτερικού for loop, τα επόμενα 4 for loops που υπολογίζουν την αντίστοιχη γραμμή και στήλη μπορούν να υπολογιστούν παράλληλα αφού τα αποτελέσματα δεν εξαρτώνται το ένα από το άλλο, για αυτό και χρησιμοποιείται το nowait, για αποφυγή αχρείαστου συγχρονισμού των νημάτων. Ωστόσο, η δεύτερη ομάδα των for loops, δηλαδή τα διπλά for loops τα οποία επεξεργαζονται τα κομμάτια του πίνακα που βρίσκονται πάνω δεξιά, πάνω αριστερά, κάτω δεξιά και κάτω αριστερά, χρειάζονται τα αποτελέσματα των πάνω for loops, ακριβώς όπως και πάνω, για αυτό και τοποθετούμε ανάμεσα στος 2 ομάδες for loops ένα barrier, για τον αναγκαίο συγχρονισμό των κομματιών του πίνακα. Επιπλέον χρησιμοποιούμε και το clause collapse(2). Το clause collapse(2) είναι μια παράμετρος που χρησιμοποιείται για να βελτιώσει την απόδοση σε επαναληπτικές δομές (loops) που είναι φωλιασμένες. Με το collapse(2), το OpenMP θεωρεί τους δύο βρόχους ως έναν ενιαίο βρόχο με συνολικό αριθμό επαναλήψεων $N * M$, και επομένως μπορεί να μοιράσει πιο αποδοτικά το έργο στους πυρήνες του επεξεργαστή, επιτρέποντας να γίνεται καλύτερο load balancing.

Αμοιβαίος Αποκλεισμός - Κλειδώματα

Στο ερώτημα αυτό , καλούμαστε να αξιολογήσουμε διαφορετικούς τρόπους υλοποίησης κλειδώματος για αμοιβαίο αποκλεισμό. Το κρίσιμο τμήμα συνιστά η ενημέρωση του διαμοιραζόμενου πίνακα sharedNewClusters.

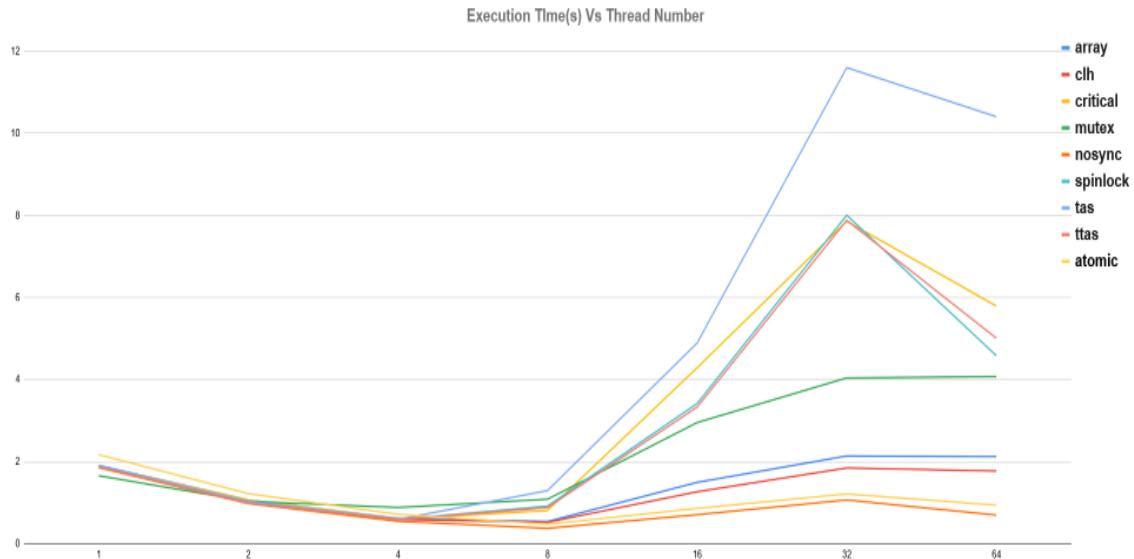
Έχουμε στη διάθεσή μας 7 διαφορετικές υλοποιήσεις του απαραίτητου κλειδώματος:

- **nosync_lock:** Η πρώτη μας υλοποίηση, η οποία δεν παρέχει κλειδώματα, ώστε να εξασφαλίσει τον αμοιβαίο αποκλεισμό μεταξύ των διαφόρων νημάτων που ταυτόχρονα τροποποιούν τον πίνακά μας. Ακριβώς επειδή η συγκεκριμένη υλοποίηση δεν παρέχει κλειδώματα, αναμένουμε να είναι και η πιο γρήγορη μέθοδος ωστόσο παράγει εσφαλμένα αποτελέσματα. Χρησιμοποιείται ως **σημείο αναφοράς** για τα υπόλοιπα κλειδώματα.
- **tas_lock:** Στη συγκεκριμένη υλοποίηση, χρησιμοποιείται το atomic operation __sync_lock_test_and_set, το οποίο εξασφαλίζει ατομική πρόσβαση στην μεταβλητή lock, έτσι ώστε μόνο ένα νήμα κάθε φορά να μπορεί να κάνει modify τον shared πίνακα.
Ουσιαστικά η λειτουργία είναι η εξής: Το νήμα, προκειμένου να κάνει acquire το lock, καλεί τη συνάρτηση lock_acquire, μέσα στην οποία καλείται η __sync_lock_test_and_set(&l->state, LOCKED). Η συνάρτηση αυτή διαβάζει το l->state και το θέτει στην τιμή LOCKED. Στη συνέχεια επιστρέφει την τελευταία τιμή του l->state πριν την ενημέρωση, Εάν η τιμή αυτή ήταν UNLOCKED, τότε σημαίνει πως το lock είναι ελεύθερο, και το νήμα το καταλαμβάνει, ενώ όλα τα υπόλοιπα σπινάρουν(busy waiting) μέχρι να απελευθερωθεί το lock ώστε να το κάνουν acquire. Εάν η τιμή αυτή ήταν LOCKED, τότε κάποιο άλλο νήμα κρατά το lock, άρα το τρέχον νήμα σπινάρει και αυτό μέχρι το κλείδωμα να γίνει ξανα διαθέσιμο.
Περιμένουμε πως αυτή η υλοποίηση κλειδώματος δεν θα είναι ιδιαίτερα αποδοτική, αφού στη διάρκεια του busy waiting τα νήματα καταναλώνουν κύκλους της CPU (καλώντας την __sync_lock_test_and_set) χωρίς να κάνουν χρήσιμη δουλειά, γεγονός που μπορεί να οδηγήσει σε performance degradation όσο τα νήματα αυξάνονται, και **εκτεταμένη χρήση των διαύλων μνήμης (cache coherence)**. Επίσης, η συγκεκριμένη υλοποίηση δεν εξασφαλίζει fairness, με αποτέλεσμα κάποιο νήματα να οδηγηθούν δυνητικά σε starvation. Ωστόσο, για μικρό αριθμό νημάτων και για μικρό αριθμό εντολών εντός του κρίσιμου τμήματος, όπου το contention είναι μικρό και δεν έχουμε εκτεταμένο spinning, θα δώσουν καλά αποτελέσματα.
- **ttas_lock:** Πρόκειται για βελτίωση του tas_lock, όπου πριν κληθεί το atomic operation __sync_lock_test_and_set, πραγματοποιείται ένα non atomic read πάνω στο lock. Το νήμα αρχικά διαβάζει το l->state και περιμένει μέχρι να γίνει διαθέσιμο. Σε αυτό το χρονικό διάστημα, κανένα atomic operation __sync_lock_test_and_set δεν γίνεται issued, έχουμε καλύτερη διαχείριση του διαύλου μνήμης και του σχετικού πρωτοκόλλου συνάφειας της cache, ωστόσο έχουμε ακόμη busy waiting(όταν γίνει το state unlocked, τότε αρχίζει να σπινάρει) και πιθανό starvation των νημάτων, επομένως θεωρούμε πως η συγκεκριμένη υλοποίηση δε θα έχει πολύ καλό speedup(ωστόσο θα είναι καλύτερο από το απλό tas_lock).

- **pthread_mutex_lock:** Η υλοποίηση αυτού του κλειδώματος είναι η εξής: Κάθε νήμα που προσπαθεί να εισέλθει στο κρίσιμο τμήμα καλεί τη συνάρτηση `pthread_mutex_lock`. Εάν ο mutex είναι ήδη acquired, το νήμα μπλοκάρει και περιμένει μέχρι το mutex να γίνει διαθέσιμο. Επομένως υλοποιείται η λογική του μπλοκαρίσματος των νημάτων, επομένως δεν έχουμε πλέον busy waiting(άρα περιμένουμε το speedup να είναι πολύ καλύτερο του TAS και καλύτερο scaling για αυξανόμενο αριθμό νημάτων), ωστόσο έχουμε context switching που πιθανώς να εισάγει overhead εάν έχουμε μικρό αριθμό νημάτων και/άρα μικρού αριθμού εντολών στο κρίσιμο τμήμα.
- **pthread_spin_lock:** Το κλείδωμα υλοποιείται μέσω spinlocks. Συγκεκριμένα το εκάστοτε νήμα καλεί την `pthread_spin_lock` πάνω στο lock->spinlock, ώστε να δεσμεύσει το lock. Εάν δεν είναι διαθέσιμο, το νήμα σπινάρει, μέχρι το lock να γίνει διαθέσιμο για να το διεκδικήσει. Προβλέπουμε ότι, όπως και στο tas lock, η συγκεκριμένη υλοποίηση θα είναι καλή για μικρό αριθμό νημάτων, όπου το contention είναι μικρό, αλλά το scaling θα είναι κακό, αφού αυξηση των αριθμού των νημάτων που θα κάνουν busy waiting σημαίνει κακή χρήση της CPU.
- **array_lock:** Πρόκειται για ένα είδος κλειδώματος, στο οποίο τα threads τροποποιούν τον πίνακα με τη σειρά, και έτσι εξασφαλίζεται το fairness. Αρχικά φτιάχνουμε έναν πίνακα από το struct `array_lock_t`, διαστάσεων όσων τα threads(που δύνανται να ζητήσουν το lock), το οποίο έχει μέσα ένα flag που λειτουργεί σαν εξατομικευμένο lock για κάθε νήμα.
Τα νήματα που ζητούν να δεσμεύσουν το Lock και να τροποποιήσουν τον πίνακα αντιστοιχίζονται σε μια θέση του πίνακα, χρησιμοποιώντας τη μεταβλητή counter που ορίζει τη σειρά τους, και κάθε ένα νήμα σπινάρει πάνω στο δικό του flag(δηλαδή μέχρι το flag να τεθεί στην τιμή TRUE), αποφεύγοντας το contention πάνω σε μία διαμοιραζόμενη μεταβλητή lock.
Όταν το νήμα ολοκληρώσει τη δουλεία του πάνω στον πίνακα, κάνει το Flag του FALSE, δηλώνοντας ότι τελείωσε η σειρά του να τροποποιήσει τον πίνακα, και θέτει κυκλικά(μέσω mod με τον αριθμό των threads) TRUE το flag του επόμενου στη σειρά νήματος.
Επιπλέον σημειώνεται πως κάθε slot στον πίνακα των νημάτων ισούται ακριβώς με το μέγεθος μιας cache line, προκειμένου να μην συμπίπτουν slots 2 νημάτων πάνω στην ίδια cache line και έτσι αποφεύγουμε το false sharing.
Ακριβώς λοιπόν επειδή πλέον κάθε νήμα σπινάρει πάνω στο δικό του flag, αποφεύγεται ο ανταγωνισμός πάνω στην shared μεταβλητή lock, και έτσι καθίσταται πιο αποδοτική από τα TAS locks ή τα spinlocks σε περιπτώσεις όπου ο αριθμός των νημάτων είναι μεγάλος, ενώ εικάζουμε πως ίσως για μικρό αριθμό νημάτων όπου τα conflicts πάνω σε μια μοιραζόμενη lock δεν είναι πολλά, εισάγει overhead.
- **clh_lock:** Η λογική που ακολουθείται εδώ είναι παρόμοια με τη λογική του array based lock, όπου τα νήματα τροποποιούν τον πίνακα με FIFO order, με τη διαφορά ότι εδώ χρησιμοποιείται μια απλά συνδεδεμένη λίστα και κάθε νήμα που επιθυμεί να πειράξει τον πίνακα φτιάχνει ένα node και κάνει τον εαυτό του attach στο τέλος της λίστας. Το πλεονέκτημα αυτής της μεθόδου είναι πως τα nodes είναι thread local, μειώνεται το cache coherence traffic σε σχέση με το array based lock, αφού εδώ κάθε νήμα χρησιμοποιεί μόνο κοντινή μνήμη για τη δημιουργία και τη διαχείριση του node, σε σχέση με την προηγούμενη περίπτωση, όπου έπρεπε τα νήματα να διαχειριστούν έναν global πίνακα για τα flags τους. Περιμένουμε λοιπόν να δούμε καλύτερη επίδοση(speedup) σε σχέση με το array_lock.

- **omp_naive**: Χρησιμοποιήσαμε pragma omp atomic που υλοποιείται με ατομικές εντολές σε επίπεδο hardware, γεγονός που το καθιστά πιο γρήγορο και αποδοτικό
- **omp_critical**: Χρησιμοποιούμε το directive #pragma omp critical προκειμένου να εξασφαλίσουμε αμοιβαίο αποκλεισμό των νημάτων και προστασία του κρίσιμου τμήματος του κώδικα. Ωστόσο υλοποιείτα κατά το runtime του Openmp μέσω mutexes επομένως δεν περιμένουμε να δούμε ιδιαίτερα καλύτερο speedup από την υλοποίηση με pthread_mutex, ίσα ίσα έχει overhead (σημαντικό)..

Παρακάτω παρατίθενται τα διαγράμματα των χρόνων και του speedup των υλοποιήσεων κλειδωμάτων που αναλύσαμε παραπάνω:



CPU\Time(s)	nosync	array	clh	critical	mutex	spinlock	tas	ttas	atomic
1	1.8445	1.9006	1.8599	1.8734	1.6489	1.8687	1.8724	1.8576	2.1629
2	0.9705	1.061	1.0381	1.0571	1.0258	0.9973	1.0256	0.9817	1.2111
4	0.5392	0.6129	0.5843	0.5928	0.8754	0.585	0.5854	0.5818	0.709
8	0.3687	0.5365	0.5163	0.8004	1.0777	0.9094	1.2875	0.8716	0.4662
16	0.6998	1.4897	1.2593	4.2762	2.9431	3.4069	4.8809	3.3202	0.8531
32	1.0583	2.1336	1.839	7.8453	4.0322	7.9946	11.5898	7.8697	1.2043
64	0.6902	2.1192	1.7628	5.78	4.0618	4.5708	10.3951	4.9968	0.9298

Όπως διαπιστώνουμε, για μικρό αριθμό threads (4), η η υλοποίηση atomic/mutex είναι η χειρότερη, ενώ με ttas, clh και critical έχουμε το βέλτιστο χρόνο.

Για μεγαλύτερο αριθμό threads (άρα και scalability) λαμβάνονται από τις υλοποιήσεις atomic και clh lock, ενώ ακολουθεί το array based lock, που πιάνουν μέγιστο speedup για 8 νήματα. Όπως εξηγήσαμε και στην προηγούμενη άσκηση, το σύστημά μας είναι NUMA aware, και το καλύτερο speedup εξασφαλίζεται στην περίπτωση του καλύτερου utilization ενός signle NUMA node, άρα στα 8 threads(με βάση το configuration μας GOMP_CPU_AFFINITY="0-63", μόλις περάσουμε τα 16 nodes ξεκινάω να χρησιμοποιώ και ένα δεύτερο NUMA node, και άρα το performance πέφτει).

Στη συνέχεια παρατηρούμε πως τον επόμενο καλύτερο χρόνο κάνουν τα mutex(δεν έχουμε καθόλου spinning των νημάτων παρά μόνο το overhead του content switch το οποίο μπορεί να οδηγήσει σε degradation μόνο σε μικρό αριθμό νημάτων όπως εξηγήσαμε παραπάνω) και ttas lock(όπου έχουμε λιγότερο busy waiting όπως εξηγήσαμε παραπάνω).

Τον χειρότερο χρόνο και scaling, κάνουν όπως προβλέψαμε οι υλοποιήσεις tas lock και τα spinlocks, λόγω του πολύ αυξημένου busy waiting όσο αυξάνεται και ο αριθμός των νημάτων.

Επίσης παρατηρούμε ότι γενικά τον χειρότερο χρόνο τον κάνουν όλες οι υλοποιήσεις για 32 threads, το οποίο και πάλι σχετίζεται μετο GOMP_CPU_AFFINITY="0-63", αφού για 32 threads γίνεται χρήση και των 4 NUMA nodes, και άρα εισάγεται cache coherence overhead από την επικοινωνία των νημάτων σε όλα τα cores.

Συνολικά οι καλύτερες υλοποιήσεις κλειδώματος φαίνονται να είναι το clh_lock, το array_lock καθώς και η χρήση των εντολών pragma omp atomic(που είναι lock free και εξασφαλίζει ατομική πρόσβαση σε επίπεδο υλικού).

Ενότητα 2.3: Ταυτόχρονες Δομές Δεδομένων

Σε αυτό το σημείο θα μελετήσουμε διάφορες υλοποιήσεις μιας **απλά συνδεδεμένης λίστας**.

Έχουμε τις εξής υλοποιήσεις:

- **Coarse-grain locking:** Χρησιμοποιείται ένα lock για όλους τους κόμβους της λίστας, πρακτικά δηλαδή μόνο ένα νήμα μπορεί να επεξεργάζεται τη λίστα κάθε φορά.
Καταλαβαίνουμε ότι αυτή η υλοποίηση γενικά θα παράγει κακά αποτελέσματα, αφού ακόμη και νήματα που θέλουν να επεξεργαστούν διαφορετικούς κόμβους και το ένα modification δεν κάνει interfere με το άλλο, δεν μπορούν να τρέξουν παράλληλα, άρα περιμένουμε το throughput να μειώνεται για πολλά νήματα..
- **Fine-grain locking:** Χρησιμοποιείται ένα lock για κάθε κόμβο πάνω στη λίστα, και έτσι πλέον δεν κλειδώνω ολόκληρη τη λίστα, αλλά μόνο τους κόμβους στους οποίους δουλευει. Η ιδέα είναι η εξής: αρχικά κλειδώνω τον κόμβο πάνω στον οποίο θέλω να κάνω ένα operation και στη συνέχεια κλειδώνω και τον επόμενο κόμβο, εξασφαλίζοντας ότι κάποιο άλλο νήμα δεν θα πειράζει τους κόμβους που πειράζει το current thread. Οι μέθοδοι contains, add, remove χρησιμοποιούν το macro TRAVERSE_LIST(), η οποία και λόγω των κλειδωμάτων που έχει(hand over hand locking) συνιστά το bottleneck, και όχι η ll_add ή η ll_remove. Όσο ο αριθμός των νημάτων που καλούνται να διασχίσουν ταυτόχρονα τη λίστα αυξάνεται, αυτά να κολλάνε το ένα πίσω από το άλλο λόγω ακριβώς του hand over hand locking, πράγμα που οδηγεί σε performing degradation(το throughput είναι πολύ κακό για 128 νήματα). Σε σχέση ωστόσο με την coarse grain υλοποίηση θεωρούμε πως το throughput θα είναι καλύτερο για μεγαλύτερο αριθμό νημάτων(για μικρότερο αριθμό νημάτων, το overhead των πολλαπλών κλειδωμάτων κάνει το fine grain throughput χειρότερο από αυτό του coarse grain), αφού πλέον δεν έχω όλα τα νήματα να ανταγωνίζονται για 1 lock, άρα συνολικά θα έχω λιγότερο contention, ωστόσο και πάλι η υλοποίηση αυτή δεν είναι η καλύτερη δυνατή αφού έχω μη αποδοτική διάσχιση της λίστας από πολλά νήματα ταυτόχρονα.
- **Optimistic synchronization:** Σε αυτό την υλοποίηση της λίστας, υποθέτουμε ότι τα conflicts που δύνανται να προκύψουν μεταξύ των νημάτων είναι σπάνια. Επομένως, κάθε νήμα αρχικά κάνει την αναζήτηση του στοιχείου(δηλαδή traverse, που δεν έχει κλειδώματα), κάνει το operation που θέλει(π.χ. εισαγωγή / διαγραφή κόμβου) με κατάλληλα κλειδώματα και έπειτα ελέγχει για conflicts/inconsistencies που πιθανόν έχουν προκύψει και κάνουν τη λίστα μη συνεπή. Εάν όντως βρεθούν conflicts, τότε το operation αναιρείται και και το ξαναπροσπαθεί. Το bottleneck αυτής της υλοποίησης είναι το ότι διασχίζουμε όλη τη λίστα μέχρι το σημείο που μας ενδιαφέρει δύο φορές, μία χωρίς τα locks(traverse_list) και μία κρατώντας τα(validate, που συμβαίνει μετά τα locks), άρα καθυστερούμε όσα locks θέλουν να μας προσπεράσουν και να πάνε παρακάτω. Παρατηρούμε ότι για αύξηση του αριθμού των νημάτων έχω καλό scaling. Ακόμη, παρατηρούμε γενικά βελτίωση έως τα 64 νήματα αλλά πτώση για τα 128 νήματα του throughput για όλα τα workloads, εικάζουμε λόγω του context switching, αφού ενδέχεται να κρατάει το lock σε κάποια χρονική στιγμή κάποιο νήμα το οποίο πρέπει να γίνει schedule out για να μπει άλλο στο core που τρέχει.
- **Lazy synchronization:** Η ιδέα της υλοποίησης είναι η εξής: έχω μία τιμή boolean η οποία μου δείχνει εάν ο κόμβος είναι λογικά διαγραμμένος ή όχι. Αυτή η επιλογή στοχεύει στο να μην κάνω φυσική διαγραφή κόμβων συνεχώς, καθυστερώντας έτσι

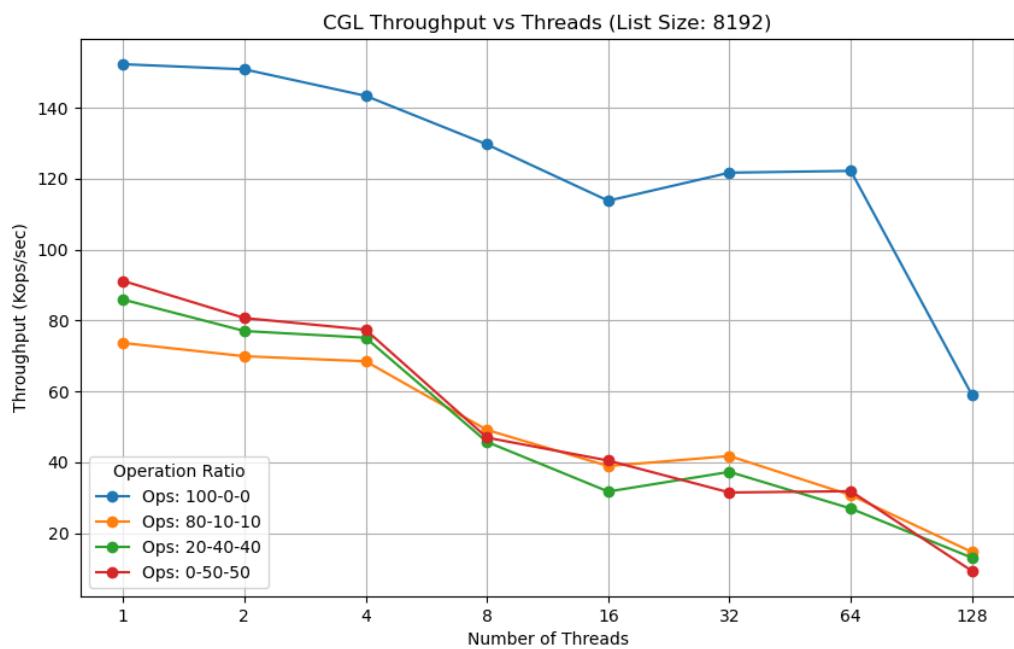
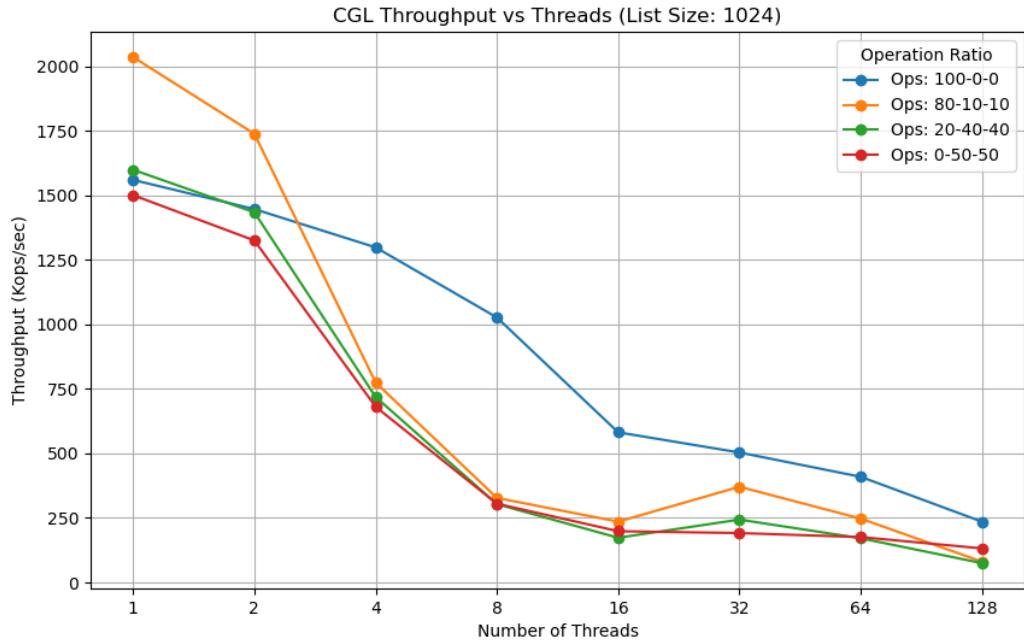
τον συγχρονισμό μέχρι να παραστεί ανάγκη. Έτσι η μέθοδος `contains` διατρέχει τη λίστα χωρίς κλειδώματα και ελέγχει αυτά τα μπιτάκια, η `add` διατρέχει τη λίστα και κάνει τα απαραίτητα κλειδώματα, και ελέγχει τη συνέπεια με τη `validate`, ενώ η `remove` είναι η πραγματικά `lazy` μέθοδος, αφού στο πρώτο βήμα αφαιρεί λογικά τον κόμβο(όχι φυσική διαγραφή -> όχι `locks` αφού δε προκύπτουν θέματα συνέπειας), στο 2ο βήμα διαγράφει και φυσικά πειράζοντας τους δείκτες. Ακριβώς λοιπόν επειδή στη συγκεκριμένη υλοποίηση έχουμε τα λιγότερα δυνατά κλειδώματα, η υλοποίηση αυτή θα παρουσιάζει πλεονέκτημα σε σχέση με την `optimistic` υλοποίησεις για πολλά `insertions&deletes`, αφού κάνει `local locking` για να ελέγχει για συνέπεια και δεν ξαναδιασχίζει τη λίστα. Ακόμη παρατηρούμε πολύ καλό `scaling` για όσο αυξάνεται το `concurrency`. Το `throughput` για το `workload 100_0_0` είναι πάρα πολύ καλό(αυξάνεται μια τάξη μεγέθους σε σχέση με αυτό του ίδιου `workload` για την `optimistic` υλοποίηση) αφού δεν έχουμε καθόλου κλειδώματα, και απλά τσεκάρω το `marked bit` και το `key`, αντί να κλειδώνουμε & να ξεκλειδώνουμε `nodes` και να τσεκάρουμε μέσω της `validate` τους `curr` & `next`.

- **Non-blocking synchronization:** Η συγκεκριμένη υλοποίηση εξαλείφει τα `locks` από όλες τις μεθόδους, ώστε να αποφύγουμε το `overhead` των `locks` και να κάνουμε την υλοποίηση ακόμη πιο γρήγορη. Η ιδέα που εφαρμόζεται σε αυτή την υλοποίηση είναι η εξής: αντί για να περιμένω για λήψη ενός κλειδώματος, θα κάνω “έλεγχο και επαναπροσπάθεια”. Χρησιμοποιούνται τα `atomic operations Compare and Swap(CAS)` για να κάνουμε τις απαραίτητες αλλαγές χωρίς χρήση κλειδωμάτων, και έτσι χωρίς να μπλοκάρονται νήματα.

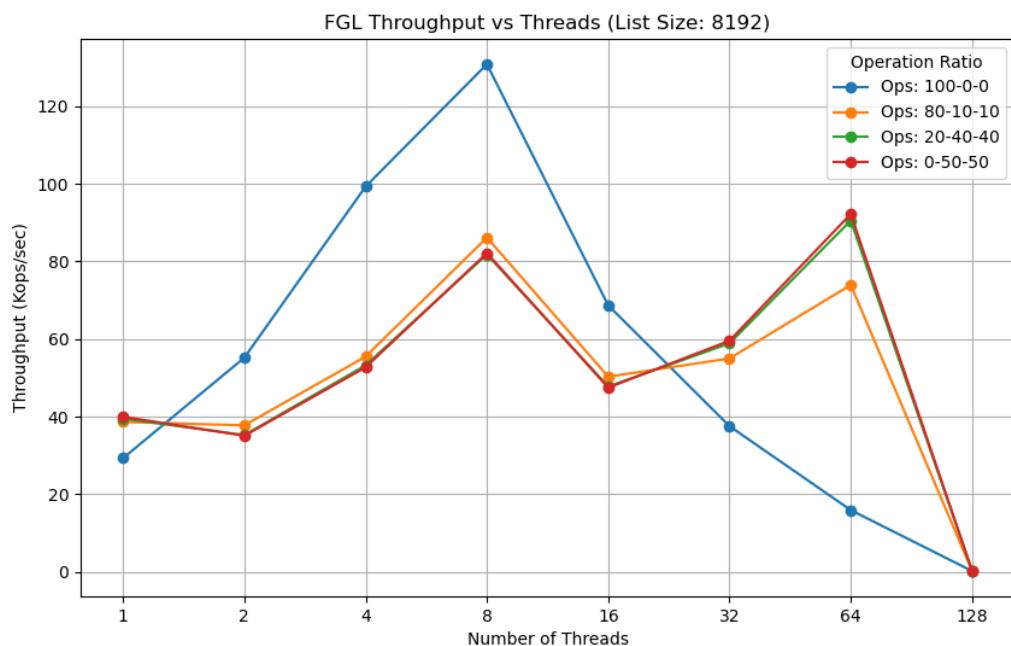
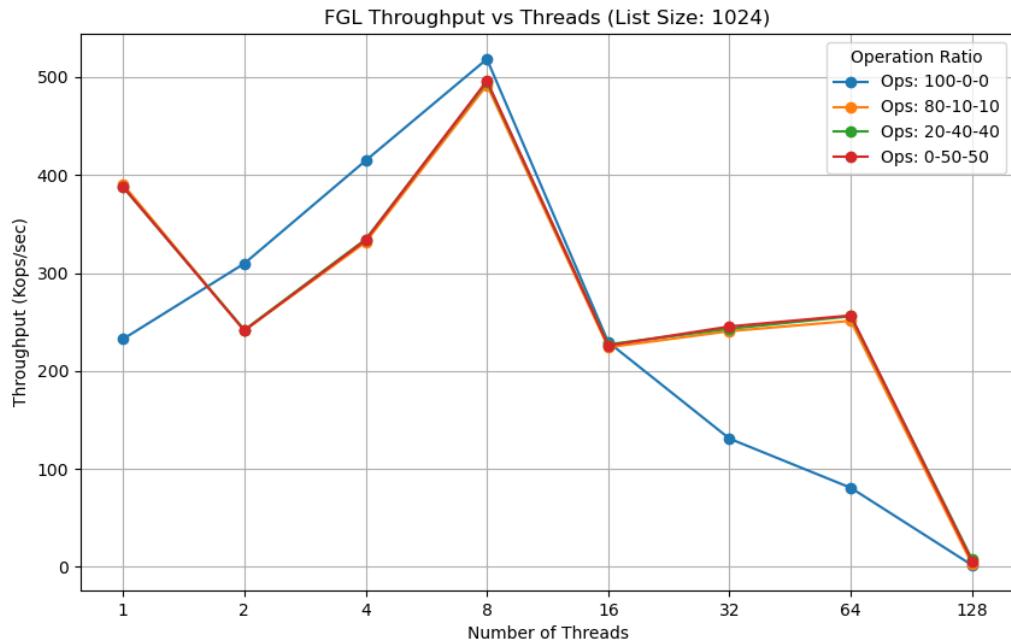
Γενικά θεωρούμε πως η συγκεκριμένη υλοποίηση θα δώσει καλό `throughput` για πολλά νήματα και θα έχουμε καλό `scaling`, αφού επιτρέπει στα νήματα να δουλεύουν ταυτόχρονα πάνω στη λίστα, χωρίς να μπλοκάρουν το ένα το άλλο, κάτι το οποίο μπορεί να είναι `bottleneck` για `high concurrency` περιβάλλοντα, λόγω π.χ. του `latency` που εισάγεται από τα νήματα που περιμένουν να καταλάβουν τα `locks`. Τα 128 νήματα παρουσιάζουν και πάλι μια βύθιση, ωστόσο παρατηρούμε καλύτερους χρόνους σε σχέση με άλλες υλοποιήσεις. Η βασική επιβάρυνση σε αυτή την υλοποίηση είναι η `ll_contains`, η οποία κατά τη διάσχιση της λίστας τσεκάρει ποια `nodes` είναι μαρκαρισμένα και τα αφαιρεί και φυσικά.

Τα διαγράμματα που προκύπτουν είναι τα εξής:

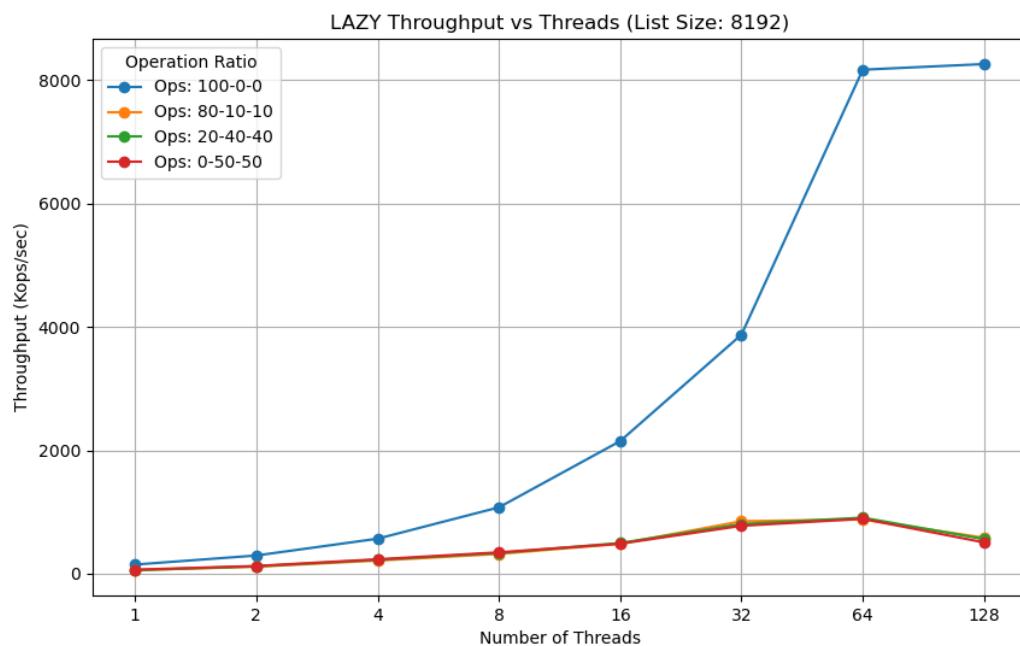
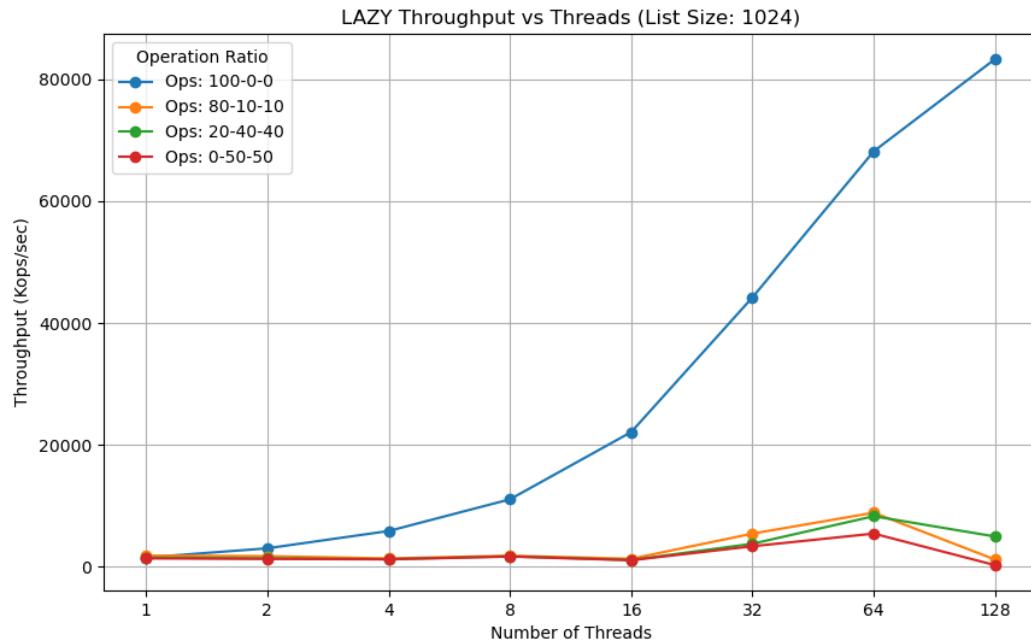
Coarse Grain{1024, 2048}:



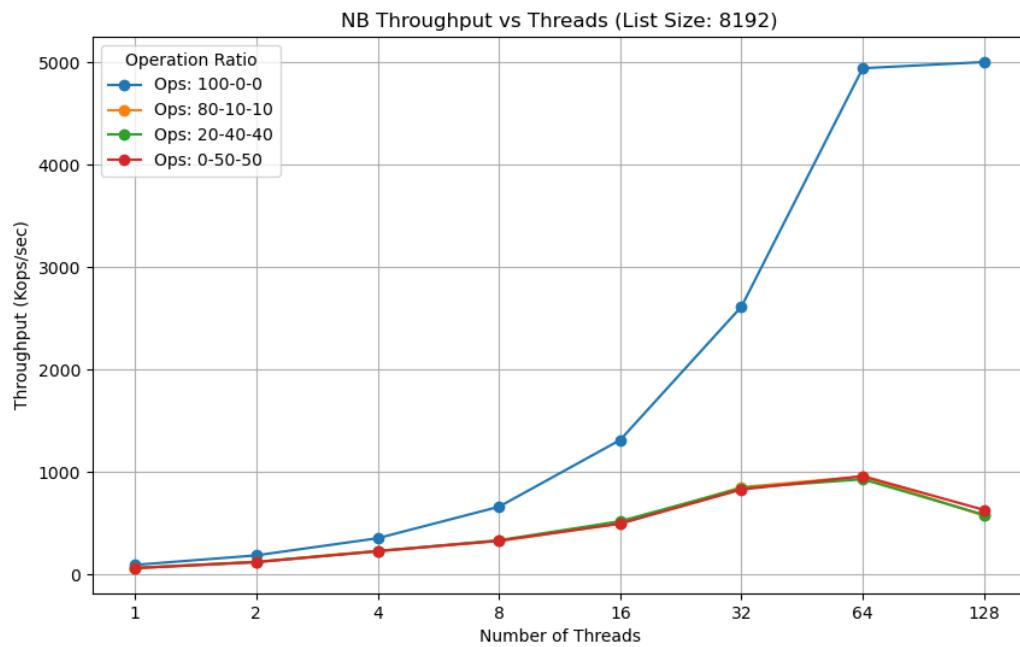
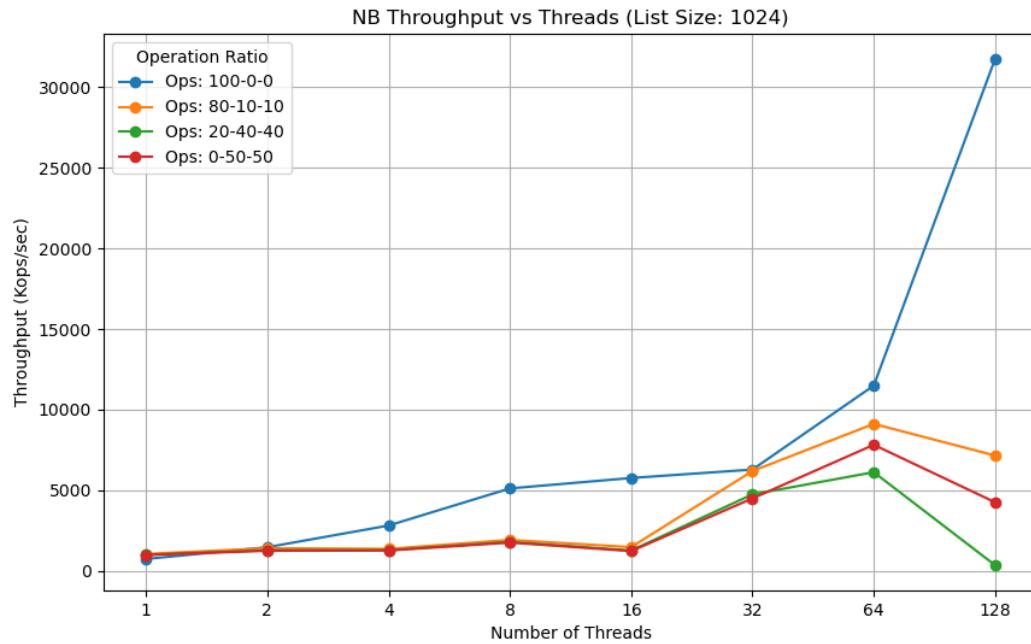
Fine Grain{1024,2048}:



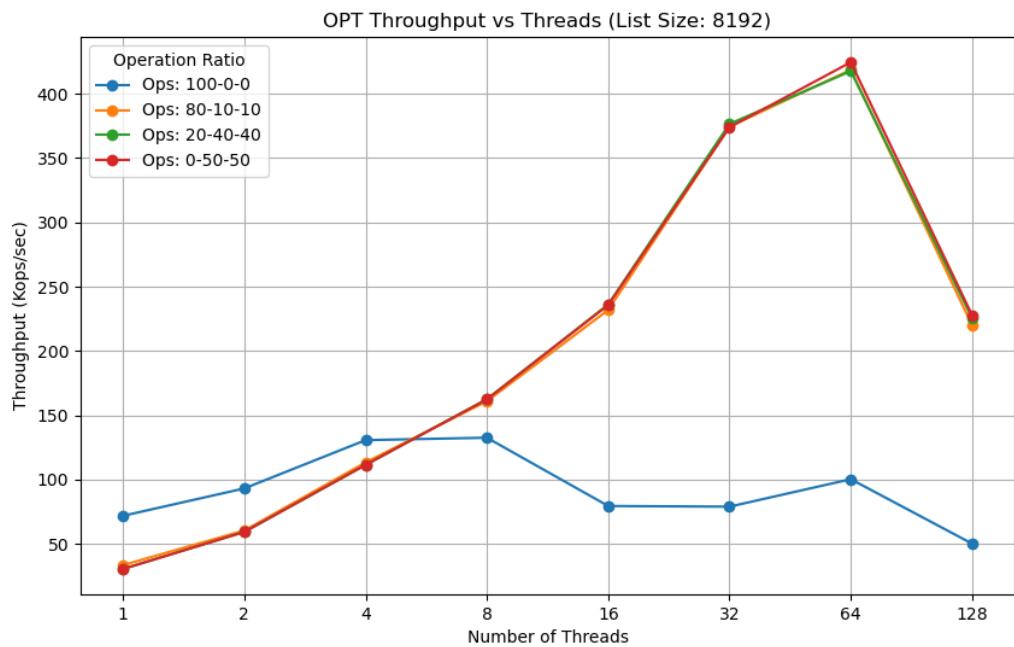
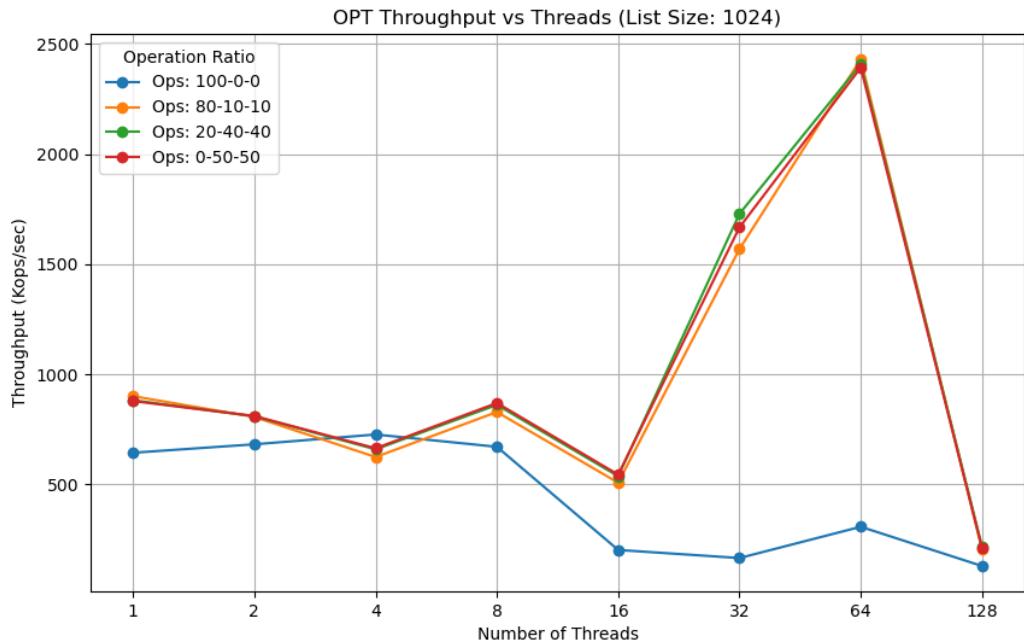
Lazy implementation{1024, 2048}:



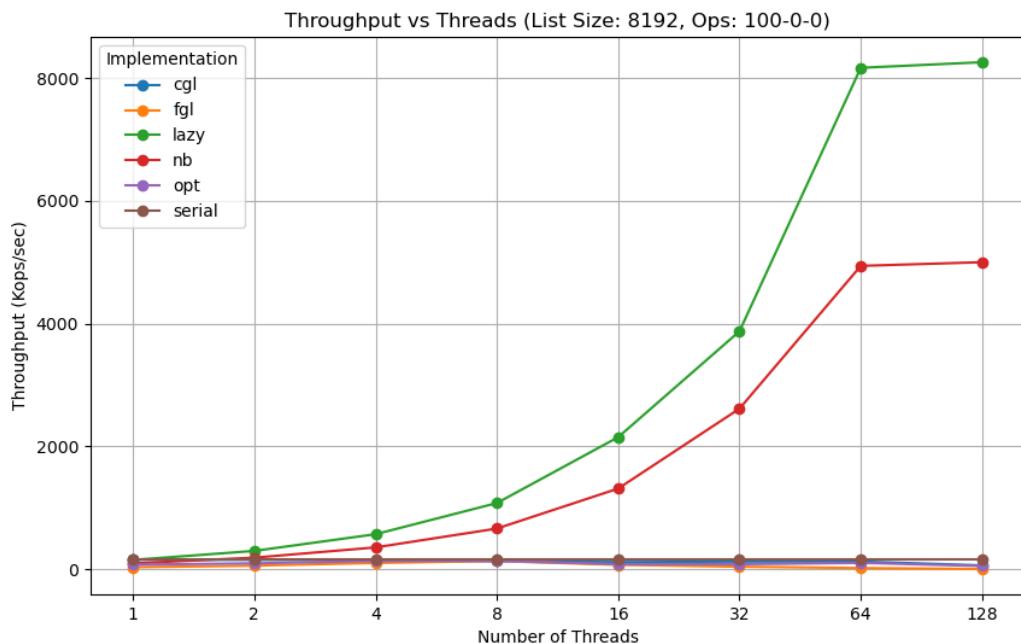
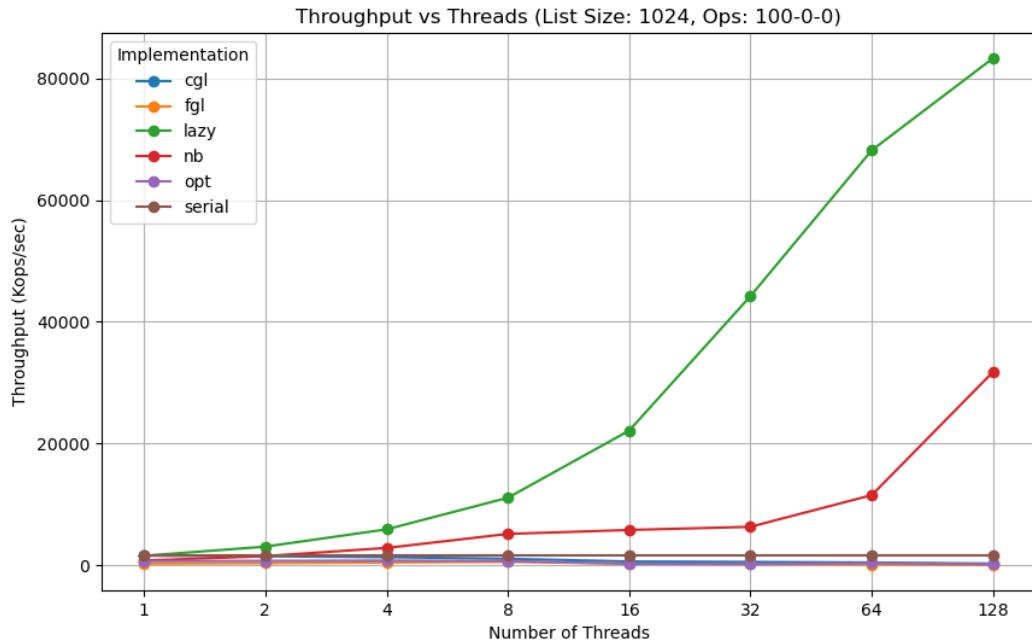
Non blocking{1024, 2048}:



Optimistic Synchronization {1024, 2048}:



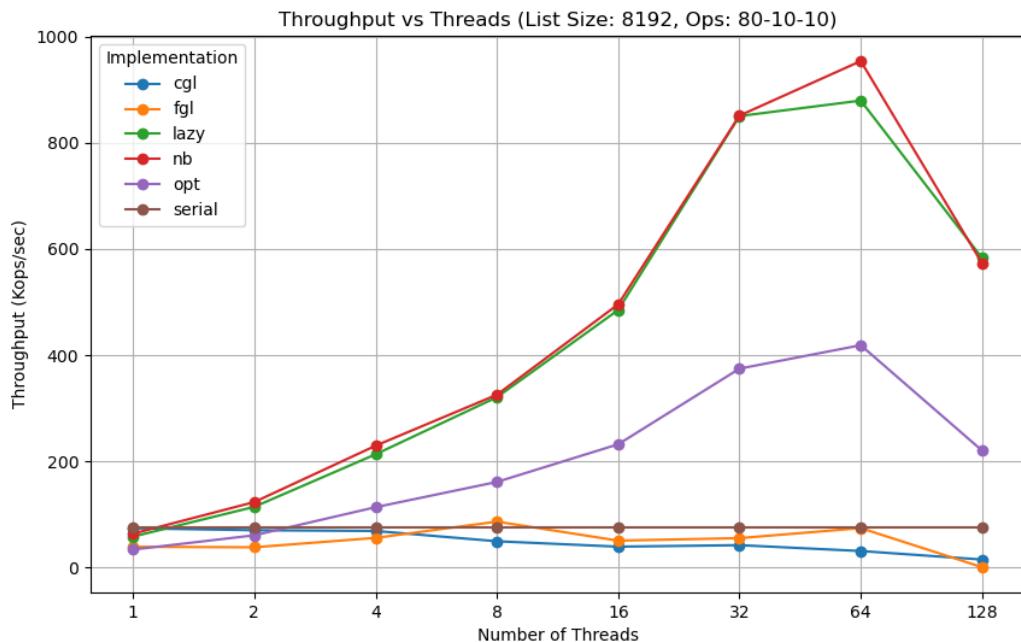
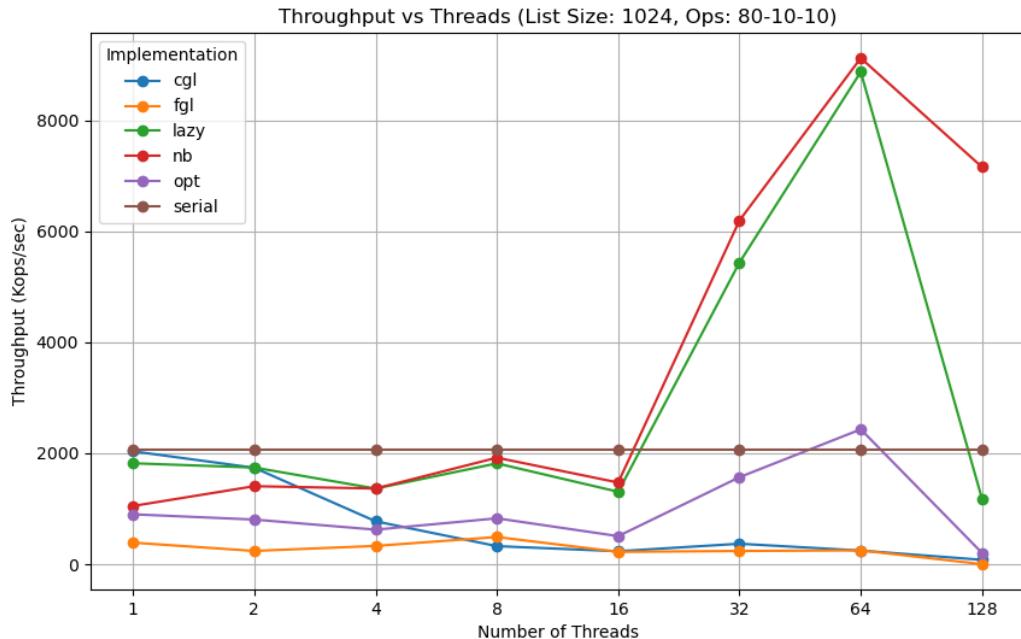
Παραθέτουμε επίσης τα διαγράμματα ανάλογα με τα διάφορα workloads:
Workload 100-0-0:



Όπως βλέπουμε στο 100-0-0, η υλοποίηση με την καλύτερη κλιμάκωση είναι η lazy και ακολουθεί η non blocking, λόγω της έλλειψης των κλειδωμάτων. Το bottleneck στην nb σε σχέση με την lazy είναι η χρήση της list_search, η οποία ξαναδιασχίζει όλη τη λίστα, ενώ στην lazy έχω μόνο τοπικούς ελέγχους κόμβων. Ωστόσο έχουμε μεγαλύτερο throughput από αυτες τις υλοποιήσεις για 2 ταξις μεγέθους σε σχέση με τις υπόλοιπες υλοποιήσεις. Βλέπουμε ότι γενικά ολες οι υπόλοιπες υλοποιήσεις έχουν αρκετά κακά αποτελέσματα για αυτό το workload λόγω του overhead των κλειδωμάτων, ενώ επίσης το μέγεθος της λίστας

παίζει σημαντικό ρόλο, αφού για μεγαλύτερη λίστα θα χρειαστεί να προσπελάσω περισσότερους κόμβους (άρα μεγαλύτερες καθυστερήσεις ιδίως όταν τα νήματα μπλοκάρονται το ένα πίσω από το άλλο) και θα χρειαστώ περισσότερα κλειδώματα.

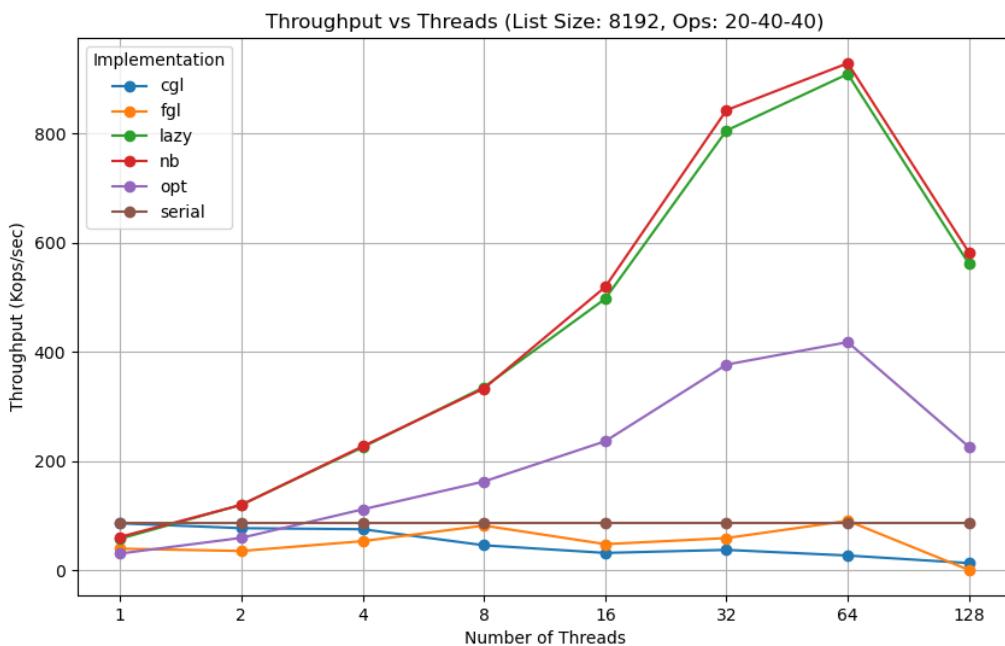
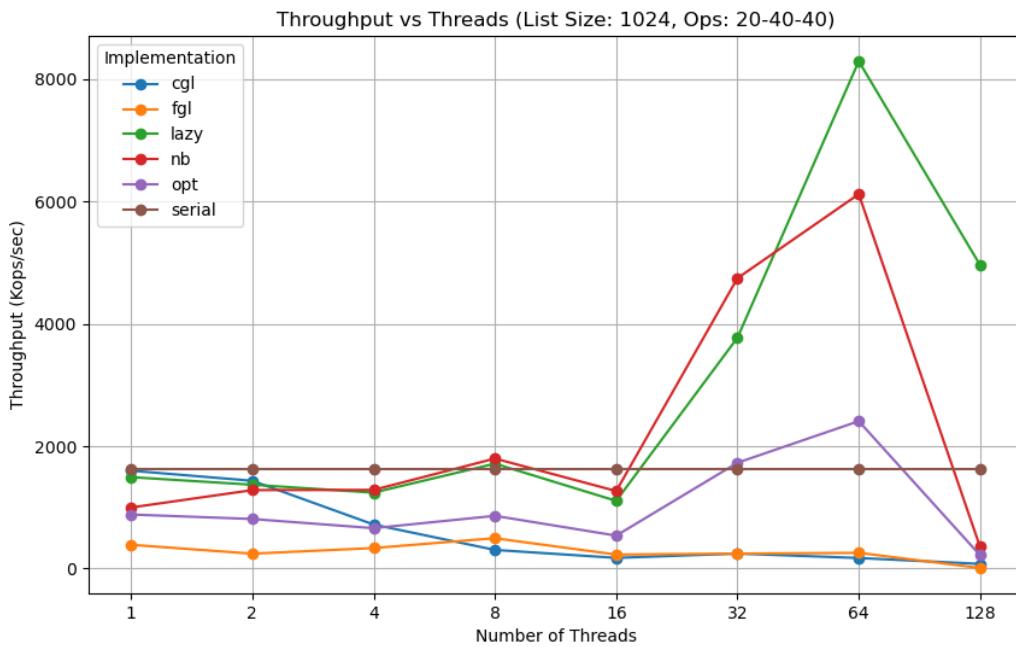
Workload 80-10-10{1024,2048}:



Εδώ έχουμε ως καλύτερη την υλοποίηση non blocking, με την lazy να ακολουθεί, που είναι λογικό αφού η non blocking με την παντελή ελλειψη κλειδωμάτων υπερβαίνει το throughput της lazy, η οποία κάνει τα λιγότερα δυνατά κλειδώματα, και έπειται η optimistic με επίσης καλό throughput, ώστόσο το πλεονέκτημα των υλοποιήσεων σε αυτό το workload δεν είναι

στον ίδιο βαθμό σε σχέση με το προηγούμενο workload. Παρατηρούμε επίσης ότι για μεγάλο αριθμό Threads, πέραν των 3 προαναφερθέντων υλοποιήσεων, το throughput γίνεται κακό, ενώ και πάλι το αυξανόμενο μέγεθος της λίστας ρίχνει την απόδοση.

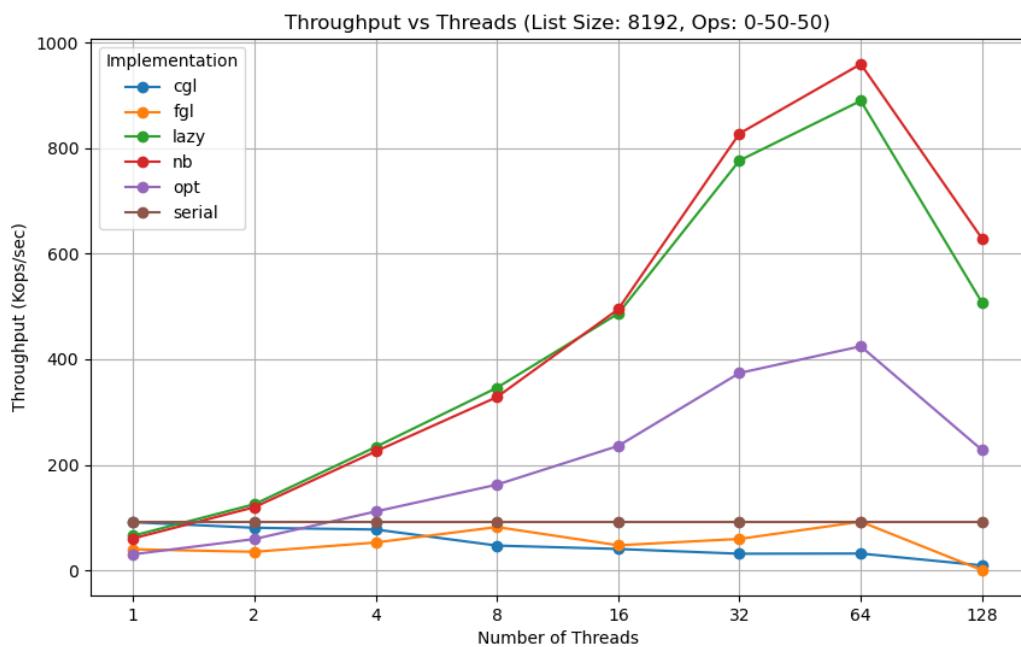
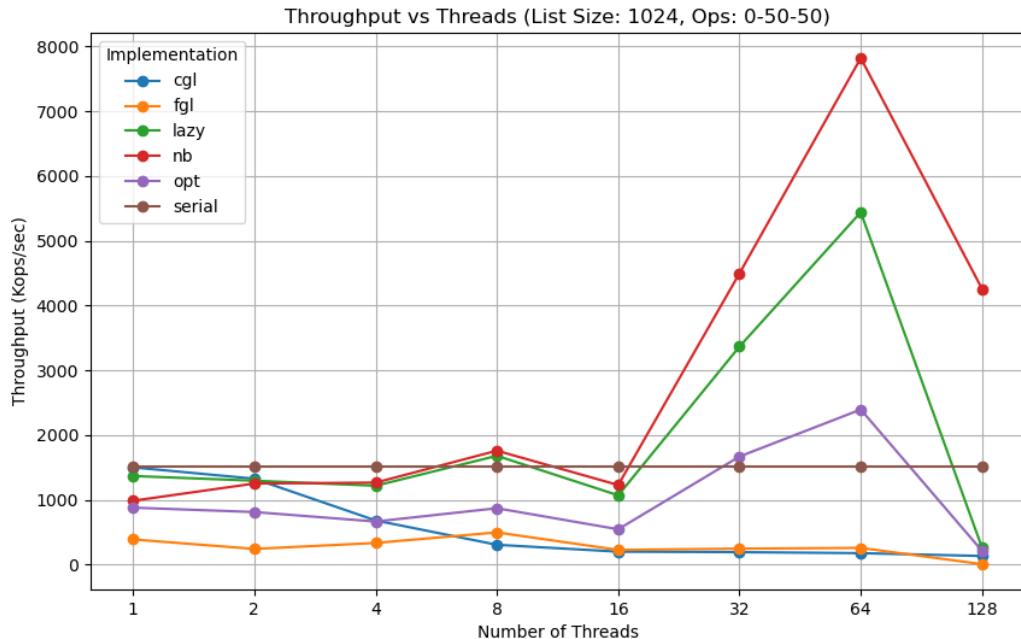
Workload 20-40-40{1024,2048}:



Παρατηρούμε σχετικά παρόμοια συμπεριφορά σε σχέση με την παραπάνω υλοποίηση, ωστόσο βλέπουμε ότι η lazy synchronization πέφτει, αφού πλέον υπερτερούν τα add & remove, επομένως υπάρχουν και περισσότερα κλειδώματα σε σχέση με το να έχουμε

περισσότερα contains(προηγούμενο workload). Παρατηρούμε και πάλι ότι το μεγαλύτερο μέγεθος της λίστας μας οδηγεί σε μικρότερο throughput.

Workload 0-50-50{1024,2048}:



Και πάλι οι υλοποιήσεις που αποφεύγουν τα κλειδώματα ή τα χρησιμοποιούν σε μικρό βαθμό(nb, lazy, opt) κυριαρχούν, αφου η διαχείριση των νημάτων για συνέπεια της λίστας είτε μέσω κλειδωμάτων είτε μέσω ατομικών εντολών για τα insert&delete γίνεται κρίσιμη(έχουμε read-writes). Επομένως και η nb έχει μεγαλύτερη διαφορά όσον αφορά το throughput σε σχέση με τις υπόλοιπες υλοποιήσεις.

Συνολική παρατήρηση για όλα τα διαγράμματα είναι ότι η coarse grain υλοποίηση ξεκινά καλά, αλλά καταρρέει όσο ο αριθμός των νημάτων αυξάνεται, λόγω των conflicts που προκύπτουν πάνω στο κοινό lock για όλη τη λίστα.

Άσκηση 4

Παραλληλοποίηση και βελτιστοποίηση k-means σε κάρτες γραφικών

3.1

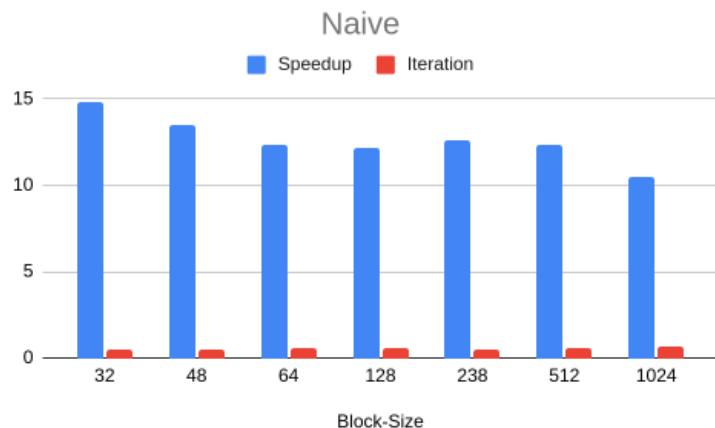
Βλέπουμε ότι έχουμε 2 κάρτες γραφικών, μια μικρή για έλεγχο και debugging και μια μεγάλη για την τελική λήψη αποτελεσμάτων. Αρχικά με κατάλληλα σκριπτς παίρνουμε τα δεδομένα κάθε GPU, έτσι έχουμε:

```
=====
== GPU Device 1 ==
Name: Tesla V100-SXM2-32GB
CUDA Compute Capability: 7.0
Number of SMs: 80
Warp Size: 32
Total Global Memory: 32494 MB
Shared Memory Per Block: 48 KB
Max Threads Per Block: 1024
Max Threads Per SM: 2048
Max Grid Dimensions: [2147483647, 65535, 65535]
Max Threads Dimensions: [1024, 1024, 64]
Clock Rate: 1530 MHz
Memory Clock Rate: 877 MHz
Memory Bus Width: 4096 bits
=====
===== GPU Device 2 =====
Name: NVIDIA GeForce GTX 1060 6GB
CUDA Compute Capability: 6.1
Number of SMs: 10
Warp Size: 32
Total Global Memory: 6065 MB
Shared Memory Per Block: 48 KB
Max Threads Per Block: 1024
Max Threads Per SM: 2048
Max Grid Dimensions: [2147483647, 65535, 65535]
Max Threads Dimensions: [1024, 1024, 64]
Clock Rate: 1809.5 MHz
Memory Clock Rate: 4004 MHz
Memory Bus Width: 192 bits
=====
```

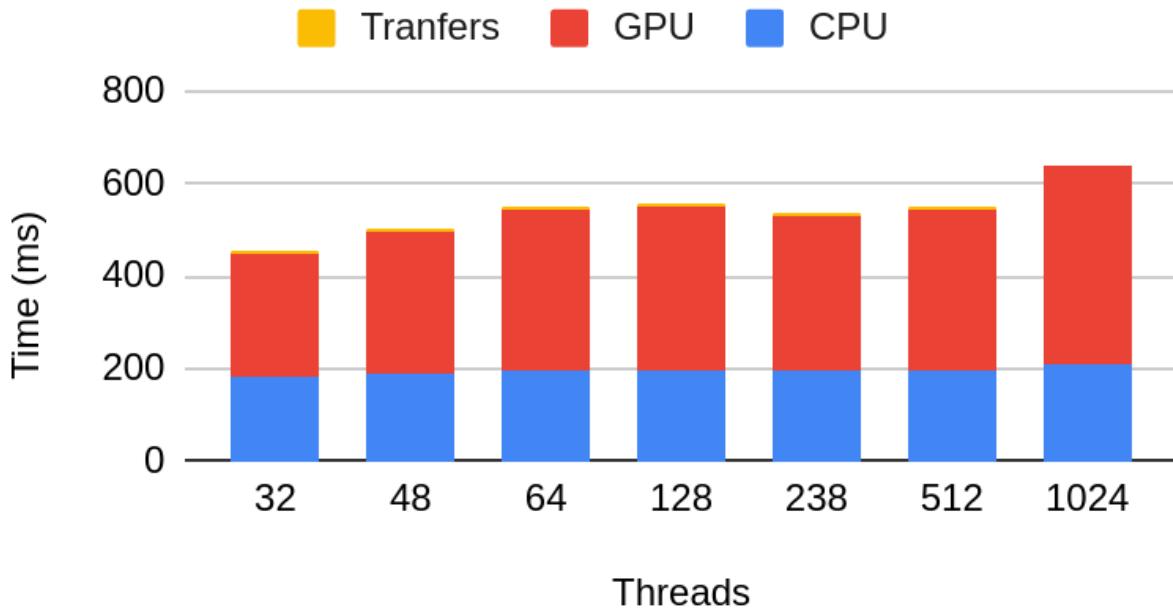
Αυτά τα χαρακτηριστικά θα μας χρειαστούν στην συνέχεια.

Naive Version

Αρχικά γράφουμε τα TO-DOS και αφού επιβεβαιώσουμε ότι είναι ορθά με τα Validations, προκύπτουν τα εξείς νούμερα για τους χρόνους εκτέλεσης:



Naive Version



3.2 Για αρχή βλέπουμε πολύ καλά αποτελέσματα όσο αφορά το speedup, καθώς μόνο η εισαγωγή στη GPU του kmeans δίνει στη χειρότερη περίπτωση speedup πάνω από 10. **Θυμίζω με το openmp είχαμε περίπου 5.**

3.3 Transfer Time & CPU Time: Οι χρόνοι αυτοί δε διαφέρουν σημαντικά με το block-size και αυτό είναι αναμενόμενο, καθώς το block size επηρεάζει τον τρόπο που η GPU επιταχύνει την εκτέλεση του kmeans και αφού υπάρχει και η εντολή `cudaDeviceSynchronize()`; περιμένουμε να τελειώσει την εκτέλεση η GPU για να συνεχίσει η CPU.

GPU Time: Γενικά παρατηρούμε πως τα block sizes διαφέρουν (λίγο, αλλά διαφέρουν) στον συνολικό χρόνο εκτέλεσης της GPU, αλλά η επιρροή αυτή δεν είναι και πολύ μεγάλη, και αυτό κυρίως έγκειται στο γεγονός πως ο αλγόριθμός μας kmeans είναι ένας memory bound αλγόριθμος, δηλαδή κυρίως επηρεάζουν οι προσβάσεις στη μνήμη(τις οποίες και θα επιχειρήσουμε να βελτιώσουμε στη συνέχεια) και γενικότερα η διαχείριση της μνήμης, παρά τα computations τα οποία δεν είναι υπολογιστικά βαριά. Παρατηρούμε μόνο μια σχετικά μεγάλη αύξηση του χρόνου για το block size 1024, και εικάζουμε πως αυτό συμβαίνει επειδή δεν έχουμε αρκετούς registers/thread block για να ευοηρετήσουμε και τα 1024 νήματα, με αποτέλεσμα να έχουμε spilling στην local memory.

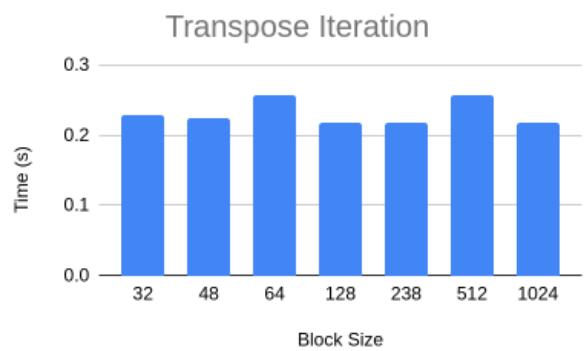
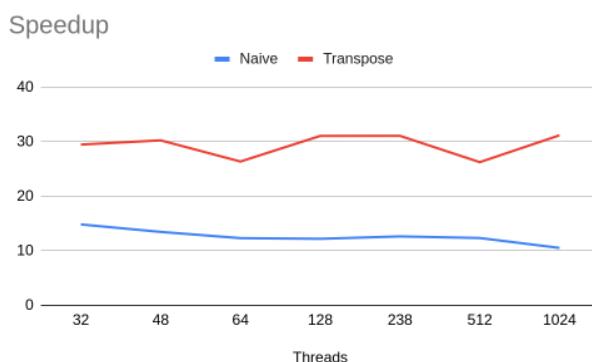
Γενικά παρατηρούμε σίγουρα μια καλύτερη επίδοση του αλγορίθμου σε σχέση με το να τρέξουμε όλο μας τον κώδικα σε CPU, ωστόσο η υλοποίηση αυτή δεν είναι ιδανική καθώς θα έπρεπε ο χρόνος μας να είναι πολύ μικρότερος. Μέρος του προβλήματος θεωρούμε πως είναι ο τρόπος που κάνουμε accesses στη μνήμη και τα latencies που δημιουργούνται τα οποία δεν

μπορούμε να κρύψουμε βάζοντας να τρέξουν στο ενδιάμεσο άλλα warps(oversubscription) μέχρι τα δεδομένα να έρθουν για το πρώτο warp. Το καλύτερο speedup το παίρνουμε για 32 νήματα, και αυτό υποθέτουμε ότι μπορεί να συμβαίνει επειδή για μικρότερα block sizes έχουμε περισσότερα blocks, και άρα χρησιμοποιούμε το σύνολο των διαθέσιμων SM, ενώ το oversubscription όπως αναφέρθηκε μας βοήθησε να κρύψουμε τα memory latencies.

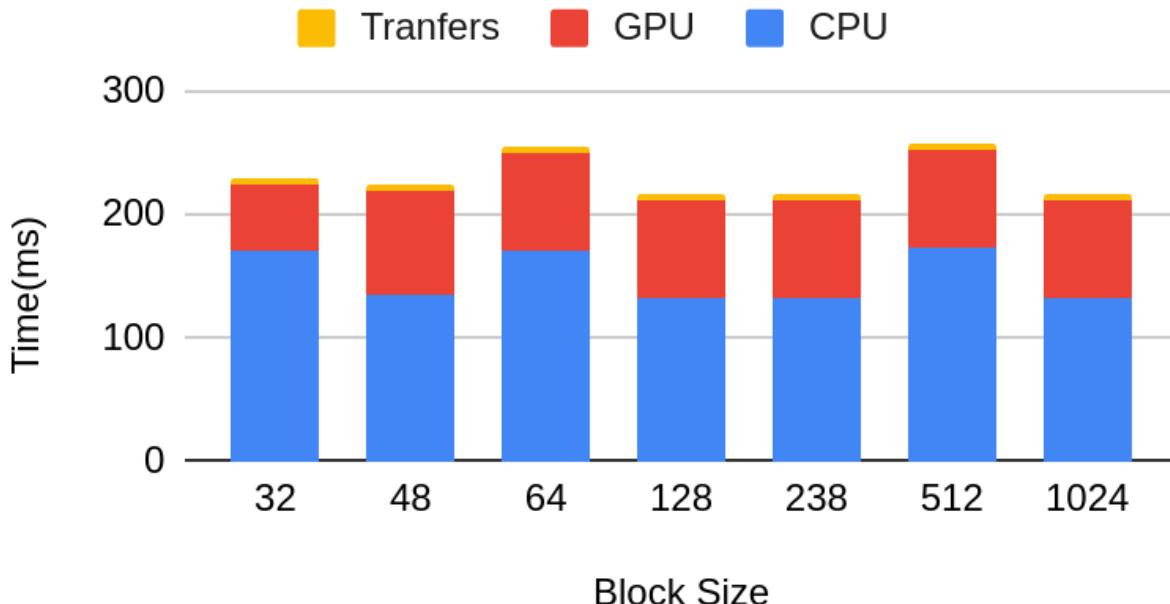
Αξίζει να σημειώσουμε ότι για block size που δεν είναι πολλαπλάσιο του 32 εδώ τα 48 και 238 γνωρίζουμε ότι δεν αξιοποιούν πλήρως τις ικανότητες της GPU, δεδομένου ότι οι εντολές εκτελούνται σε warps και τα warps οργανώνονται σε 32άδες μέσα στα μπλοκ, με αποτέλεσμα να μένουν πιθανώς στο τελευταίο warp ενός block idle threads. Παρόλα αυτά βλέπουμε ότι οι SMs κάνουν καλή δουλειά και τελικά αυτό δεν έχει πολύ μεγάλη σημασία όσο αφορά το συνολικό χρόνο εκτέλεσης.

Transpose Version

Αφού συμπληρώσαμε τα TO-DOs και επαληθεύσαμε τα αποτελέσματα, πήραμε τις μετρήσεις.



Transpose



CPU&transfer times: Παρατηρούμε βελτιωμένη επίδοση σε σχέση με την naive version για την CPU, η οποία βελτίωση θεωρούμε ότι σχετίζεται με τον τρόπο που πλέον έχουμε αποθηκεύσει τον πίνακα στη μνήμη και το πως τον διαχειρίζεται η cache μας. Τα transfer times και πάλι έχουν σταθερό χρόνο (και μειωμένο από τη naive εκτέλεση) όπως και είναι λογικό, αφού οι πίνακες που μεταφέρουμε δεν αλλάζουν με το Block Size, όσον αφορά τα transfers αυτό που παίζει ρόλο είναι το μέγεθος που μεταφέρεται, καθώς και η δομή των πινάκων.

GPU Time: Το performance σε κάθε περίπτωση είναι σημαντικά μειωμένο από την naive version, και η διαφορά έγκειται κυρίως στον τρόπο που πλέον υπολογίζεται η απόσταση του κάθε object το οποίο έχει αναλάβει κάθε νήμα από τα clusters, καθώς και στον τρόπο που αποθηκεύονται οι πίνακες. Πιο συγκεκριμένα, γνωρίζουμε πώς ο τρόπος με τον οποίο γίνεται access η μνήμη είναι ο εξής, συμβαίνει ένα transaction 128 bytes το οποίο η μνήμη στέλνει στο νήμα που ζήτησε μνήμη. Ωστόσο εμείς γνωρίζουμε ότι, όσον αφορά ένα half warp, τα νήματα που ανήκουν σε αυτό εκτελούν την ίδια εντολή σε ένα cycle. Επομένως όταν το thread 0 ζητά το στοιχείο coord1[0] ενός πίνακα(στη δική μας περίπτωση οι πίνακες είναι το dimObjects[numCoords][numClusters] & dimClusters[numCoords][numClusters], και γνωρίζοντας πώς κάθε element είναι double=8 bytes), σε ένα single transaction θα έρθουν 128/8=16 στοιχεία του πίνακα(coord1[0]-coord1[15]), τα οποία θα μπορέσουν να χρησιμοποιηθούν και από τα threads 1 έως 15 στον ίδιο κύκλο, άρα το bandwidth μεγιστοποιείται, αφού εκμεταλλευόμαστε όλο το transaction λόγω της coalesced access από τα νήματα του half warp.

Αντίθετα στη naive υλοποίηση, σε 1 transaction έρχονταν τα στοιχεία

coord1[0], coord2[0], coord3[0], ... coordn[0], όπου n = numCoords, τα οποία στοιχεία μπορούν να χρησιμοποιηθούν μόνο από το νήμα 0 που τα ζήτησε, και άρα θα χρειαζόμασταν 2o transaction για το νήμα 2 προκειμένου να προχωρήσει η εκτέλεση, 3o transaction για το νήμα 3 κλπ άρα έχουμε κακή αξιοποίηση της L1 cache.

Παρατηρούμε πως το Block size 1024 μας δίνει το καλύτερο speedup, ενώ γενικά δεν φαίνεται το block size να επηρεάζει σημαντικά την απόδοση του αλγορίθμου, υποθέτουμε και πάλι λόγω της memory bound φύσης του αλγορίθμου.

Shared Memory Version

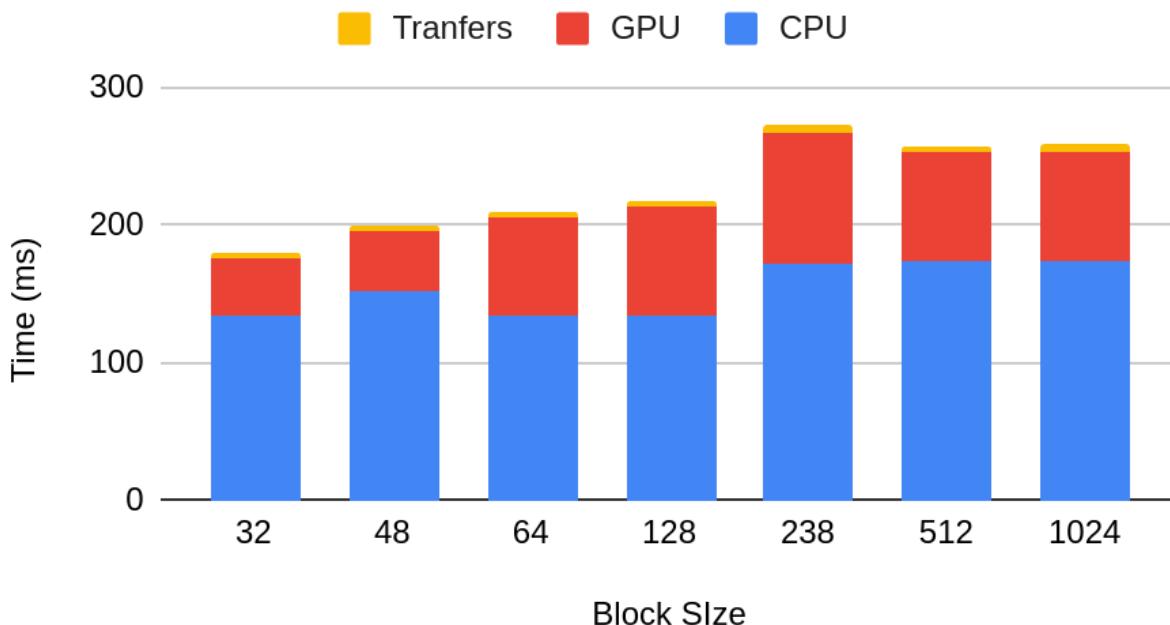
Τι είναι η Shared Memory

Η **shared memory** στις NVIDIA GPUs είναι μια μικρή, γρήγορη μνήμη που μοιράζονται όλα τα threads μέσα σε ένα block, και η οποία είναι πλήρως διαχειρίσιμη από τον προγραμματιστή, σε σχέση με την L1 cache, με την οποία έχει αρκετές ομοιότητες. Η ιδέα της verison αυτής, είναι να βάλουμε στην shared Memory τον πίνακα Device clusters, ο οποίος χρησιμοποιείται από όλα τα threads στο μπλοκ- είναι κοινός και read-only. Βέβαια, αξίζει να σημειώσουμε ότι υπάρχει και η cache-L1 η οποία είναι μαζί με την shared memory, άρα και να μην την δηλώσουμε explicit, αυτόματα cach-άρεται ο πίνακας στην L1, αυτό γίνεται στην transposed. Εδώ όμως που το κάνουμε explicitly, περιμένουμε να έχουμε καλύτερη συμπεριφορά.

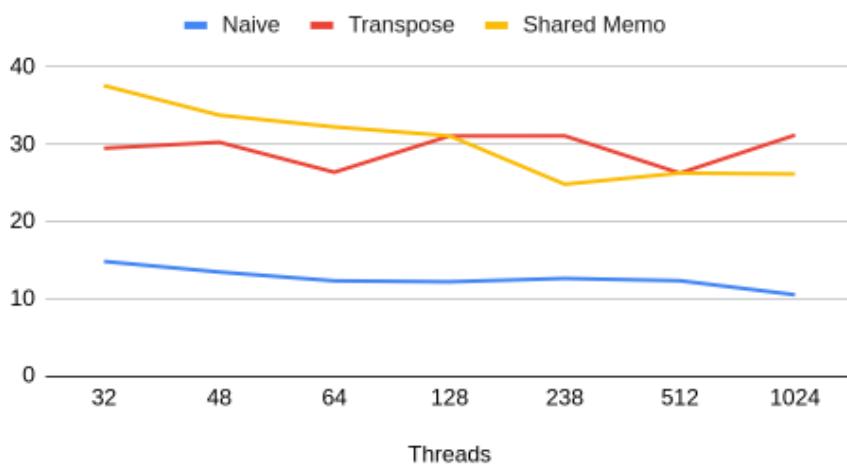
Εκτέλεση

Το κομβικό κομμάτι είναι να γραφτεί στην αρχή του kernel η μεταφορά από του πίνακα Device-Clusters στην shared memory. Αυτό μπορεί είτε να γίνει από ένα thread του κάθε μπλοκ ή να μοιραστεί σε όλα τα threads του μπλοκ, εμείς επιλέξαμε να γράφεται από όλα τα threads παράλληλα, αν και μερικές φορές η εγγραφή από ένα μόνο thread μπορεί να είναι ταχύτερη, αφού για μικρούς πίνακες πιθανότατα τα bank conflicts που προκύπτουν πάνω στη shared memory εισάγουν overhead, το οποίο η αντιγραφή από ένα μόνο νήμα μπορεί να υπερκεράσει, και να είναι άρα πιο γρήγορη. Έτσι καταλήξαμε στα παρακάτω νούμερα:

Shared Memo



Speedup



CPU & Transfer Time: Παρατηρούμε πώς ο χρόνος της CPU κυμαίνεται στα ίδια νούμερα σε σχέση με την transpose υλοποίηση, ενώ επίσης βλέπουμε ότι όπως και στη transpose υλοποίηση, η CPU αποτελεί το bottleneck της υλοποίησης, και άρα σκεφτόμαστε πως σε επόμενα στάδια, κομμάτι της CPU πρέπει να ανατεθεί στη GPU προκειμένου να αυξήσουμε την απόδοσή μας.

GPU Time: Εδώ, όπως αναφέραμε, χρησιμοποιούμε τον πίνακα shmemClusters, ο οποίος είναι ξεχωριστός για κάθε thread block, προκειμένου να διαβάζουμε τα κέντρα από μια πιο γρήγορη on chip shared μνήμη μεταξύ των threads του block, άρα να επιταχυνθεί το kernel μας. Ωστόσο

αυτό οδηγεί στην ανάγκη συγχρονισμού, μεταξύ των νημάτων του block(με το `__syncthreads()`). Επομένως τώρα το block size θα ορίζει πόσα νήματα ταυτόχρονα θα βλέπουν τον shmemClusters (άρα πόσα νήματα θα πρέπει να συγχρονιστούν για την αρχικοποίηση). Βλέπουμε ότι τον καλύτερο speedup έχουμε για 32 νήματα. Αυτό μπορεί να συμβαίνει για τους εξής λόγους: αρχικά για 32 νήματα το block size ταυτίζεται με το warp size, διασφαλίζοντας πλήρη αξιοποίηση του warp χωρίς ανενεργά νήματα, ενώ για μεγαλύτερο αριθμό μπλοκ μπορεί τα warps να μη μπορούν να χρησιμοποιηθούν πλήρως λόγω περιορισμών πόρων (π.χ. Όχι καλός διαμοιρασμός των καταχωρητών στα threads του block), μειώνοντας το occupancy. Επίσης, μικρότερο μπλοκ σημαίνει λιγότερα νήματα που ανταγωνίζονται για την κοινή μνήμη, αποφεύγοντας τα bank conflicts. Τέλος, το block size 32 επιτρέπει περισσότερα blocks/SM, μεγιστοποιώντας την χρήση του SM.

Επιπλέον σημειώνουμε ότι όσον αφορά τις transpose & shared versions, τα speedups είναι αρκετά παραπλήσια, αφού η μία χρησιμοποιεί τη cache ενώ η άλλη τη shared memory, προκειμένου να διαβάζουν πιο γρήγορα τα δεδομένα που παίρνουν από τη RAM.

Σύγκριση υλοποίησεων /bottleneck Analysis

Όσον αφορά την παίνε υλοποίηση, παρατηρούμε ότι το bottleneck της υλοποίησης συνιστά το κομμάτι του υπολογισμού από την GPU. Πιθανολογούμε πως αυτό οφείλεται στο γεγονός πως δεν έχουμε coalesced accesses στη μνήμη προκειμένου να εκμεταλλευτεί ένα single transaction όλο το warp σε ένα κύκλο εντολής. Ακολουθεί το κομμάτι υπολογισμού της CPU, και τελευταίο είναι ο χρόνος μεταφοράς δεδομένων μεταξύ των CPU-GPU(ο οποίος επηρεάζεται από το μέγεθος των πινάκων που αντιγράφονται).

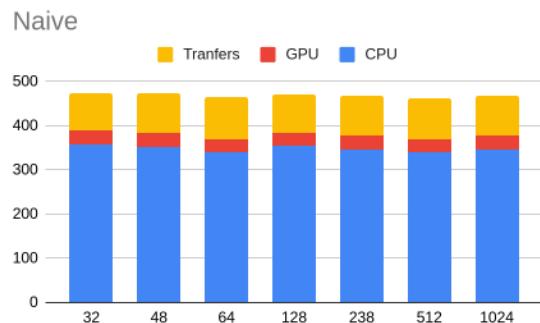
Όσον αφορά την transpose υλοποίηση, παρατηρούμε πως το bottleneck είναι ο χρόνος που καταναλώνει η CPU, ενώ βλέπουμε πως η GPU τα πάει αισθητά καλύτερα. Αρχίζουμε λοιπόν να υποψιαζόμαστε ότι θα πρέπει να σκεφτούμε τρόπους για να παραλληλοποιήσουμε κομμάτι της CPU και να το αναθέσουμε στη GPU. Ο δεύτερος υψηλότερος χρόνος είναι αυτός της GPU, ενώ ακολουθούν τα transactions GPU-CPU.

Τέλος, όσον αφορά την shared version, και πάλι παρατηρούμε συμπεριφορά παρόμοια με την transpose, με το κομμάτι της CPU να καταναλώνει το μεγαλύτερο μέρος του συνολικού χρόνου. Ακολουθούν ο χρόνος της GPU, και τελευταία και λιγότερο χρονοβόρα είναι και πάλι τα transactions.

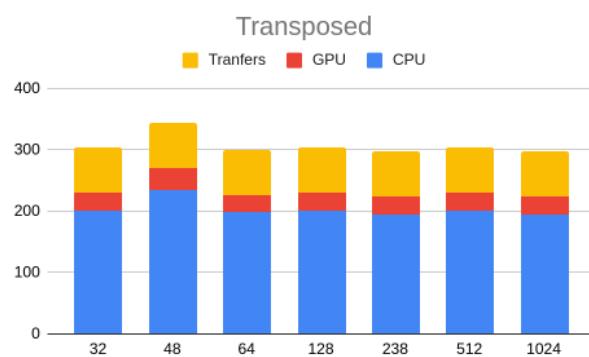
Θεωρούμε ότι ο λόγος για τον οποίο ο αλγόριθμός μας υποφέρει στις 2 τελευταίες υλοποιήσεις, όπου έχουμε αντιστρέψει τις διαστάσεις των πινάκων, έιναι επειδή μπορεί αυτό να διευκολύνει μεν την GPU, αλλά καταστρέψει το locality των caches της CPU, η οποία αποτελεί και το bottleneck.

2. Διαγράμματα για το configuration {Size, Coords, Clusters, Loops} = {1024, 2, 64, 10}:

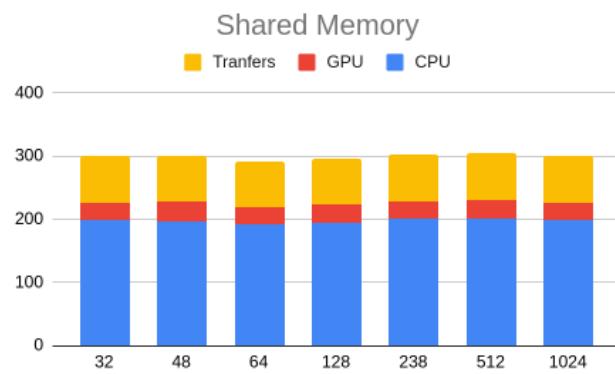
Naive:



Transposed:



Shared Memory:



Βλέπουμε εδώ πολύ διαφορετική συμπεριφορά όσον αφορά τους χρόνους των transactions και για τις 3 υλοποιήσεις(έχει γίνει σε όλες η δευτερη πιο χρονοβόρα διαδικασία). Γενικά μεταφέρονται οι πίνακες clusters και memberships, καθώς και το delta. Το clusters h allagh poly mikroterh ενώ το delta είναι μόνο ένα στοιχείο άρα εστιάζουμε στο memberships με μέγεθος numObjs.

Η αλλαγή συνέβη στο numCoords, που από 32 πήγε σε 2. Ανατρέχοντας στο main_gpu.cu βλέπουμε ότι

```
numObjs = (dataset_size*1024*1024) / (numCoords*sizeof(double));
```

Επομένως ο αριθμός των συνολικών objects αυξάνεται 16 φορές, και αντίστοιχα ο αριθμός των δεδομένων που πρέπει να αντιγραφούν από τη CPU στη GPU και πίσω για τον πίνακα memberships και άρα μπορούμε να παρατηρήσουμε από τους χρόνους ότι και οι μεταφορές έχουν στο περίπου δεκαεξαπλασιαστεί.

Όσον αφορά την CPU, ο χρόνος της αυξάνεται και στις 3 περιπτώσεις, όπως συμβαίνει και με το sequential. Xrhsimopoiei to membership pou einai megaltero sto deytero config ara pio polles praksei sara pio argh

Για τη GPU, παρατηρούμε ότι ο χρόνος που καταναλώνει μειώνεται σε μικρό βαθμό και θεωρούμε ότι αυτό συμβαίνει επειδή πλέον κάθε νήμα έχει μόνο να λάβει υπόψιν του μόνο τις 2 διαστάσεις του αντικειμένου που έχει αναλάβει. Παρόλο που ο αριθμός των numObjs γίνεται 16 φορές μεγαλύτερος, ο αριθμός των thread blocks επιλέγεται έτσι ώστε η δουλειά να παραληλοποιηθεί (και να έχουμε και την ανάλογη αύξηση του αριθμού των νημάτων που θα τρέξουν σε όλο το SM), άρα αξιοποιούμε καλύτερα τα SMs ενώ το oversubscription μας βοηθά να καλύψουμε τις ενδεχόμενες καθυστερήσεις των προσβάσεων στη μνήμη.

Η παρούσα shared uλοποίηση δεν είναι και η καλύτερη δυνατή, αφού ξέρουμε ότι έχουμε συγκεκριμένα 48KB διαμοιραζόμενης μνήμης για κάθε μπλοκ, επομένως σε configurations αυθαίρετα δεν ξέρουμε αν μπορεί να ικανοποιήσει τις ανάγκες μας αυτό το μέγεθος shared μνήμης, Ακόμη, θεωρούμε πως ένα κομμάτι του υπολογισμού thw CPU μπορεί να παραληλοποιηθεί και να ανατεθεί στην GPU, ενώ επίσης ο υπολογισμός του delta μπορεί να γίνει με reduction, αντί για atomicAdd.

3. Χρησιμοποιήσαμε το cudaOccupancyMaxPotentialBlockSize, και για τις 3 uλοποιήσεις και ο αριθμός για το βέλτιστο block size προκειμένου να βελτιστοποιηθεί το occupancy είναι 1024.

Θυμίζουμε ότι το occupancy ισούται με τον αριθμό των active warps προς τον συνολικό αριθμό των warps ενός SM.

Ξέρουμε ότι:

Table 2. Compute Capabilities: GK180 vs GM200 vs GP100 vs GV100

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255 ¹
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

¹ The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.

<https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

Άρα για block size = 1024 threads/block:

warps/block = (block size)/(warp size) = 1024/32 = 32 warps/block

Active warps/SM = 32 warps per block * 32 blocks per SM = 1024 warps per SM

Occupancy = Active warps per SM / Max warps per SM * 100% = 1024/1024 * 100% = 100%,

Άρα μεγιστοποιείται το theoretical occupancy.

Ωστόσο υπάρχουν και άλλοι παράγοντες οι οποίοι επηρεάζουν την μεγιστοποίηση του πραγματικού occupancy, όπως πχ ο αριθμός των registers που ανατίθενται σε κάθε thread και το μέγεθος της shared memory που ανατίθενται σε κάθε block.

Full-Offload (All-GPU) version

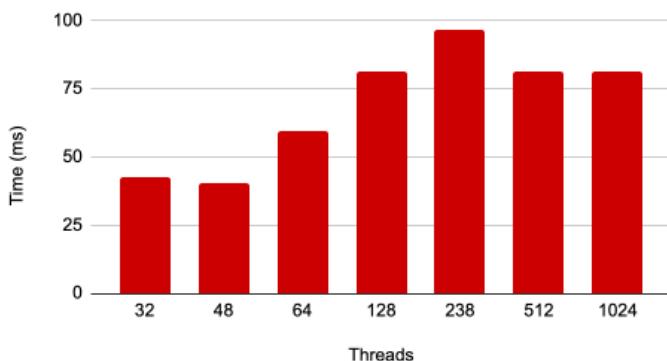
Παρατηρούμε πως πλέον ο χρόνος έχει μειωθεί και το speedup έχει αυξηθεί αρκετά σε σχέση με τις προηγούμενες 3 υλοποιήσεις, καθώς πλέον ο υπολογισμός των κεντρών(που γινόταν στην CPU σειριακά) πλέον υπολογίζεται παράλληλα από τα νήμα στα 80 cores της GPU μας.

Προκειμένου να μειώσουμε περαιτέρω τον χρόνο εκτέλεσης, πρέπει να μειώσουμε κι άλλο τον χρόνο εκτέλεσης στην CPU και αντ' αυτού να αξιοποιήσουμε τις δυνατότητες της GPU.

Η ιδέα ήταν η εξής: κρατήσαμε γενικά την `find_nearest_cluster` όπως ήταν στην GPU, ενώ στο τέλος της προσθέσαμε το ακόλουθο βήμα: κάθε νήμα θα αντιστοιχίζει το σωστό membership(μέσω του index, το οποίο αναφέρεται ένα συγκεκριμένο cluster) στο `objectId` που έχει αναλάβει, αυξάνει ατομικά το μέγεθος του συγκεκριμένου cluster(και πάλι μέσω του index), και τέλος κάνει `iterate` πάνω στις διαστάσεις και προσθέτει ατομικά τις διαστάσεις του `objectId` που έχει αναλάβει στις αντίστοιχες διαστάσεις του cluster στο οποίο ανήκει προκειμένου να υπολογιστεί σε επόμενο στάδιο ο μέσος όρος. Θα μπορούσαμε σε αυτό το σημείο να βάλουμε ένα `if branch` στο οποίο να μπαίνει μόνο ένα νήμα για τον υπολογισμό του μέσου όρου μέσα στην `find_nearest_cluster`, αλλά το απόφυγαμε αφού μπορούμε να το κάνουμε σε επόμενο βήμα και να αποφύγουμε το divergence στο οποίο θα οδηγούνταν νήματα του ίδιου warp.

Συμπληρώσαμε και τον κώδικα του `update_centroids`, όπου ουσιαστικά κάθε νήμα(το οποίο πλέον δεν αναλαμβάνει ένα `objectId` αλλά μια διάσταση κάθε ενός `clusterId`, δηλαδή στον πίνακα `deviceClusters[numCoords][numClusters]` κάθε νήμα πειράζει μόνο ένα `element`), σύμφωνα με το νέο `launch configuration`. Έχουμε επομένως τη μέγιστη παραλληλοποίηση του αλγορίθμου, αφού κάθε νήμα αναλαμβάνει μόνο ένα κουτάκι του πίνακα, και υπολογίζει τον μέσο όρο για τη συγκεκριμένη διάσταση του συγκεκριμένου cluster(το `clusterId` βρίσκεται με πράξη % με το `numClusters`). Άρα πλέον εξαλειφουμε το bottleneck της CPU η οποία υπολόγιζε σειριακά τα νέα κέντρα, κάθως τώρα ο υπολογισμός αυτός παραλληλοποιείται. Επιπλέον, Γλιτώνουμε και όλον τον χρόνο μεταφοράς δεδομένων από τη CPU στη GPU και ανάποδα, αφού πλέον όλα τα δεδομένα μας βρίσκονται στη GPU.

All GPU Coord-32 -GPU Time



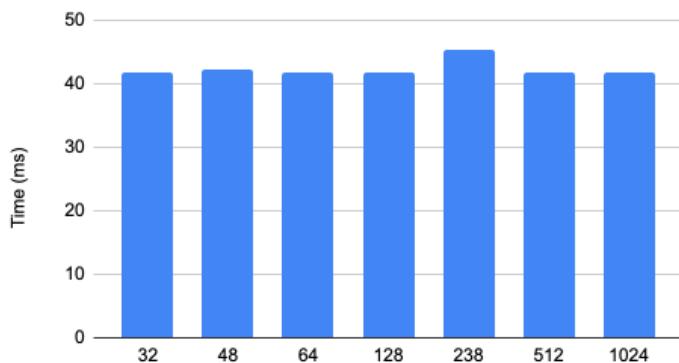
Παρατηρούμε πως όσο ο αριθμός των threads μέσα στα blocks αυξάνεται έχουμε μικρή αύξηση και στον χρόνο, ιδίως για μεγάλα block sizes(512, 1024 threads per blocks) που πιθανότατα οφείλεται στην ανάγκη συγχρονισμού των νημάτων πάνω στην shared

memory του πρώτου kernel μέσα στα blocks.

Για το πρώτο configuration με 32 coords έχουμε ότι για το kernel update_centroids θα παραχθούν blocks με μέγεθος το πολύ $32*64 = 2048$ threads, ενώ για το δεύτερο με 2 coords, blocks με $2*64 = 128$ threads. Ωστόσο θεωρούμε πως οι προσβάσεις στη μνήμη είναι αυτές που καθορίζουν και τη συμπεριφορά του αλγορίθμου ως προς το χρόνο, και όχι τόσο το size.

Για μικρά block sizes, το πρώτο config με numCoords=32 είναι γρηγορότερο σε σχέση με το config με numCoords=2, ενώ τα κομμάτια των μεταφορών και της CPU, τα οποία στη shared version κυριαρχούσαν και έπιαναν μεγάλο μέρος του συνολικού χρόνου σε όλα τα block sizes τώρα εκμηδενίζονται.

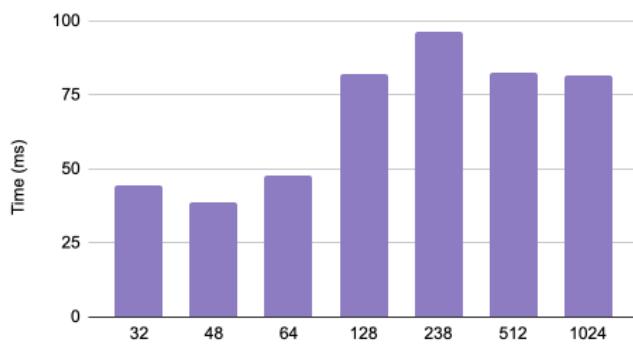
All GPU Coord-2 -GPU Time



```
const unsigned int update_centroids_block_sz = (numCoords * numClusters > blockSize) ? blockSize : numCoords * numClusters;
```

Bonus 2: Delta reduction (All-GPU) version

All GPU- Reduction Coord-32 -GPU Time

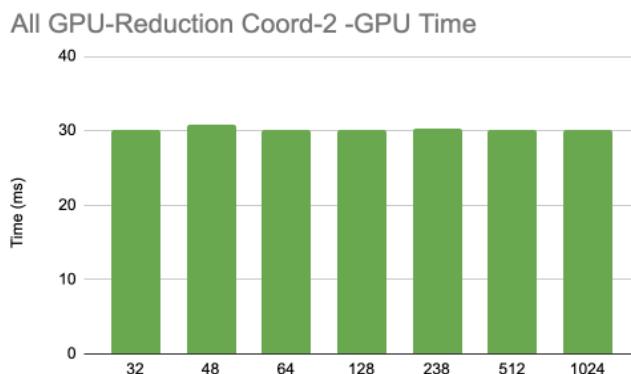


Στο σημείο αυτό αντικαθιστούμε την atomicAdd με reduction πάνω στην μεταβλητή delta, προκειμένου να αποφύγουμε το overhead που εισάγεται για thread-safe write πάνω στο delta με χρήση ατομικής εντολής. Χρησιμοποιήσαμε reduction ανάμεσα στα thread blocks στην κύρια μνήμη. Χρησιμοποιείται βέβαια και εδώ η ατομική εντολή, αλλά όχι από όλα

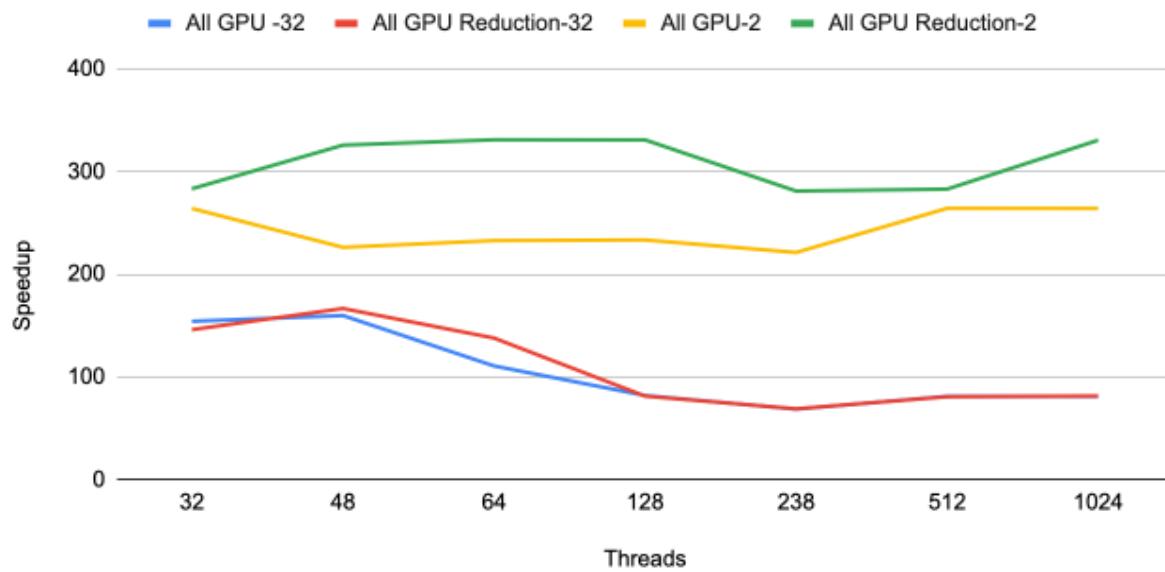
τα νήματα όλων των blocks πάνω στην κοινή μεταβλητή delta, αλλά μονάχα από τα πρώτα νήματα κάθε thread block, τα οποία επιφορτίζονται με τον ρόλο να ενημερώσουν σωστά το shared delta στο τέλος του iteration, και έτσι περιμένουμε να δούμε καλύτερη επίδοση, αφού λιγότερα νήματα ανταγωνίζονται για να γράψουν πάνω στην κοινή μεταβλητή. Βλέπουμε ότι η αποφυγή αυτού του overhead όντως μας δίνει μικρότερους χρόνους σε σχέση με το

προηγούμενο ερώτημα(για για μικρά block sizes. Από την άλλη για μεγαλύτερα μεγέθη, βλέπουμε ότι ο χρόνος παρουσιάζει μια μικρή αύξηση σε σχέση με το προηγούμενο ερώτημα, και αυτό μπορεί να συμβαίνει λόγω του overhead που εισάγεται λόγω της ανάγκης συγχρονισμού του μεγάλου αριθμού από threads μέσα σε κάθε block στο τέλος κάθε iteration του while loop με το οποίο υπολογίζεται το delta κάθε block.

Συνολικά Αποτελέσματα



All GPU & All GPU Reduction for Coord- 32 & 2



Άσκηση 5

Παραλληλοποίηση και βελτιστοποίηση αλγορίθμων σε αρχιτεκτονικές κατανεμημένης μνήμης

K-means

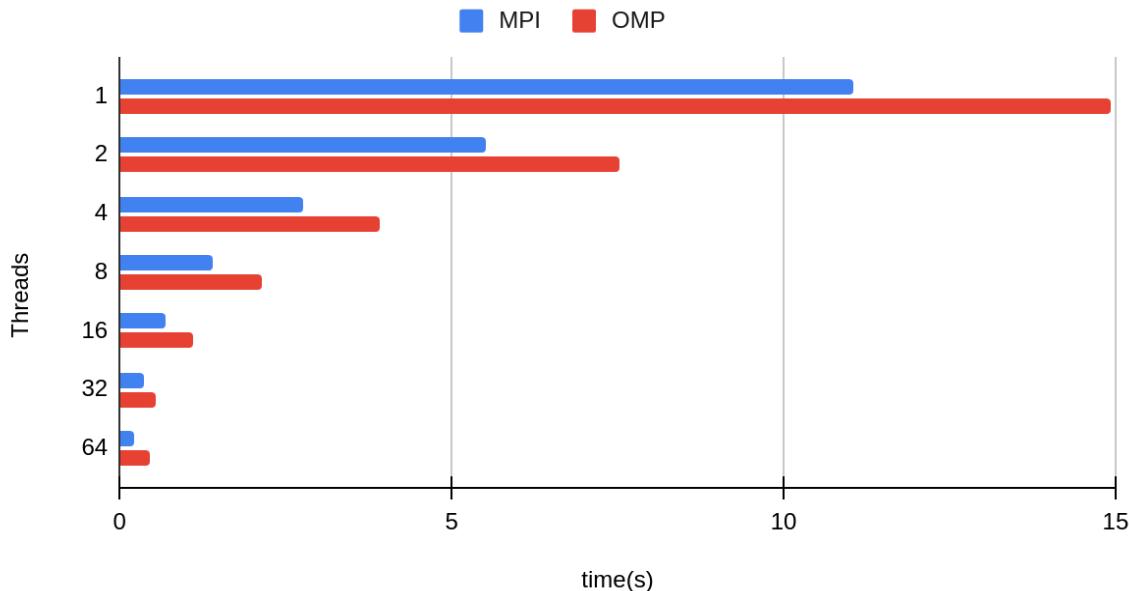
Για την υλοποίηση του k-means:

- Συμπληρώσαμε στο file_io.c την συνάρτηση dataset_generation, η οποία φτιάχνει το συνολικό σετ δεδομένων και το μοιράζει στις διεργασίες. Για τον διαμοιρασμό επιλέξαμε την Scaterv, καθώς μας προσφέρει ευελιξία στο πλήθος των δεδομένων που στέλνουμε σε κάθε διεργασία.
- Η συνάρτηση kmeans δουλεύει, όπως και στο k-means με reduction, η κάθε διεργασία αναλαμβάνει ένα κομμάτι των Objects. Σε κάθε iteration υπολογίζον το membership των τοπικών Object και όλες μαζί “συνθέτουν” με reduction τα νέα clusters & delta
- Τέλος η main.c συνθέτει τους πίνακες των membership (τελικό output) των διεργασιών με gatherv.

Συνολικά, αναμένουμε πολύ καλή απόδοση με τη μέθοδο MPI, καθώς η επικοινωνία μεταξύ διεργασιών ελαχιστοποιείται κατά την εκτέλεση των iterations. Πλέον ο πίνακας objects χωρίζεται σε κομμάτια, καθένα από τα οποία βρίσκεται στη μνήμη του node που τα χρησιμοποιεί για να κατατάξει κάθε σημείο στο σωστό κέντρο(κατανεμημένη μνήμη). Επιπλέον, το reduction υλοποιείται με δενδρική δομή, σε αντίθεση με το OpenMP, όπου το master thread αναλάμβανε τη σύνθεση του shared array από τους local και στη συνέχεια τον υπολογισμό του τελικού πίνακα clusters. Το μόνο μειονέκτημα είναι ότι όλες οι διεργασίες υπολογίζουν τους πίνακες clusters με τα νέα δεδομένα (newClusters/newClustersSize), προσθέτοντας επιπλέον κόστος, αφού πλέον ο πίνακας clusters δεν είναι διαμοιραζόμενος, αλλά αντίγραφό του υπάρχει στη μνήμη κάθε node.

Σε κάθε περίπτωση, βλέπουμε γραμμική επιτάχυνση του προγράμματός μας, πράγμα που συνάδει με τη κλιμακωσιμότητα η οποία γνωρίζουμε πως διακρίνει το MPI σε συστήματα κατανεμημένης μνήμης.

K-menas MPI Vs OMP



Παρατηρούμε πως το MPI μας δίνει μικρότερους χρόνους αλλά και καλύτερο(πιο γραμμικό) speedup σε σχέση με την παράλληλη υλοποίηση σε OpenMP. Θεωρούμε πως αυτό οφείλεται στο γεγονός ότι πως για το OpenMP, ο συγχρονισμός των νημάτων πάνω στη shared memory εισάγει ένα overhead, το οποίο για μεγάλο αριθμό από σημεία στον πίνακα objects γίνεται αρκετά σημαντικό. Έτσι υποθέτουμε πως η συμφόρηση στο memory bus και το cache coherence protocol ξεπερνούν το overhead που δίνει η επικοινωνία μέσω του δικτύου διασύνδεσης. Επιπλέον γνωρίζουμε ότι το MPI είναι ιδανικό για μεγάλα datasets, αφού μειώνει το contention στη μνήμη και στις CPU resources, ωστόσο θεωρούμε πως σε μικρότερα datasets θα εισήγαγε αχρείαστα overheads τα οποία με χρήση του openMP θα εξαλείφονταν.

Διάδοση Θερμότητας σε 2 διαστάσεις

Μέθοδος Jacobi

Για την υλοποίηση έχουμε να συμπληρώσουμε στον σκελετό 7 βήματα:

- Διαμοιρασμό του πίνακα σε μικρότερους υποπίνακες - blocks- στις διεργασίες (ranks)

```
//*****[TODO]*****//  
  
/*Fill your code here*/  
MPI_Scatterv( &(U[0][0]), scattercounts, scatteroffset,  
global_block, & (u_previous[1][1]),  
1, local_block, 0, MPI_COMM_WORLD);  
  
for (i = 1; i <= local[0]; i++) {  
    for (j = 1; j <= local[1]; j++) {  
        u_current[i][j] = u_previous[i][j];  
    }  
}
```

- Οι πίνακες u_previous και u_current αποθηκεύουν τιμές σε δύο διαδοχικές στιγμές. Για να διευκολυνθεί η ανταλλαγή δεδομένων μεταξύ διεργασιών, οι πίνακες έχουν επιπλέον περιθώριο (halo) στις άκρες. Σε κάθε επαναληπτικό βήμα, οι διεργασίες ανταλλάσσουν τις ακραίες τιμές των πινάκων με τους γειτονικούς υπολογιστές, ώστε να εξασφαλίζεται η σωστή ενημέρωση των οριακών σημείων (halo communication).

- Ορισμός MPI datatypes
row/column type,
εξασφαλίζουν εύκολη
επικοινωνία μεταξύ
γειτονικών διεργασιών.

```
MPI_Datatype row_type;  
MPI_Type_contiguous(local[1], MPI_DOUBLE, & row_type);  
MPI_Type_commit( & row_type);  
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);  
  
//*****[TODO]*****//  
  
//----Find the 4 neighbors with which a process exchanges messages----//  
  
//*****[TODO]*****//  
int north, south, east, west;  
  
MPI_Cart_shift(CART_COMM, 0, 1, & north, & south); // 0 -> axons y +-1 neighbour  
MPI_Cart_shift(CART_COMM, 1, 1, & west, & east); // 1 -> axons x +-1 neighbour
```

- Εύρεση των στοιχείων του κάθε μπλοκ που θα γίνει επεξεργασία. Σε αυτό το σημείο πρέπει να λάβουμε υπόψη τόσο τα ghost cells όσο και τα όρια του πίνακα τα οποία κατα dirichlet συνθήκη μένουν αναλλοίωτα στην επεξεργασία -i/j_min/max.
- Επικοινωνία κάθε διεργασίας με τις γειτονικές διεργασίες, ώστε να ανταλλάξει τα ghost cells, έπειτα εφαρμογή του βήματος jacobi και έλεγχος σύγκλισης. Η επικοινωνία γίνεται στην αρχή (*u_previous*), πριν τον υπολογισμό των νέων τιμών. Έπειτα υπολογίζονται οι νέες τιμές μέσω της συνάρτησης και ο έλεγχος της σύγκλισης.
- Σύνθεση του τελικού πίνακα εξόδου από τις επιμέρους διεργασίες με GatherV και του χρόνου με Reduction.

```

//*****[TODO]*****//
swap = u_previous;
u_previous = u_current;
u_current = swap;

MPI_Sendrecv( & u_previous[1][1], 1, row_type, north, 0, &
    u_previous[0][1], 1, row_type, north, 0,
    CART_COMM, MPI_STATUS_IGNORE);
MPI_Sendrecv( & u_previous[local[0]][1], 1, row_type, south, 0, &
    u_previous[local[0] + 1][1], 1, row_type, south, 0,
    CART_COMM, MPI_STATUS_IGNORE);
MPI_Sendrecv( & u_previous[1][local[1]], 1, column, east, 0, &
    u_previous[1][local[1] + 1], 1, column, east, 0,
    CART_COMM, MPI_STATUS_IGNORE);
MPI_Sendrecv( & u_previous[1][1], 1, column, west, 0, &
    u_previous[1][0], 1, column, west, 0,
    CART_COMM, MPI_STATUS_IGNORE);

gettimeofday( & tcs, NULL);

for (i = i_min; i <= i_max; i++) {
    for (j = j_min; j <= j_max; j++) {
        u_current[i][j] = (u_previous[i - 1][j] + u_previous[i + 1][j] +
            u_previous[i][j - 1] + u_previous[i][j + 1]) / 4.0;
    }
}

/*Add appropriate timers for computation*/

gettimeofday( & tcf, NULL);

tcomp += (tcf.tv_sec - tcs.tv_sec) + (tcf.tv_usec - tcs.tv_usec) * 0.000001;

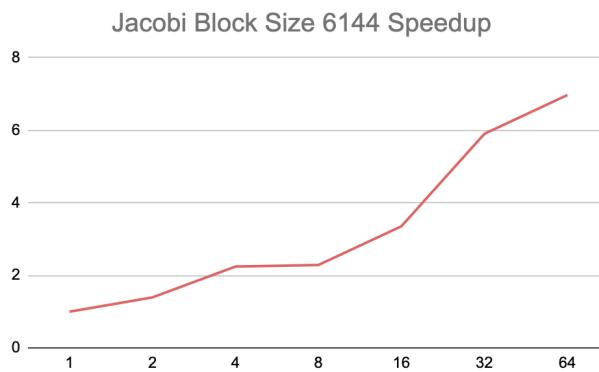
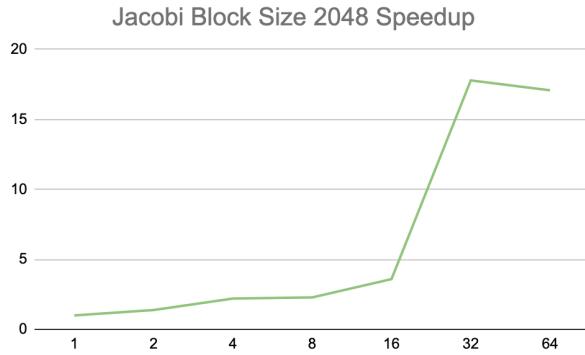
#ifdef TEST_CONV
if (t % C == 0) {
    int converged;
    converged = converge(u_previous, u_current, i_min, i_max, j_min, j_max);
//printf("Process %d: Before MPI_Allreduce, converged = %d\n", rank, converged);
//fflush(stdout);
    MPI_Allreduce( &converged, &global_converged, 1, MPI_INT, MPI_LAND, CART_COMM);
}
#endif

```

Αποτελέσματα Μεθόδου Jacobi

Αρχικά βλέπουμε ότι για τον πίνακα 512*512 για να επιτύχουμε σύγκλιση χρειάζονται **236001** iterations και χρόνος **58,38sec** για πλέγμα **8*8**, άρα έχουμε 0,2473ms ανα επανάληψη.

Για σταθερό αριθμό επαναλήψεων 256 έχουμε για το Speedup και τον χρόνο εκτέλεσης:



Συμπεράσματα

Το πρόγραμμα απαιτεί μεγάλο αριθμό επαναλήψεων για να συγκλίνει, αλλά το κόστος κάθε επανάληψης είναι χαμηλό. Αυτό οφείλεται στην άμεση εκτέλεση της επικοινωνίας πριν από τον υπολογισμό, επιτρέποντας σε κάθε rank να λειτουργεί ανεξάρτητα από τα υπόλοιπα, μόλις ολοκληρωθεί η αρχική ανταλλαγή halo δεδομένων.

Για μεγάλα block sizes, όπου κυριαρχεί το υπολογιστικό φορτίο, η απόδοση κλιμακώνεται γραμμικά. Αντίθετα, για μικρότερα block sizes, η επικοινωνία μεταξύ διεργασιών γίνεται το κυρίαρχο κόστος, προκαλώντας συμφόρηση στον δίαυλο μνήμης και περιορίζοντας την κλιμάκωση του προγράμματος.

Τέλος, αξίζει να παρατηρήσουμε τη γραμμική σχέση του χρόνου εκτέλεσης με το μέγεθος του πίνακα στη σειριακή εκτέλεση. Για πίνακα 2048×2048 , ο χρόνος εκτέλεσης είναι X, ενώ για 4096×4096 , παρατηρούμε περίπου τετραπλάσιο χρόνο (4X). Αντίθετα, για 6144×6144 , ο χρόνος αυξάνεται περίπου 1,5 φορές σε σχέση με τον 4096×4096 , φτάνοντας τα 6X.

Μέθοδος Gauss-Seidel-SOR

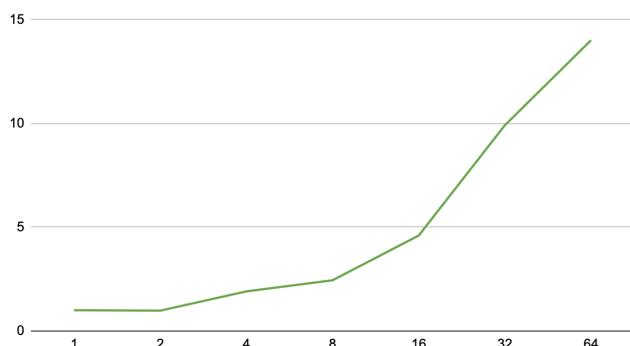
Για την υλοποίηση, ακολουθήσαμε τη δομή του Jacobi όσον αφορά τον σκελετό και τη διαχείριση της μνήμης σε blocks, αλλά τροποποιήσαμε τη λογική της επικοινωνίας και των υπολογισμών. Κάθε διεργασία στέλνει, όπως πριν, τα ακραία στοιχεία της (halo cells) στον βόρειο και δυτικό γείτονα και λαμβάνει από τον νότιο και ανατολικό. Αφού λάβει τα δεδομένα από τον βόρειο και δυτικό γείτονα, υπολογίζει τις νέες τιμές του πίνακα για την τρέχουσα χρονική στιγμή και στη συνέχεια αποστέλλει τις ενημερωμένες τιμές στον νότιο και ανατολικό γείτονα, επιτρέποντας στους γείτονες να ξεκινήσουν τον υπολογισμό με τα σωστά u_{current} .

Αυτή η ανάγκη κάθε διεργασίας να περιμένει δεδομένα από δύο γειτονικές διεργασίες, οι οποίες πρέπει πρώτα να ολοκληρώσουν τους υπολογισμούς τους, περιμένουμε να εισάγει καθυστέρηση και να περιορίσει την κλιμάκωση του προγράμματος.

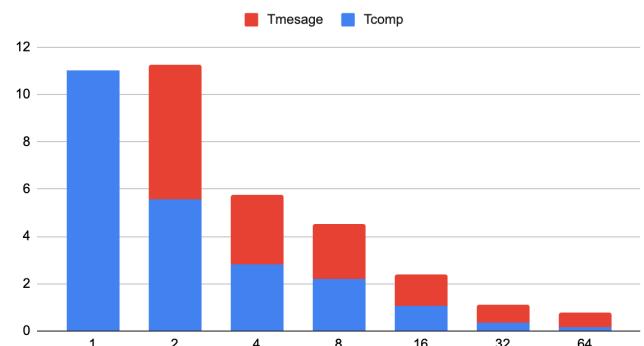
Παρόλα αυτά, το πρόγραμμα επιτυγχάνει σύγκλιση σε χρόνο **0.482838** με **1501** επαναλήψεις. Παρότι ο χρόνος ανά επανάληψη αυξήθηκε σε 0.4594ms για πλέγμα 8×8 , το **συνολικό όφελος είναι σημαντικό, καθώς η σύγκλιση επιτυγχάνεται ταχύτερα**.

Για σταθερό αριθμό επαναλήψεων 256 βλέπουμε:

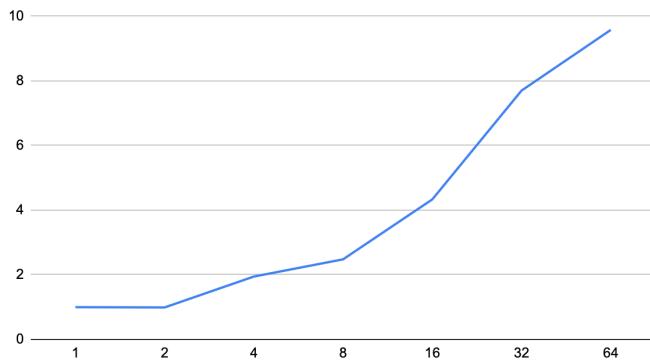
Gauss Block Size 2048 Speedup



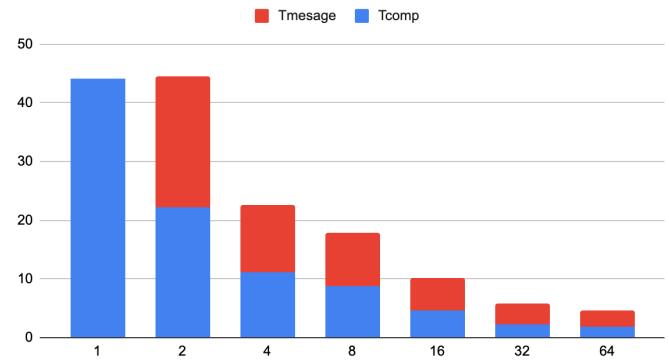
Gauss Block Size 2048

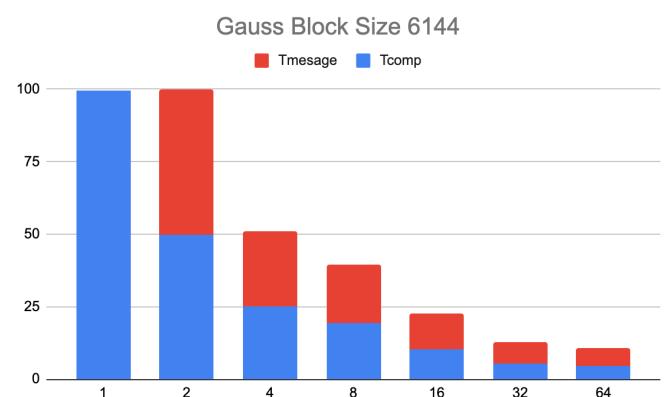
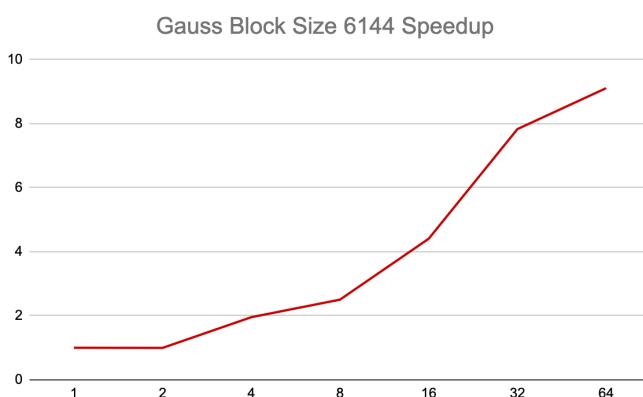


Gauss Block Size 4096 Speedup



Gauss Block Size 4096





Συμπεράσματα

Όπως αναμενόταν, τα διαγράμματα δείχνουν ότι περισσότερο από τον μισό χρόνο δαπανάται στην επικοινωνία. Αυτό οδηγεί σε αδρανή νήματα που περιμένουν να λάβουν τις ενημερωμένες τιμές των ακραίων κελιών από τους γείτονες, περιορίζοντας την αποδοτικότητα.

```
// Send data to north
MPI_Send(&u_previous[1][1], 1, row_type, north, 0, CART_COMM);
// Send data to west
MPI_Send(&u_previous[1][1], 1, column, west, 0, CART_COMM);
// Receive data from south
MPI_Recv(&u_previous[local[0] + 1][1], 1, row_type, south, 0, CART_COMM, MPI_STATUS_IGNORE);
// Receive data from east
MPI_Recv(&u_previous[1][local[1] + 1], 1, column, east, 0, CART_COMM, MPI_STATUS_IGNORE);

// Receive data from north
MPI_Recv(&u_current[0][1], 1, row_type, north, 0, CART_COMM, MPI_STATUS_IGNORE);
// Receive data from west
MPI_Recv(&u_current[1][0], 1, column, west, 0, CART_COMM, MPI_STATUS_IGNORE);

gettimeofday( & tcs, NULL);

for (i = i_min; i <= i_max; i++) {
    for (j = j_min; j <= j_max; j++) {
        u_current[i][j] = u_previous[i][j] + (omega/4.0) * (u_current[i - 1][j] + u_current[i][j-1] +
        u_previous[i+1][j] + u_previous[i][j + 1] - 4.0 * u_previous[i][j]);
    }
}
//send dat to south
MPI_Send(&u_current[local[0]][1], 1, row_type, south, 0, CART_COMM);
// Send data to east
MPI_Send(&u_current[1][local[1]], 1, column, east, 0, CART_COMM);
```

Black & Red

Με αυτή τη μέθοδο, στοχεύουμε να διατηρήσουμε την ταχεία σύγκλιση της μεθόδου Gauss-SOR, επιτρέποντας παράλληλα μεγαλύτερη παραλληλοποίηση και μειώνοντας τον χρόνο αδράνειας των νημάτων. Η επεξεργασία χωρίζεται σε δύο φάσεις – την κόκκινη και τη μαύρη – με δύο ανταλλαγές μηνυμάτων με τους γείτονες: μία πριν από την κόκκινη φάση και μία πριν από τη μαύρη. Έτσι, αποφεύγεται η αδράνεια των νημάτων που περιμένουν νέα halo cells, ενώ παράλληλα ο μισός πίνακας ενημερώνεται με τις πιο πρόσφατες τιμές, επιταχύνοντας τη γρήγορη σύγκλιση.

```
void RedSOR(double ** u_previous, double ** u_current, int X_min, int X_max,
           int Y_min, int Y_max, double omega) {
    int i,j;
    for (i=X_min;i<=X_max;i++)
        for (j=Y_min;j<=Y_max;j++)
            if (((i+j)%2==0))
                u_current[i][j]=u_previous[i][j]+(omega/4.0)*(u_previous[i-1][j]+u_previous[i+1][j]
                    +u_previous[i][j-1]+u_previous[i][j+1]-4*u_previous[i][j]);
}

void BlackSOR(double ** u_previous, double ** u_current, int X_min, int X_max,
              int Y_min, int Y_max, double omega) {
    int i,j;
    for (i=X_min;i<=X_max;i++)
        for (j=Y_min;j<=Y_max;j++)
            if (((i+j)%2==1))
                u_current[i][j]=u_previous[i][j]+(omega/4.0)*(u_current[i-1][j]+u_current[i+1][j]+u_current[i][j-1]
                    +u_current[i][j+1]-4*u_previous[i][j]);
}
```

```
/*Compute and Communicate*/
/*Add appropriate timers for computation*/
//here is the red phase
MPI_Sndrecv(&u_previous[1][1], local[1], MPI_DOUBLE, north, 0, &u_previous[0][1], local[], MPI_DOUBLE, north, 0, MPI_COMM_WORLD, &status );
MPI_Sndrecv(&u_previous[local[0][1]], local[], MPI_DOUBLE, south, 0, &u_previous[local[0]+1][1], local[], MPI_DOUBLE, south, 0, MPI_COMM_WORLD, &status );
MPI_Sndrecv(&u_previous[1][local[1]], 1, col_type, east, 0, &u_previous[1][local[1]+1], 1, col_type, east, 0, MPI_COMM_WORLD, &status );
MPI_Sndrecv(&u_previous[1][1], 1, col_type, west, 0, &u_previous[1][0], 1, col_type, west, 0, MPI_COMM_WORLD, &status );
gettimeofday(&tcs, NULL);

RedSOR(u_previous, u_current, i_min, i_max, j_min, j_max, omega);
gettimeofday(&tcf, NULL);

tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;

MPI_Sndrecv(&u_current[1][1], local[], MPI_DOUBLE, north, 0, &u_current[0][1], local[], MPI_DOUBLE, north, 0, MPI_COMM_WORLD, &status );
MPI_Sndrecv(&u_current[local[0][1]], local[], MPI_DOUBLE, south, 0, &u_current[local[0]+1][1], local[], MPI_DOUBLE, south, 0, MPI_COMM_WORLD, &status );
MPI_Sndrecv(&u_current[1][local[1]], 1, col_type, east, 0, &u_current[1][local[1]+1], 1, col_type, east, 0, MPI_COMM_WORLD, &status );
MPI_Sndrecv(&u_current[1][1], 1, col_type, west, 0, &u_current[1][0], 1, col_type, west, 0, MPI_COMM_WORLD, &status );
gettimeofday(&tcs, NULL);

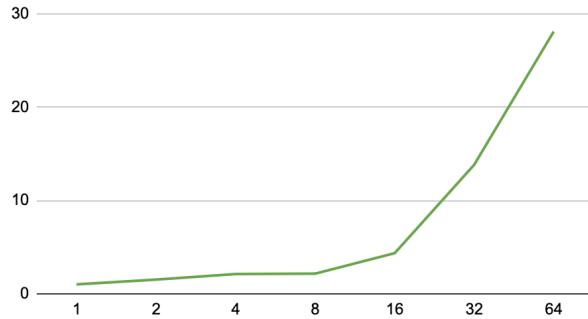
BlackSOR(u_previous, u_current, i_min, i_max, j_min, j_max, omega);
gettimeofday(&tcf, NULL);

tcomp += (tcf.tv_sec-tcs.tv_sec)+(tcf.tv_usec-tcs.tv_usec)*0.000001;
```

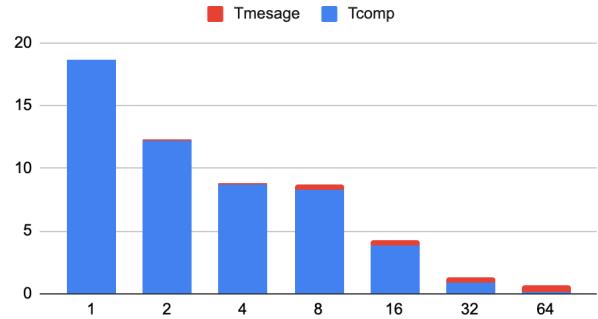
Βλέπουμε ότι με αυτή την υλοποίηση, για πίνακα 512*512 για πλέγμα 8*8 χρειαζόμαστε **1201** επαναλήψεις, με **συνολικό χρόνο 0.798550**, δίνοντας μας χρόνο 0,664ms ανά επανάληψη.

Για σταθερό αριθμό επαναλήψεων 256 βλέπουμε:

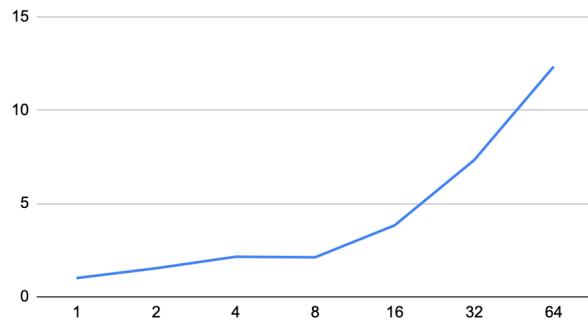
Red & Black Block Size 2048 Speedup



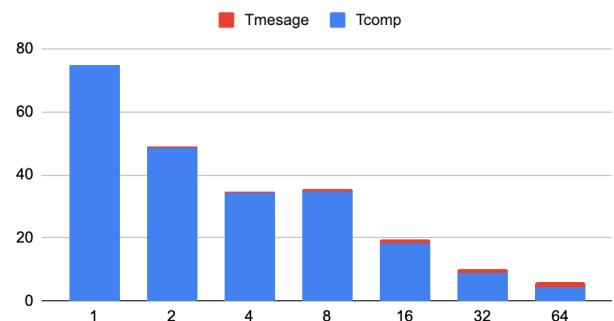
Red & Black Block Size 2048



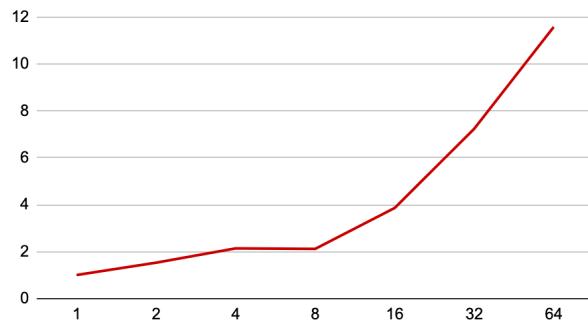
Red & Black Block Size 4096 Speedup



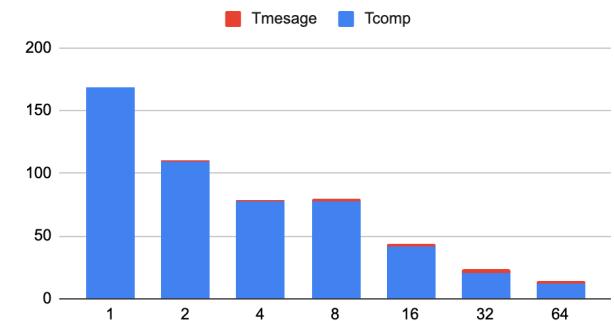
Red & Black Block Size 4096



Red & Black Block Size 6144 Speedup

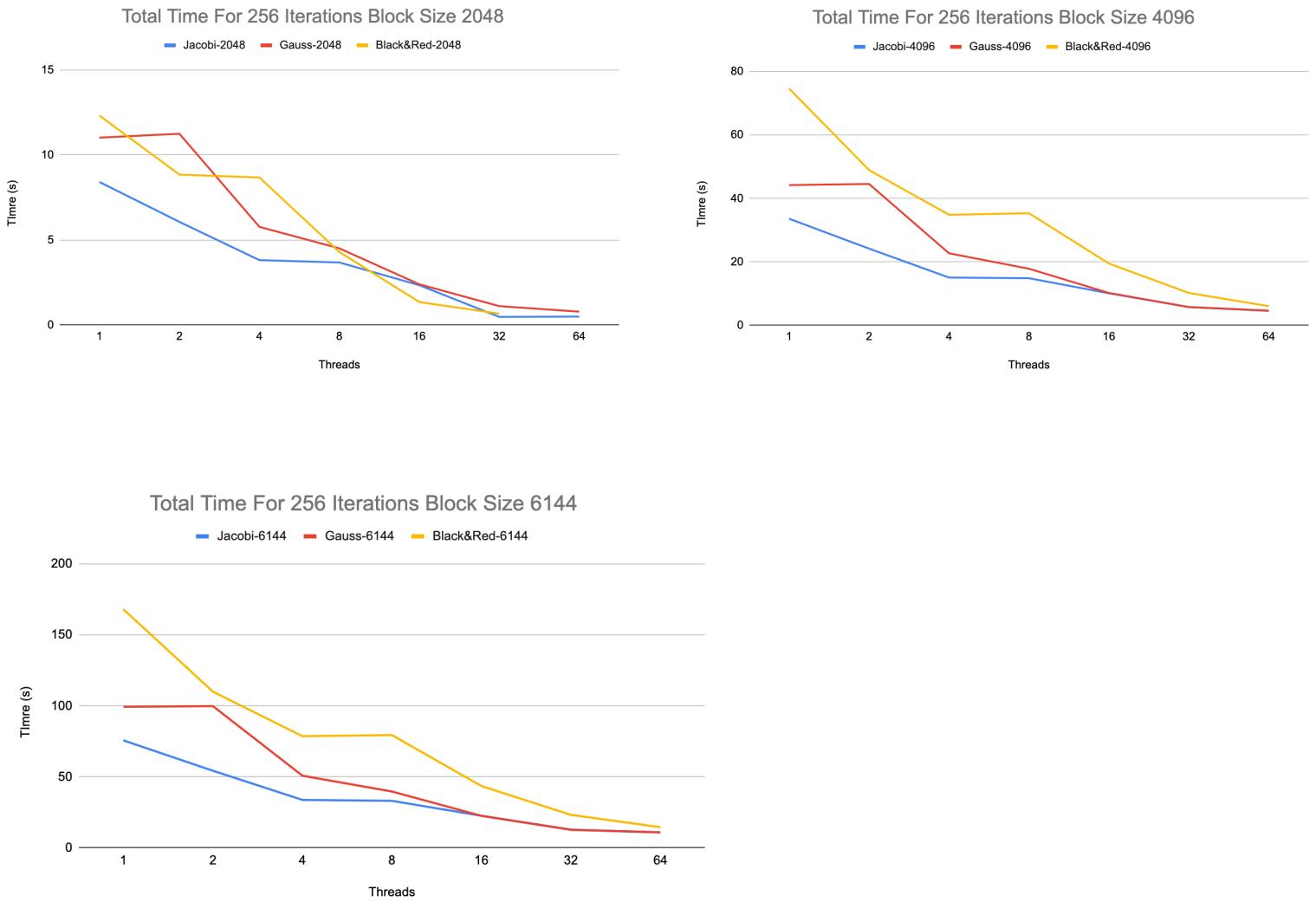


Red & Black Block Size 6144



Με τη συγκεκριμένη υλοποίηση, παρατηρούμε εξαιρετική συμπεριφορά, καθώς η σύγκλιση επιτυγχάνεται γρήγορα και οι διεργασίες αφιερώνουν περισσότερο χρόνο στους υπολογισμούς παρά στην επικοινωνία. Συνεπώς, το πρόγραμμα κλιμακώνει αποτελεσματικά.

Τελική Σύγκριση Υλοποιήσεων



Συνολικά, η υλοποίηση με την καλύτερη κλιμάκωση είναι το RedBlack, ενώ η Jacobi δεν κλιμακώνει καλά, καθώς, όπως εξηγήσαμε, γίνεται γρήγορα memory-bound. Ωστόσο διατηρεί τους μικρότερους χρόνους ανά επανάληψη, καθώς έχει τον μικρότερο δυνατό συγχρονισμό.

Για μικρά μεγέθη πινάκων, όπου η επικοινωνία αποτελεί σημαντικό ποσοστό του κόστους, η μέθοδος RedBlack αποδίδει καλύτερα όσον αφορά τον χρόνο ανά επανάληψη. Αντίθετα, για μεγαλύτερα πλέγματα, οι μέθοδοι Gauss και Jacobi παρουσιάζουν πιο συστηματικά τους μικρότερους χρόνους ανά επανάληψη.