Εθνικο Μετσοβιο Πολυτεχνειο

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

4η ΑΣΚΗΣΗ

Αχλάτης Στέφανος-Σταμάτης (03116149)

<el16149@central.ntua.gr>

Ιούνιος 2020

ΠΕΡΙΕΧΟΜΕΝΑ

Εισαγωγή

Το πρώτο μέρος της παρούσας άσκησης έχει ως αντικείμενο την αξιολόγηση διαφόρων μηχανισμών συγχρονισμού και πρωτοκόλλων συνάφειας κρυφής μνήμης (**cache coherence protocols**) σε σύγχρονες πολυπύρηνες αρχιτεκτονικές. Έτσι, με βάση δοσμένο πολυνηματικό κώδικα με θα υλοποιηθούν διάφοροι μηχανισμοί συγχρονισμού, οι οποίοι έπειτα θα αξιολογηθούν σε ένα πολυπύρηνο προσομοιώμενο σύστημα με τη βοήθεια του Sniper. Πιο συγκεκριμένα, θα αξιολογήσουμε πειραματικά την κλιμακοσιμότητα των πρωτόκολλων συγχρονισμού Test- and-Set, Test-and-Test-and-Set και Mutex (test-set ή compare-and-swap) με διάφορα μεγέθη κρίσιμης περιοχής, επαναλήψεων και αριθμού πυρήνων.
Το δεύτερο μέρος της παρούσας άσκησης, έχει ως αντικείμενο την εφαρμογή του **Tomasulo Algorithm** για ένα δεδομένο υπολογιστικό σύστημα με συγκεκριμένα χαρακτηριστικά και για ένα συγκεκριμένο κομμάτι κώδικα που δίνεται σε assembly μορφή.

Μέρος Α

Εργαλεία

PIN TOOL

Για την πραγματοποίηση της άσκησης χρησιμοποιήθηκε το εργαλείο PIN, το οποίο είναι ένα εργαλείο ανάλυσης εφαρμογών που αναπτύσσεται και συντηρείται από την Intel. Η χρήση του PIN δίνει τη δυνατότητα για dynamic binary instrumentation, το οποίο σημαίνει πως εισάγεται δυναμικά κώδικας κατά τη διάρκεια της εκτέλεσης των μετροπρογραμμάτων ανάμεσα στις εντολές της εφαρμογής αυτής έτσι ώστε να συλλεχθούν πληροφορίες σχετικές με την εκτέλεση, όπως ο συνολικός αριθμός εντολών, ο συνολικός αριθμός ευστοχιών στην κρυφή μνήμη κ.ο.κ.
Η έκδοση του PIN στην οποία πραγματοποιήθηκε η προσομοίωση είναι η **PIN 98189**, σε **Ubuntu Linux 18.04** με έκδοση **πυρήνα 4.4**.

Sniper Multicore Simulator

Θα χρησιμοποιήσουμε τον προσομοιωτή “Sniper Multicore Simulator”, ο οποίος αξιοποιεί το εργαλείο “PIN” που αναφέρεται παραπάνω. Με τη βοήθεια του Sniper Multicore Simulator θα μελετηθούν τα χαρακτηριστικά των σύγχρονων superscalar, out-of- order επεξεργαστών και ο τρόπος με τον οποίο επηρεάζουν την απόδοση του συστήματος, την κατανάλωση ενέργειας και το μέγεθος του chip του επεξεργαστή.
Η έκδοση του Sniper που χρησιμοποιήθηκε είναι η **Sniper 7.3**.

McPAT

Το McPAT (Multi-core Power, Area, Timing) είναι ένα εργαλείο το οποίο χρησιμεύει στη μοντελοποίηση των χαρακτηριστικών ενός επεξεργαστή, όπως για παράδειγμα η κατανάλωση ενέργειας και το μέγεθος που καταλαμβάνουν στο τσιπ οι διαφορετικές δομικές μονάδες του επεξεργαστή.
Μάλιστα, ενώ ο Snipet προσομοιώνει τις παραπάνω μετρικές, ο McPAT έχει συνεπικουρικό ρόλο καθώς ενσωματώνεται στον Snipet, ενώ εξάγει στατιστικά για την προσομοίωση που έχει ολοκληρωθεί!

Θεωρητικό Υπόβαθρο

Οι πολυεπεξεργαστές κοινόχρηστης μνήμης (shared memory multiprocessors) αποτελούνται από πολλαπλούς επεξεργαστές που όλοι "βλέπουν" μιά κοινή μνήμη ή ενός συστήματος από κοινές μνήμες, μέσω της οποίας επικοινωνούν μεταξύ τους και μπορούν να συνεργάζονται (π.χ. όταν εκτελούν ένα παράλληλο πρόγραμμα). Η επικοινωνία τους προκύπτει όταν ένας επεξεργαστής γράφει κάποια νέα αποτελέσματα σε ορισμένες θέσης της κοινής μνήμης, και ένας ή περισσότεροι άλλοι επεξεργαστές διαβάζουν αυτές τις νέες τιμές από εκείνες τις θέσεις μνήμης, κάτι που θυμίζει αρκετά και την επικοινωνία με τις συσκευές εισόδου/εξόδου. Οι πιό συχνοί τέτοιοι πολυεπεξεργαστές σήμερα είναι οι πολυπύρηνοι επεξεργαστές (multicore processors), που έχουν πολλαπλούς πυρήνες (cores), δηλαδή επεξεργαστές όλους μέσα στο ίδιο chip.

Επειδή οι μνήμες είναι συνήθως μονόπορτες (γιά να μην κοστίζουν πολύ), θα ήταν πολύ αργό εάν όλοι οι επεξεργαστές σ' ένα τέτοιο σύστημα εργάζονταν συνεχώς με την μία κοινόχρηστη μνήμη. Αντ' αυτού, λοιπόν, ο κάθε επεξεργαστής έχει την δική του "ιδιωτική" (private) κρυφή μνήμη, και μόνον οι αστοχίες αυτών των κρυφών μνημών πηγαίνουν στην κοινόχρηστη μνήμη. Αυτήν την πόρτα της κοινόχρηστης μνήμης την βλέπουμε συνήθως σαν μία αρτηρία (bus) που ενώνει όλες τις κρυφές μνήμες όλων των πυρήνων καθώς και τη μνήμη. Όλες οι αστοχίες (misses) των κρυφών μνημών περνάνε από αυτή την αρτηρία και άρα, όποιος θέλει και κοιτάζει μπορεί να τις βλέπει.

Όποτε δημιουργούμε πολλαπλά αντίγραφα της ίδιας πληροφορίας και κάποιος αλλάζει ένα από τα αντίγραφα (ή το πρωτότυπο) ενώ άλλοι έχουν και βλέπουν άλλα αντίγραφα, τότε εντοπίζουμε ένα κίνδυνο. Αυτό ισχύει και στο υλικό (κρυφές μνήμες), και στο λογισμικό (π.χ. web browser caches).

Την επιθυμητή συνοχή (coherence) τέτοιων κρυφών μνημών, δηλαδή το να βλέπουν όλοι οι επεξεργαστές την ίδια τιμή στην ίδια διεύθυνση (σχεδόν) ανά πάσα στιγμή, την εξασφαλίζουν ειδικά πρωτόκολλα συνοχής, συνήθως υλοποιημένα σε υλικό (χωρίς να αποκλείονται και πρωτόκολα σε λογισμικό), τα απλούστερα από τα οποία είναι τα πρωτόκολλα "snooping" όταν όλες οι κρυφές μνήμες συνδέονται πάνω στην ίδια αρτηρία (bus) και παρακολουθούν εκεί "όλα τα νέα της γειτονιάς".

Στην αρχιτεκτονική υπολογιστών, η συνοχή της κρυφής μνήμης (**Cache Coherence**) αφορά την ομοιομορφία των κοινών πόρων που δεν αποτελούν κομμάτι μόνο μίας κρυφής μνήμης αλλά αντιθέτως μπορούν να αποθηκευτούν σε πολλαπλές κρυφές μνήμες

Όταν οι διαφορετικοί πυρήνες σε ένα σύστημα διατηρούν προσωρινά αποθηκευμένα στοιχεία ενός κοινού πόρου μνήμης, ενδέχεται να προκύψουν προβλήματα με **ασυνεπή δεδομένα**, κάτι που συμβαίνει ιδιαίτερα με τις CPU σε ένα σύστημα πολλαπλών επεξεργαστών.

Για παράδειγμα έστω ότι 2 πυρήνες έχουν αντίγραφο του ίδιου κοινού πορού, έστω ενός block μνήμης. Θεωρητικά για αποφυγή του παραπάνω προβλήματος, όταν κάποιος εκ των 2 πυρήνων προβεί σε ενημέρωση του block, θα πρέπει ιδανικά ταυτόχρονα να γίνει και ενημέρωση του block του άλλου πυρήνα ώστε να μην απομείνει με **μη έγκυρη μνήμη cache**. Η αντιμετώπιση τέτοιου είδους conflicts, με κατάλληλη διαχείριση των κοινών πόρων σε παραπάνω από μία κρυφές μνήμης, κρίνεται αναγκαία για την εξασφάλιση της **συνοχής της κρυφής μνήμης.**

3.1 Σύγκριση υλοποιήσεων

Αρχικά, να θυμίσουμε τον κώδικα της κρίσιμης περιοχής. Σκεφτείτε ένα σύστημα που αποτελείται από n διεργασίες, κάθε διεργασία έχει ένα τμήμα κώδικα, που ονομάζεται κρίσιμη περιοχή, στην οποία η διεργασία μπορεί να μεταβάλει κοινές μεταβλητές, να ενημερώνει ένα πίνακα, να γράφει σε ένα αρχείο κοκ.
Η διαδικασία του κλειδώματος είναι απαραίτητη για την αντιμετώπιση του προβλήματος των κρίσιμων περιοχών και με βάση αυτή επιτυγχάνουμε την προστασία των κρίσιμων περιοχών μέσω κλειδωμάτων. Δηλαδή για να μπει μια διεργασία στην κρίσιμη περιοχή πρέπει να "ξεκλειδωθεί" η κρίσιμη περιοχή και να αποκτήσει δικαίωμα πρόσβασης σε αυτή.
Στο σημείο αυτό θα αναφερθούμε στο **μηχανισμούς κλειδώματος** (**locks)** τους οποίους υλοποιήσαμε. Πρόκειται για τους μηχανισμούς κλειδώματος **Test-and-Set** (TAS) και **Test-and-Test-and-Set** (TTAS).Η υλοποίηση των ρουτίνων για τα TAS και TTAS πραγματοποιήθηκαν χρησιμοποιώντας τις εξής δύο συναρτήσεις από τα **atomic intrinsics του gcc** αντίσοιχα:
int __sync_lock_test_and_set(int *ptr, int newval);
int __sync_val_compare_and_swap(int *ptr, int oldval, int newval);

Ο δοσμένος κώδικας περιλαμβάνει τη βιβλιοθήκη Pthreads (Posix Threads) για τη δημιουργία και διαχείριση των νημάτων λογισμικού. Δημιουργείται ένας αριθμός από νήματα, τα οποία εκτελούν μια κρίσιμη περιοχή για ένα συγκεκριμένο αριθμό επαναλήψεων. Η είσοδος των νημάτων στην κρίσιμη περιοχή ελέγχεται από **μία κοινή μεταβλητή κλειδιού**. Πριν την είσοδο, κάθε νήμα εκτελεί την κατάλληλη ρουτίνα για την απόκτηση του κλειδιού (**lock aqcuire**), προκειμένου να λάβει την αποκλειστικότητα εισόδου στην περιοχή όπου αφού εκτελέσει έναν **cpu-intensive υπολογισμό** δηλαδή μια βαριά υπολογιστική εργασία που λαμβάνει χώρα εντός της cpu και δεν αφορά διεργασίες εισόδου/εξόδου, έπειτα κατά την έξοδο εκτελεί την κατάλληλη ρουτίνα για την απελευθέρωση του κλειδιού (**lock release**).
Οι προσωμειώσεις διεξήχθησαν με τις παρακάτω τρεις παραμέτρους:

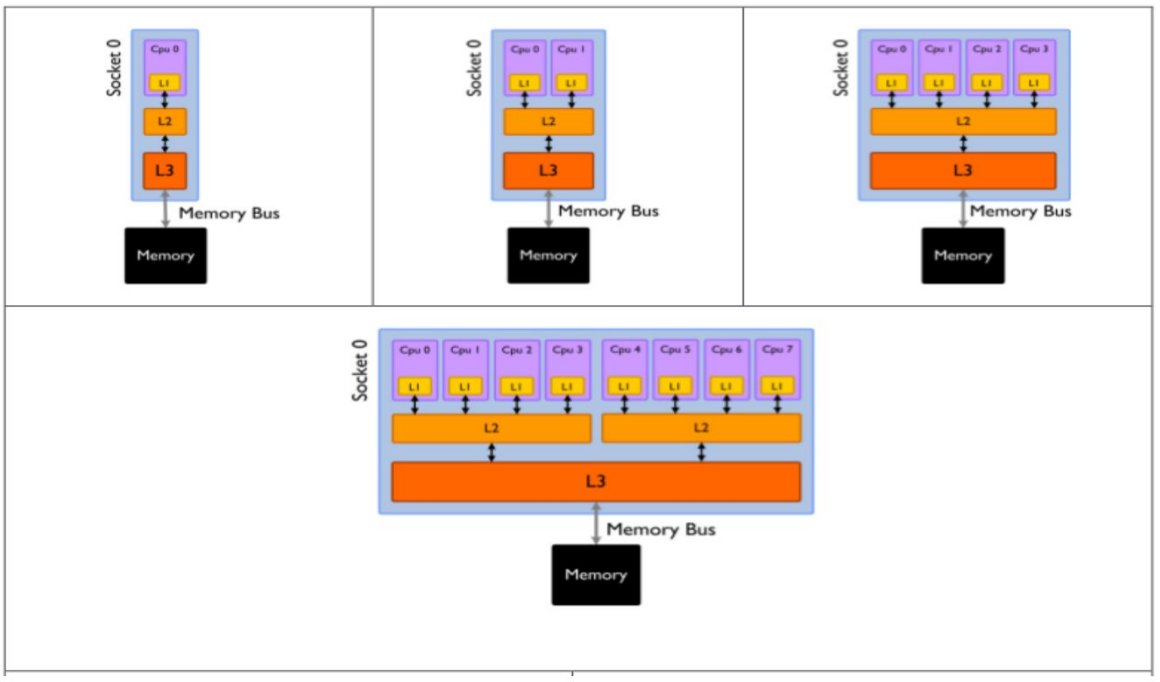
- **nthreads**: ο αριθμός των threads που θα δημιουργηθούν
- **iterations**: ο αριθμός των επαναλήψεων που θα εκτελεστεί η κρίσιμη περιοχή από κάθε νήμα
- **grain_size**: ο όγκος δηλαδή το φόρτο των cpu-intensive υπολογισμών, και κατ' επέκταση το μέγεθος της κρίσιμης περιοχής. Δηλαδή μπορούμε να ορίσουμε δυναμικά το ογκο του μεγέθους της δυναμικής περιοχής για να κανουμε κατάλληλους υπολογισμούς.

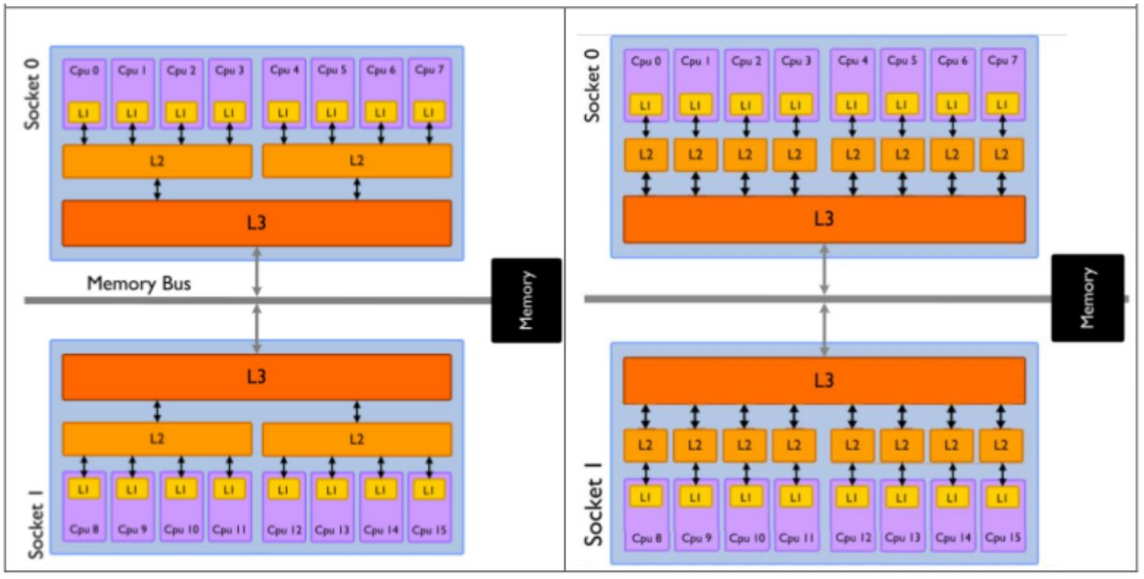
Ερώτημα 3.1.1 Για κάθε grain size, δώστε το διάγραμμα της κλιμάκωσης του συνολικού χρόνου εκτέλεσης της περιοχής ενδιαφέροντος σε σχέση με τον αριθμό των νημάτων. Συγκεκριμένα, στον x-άξονα πρέπει να έχετε τον αριθμό των νημάτων και στον y-άξονα τον χρόνο εκτέλεσης σε κύκλους. Στο ίδιο διάγραμμα θα πρέπει να συμπεριλάβετε τα αποτελέσματα και για τις 5 εκδόσεις.

Είναι προφανές ότι όταν έχουμε πολλά νήματα δημιουργούνται έντονα θέματα με την διαχείριση της κρίσιμης περιοχής όσο και με τα δεδομένα των κρυφών μνημών, από την άλλη μερία αν έχουμε περισσότερα νήματα μπορούμε να κάνουμε πιο παράλληλα τις διεργασίες μας.
Άρα επειδή δεν είναι τόσο απλη η απάντηση ορίζουμε τον όρο της κλιμακωσιμότητας (**scalability**) που περιγράφει το **πόσο καλά αποδίδει** ένα παράλληλο πρόγραμμα όσο αυξάνεται ο αριθμός των νημάτων από τα οποία αποτελείται.
Να σημειωθεί πως ακόμα και αν ένα παράλληλο πρόγραμμα είναι σχεδιασμένο για να κλιμακώνει ιδανικά (π.χ., ο χρόνος εκτέλεσης με N νήματα να ισούται με 1/N του χρόνου με 1 νήμα), υπάρχουν διάφοροι εξωγενείς παράγοντες (overhead, bandwidth πρόσβασης στην μνήμη) που μπορούν να περιορίσουν την κλιμακωσιμότητά του πολύ κάτω του ιδανικού, με συνέπεια το κόστος μιας λειτουργίας ενός νήματος να αυξάνεται όσο αυξάνεται ο αριθμός των νημάτων.
Παρακάτω αξιολογείται η κλιμακωσιμότητα των μηχανισμών TAS_CAS, TAS_TS, TTAS_CAS, TTAS_TS και Mutex, χρησιμοποιώντας προσομοιωμένα πολυπύρηνα συστήματα με διάφορες πολιτικές διαμοιρασμού των caches. Παρακάτω παρουσιάζονται οι διάφορες τιμές των παραμέτρων της προσωμοίωσης :

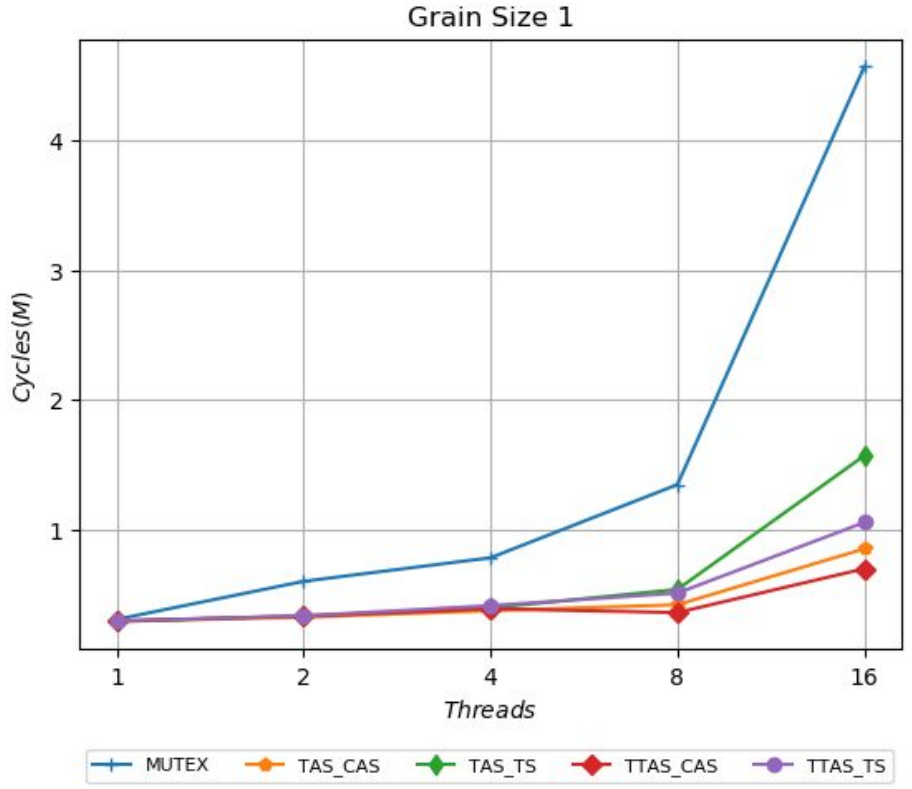
1. **εκδόσεις προγράμματος: TAS_CAS, TAS_TS, TTAS_CAS, TTAS_TS, MUTEX**
2. **iterations: 1000**
3. **nthreads: 1, 2, 4, 8, 16.1, 16.2 (2 συνδυασμοί)**

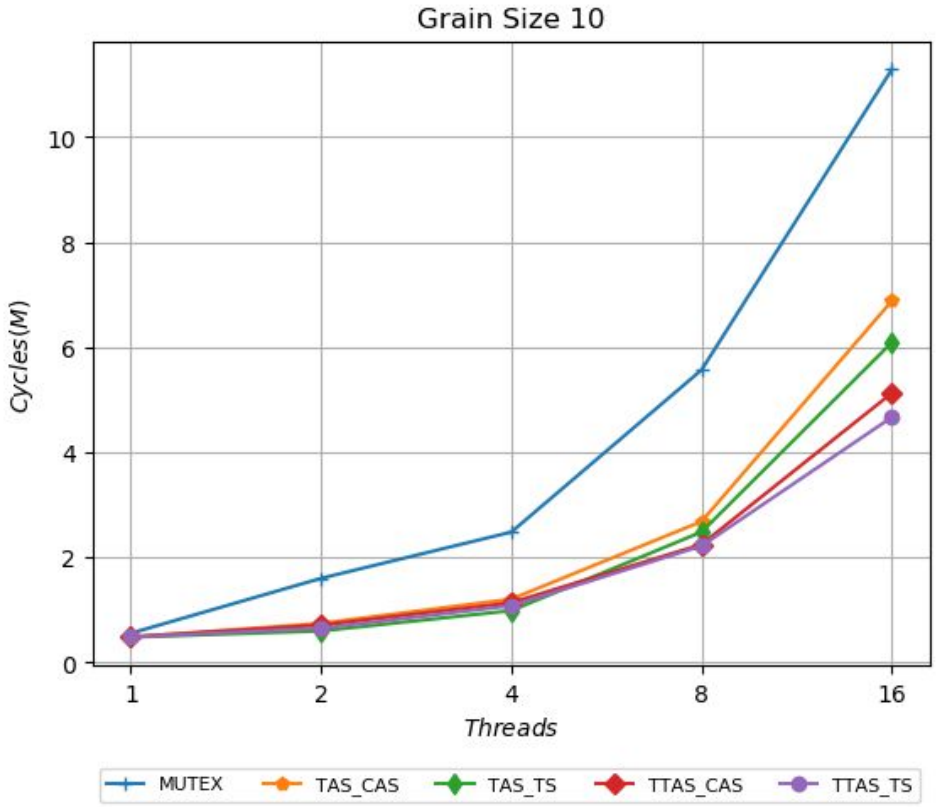
Στη συνέχεια παρατίθενται σχηματικά οι **πολιτικές διαμοιρασμού των caches** όπως αυτές δίνονται στην εκφώνηση της άσκησης:

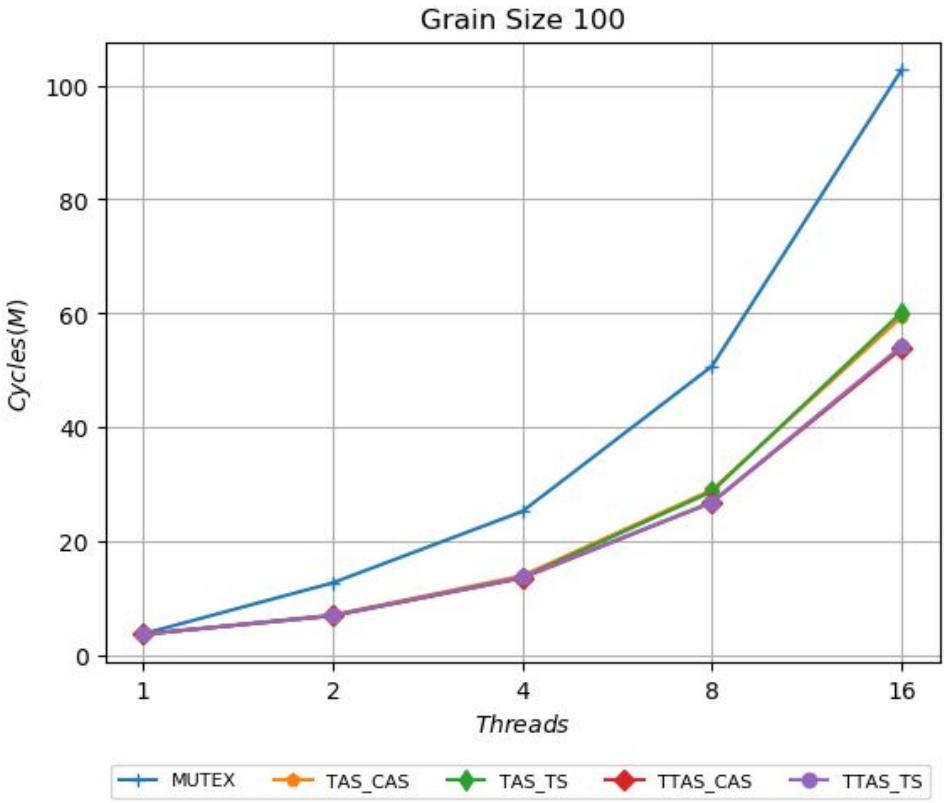




Αρχικά, για κάθε grain size (μέγεθος κρίσιμης περιοχής), θα παρουσιάσουμε τα διαγράμματα κλιμάκωσης χρόνου εκτέλεσης ανάλογα με τον αριθμό των νημάτων που χρησιμοποιούνται.





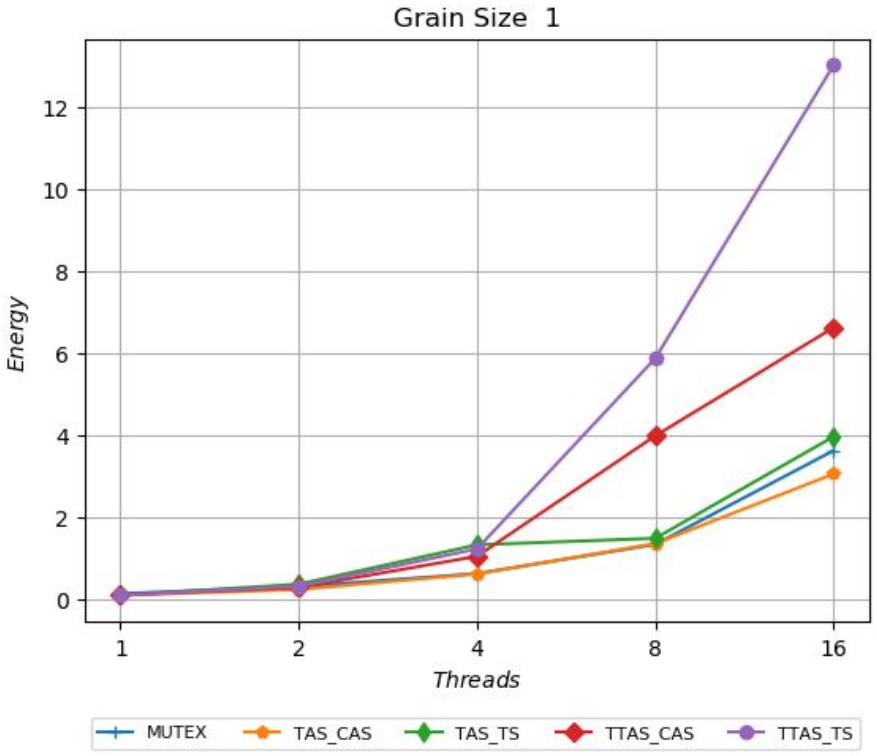

Grain Size 100
100
80
60
40
20
0
Cycles(M)
1
2
4
8
16
Threads
MUTEX
TAS_CAS
TAS_TS
TTAS_CAS
TTAS_TS

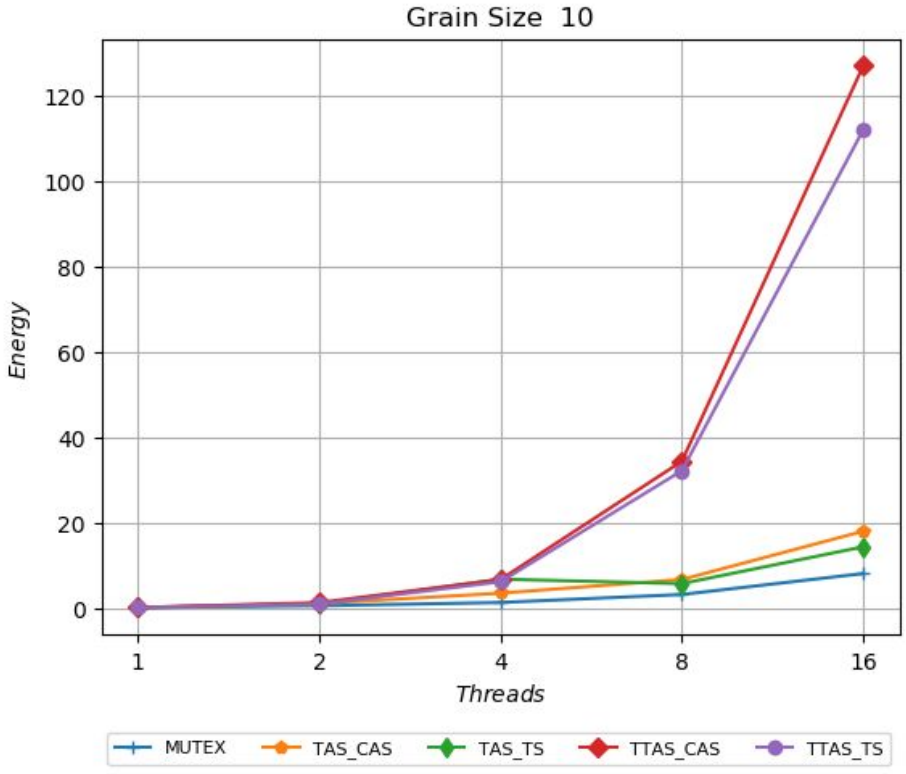
Ερώτημα 3.1.2. Τι συμπεραίνετε για την κλιμάκωση του χρόνου εκτέλεσης σε σχέση με τη φύση της εκάστοτε υλοποίησης; Τι συμπεραίνετε για την κλιμάκωση του χρόνου εκτέλεσης σε σχέση με το grain size; Δικαιολογήστε τις απαντήσεις σας.

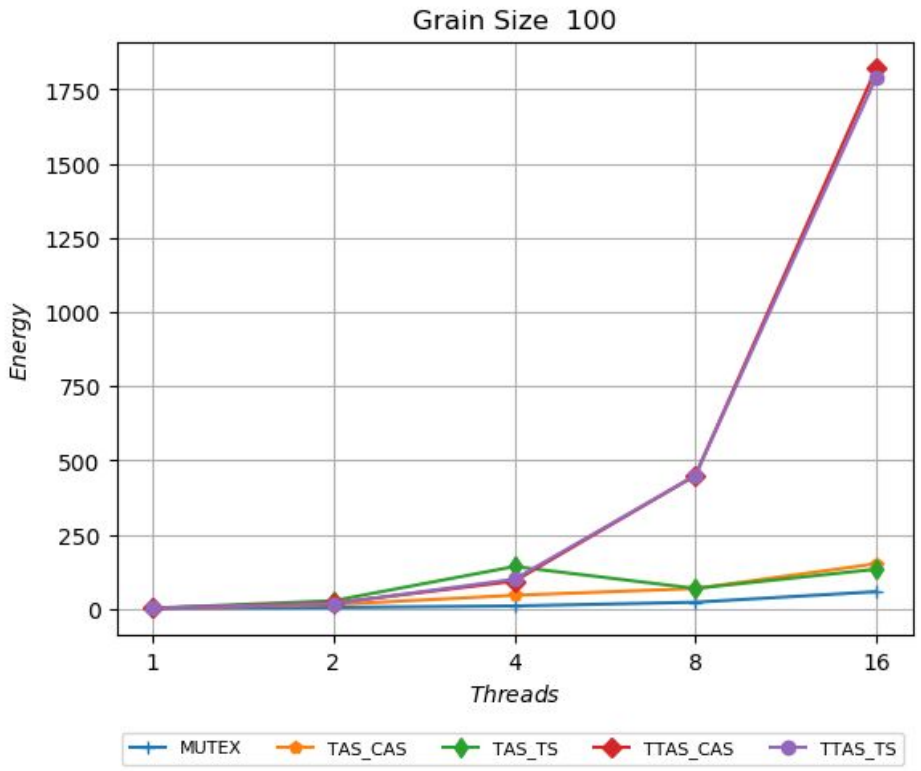
- Αυτό που φαίνεται από τα παραπάνω διαγράμματα είναι ότι για μεγάλο αριθμό νημάτων, ο μηχανισμός **MUTEX** είναι σταθερά **πιο αργός** σε όλους τους συνδυασμούς λόγω του overhead που προκύπτει από την διαχείριση της ουράς των διεργασιών. Να σημειωθεί ότι ο κλασσικός MUTEX χρησιμοποιεί busy way αλλά ο mutex που χρησιμοποιείται μέσω της pthread δεν χρησιμοποιει busy wait. Επομένως, ο κλασικός MUTEX θα είχε ακόμη χειρότερο αποτέλεσμα, αφού θα είχαμε και επιπλέον κύκλους μηχανής λόγω του busy wait.
- Παρατηρούμε επίσης ότι οι **TAS** έχουν **χειρότερη απόδοση** από τους **TTAS**. Αυτό έχει προφανή εξήγηση το γεγονός ότι ο TAS σε κάθε εκτέλεσή του από ένα νήμα εκτελεί μία εντολή store η οποία μπορεί να «χαλάει» την cache των πυρήνων που εκτελούν τα υπόλοιπα νήματα αλλά και να δημιουργεί αχρείαστη κίνηση στον δίαυλο. Άρα θα έχει μεγαλύτερες καθυστερήσεις από τον TTAS, ο οποίος παρακολουθεί την μεταβλητή «κλειδί» και μόνο όταν δει ότι αυτή είναι ελεύθερη κινείται να την πάρει, και άρα μόνο τότε εκτελεί store. Αυτό σημαίνει ότι η υλοποίηση που χρησιμοποιεί TAS θα κάνει πολλές store και όταν κάποια στιγμή θα μπαίνει στην κρίσιμη περιοχή το εκάστοτε νήμα θα εκτελεί εντολές, ενώ με τον TTAS περίπου για κάθε store που θα εκτελεί ένα νήμα, θα μπαίνει και στην κρίσιμη περιοχή.
- Κάτι άλλο που παρατηρείται σε αυτά τα διαγράμματα είναι ότι για **μικρό αριθμό νημάτων**, δεν υπάρχει μεγάλη διαφοροποίηση απόδοσης ανάμεσα στους TAS και τους TTAS, κάτι που είναι λογικό με βάση αυτά που εξηγήσαμε νωρίτερα για την λειτουργία των δύο μηχανισμών. Όσο τα νήματα γίνονται περισσότερα, τόσο περισσότερα γίνονται και τα νήματα που θα είναι κάθε φορά εκτός κρίσιμης περιοχής και άρα θα προκαλούν καθυστερήσεις στους TAS για λόγους τους οποίους εξηγήσαμε. Ενώ στους TTAS αυτές οι καθυστερήσεις θα είναι σαφώς μικρότερες.
- Τέλος η αύξηση του grain size επιφέρει και συνολική αύξηση στον αριθμό των απαιτούμενων κύκλων, αλλά αυτό φαίνεται να συμβαίνει με ομοειδή τρόπο για όλα τα κλειδώματα. Πράγμα λογικό καθώς ουσιαστικά αυξάνουμε το μέγεθος της κρίσιμης περιοχής.
- Ας τονίσουμε ότι καλη κλιμακωσιμότητα για το συγκεκριμένο πρόγραμμα σημαίνει ότι όσο ανεβαίνουν τα νήματα χειροτερεύει το λιγότερο δυνατό το πλήθως των cycles. Επομένως, μπορούμε να πούμε ότι την **καλύτερη κλιμακωσιμότητα** παρουσιάζει το **TTAS** μετά το **TAS** και τέλος την **χειρότερη κλιμακωσιμότητα** το **MUTEX**.

Ερώτημα 3.1.3. Χρησιμοποιώντας όπως και στην προηγούμενη άσκηση το McPAT, συμπεριλάβετε στην ανάλυση σας εκτός από το χρόνο εκτέλεσης και την κατανάλωση ενέργειας (Energy, EDP κτλ.)

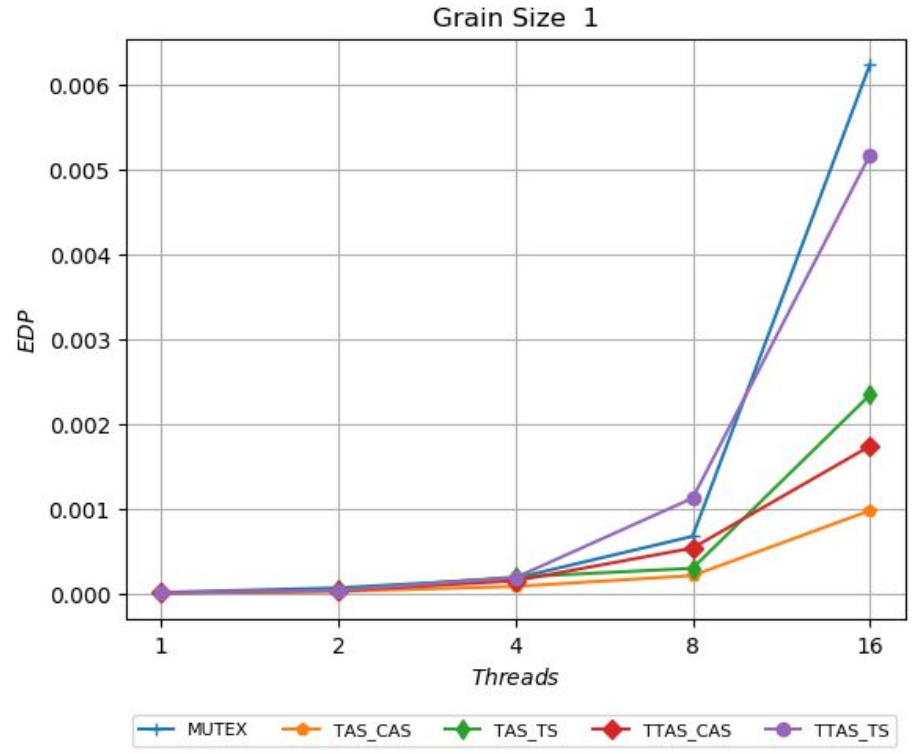
Αποτελέσματα για το **Energy**:

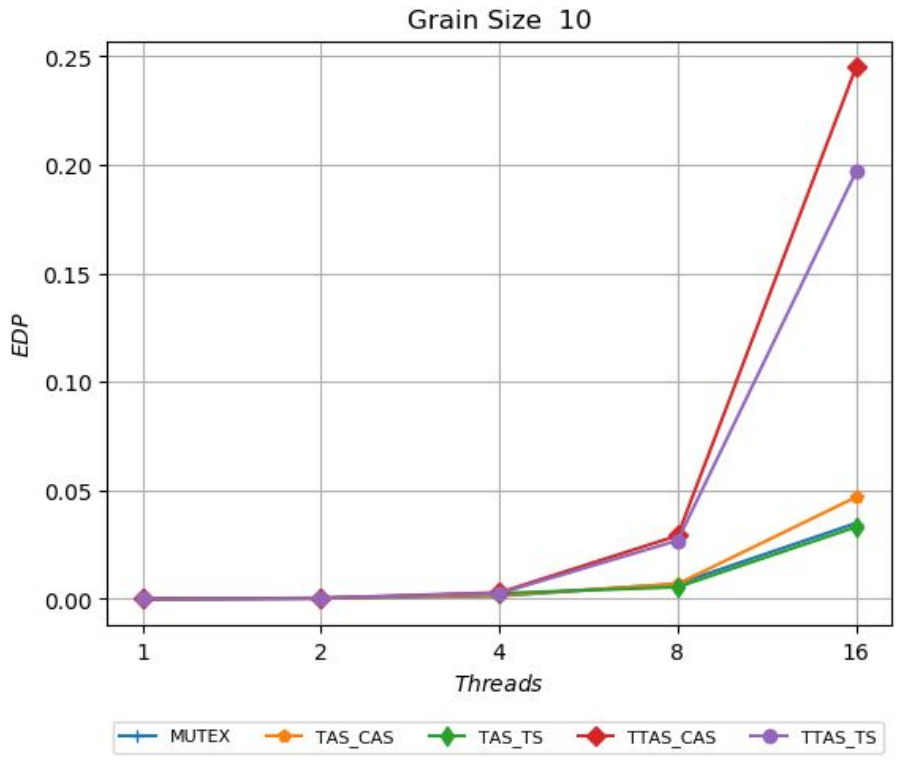


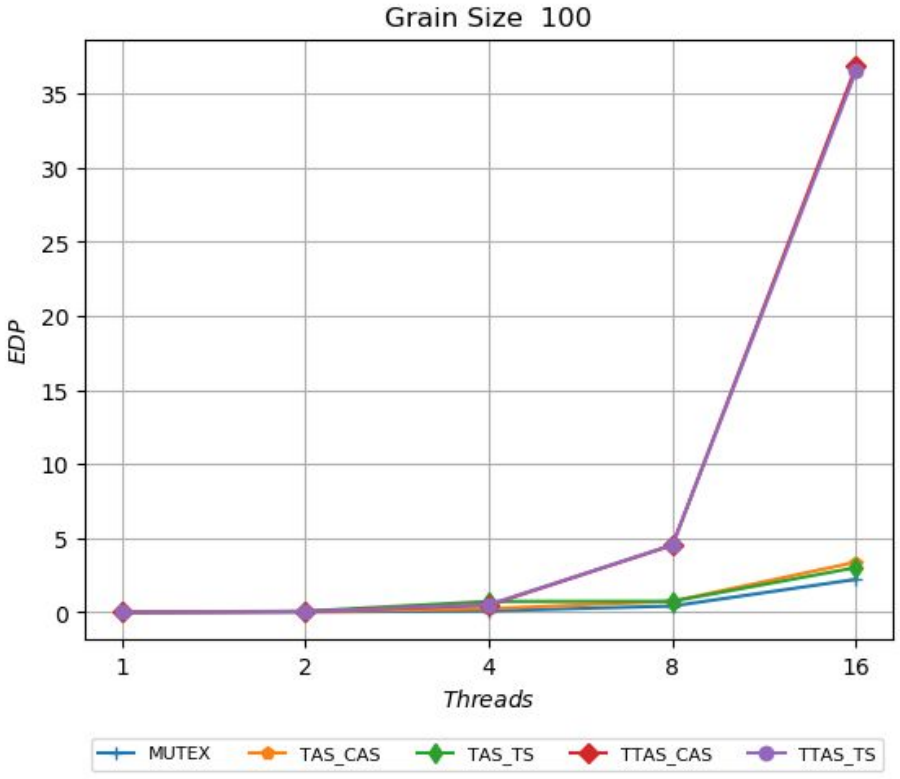




Αποτελέσματα για το **EDP**:




Grain Size 10
EDP
0.25
0.20
0.15
0.10
0.05
0.00
1
2
4
8
16
Threads
MUTEX
TAS_CAS
TAS_TS
TTAS_CAS
TTAS_TS


Grain Size 100
EDP
35
30
25
20
15
10
5
0
1
2
4
8
16
Threads
MUTEX
TAS_CAS
TAS_TS
TTAS_CAS
TTAS_TS

Συμπεράσματα:
Από την πλευρά της ενέργειας παρατηρούμε ότι **λιγότερο αποδοτικός** είναι ο μηχανισμός **TTAS** γιατί το busy-wait που κάνει με την πρόσθετη συνθήκη απασχολεί αρκετά περισσότερο τον επεξεργαστή με αποτέλεσμα να καταναλώνει περισσότερη ενέργεια. Αντίθετα ο **πιο αποδοτικός** μηχανισμός όσο αφορά την ενέργεια είναι ο **MUTEX**, εκτιμάμε όμως πως αν συνεκτιμηθεί και ο χρόνος στην παραπάνω γραφική (Joule * sec) το MUTEX θα εμφανίζει υψηλότερες τιμές. Μια **μέση κατάσταση** παρουσιάζει το **TAS**.
Για μικρό grain-size (ίσο με 1) το mutex παρουσιάζει κακο EDP σε σχέση με τις busy-wait λύσεις. Ο λόγος που συμβαίνει αυτό είναι επειδή το μήκος της κρίσιμης περιοχής (region of interest) είναι αρκετά μικρό με συνέπεια να είναι άσκοπο το να κοιμηθούν οι διεργασίες και να υπερισχύει το overhead, πρακτικά σπάνια θα είναι τοσο μικρό το μήκος της κρίσημης περιοχής. Καθώς το grain size αυξάνεται, το mutex υπερισχύει σε σχέση με τις υπόλοιπες υλοποιήσεις που είναι τύπου busy-waiting τόσο σε κατανάλωση ενέργειας όσο και σε EDP.Συγκριτικά για τις άλλες τεχνικές, παρατηρούμε ότι η tas είναι καλύτερη απο την ttas στον τομέα της κατανάλωσης ενέργειας και του EDP. Ένας πιθανός λόγος που συμβαίνει αυτό είναι ότι η tas εκτελεί συνεχόμενες εγγραφές (write) που πιθανώς προκαλούν stalls για να γίνει το cache line transfer σε αντίθεση με το ttas εκτελεί busy-loop διαβάζοντας τη μνήμη αδιάκοπα και καταναλώνοντας περισσότερη ενέργεια.
Πιο αναλυτικά:
Ενέργεια:
Όσον αφορά την ενέργεια, παρατηρούμε ότι το MUTEX είναι πολύ καλύτερο σε σύγκριση με τους υπόλοιπους μηχανισμούς. Αυξάνει λίγο όσο αυξάνει το grain size. Για το μηχανισμό TTAS, παρατηρούμε ότι η κατανάλωση ενέργειας κλιμακώνεται πολύ απότωμα με την αύξηση των νημάτων και αυξάνεται αρκετά όσο αυξάνεται το grain size. Όσον αφορά το TAS, ακολουθεί μια μέση κατάσταση.Η μέθοδος κλειδώματος επηρεάζει επίσης σε έναν πολύ μικρό βαθμό τη κατανάλωση ενέργειας όπου το CAS καταναλώνει περισσότερη.
EDP:
Όσον αφορά το EDP, για grain size ίσο με 1 το MUTEX έχει υψηλό EDP, αν και η κατανάλωση ενέργειας ήταν μικρή λόγω της μεγάλης διάρκειας εκτέλεσης που φαίνεται και από την υπαρξει πολύ παραπάνω κύκλων όπως είδαμε και στο πρώτο ερώτημα. Τα υπόλοιπα έχουν ίδιο EDP, αν και για μεγάλο αριθμό νημάτων, τα TAS έχουν λίγο καλύτερο EDP. Για grain size ίσο με 10, το TTAS-TS έχει καλύτερο EDP από τα υπόλοιπα διότι έχει σχετικά καλή κατανάλωση ενέργειας, χωρίς όμως να έχει και μεγάλο χρόνο εκτέλεσης, όπως το DMUTEX. Για grain size ίσο με 100, βλέπουμε ότι τα TTAS έχουν πολύ μεγαλύτερο EDP σε σχέση με τους υπόλοιπους μηχανισμούς. Αυτό συμβαίνει, διότι η κατανάλωση ενέργειάς τους, αυξάνεται εκθετικά. Τα TAS έχουν καλό EDP όμως το DMUTEX έχει το καλύτερο.

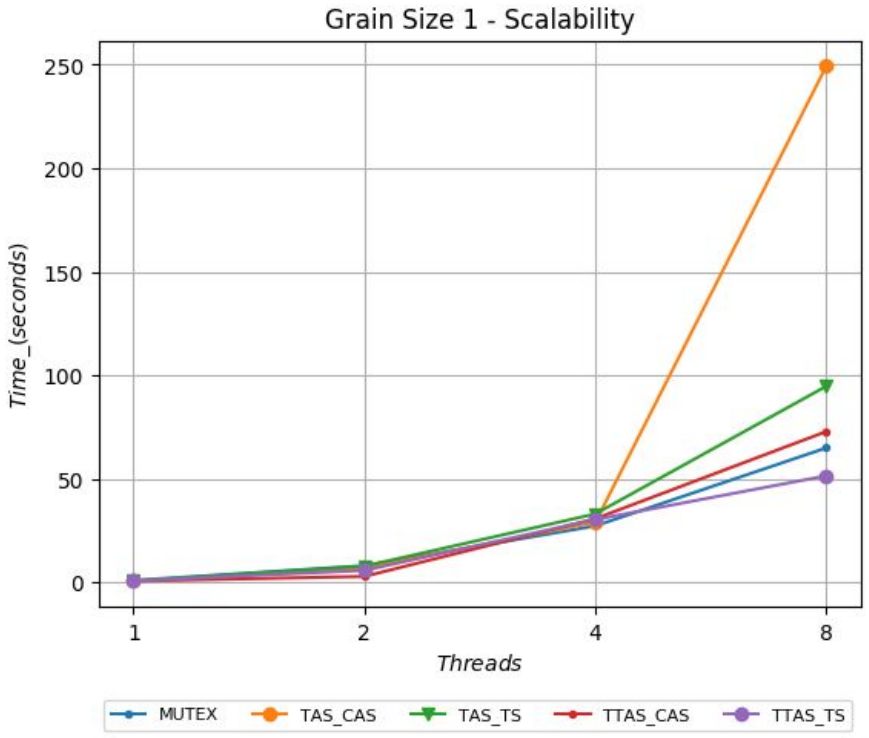
Τι επιλέγουμε (**trade-off**);
Από την πρώτη ανάλυση συμπεραίνουμε ότι η καλύτερη επιλογή θα ήταν να επιλέξουμε την στρατηγική ttas αλλά βλέπουμε ότι καταναλώνει την περισσότερη ενέργεια. Από την άλλη από αυτο το ερώτημα βλέπουμε ότι το mutex που καταναλώνει την λιγότερη ενέργεια έχει τους περισσότερους κύκλους.

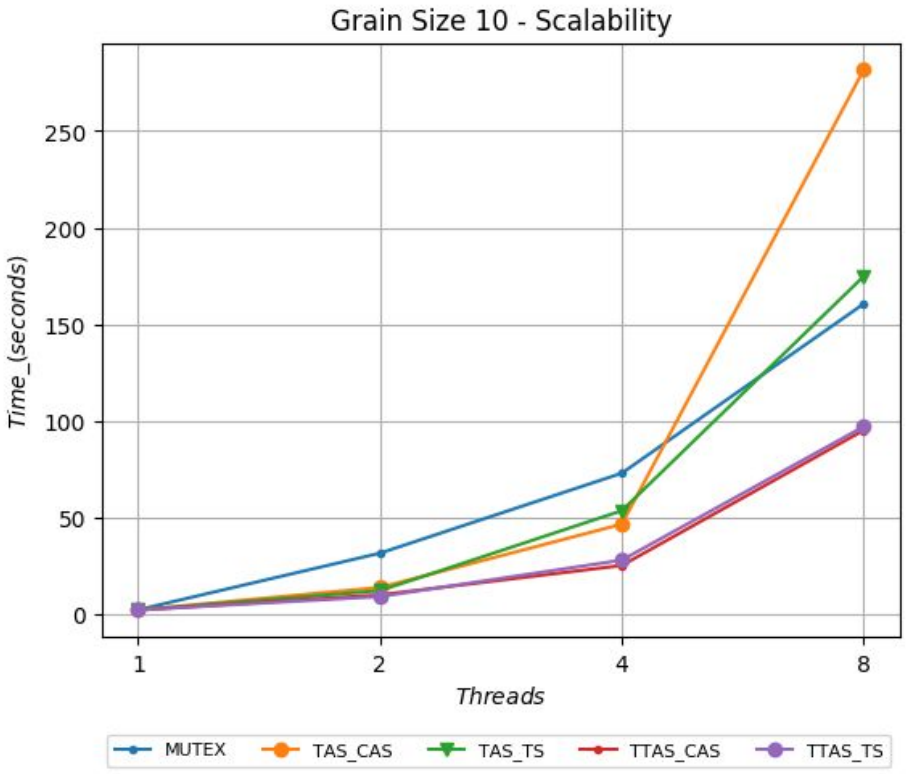
Ερώτημα 3.1.4. Μεταγλωττίστε τις διαφορετικές εκδόσεις του κώδικα για πραγματικό σύστημα. Εκτελέστε τα ίδια πειράματα με πριν σε ένα πραγματικό σύστημα, εφόσον αυτό διαθέτει πολλούς πυρήνες, ή σε ένα πολυπύρηνο vm που έχετε στον ~okeanos. Χρησιμοποιήστε τους ίδιους αριθμούς νημάτων (με μέγιστο αριθμό νημάτων ίσο με τον αριθμό των πυρήνων που διαθέτει το μηχάνημά σας) και τα ίδια grain sizes με πριν. Αυτή τη φορά δώστε έναν αρκετά μεγαλύτερο αριθμό επαναλήψεων ώστε ο χρόνος της εκτέλεσης να είναι επαρκώς μεγάλος για να μπορεί να μετρηθεί με ακρίβεια (π.χ.φροντίστε ώστε η εκτέλεση με 1 νήμα να είναι της τάξης των μερικών δευτερολέπτων). Δώστε τα ίδια διαγράμματα με το ερώτημα 3.1.1. Πώς συγκρίνεται η κλιμακωσιμότητα των διαφορετικών υλοποιήσεων στο πραγματικό σύστημα σε σχέση με το προσομοιωμένο; Δικαιολογήστε τις απαντήσεις σας.

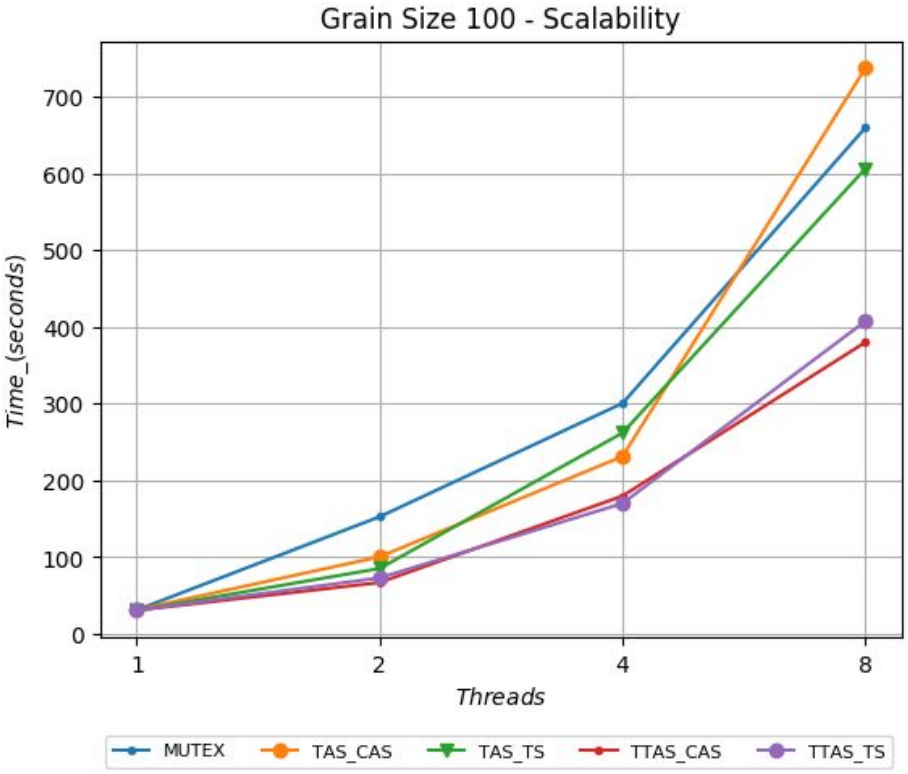
Οι προσομοιώσεις έγιναν σε έναν 4πύρηνο υπολογιστή με δυνατότητα για 8 νήματα.
Θα τρέξουμε μια προσομοίωση για 8 νήματα και δεν έχει νόημα να τρέξουμε για παραπάνω νήματα, επίσης θα τρέξουμε μια προσομοίωση για 4 νήματα.

Επομένως τρέχουμε τις προσομοιώσεις πρώτα για:

- πληθος επαναλήψεων: 50.000.000
- νήματα: 8

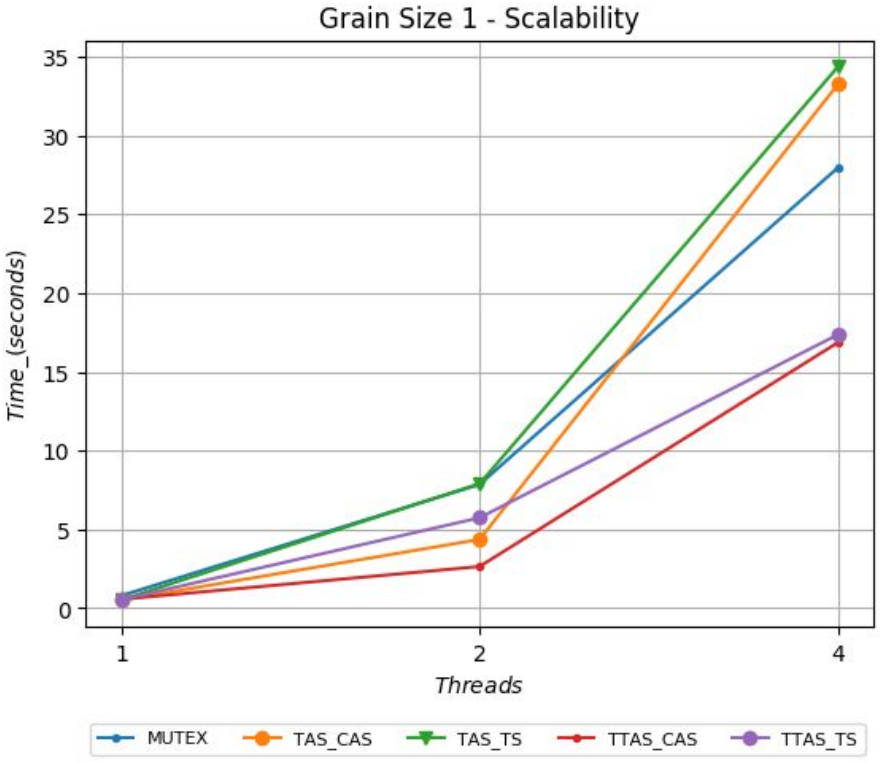


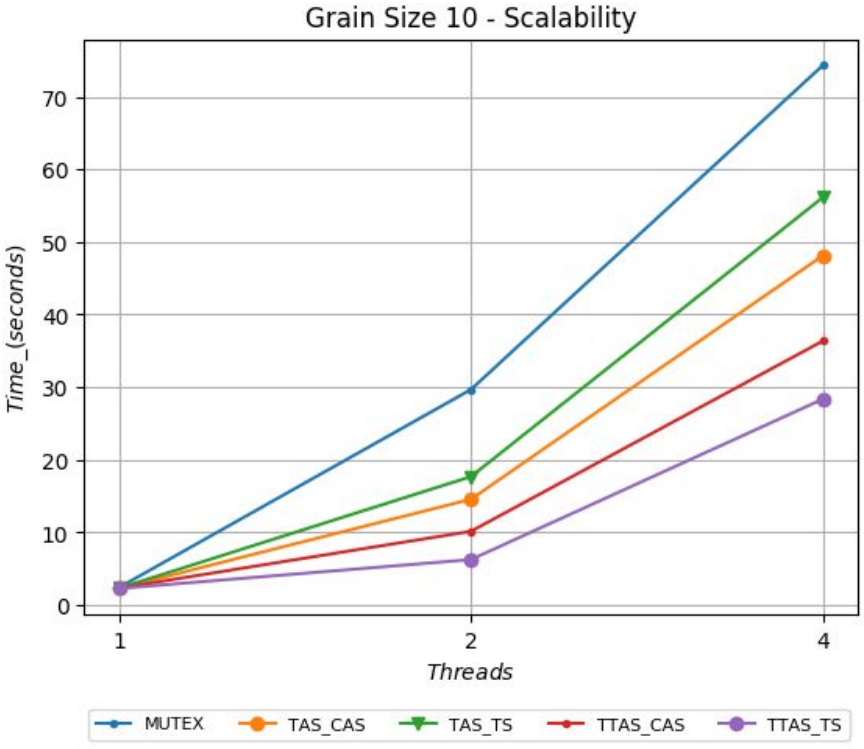

Grain Size 10 - Scalability
Time_(seconds)
0
50
100
150
200
250
1
2
4
8
Threads
MUTEX
TAS_CAS
TAS_TS
TTAS_CAS
TTAS_TS

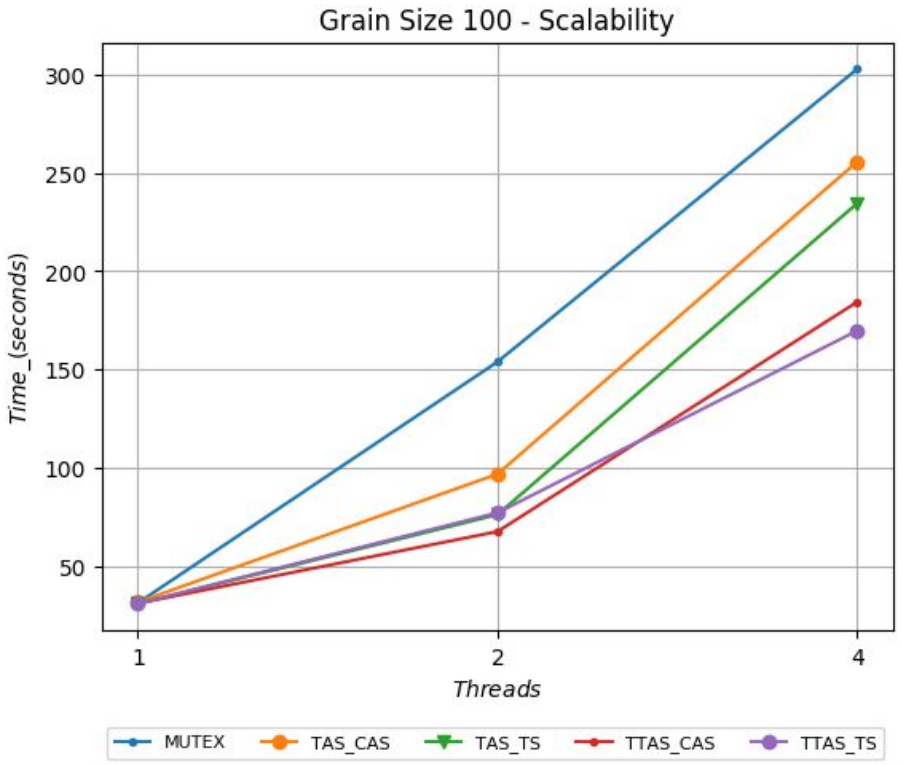

Grain Size 100 - Scalability
Time_(seconds)
0
100
200
300
400
500
600
700
1
2
4
8
Threads
MUTEX
TAS_CAS
TAS_TS
TTAS_CAS
TTAS_TS

Στην συνέχεια τρέχουμε τις προσομοιώσεις για:

- πληθος επαναλήψεων: 50.000.000
- νήματα: 4







Συμπεράσματα:
Το σύστημα έχει 4 φυσικούς πυρήνες, οπότε μέχρι 4 νήματα τρέχουν ανεμπόδιστα. Υποστηρίζει Simultaneous Multithreading (Hyperthreading στην ορολογία Intel) με 2 νηματα ανά πυρήνα, οπότε και 8 δίνει σωστά αποτελέσματα, αλλά μοιράζονται πόρους.
Αυτό το αποτέλεσμα του διαμοιρασμού πόρων φαίνεται στις προηγούμενες γραφικές και επηρεάζει την επιλογή μας.
Συνεχίζουμε την ανάλυση για 4 threads:
Καταρχάς οι κύκλοι αυξάνουν συνολικά και στους τρεις μηχανισμού όσο αυξάνει το grain size, κάτι που είναι λογικό και δεν συνέβαινε στον ίδιο βαθμό στις προσομοιώσεις. Ακόμα βλέπουμε πως με την αύξηση του αριθμού των νημάτων αυξάνει απότομα ο αριθμός των κύκλων εκτέλεσης για όλους τους μηχανισμούς κυκλώματος και όχι μόνο για το Mutex όπως συνέβαινε στο πρώτο ερώτημα. Επίσης παρατηρούμε ότι ο Mutex και εδώ είναι ο χειρότερος μηχανισμός σε κάθε περίπτωση, με εξαίρεση την πρώτη όπου το μέγεθος του κώδικα είναι αρκετά μικρό και επομένως σπάνια συναντούμε τέτοιες περιπτώσεις σε πραγματικά συστήματα. Επίσης πάλι παρατηρούμε ότι ο μηχανισμός TTAS δίνει το καλύτερο αποτέλεσμα, ενώ ο TAS πάλι δίνει την μέση λύση.
Ωστόσο παρουσιάζουν παραπλησια

3.2 Τοπολογία νημάτων

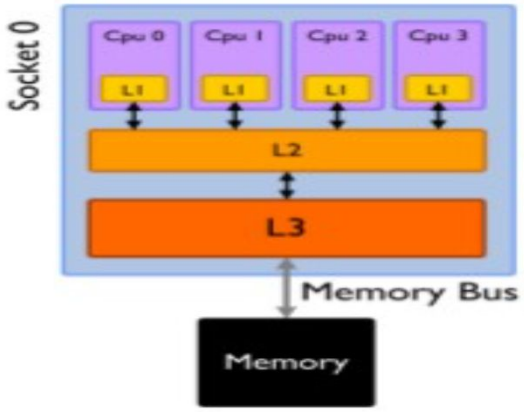
Στο ερώτημα αυτό θα αξιολογήσουμε την κλιμάκωση των διαφόρων υλοποιήσεων όταν τα νήματα εκτελούνται σε πυρήνες με διαφορετικά χαρακτηριστικά ως προς το διαμοιρασμό των πόρων.

Οι τιμές των πειραματικών παραμέτρων που θα χρησιμοποιηθούν είναι οι παρακάτω:

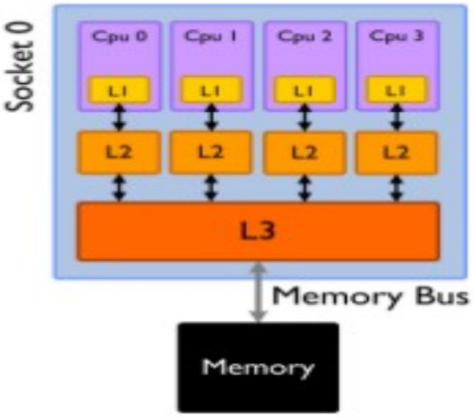
- εκδόσεις προγράμματος: TAS_CAS, TAS_TS, TTAS_CAS, TTAS_TS, MUTEX
- iterations: 1000
- nthreads: 4
- grain_size: 1

Οι τοπολογίες που θα μελετηθούν είναι οι παρακάτω:

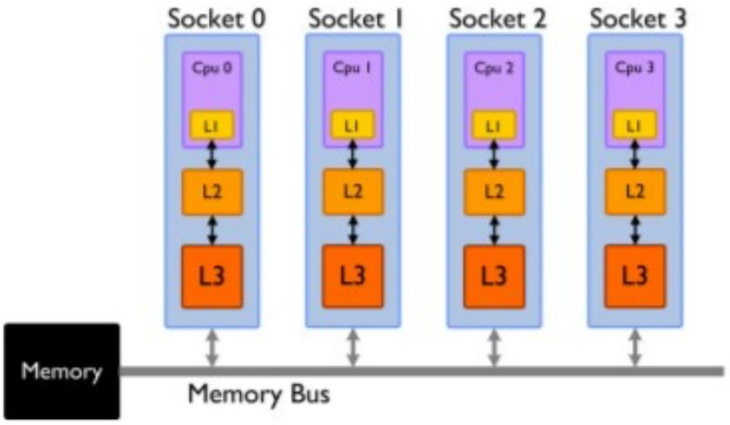
share-all: τα 4 νήματα βρίσκονται σε πυρήνες με κοινή L2 cache



share-L3: τα 4 νήματα βρίσκονται σε πυρήνες με κοινή L3 cache, αλλά όχι κοινή L2

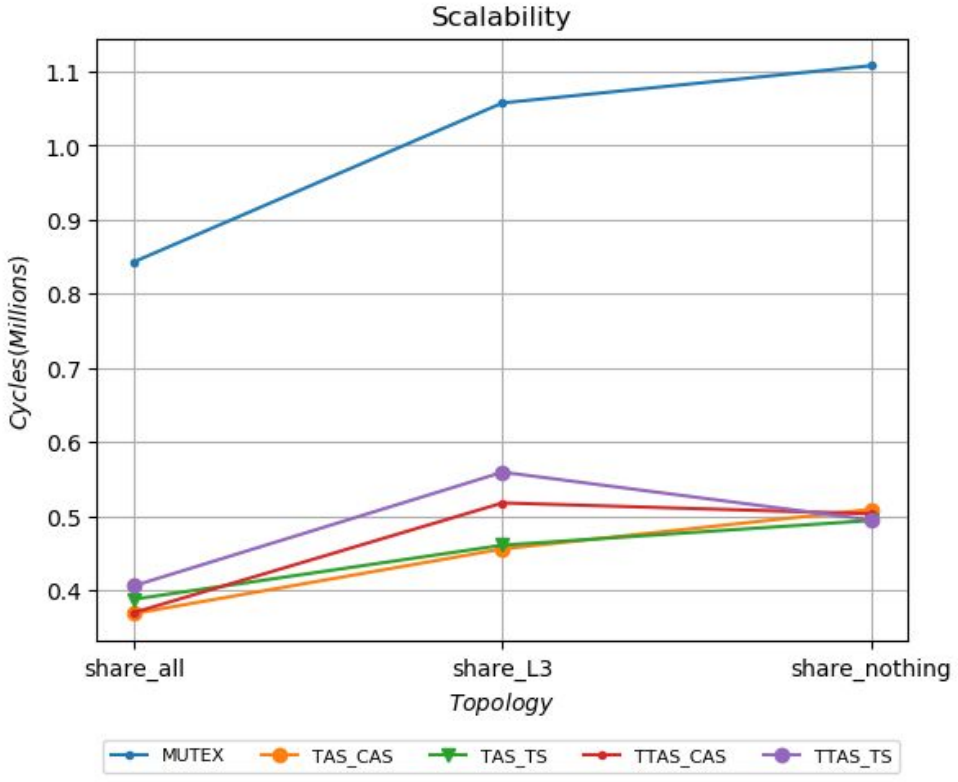


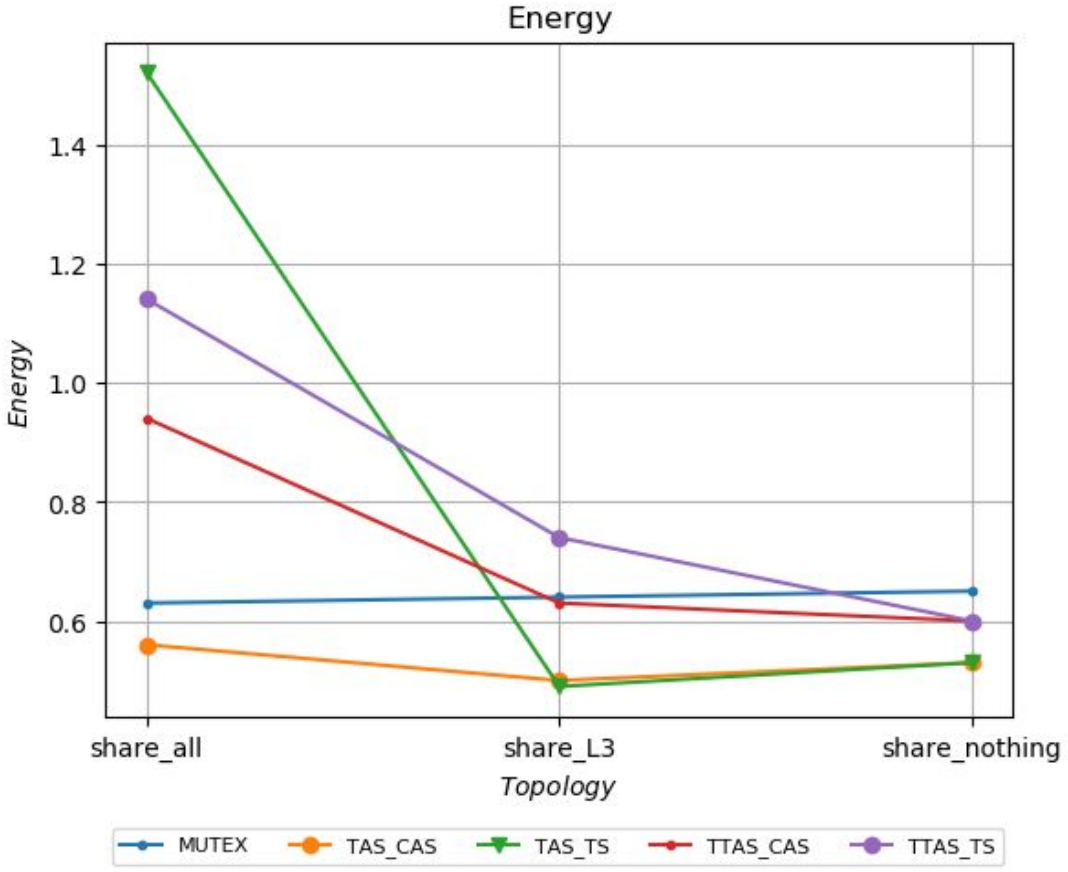
share-nothing:τα 4 νήματα βρίσκονται σε πυρήνες με διαφορετική L3 cache



Ερώτημα 3.2.1. Για τις παραπάνω τοπολογίες, δώστε σε ένα διάγραμμα το συνολικό χρόνο εκτέλεσης της περιοχής ενδιαφέροντος για όλες τις υλοποιήσεις. Χρησιμοποιώντας το McPAT συμπεριλάβετε στην αξιολόγηση σας και την κατανάλωση ενέργειας (Energy, EDP κτλ.). Τι συμπεράσματα βγάζετε για την απόδοση των μηχανισμών συγχρονισμού σε σχέση με την τοπολογία των νημάτων; Δικαιολογήστε τις απαντήσεις σας.

Τα αποτελέσματα συνοψίζονται παρακάτω στις γραφικές παραστάσεις, η πρώτη για τους κύκλους και η δεύτερη για την ενέργεια και η τρίτη για το EDP.





Συμπεράσματα

Κύκλους συστήματος:

Παρατηρούμε καταρχάς ότι η share-all τοπολογία νημάτων είναι πιο γρήγορη σε κάθε περίπτωση (λιγότερα cycles). Ένας παράγοντας που επιφέρει περισσότερους κύκλους στις υπόλοιπες τοπολογίες είναι ότι στις υπόλοιπες περιπτώσεις απαιτείται η μεταφορά του block του Lock μεταξύ των cache των επεξεργαστών.

Την χειρότερη επίδοση για τους busy-wait μηχανισμούς παρουσιάζει η share-L3 ενώ για το mutex έχουμε ότι την χειρότερη συμπεριφορά παρουσιάζει το share-nothing.

Ενέργεια:

Αρχικά παρατηρούμε ότι η mutex καταναλώνει λιγότερη ενέργεια σε share all τοπολογίες παρόλα αυτά η διαφορά της κατανάλωσης ενέργειας στην mutex στις διαφορες τοπολογίες είναι πολύ μικρή, η τεχνική TAS στην share-L3 ενώ η TTAS στην share-all.

Πιο ενεργειακά κοστοβόρα διάταξη για τις busy-wait είναι η share-all ενώ για την mutex είναι η share-nothing.

Σαν γενικό συμπέρασμα καταλήγουμε σε ένα trade off όπου βρίσκουμε ότι γενικά η πιο γρήγορη τοπολογία είναι η share-all η οποία όμως καταναλώνει και την περισσότερη ενέργεια.

Μέρος Β

Με βάση το δεδομένο συστημα που δίνεται από την εκφώνηση έχουμε την εξής εκτέλεση του Αλγορίθμου του Tomasulo:

	OP	IS	EX	WR	CMT	Comments
1	LD F0, 0(R1)	1	2-5	6	7	cache miss
2	ADDD F4, F4, F0	1	7-9	10	11	RAW F0
3	LD F1, 0(R2)	2	3-6	7	11	cache miss
4	MULD F4, F4, F1	2	11-15	16	17	RAW F1, F4
5	ANDI R9, R8, 0x2	3	4-5	8	17	CDB FULL (x2)
6	BNEZ R9, NEXT	3	9-10	11	18	Prediction T, RAW R9
7	LD F5, 8(R1)	7	8	9	x	LQ FULL, cache hit
8	ADDD F4, F4, F5	7	x	x	x	RAW F4,F5
9	ADDI R1, R1, 0x8	8	9-10	x	x	
10	SUBI R8, R8, 0x1	9	10-11	x	x	RS FULL
11	BNEZ R8, LOOP	x	x	x	x	ROB FULL
12	LD F2, 16(R2)	12	13-16	17	18	cache miss
13	MULD F2,F2,F5	12	18-22	23	24	FU FULL, RAW F2
14	ADDD F4, F4, F2	13	24-26	27	28	RAW F2,F4
15	LD F5, 8(R1)	13	14	15	28	cache hit
16	ADDD F4,F4,F5	14	28-30	31	32	RAW F4,F5
17	ADDI R1, R1, 0x8	14	15-16	18	32	
18	SUBI R8, R8, 0x1	18	19-20	21	33	ROB FULL
19	BNEZ R8, LOOP	18	22-23	24	33	ROB FULL, Prediction taken,RAW R8
20	LD F0, 0(R1)	19	20	22	x	cache hit, CDB FULL
21	ADDD F4, F4, F0	19	x	x	x	RAW F0, F4
22	LD F1, 0(R2)	x	x	x	x	ROB FULL
23	SD F4, 8(R2)	25	32-35	36	37	RAW F4, cache miss

Το τελικό περιεχόμενο της cache είναι το εξής:

A[0]	B[1]
A[1]	B[2]

Παράρτημα

```
#ifndef LOCK_H_
#define LOCK_H_

typedef volatile int spinlock_t;

#define UNLOCKED  0
#define LOCKED    1

static inline void spin_lock_init(spinlock_t *spin_var)
{
	*spin_var = UNLOCKED;
}

static inline void spin_lock_tas_cas(spinlock_t *spin_var)
{
	while (__sync_val_compare_and_swap(spin_var, UNLOCKED, LOCKED) == LOCKED);
}

static inline void spin_lock_ttas_cas(spinlock_t *spin_var)
{
	while (1) {
		while(*spin_var == LOCKED);

		if (__sync_val_compare_and_swap(spin_var, UNLOCKED, LOCKED) == UNLOCKED) break;
	}
}

static inline void spin_lock_tas_ts(spinlock_t *spin_var)
{
	while (__sync_lock_test_and_set(spin_var, LOCKED) == LOCKED);
}

static inline void spin_lock_ttas_ts(spinlock_t *spin_var)
{
	while (1) {
		while (*spin_var == LOCKED);

		if (__sync_lock_test_and_set(spin_var, LOCKED) == UNLOCKED) break;
	}
}

static inline void spin_unlock(spinlock_t *spin_var)
{
	__sync_lock_release(spin_var);
}


#endif
```