Ατομική Διπλωματική Εργασία

Dataflow Υλοποίηση της εφαρμογής OCEAN με την χρήση πλατφόρμας TFlux

Έλση Πραιτωρίτη

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Τίτλος Ατομικής Διπλωματικής Εργασίας

Έλση Πραιτωρίτη

Επιβλέπων Καθηγητής
Pedro Trancoso

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2013

Ευχαριστίες

Με την ολοκλήρωση των προπτυχιακών μου σπουδών και της ατομικής διπλωματικής εργασίας θα ήθελα να ευχαριστήσω όλους όσους συνέβαλαν σε όλο αυτό. Πρώτα από όλα θα ήθελα να ευχαριστήσω και να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα καθηγητή μου κ. Pedro Trancoso που μου έδωσε την ευκαιρία να ασχοληθώ με το θέμα αυτό. Ακολούθως να ευχαριστήσω τους διδακτορικούς φοιτητές Ανδρέα Διαβαστό και Κωνσταντίνο Χριστοφή για τον πολύτιμό χρόνο που μου αφιέρωσαν και καθώς και στην πολύτιμή βοήθεια τους.

Ένα ακόμη ευχαριστώ το χρωστάω στην μητέρα μου για την απεριόριστή συμπαράσταση και στήριξη κατά την διάρκεια των σπουδών μου. Και τέλος θα ήθελα να ευχαριστήσω τους λίγους και εκλεκτούς μου φίλους για την μεγάλη κατανόηση και στήριξη που μου έδωσαν και συνεχίζω να λαμβάνω .

Περίληψη

Η εφαρμογή Ocean είναι η προσομοίωση των ωκεάνιων ρευμάτων που προσομοιώνει την κίνηση των ρευμάτων του νερού στον ωκεανό. Αυτά τα ρεύματα αναπτύσσονται και να εξελίσσονται κάτω από την επίδραση διαφόρων φυσικών δυνάμεων, περιλαμβανομένων των ατμοσφαιρικών επιπτώσεων, τον άνεμο, και η τριβή με το πυθμένα του ωκεανού. Για λόγους απλότητας, ο ωκεανός μοντελοποιείται ως ένα ορθογώνιο σχήμα οπού τα σημεία του πλέγματος θεωρούνται ότι απέχουν ίσα , και οι εξισώσεις της κίνησης επιλύετε σε όλα τα σημεία του πλέγματος . Η εφαρμογή Ocean είναι ένας αλγόριθμός βασισμένος στο πρότυπο Wavefront το οποίο που εμφανίζετε σε επιστημονικές εφαρμογές . Τα στοιχεία των δεδομένων κατανέμονται σε πολυδιάστατα πλέγματα . Οι Wavefront τεχνικές χρησιμοποιούνται σε αλγορίθμους οι οποίοι έχουν επανεμφανίσεις με το σπάσιμο του υπολογισμού σε τμήματα και γίνετε εφαρμογή pipelining στα τμήματα αυτά μέσω πολλαπλών επεξεργαστών.

Η πλατφόρμα TFlux έχει ως στόχο την εκτέλεση προγραμμάτων χρησιμοποιώντας το Data-Driven Multithreading(DDM) μοντέλο . Πρόκειται για μια πλατφόρμα ανεξάρτητη της αρχιτεκτονικής και λειτουργικού συστήματος. Η λειτουργία της βασίζετε σε ένα προ-επεξεργαστή(Preprocessor), τον TFlux C Preprocessor, ο οποίος παίρνοντας ως είσοδο DDM προγράμματα , παράγει ως έξοδο αντίστοιχο C κώδικα για να εκτελεστεί σε οποιοδήποτε κοινό σύστημα. Βασικό συστατικό της πλατφόρμας αυτή είναι τα DTthreads , τα οποία για την χρονοδρομολόγηση αυτών καθώς επίσης και των υπολογισμών υπεύθυνο είναι η ξεχωριστή μονάδα συγχρονισμού Threads Synchronization Unit (TSU) που είναι υλοποιημένο είτε σε λογισμικό (software) είτε σε υλικό(hardware). Με την εργασία αυτή ασχοληθήκαμε για τις υλοποιήσεις μας μόνο με το TFluxSoft , στο οποία το TSU είναι υλοποιημένο σε λογισμικό.

Στόχος μας είναι με την βοήθεια της πλατφόρμας TFlux να εφαρμόσουμε data-dependencies στην εφαρμογή αυτή έτσι ώστε για να υπολογίζει ένα σημείο την τιμή του ,να γίνετε αφού τα γειτονικά σημεία στα οποία εξαρτάτε να έχουν ενημερωθεί έτσι ώστε το σημείο αυτό να χρησιμοποιεί σωστές και ανανεωμένες τιμές για τον υπολογισμό του. Πέρα από αυτό να παρατηρήσουμε την συμπεριφορά του και την επίδοση του συγκρινόμενου, με ένα μοντέλο παράλληλου προγραμματισμού , την OpenMP.Τα αποτελέσματα έδειξαν ότι η υλοποίηση με την πλατφόρμα TFlux σε γενικές γραμμές υπερίσχυσε της υλοποίησης με OpenMP. Αυτό γιατί όσο αυξανόταν ο αριθμός των blocks , τότε περισσότερα είχανε την ευκαιρία να τρέξουνε παράλληλα και να εκμεταλλευτούν τον παραλληλισμό.

Κεφάλαιο 1

Εισαγωγή

1.1 Παράλληλη Επεξεργασία και Αρχιτεκτονικές Επεξεργαστών

Η συνεισφορά των υπολογιστών αποτελεί σήμερα ανεκτίμητης αξίας μίας και μέσω αυτού μπορούν να γίνουν σχεδόν τα πάντα. Τι μπορούμε όμως να κάνουμε με γρήγορούς υπολογιστές . Μπορούμε να λύνουμε τα προβλήματα γρηγορότερα μιας και υπάρχει η δυνατότητα να μειωθεί ο χρόνος των μεγάλων εργασιών , καθώς και επίσης να πάρουμε καλύτερες λύσεις στο ίδιο χρονικό διάστημα κάνοντας τα μοντέλα μας πιο εξελιγμένα.

Ένας σημαντικός όρος αποτελεί ο παράλληλος υπολογισμός και σε αυτόν στηρίζονται όλες οι γλώσσες και οι βιβλιοθήκες που δημιουργηθήκαν για την ανάπτυξη του παραλλήλου προγραμματισμού. Σαν δεδομένο έχουμε ολόκληρο το σειριακό έργο. Αφού το διαιρέσουμε σε επιμέρους εργασίες τότε τις εκτελούμε ταυτόχρονα σε πολλούς πυρήνες . Η όλη διαδικασία για να επιτύχει , απαιτεί να κατανοήσουμε στο που ο παραλληλισμός μπορεί να είναι αποτελεσματικός και να έχουμε εις γνώση για το πώς σχεδιάζονται και υλοποιούνται καλές λύσεις. Σίγουρα αν θέλουμε να πάρουμε καλύτερη απόδοση από ένα σύγχρονο επεξεργαστή θα πρέπει να καταλάβουμε πολλά για την αρχιτεκτονική και τον μεταγλωττιστή.

Η μεθοδολογία που υπάρχει στο θέμα αυτό είναι ότι αφού έχουμε ξεκινήσει με την δήλωση ενός (σειριακού) προβλήματος ή ένα τμήμα κώδικα θα ψάξουμε για ανεξάρτητες εργασίες ή δραστηριότητες που μπορούν να εκτελεστούν παράλληλα. Στόχος είναι να κρατηθούν όλοι οι διαθέσιμοί επεξεργαστές απασχολημένοι με χρήσιμο έργο. Κρατώντας τους επεξεργαστές να κάνουν χρήσιμό έργο αυτό σημαίνει ελαχιστοποιεί τους περιττούς υπολογισμούς και τους μη χρήσιμούς εκσυγχρονισμούς μεταξύ των επεξεργαστών

Υπάρχουν αρκετοί τρόποι αξιολόγησης του παραλληλισμού όπως με αποσύνθεση πεδίου(Domain Decomposition),αποσύνθεση εργασίας(Task Decomposition) η ακόμη και διασωλήνωση(Pipeline).

Αποσύνθεση χώρου/πεδίου σημαίνει η διαίρεση των στοιχείων των δεδομένων , ώστε να χωρίζονται σε κομμάτια και στην συνέχεια να αναθέτουμε τα νήματα σε αυτά τα κομμάτια.
Στην προσέγγιση αποσύνθεση εργασίας(Task Decomposition) έχουμε την διαίρεση των διάφορών εργασιών που πρέπει να εκτελούνται μεταξύ των επεξεργαστών και στοχεύει για τις λειτουργίες που μπορούν να εκτελεστούν παράλληλα . Διασωλήνωση είναι ο τρίτος τρόπος για αξιοποίηση του παραλληλισμού σε ένα πρόβλημα. Η διασωλήνωση είναι ένα είδος της αποσύνθεσης έργου. Σε μια εφαρμογή διασωλήνωσης , η έξοδος της κάθε λειτουργίας είναι η είσοδος για την επόμενη λειτουργία. Η διασωλήνωση είναι δύσκολή , επειδή η απόδοση περιορίζεται από την πιο αργή φάση και έτσι αν όλα τα στάδια δεν τρέχουν με την ίδια ταχύτητα είναι μη αποτελεσματικό

Η σημερινή τάση που επικρατεί για την κατασκευή νέων επεξεργαστών είναι η αύξηση του αριθμού των επεξεργαστών (πανομοιότυπων ή διαφορετικών) που τοποθετούνται σε ένα chip. Και αυτό διότι η αρχική φιλοσοφία που επικρατούσε δηλαδή η αύξηση της συχνότητας και της πολυπλοκότητας σε ένα επεξεργαστή δεν μπορούσε να κρατήσει για πολύ. Ο λόγος είναι εκτός του ότι υπήρχε ανάγκη για κατανάλωση περισσότερης ενεργείας από τον επεξεργαστή με την αρχική φιλοσοφία που επικρατούσε , σαν συνεπακόλουθο πρόβλημα ήταν και η συντήρηση/ψύξη του συστήματος . Επίσης οι συχνότητες των απλών επεξεργαστών έχουν ένα άνω όριο το γνωστό «power wall» και έτσι δεν μπορούσε η συχνότητα να αυξανόταν περισσότερο από αυτό το όριο.
Με την σημερινή τάση με την οποία οι εταιρίες παραγωγής επεξεργαστών υιοθέτησαν , επιτυγχάνετε η βελτίωση της επίδοσης μιας και υπάρχουν διαθέσιμές περισσότερες υπολογιστικές μονάδες και ξεπέρασε τα προβλήματα της ψηλής κατανάλωσης ενέργειας και όλως των συνακολούθου της.
Σήμερα η νέα αυτή φιλοσοφία έχει για τα καλά εδραιωθεί πλέον στην κατασκευή πολυπύρηνων επεξεργαστών και όχι μόνο ,μιας και μεταφερόμαστε σιγά-σιγά στους many-cores επεξεργαστές οι οποίοι όπως είναι φυσικό διαθέτουν πολύ περισσότερους επεξεργαστές από τους πολυπύρηνους επεξεργαστές.

1.2 Η εφαρμογή OCEAN

Η εφαρμογή Ocean είναι ένας αλγόριθμός βασισμένος στο πρότυπο Wavefront[28],[33-35] . Wavefront είναι ένα μοτίβο προγραμματισμού που εμφανίζετε σε επιστημονικές εφαρμογές και σε αυτό το μοτίβο , τα στοιχεία των δεδομένων κατανέμονται σε πολυδιάστατα πλέγματα που αντιπροσωπεύουν ένα λογικό διάστημα. Τα στοιχεία θα πρέπει να υπολογίζονται σε σειρά επειδή υπάρχουν εξαρτήσεις μεταξύ τους.

Η εφαρμογή Ocean είναι η προσομοίωση των ωκεάνιων ρευμάτων που προσομοιώνει την κίνηση των ρευμάτων του νερού στον ωκεανό. Αυτά τα ρεύματα αναπτύσσονται και να εξελίσσονται κάτω από την επίδραση διαφόρων φυσικών δυνάμεων, περιλαμβανομένων των ατμοσφαιρικών επιπτώσεων, τον άνεμο, και η τριβή με το πυθμένα του ωκεανού. Για λόγους απλότητας, ο ωκεανός μοντελοποιείται ως ένα ορθογώνιο σχήμα όπου τα σημεία του πλέγματος θεωρείται ότι απέχουν ίσα και οι εξισώσεις της κίνησης επιλύετε σε όλα τα σημεία του πλέγματος σε ένα χρονικό βήμα ,η κατάσταση των μεταβλητών ενημερώνεται ως αποτέλεσμα, και οι εξισώσεις κίνησης επιλύονται πάλι για το επόμενο χρονικό βήμα, και ούτω καθεξής επανειλημμένα.

1.3 Φορητή Πλατφόρμα TFlux

Η πλατφόρμα TFlux έχει ως στόχο την εκτέλεση προγραμμάτων χρησιμοποιώντας το Data-Driven Multithreading (DDM) μοντέλο . Πρόκειται για μια πλατφόρμα απεξάρτηση της αρχιτεκτονικής και λειτουργικού συστήματος. Η λειτουργία της βασίζετε σε ένα προ-επεξεργαστή(Preprocessor), τον TFlux C Preprocessor, ο οποίος παίρνοντας ως είσοδο DDM προγράμματα και παράγει ως έξοδο αντίστοιχο C κώδικα για να εκτελεστεί σε οποιοδήποτε κοινό σύστημα. Βασικό συστατικό της πλατφόρμας αυτή είναι τα DTthreads , τα οποία για την χρονοδρομολόγηση αυτών καθώς επίσης και των υπολογισμών υπεύθυνο είναι η ξεχωριστή μονάδα συγχρονισμού Threads Synchronization Unit (TSU) που είναι υλοποιημένο είτε σε λογισμικό (software) είτε σε υλικό(hardware). Με την εργασία αυτή ασχοληθήκαμε για τις υλοποιήσεις μας μόνο με το TFluxSoft , στο οποίο το TSU είναι υλοποιημένο σε λογισμικό.

1.4 Κίνητρο

Η εφαρμογή OCEAN αποτέλεσε μια ενδιαφέρον εφαρμογή μιας και μελετήθηκε και μέσα από το μάθημα Παράλληλης Επεξεργασίας. Στο μάθημα είχαμε μελετήσει το πρόβλημα αυτό μέσα από μία άλλη προοπτική εφαρμόζοντας το αλγόριθμο Red-Black Gauss-Seidel . Ο αλγόριθμος αυτός είχε σαν στόχο να μειώσει το χρόνο υπολογισμού των κάθε σημείων στο πλέγμα. Τα σημεία εικονικά μοιράζονταν στα κόκκινα και μαύρα σημεία εναλλάξ μεταξύ τους και την ίδια στιγμή γίνονταν οι υπολογισμοί ,είτε στα μαύρα ,είτε στα κόκκινα αλλά όχι την ίδια στιγμή και στα δύο. Δηλαδή την μία γίνονταν οι υπολογισμοί στα κόκκινα που χρησιμοποιούσαν τις τιμές των μαύρων σημείων και μετά αντίστροφα και αυτό επαναλαμβάνονταν συνέχεια μέχρι να οι τιμές να συγκλίνουν κάποια στιγμή και να μην υπάρχουν μεγάλη διαφορά .

Η εφαρμογή όμως μπορούσε να λειτουργήσει και βλέποντας την με μία άλλη προοπτική. Σε αυτό θα βοηθούσε η πλατφόρμα TFlux που υπήρξε το κύριο κίνητρο στην διαφορετική λύση του προβλήματος αυτό, στο οποίο ήταν ένα πρόβλημα με καθαρό data-dependencies χαρακτήρα.
Μετατρέποντας το πλέγμα των σημείων σε blocks , εφαρμόσαμε τις εξαρτήσεις σε αυτά. Κάθε thread αναλάμβανε ένα block και είχε τις κατάλληλες εξαρτήσεις ανάλογα με το ποιο block αναλάμβανε σε σχέση με το ολικό πλέγμα. Όσα πιο πολλά blocks είχαμε , έδινε την δυνατότητα να εκμεταλλευτούμε καλύτερα τους πυρήνες μιας και η πλατφόρμα TFlux είχε μελετηθεί και είχε αποδείξει πως πετυχαίνει καλές επιδόσεις όταν χρησιμοποιούνται περισσότερες υπολογίστηκες μονάδες.

Θα ήταν πολύ ενδιαφέρον να καταφέρναμε την εφαρμογή αυτή με dataflow δρομολόγηση και να επιδιώξουμε διάμεσου της πλατφόρμας αυτής να έχουμε σημαντικές επιδόσεις.

1.5 Στόχος και Αναμενόμενα Αποτελέσματα

Ο δικός μας στόχος είναι να καταφέρουμε να επιλύσουμε το πρόβλημα της εφαρμογής OCEAN με την βοήθεια της πλατφόρμας TFlux, η οποία θα βοηθήσει το να λυθεί το πρόβλημα αυτό με χρήση dataflow δρομολόγηση λόγω του μοντέλου στο οποίο είναι βασισμένο.

Πρωταρχικός στόχος είναι πρώτα να έχουμε σωστά αποτελέσματα χρησιμοποιώντας την εφαρμογή TFlux και θα πρέπει να είναι τα ίδια με αυτά που θα είχαμε αν τα εκτελούσαμε σειριακά.

Πέρα από αυτό να δοκιμαστεί το πρόβλημα με διάφορα μεγέθη του πλέγματος των σημείων, ένα μικρό, μεσαίο και ένα μεγάλο και επίσης δοκιμάζοντας από 2 μέχρι 12 πυρήνες , οπού η μέγιστη υπολογιστική δυνατότητα της μηχανής στην οποία θα δοκιμάζονται είναι 12 επεξεργαστές.

Με το περάς αυτής της εργασίας και αφού έχουμε καταλήξει σε μια ορθή υλοποίηση χωρίς προβλήματα θα επικεντρωθούμε να δοκιμάσουμε στην πιθανή δυνατότητα αύξηση της επίδοσης καθώς σημαντικό και αναπόσπαστο κομμάτι είναι τα αποτελέσματα στα οποία θα πάρουμε καθώς και η σωστή και λογική δικαιολόγηση τους.

1.6 Οργάνωση και Μεθοδολογία

Έχουμε χωρίσει το έργο σε μια σειρά από βήματα με την προϋπόθεση ότι για να προχωρήσουμε στο επόμενο βήμα θα έπρεπε να είχαμε φέρει εις πέρας το προηγούμενο του. Να σημειώσουμε ότι μερικές από αυτές τις διαδικασίες που περιγράφονται στα βήματα ήταν επαναλαμβανόμενες μέχρι την ολοκλήρωση του έργου αφού ανά πάσα στιγμή μπορεί αν προέκυπταν βελτιστοποιήσεις.

Αρχικά έπρεπε να μελετήσουμε τα εγχειρίδια για την εφαρμογή καθώς και τις σχετικές δημοσιεύσεις γύρω από την εφαρμογή και της σουίτας στην οποία άνηκε , την SPLASH-2[7,8].Επίσης βοηθητικό ήταν και η μελέτη διάφορων άρθρων για την εφαρμογή OCEAN [9-10],[23]και τις παρατηρήσεις, βελτιστοποιήσεις καθώς και στα διάφορα μοντέλα παράλληλη επεξεργασίας που είχε δοκιμαστεί όπως επίσης και δημοσιεύσεων γύρων από το μοτίβο Wave-front στο οποίο στηρίζεται η εφαρμογή Ocean.

Αναγκαία ήταν και η μελέτη των άρθρων και του manual σχετικά με την πλατφόρμα TFlux[1,11,12,13], και του μοντέλου προγραμματισμού στο οποίο στηριζόταν , το DDM[3,4,5]. Στο manual υπήρχαν ήδη δοκιμασμένα παραδείγματα με την χρήση των TFlux directives που βοηθούσαν στην κατανόηση τους και την λογική που χρησιμοποιήθηκαν τα συγκεκριμένα .Επίσης μελετήσαμε άρθρα και διαφορές άλλε πηγές για το μοντέλο προγραμματισμού OpenMP για την κατανόηση του και τον τρόπο λειτουργίας του[14-23].

Βοηθητικά , επίσης , στάθηκαν και τα διάφορα DDM benchmarks, τα οποία εκτελέσαμε τόσο για να δούμε το πώς λειτουργούν και τα αποτελέσματα που έδινε στο τέλος αλλά και την μελέτη των directives χρησιμοποιήθηκε στην κάθε περίπτωση και γιατί. Αφού δοκιμάσαμε τα διάφορα DDM benchmarks και αφού μετά από βαθιά μελέτη και κατανόηση στην σωστή χρήση των directives σε ένα πηγαίο κώδικα, τα εφαρμόσαμε πιο πάνω στην εφαρμογή του OCEAN.

Στην αρχή δημιουργήσαμε υλοποίησης σε TFlux και OpenMP με μικρό μέγεθος πλέγματός.Στην υλοποίηση με TFlux το κάθε σημείο στο πλέγμα, αναλαμβάνονταν από ένα ξεχωριστό DThread και έτσι εφαρμόσαμε σε αυτά τις ακριβές εξαρτήσεις που επέβαλε ο

αλγόριθμος για να είναι σωστές οι τιμές. Στην υλοποίηση με OpenMP , ο αλγόριθμος και εδώ βασίζετε στις ακριβές εξαρτήσεις για να υπάρχει μια συνοχή. Αφού βεβαιωθήκαμε ότι τα αποτελέσματα ήτανε σωστά το επόμενο βήμα είναι να κάναμε αυτό σε μεγάλα size οπού θα μας επέτρεπε σε σύγκριση με πριν ,το μέγεθος του block να είναι μεγαλύτερο. Το βήμα αυτό ίσως να ήταν το δυσκολότερο , λόγω του ότι όσα blocks δημιουργούσαμε θα έπρεπε να αναλαμβάνονταν από ένα DThread με μοναδικό αναγνωριστικό, ξεχωριστά για το κάθε ένα ,και όλη αυτή η διαδικασία έγινε για όλα τα μεγέθη που δοκιμάστηκε με ‘ manual ‘ τρόπο αφού όταν είχαμε 16x16 blocks , είχαμε ουσιαστικά εκατοσαραντατέσσερα(256) threads . Έπρεπε να γινόταν προσεκτικά η δημιουργία τους , καθώς θα έπρεπε να άρχιζε και να τελείωνε στο σωστό σημείο η εκτέλεση του κάθε thread όπως επίσης να γινόταν προσθήκη η λίστα με το/τα thread/ threads στα οποία υπήρχε εξάρτηση όπως επίσης και τον αριθμό του πυρήνα που θα το εκτελούσε. Ακολούθως στο επόμενο βήμα ήτανε, ο ίδιος αλγόριθμός με τις ακριβές συναρτήσεις να υλοποιηθεί σε OpenMP με τα ίδια μεγέθη του πλέγματος όπως ακριβώς προηγουμένως για να είναι συγκρίσιμα. Τέλος συλλέξαμε τα αποτελέσματα από τις εκτελέσεις και τις παραλλαγές του , αναλύοντας και παρουσιάζοντας τους χρόνους εκτέλεσης και τις επιδόσεις σε γραφικές παραστάσεις.

1.7 Περιγραφή Δομής Ατομικής Διπλωματικής Εργασίας

Η εργασία αυτή είναι χωρισμένη σε οκτώ(8) κεφάλαια. Το κεφάλαιο 1 αποτελείτε από την εισαγωγή και στόχος είναι να δώσει στον αναγνώστη μια πρώτη εικόνα για το πρόβλημα ώστε να γίνει κατανοητό . Γίνεται μια μικρή αναφορά για την παράλληλη επεξεργασία , τους μεθόδους της και τις αρχιτεκτονικές επεξεργαστών. Κατόπιν αναφέρετε το κίνητρο για αυτή την εργασία και την μεθοδολογία που ακολουθήθηκε για την εκπλήρωση του. Στο Κεφαλαίο 2 περιγράφετε η σχετική δουλεία που έχει γίνει για τα θέματα που θα αναφερθούν , έτσι όπως τα έχουμε μελετήσει μέσα από τις διάφορες πηγές που αναγράφονται στην βιβλιογραφία .Στο Κεφαλαίο 3 αναλύετε με λεπτομέρεια η εφαρμογή OCEAN και ο αλγόριθμος του καθώς γίνετε η περιγραφή του αλγόριθμός Red-Black Gauss-Seidel και του αλγορίθμου με τις ακριβείς εξαρτήσεις καθώς επίσης την απλή υλοποίηση του που αναπτύξαμε με την πλατφόρμα TFlux και με το μοντέλο OpenMP. Στο Κεφάλαιο 4 μελετάτε η φορητή πλατφόρμας TFlux με το μοντέλο προγραμματισμού DDM ,τα βασικά συστατικά εκτέλεσης DDM εφαρμογής καθώς επίσης και το προγραμματιστικό μοντέλο OpenMP με τα βασικά του χαρακτηρίστηκα, την σύνταξη και τις συναρτήσεις που διαθέτει. Στο Κεφάλαιο 5 γίνετε μια αξιολόγηση των πειραμάτων που υλοποιήθηκαν και παρουσιάζονται οι γραφικές παραστάσεις με τα αποτελέσματα που έχουν παρθεί για τους χρόνους εκτελέσεις και τις επιδόσεις τους. Τέλος με το Κεφαλαίο 6 κλείνουμε την εργασία αυτή κάνοντας μια αναφορά στα συμπεράσματα και τις γνώσεις που έχουν αποκομιστεί καθ’ όλη τη διάρκεια του έργου , καθώς και στην μελλοντική δουλεία που μπορεί να γίνει για την βελτίωση του.

Κεφάλαιο 2

Σχετική Δουλεία

Για να είμαστε σε θέση να ξεκινήσουμε την υλοποίηση των στόχων μας θα έπρεπε πρωτίστως να μελετήσουμε κάποια σχετικά συγγράμματα που αφορούν τους τομείς με τους οποίους θα ασχοληθούμε , έτσι ώστε να φτιάξουμε μια όσο το δυνατό πιο πλήρη και σαφή εικόνα στο μυαλό μας και να έχουμε μία δυνατή βάση γνώσεων για να στηριχθούμε για την υλοποίηση του έργου μας. Για αυτό τον λόγο έχουν μελετηθεί κάποια επιλεκτικά άρθρα με θέματα που αποτελούν κλειδί στο έργο αυτό. Τα άρθρα αυτά αφορούν κυρίως μελέτες για την πλατφόρμα TFlux[1,2] , το μοντέλο DDM[3,4,5] , τον αλγόριθμό OCEAN[8,9] , και το μοντέλο παράλληλου προγραμματισμού OpenMP[13,14,15,16,17,19] και καταγράφονται στην βιβλιογραφία που βρίσκετε στο τελευταίο τμήμα της συγκεκριμένης μελέτης.

Η όλη ιδέα της πλατφόρμας TFlux βασίζετε στο παράλληλο μοντέλο προγραμματισμού και εκτέλεσης Data-Driven Multithreading(DDM) το οποίο μπορεί να πετύχει καλύτερη επίδοση από μια σειριακή εκτέλεση. Η γενική ιδέα του DDM λέει πως κάποιο thread επιτρέπετε να ξεκινήσει την εκτέλεση αν και μόνο αν τα δεδομένα που χρειάζεται είναι έτοιμα και βρίσκονται στην μνήμη. Με τον τρόπο αυτό οι διάφορες καθυστερήσεις υπολογισμού και συγχρονισμού ελαχιστοποιούνται .Ένας κύριος στόχος του DDM είναι να εκμεταλλευθεί την καθυστέρηση επιτρέποντας τον επεξεργαστή που είναι για τους υπολογισμούς να εκτελέσει παραγωγική δουλεία καθώς την ίδια ώρα από πίσω υπάρχει η καθυστέρηση. Αποτελέσματα έδειξαν πως το DDM μπορεί να εκμεταλλευθεί πραγματικά την καθυστέρηση επικοινωνίας και συγχρονισμού.
Παρόλα αυτά ο χρονοπρογραμματισμός βασίζετε στην διαθεσιμότητα των δεδομένων και μπορεί να έχει αρνητική επίδραση στην τοπικότητα δημιουργώντας ακανόνιστες προσβάσεις μνήμης και για αυτό τον λόγο μπορεί να εκμεταλλευτεί κάποιες πολιτικές οι οποίες είναι οι CacheFlow. Η βασική υλοποίηση των πολιτικών αυτών είναι να εφαρμόζει hardware prefetching στο να κάνει prefetching στην L2 Cache τα δεδομένα που θα χρειαστούν τα threads που θα εκτελεστούν σύντομα, δηλαδή τα δεδομένα που χρειάζεται κάποιο thread φορτώνονται στην cache πριν το thread δρομολογηθεί για εκτέλεση.

Με το DDM μοντέλο ο επεξεργαστής αποκρύπτει τις μεγάλες καθυστερήσεις αναμονής εκτελώντας κάποιο άλλο κομμάτι δουλείας και έτσι εκμεταλλεύεται σε μεγάλο βαθμό τον

παραλληλισμό της εφαρμογής. Μια εφαρμογή σε TFlux αποτελείτε από διάφορα blocks , που το κάθε ένα από τα οποία συμπεριλαμβάνει διάφορα threads . Κάθε block περιέχει το Inlet Thread και το Outlet Thread .Για το χρονοπρογραμματισμό των threads υπεύθυνο είναι το TSU (Thread Synnchronization Unit) το οποίο μπορεί να υλοποιηθεί είτε σαν hardware είτε σαν software μονάδα . Τα threads μέσα σε ένα block εκτελούνται παράλληλα αν επιτρέπετε από το TSU και μπορούν να εκτελεστούν με διαφορετική σειρά από την οποία είναι γραμμένα, ενώ σε σύγκριση με τα blocks , αυτά δρομολογούνται και συγχρονίζονται από το TSU με την σειρά με την οποία είναι γραμμένα.

Με βάση το SPLASH-2 , το application Ocean μελετά της μεγάλης κλίμακας κινήσεις του ωκεανού με βάση τα όρια και τα ρεύματα και είναι ήδη μια βελτιωμένη έκδοση του ίδιου application στην προηγούμενη του έκδοση. Υπάρχουν 3 κύριες διαφορές : (1) τα grids διαχωρίζονται σε ομάδες στηλών για βελτίωση της αναλογίας επικοινωνίας προς υπολογισμού (2) τα grids παρουσιάζονται σε 4-D πίνακες (3) και τέλος χρησιμοποιείτε ο αλγόριθμός red-black Gauss-Seidel [24-27] στην θέση του SOR. Εδώ το application υπάρχει σε δύο υλοποιήσεις σε contiguous και non contiguous . Χρήσιμες πληροφορίες για την υλοποίηση αυτή , μελετήθηκαν σε διάφορα συγγράμματα , papers που ανάλυαν τα Benchmarks της σουίτας SPLASH-2 [8,30]. Μάλιστα έρευνες που έγιναν στην εφαρμογή αυτή με την συγκεκριμένη σουίτα, απόδειξε πως μπορούσε ο κώδικας να βελτιωθεί πολύ περισσότερο αν άλλαζαν μερικές γραμμές κώδικα , οι οποίες γραμμές πι συγκεκριμένες ήταν κλειδαριές μέσα στο κώδικα για το συγχρονισμό μεταβλητών που ήτανε global [29]. Η σουίτα αυτή ήταν ευρέως γνώστη και διαθέσιμη σε όλους όσους ενδιαφέρονται να τις χρησιμοποιήσουν στα πειράματα τους. Χρησιμοποιήθηκαν είτε σε κανονικούς επεξεργαστές είτε σε προσομοιωτές ή σε οτιδήποτε άλλο [32,33,34].
Επίσης η εφαρμογή Ocean όπως και άλλες εφαρμογές και αλγορίθμους στηρίζετε στο πρότυπο Wavefront[34,35] το οποίο είναι ένα μοτίβο προγραμματισμού που εμφανίζετε σε επιστημονικές εφαρμογές και σε αυτό το μοτίβο , τα στοιχεία των δεδομένων κατανέμονται σε πολυδιάστατα πλέγματα που αντιπροσωπεύουν ένα λογικό διάστημα. Τα στοιχεία θα πρέπει να υπολογίζονται σε σειρά επειδή υπάρχουν εξαρτήσεις μεταξύ τους.

Σκοπός μας ήταν να μελετήσουμε το application Ocean πέρα από την σουίτα SPLASH-2 έτσι ώστε να χρησιμοποιηθεί ο αλγόριθμος με τις ακριβές συναρτήσεις και όχι με το red-black Gauss-Seidel που είναι υλοποιημένος στην σουίτα SPLASH-2. Εδώ είναι που με την βοήθεια τη πλατφόρμας TFlux θα χρησιμοποιηθεί ο αλγόριθμος με τις ακριβές συναρτήσεις , γιατί χάρης των ειδικών directive’s που μας προσφέρει μπορούμε να χρησιμοποιούμαι data-dependencies μεταξύ των σημείων και έτσι το κάθε σημείο , όταν θα υπολογίζετε η τιμή του να παίρνει τις πραγματικές τιμές και να γίνετε μια φορά και όχι επαναληπτικά όπως σε αντίθεση ο προηγούμενος αλγόριθμός.

Πέρα από αυτό μελετήθηκε το ίδιο πρόβλημα με το μοντέλο OpenMP , όπου εκεί η φιλοσοφία είναι να διαιρεθεί το foor loop που υπήρχε για τον υπολογισμό των σημείων του πλέγματος στα threads ανάλογα με τον αριθμό με τον οποίο δηλώναμε τα threads. Για το σκοπό αυτό μελετήσαμε την σύνταξη και την ιδέα του OpenMP μέσα από διάφορα συγγράμματα και tutorias.

Η Διαφορετικότητα των 2 μοντέλων ήταν αυτό που θα μας βοηθήσει να αναπτύξουμε τα συμπεράσματα μας για την TFlux εφαρμογή και κατά πόσο είναι καλύτερη και πιο αποδοτική από το μοντέλο OpenMP. Οι δύο υλοποιήσεις μεταξύ τους σίγουρα έχουνε αρκετές διάφορες αλλά η κυριότερη είναι ότι με το TFlux υλοποιούμε το application Ocean με data-dependencies ο οποίος ήτανε και ο στόχος μας , ενώ με το OpenMP γίνετε ένα load-balancing μεταξύ των threads.
Όλα αυτά θα μας βοηθήσουν να δημιουργήσουμε μια καλή γνώση για το έργο που έχουμε να διεκπεραιώσουμε φτάνοντας στο μέγιστο των δυνατοτήτων μας.

Κεφάλαιο 3

Προγραμματιστικό Μοντέλο OpenMp & Φορητή Πλατφόρμα TFlux

3.1.1 Τι είναι το μοντέλο OpenMP

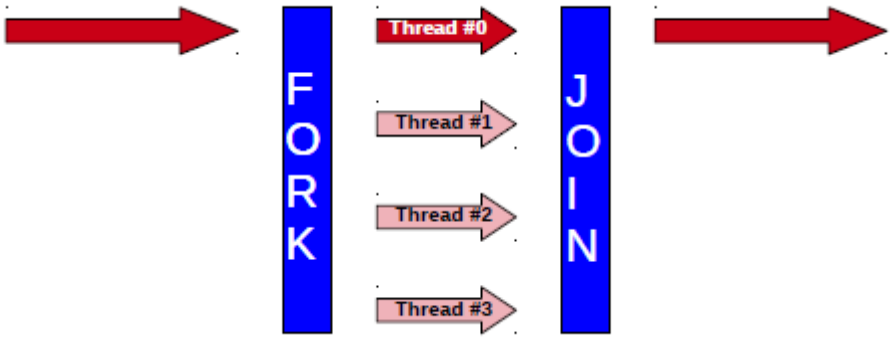
OpenMP είναι ένα φορητό επεκτάσιμο μοντέλο με απλό και ευέλικτο περιβάλλον για ανάπτυξη παράλληλων εφαρμογών. Είναι ένα πρότυπο που καθορίζει ένα σύνολο από εντολές για τον μεταφραστή , οι οποίες προστίθενται στο πηγαίο κώδικα ενός προγράμματος το οποίο έχει γραφτή έχοντας υπόψη την σειριακή εκτέλεση. Είναι μια ρητή παράλληλη προσέγγιση και έτσι ο compiler δεν μαντεύει το πώς να εκμετάλλευση το συγχρονισμό και σχετικά ένας εύκολος τρόπος να πάρουμε παραλληλισμό σε μηχανές με κοινή μνήμη.
Είναι μια συλλογή από directives του compiler και library συναρτήσεων που χρησιμοποιούνται για να δημιουργήσουμε παράλληλα προγράμματα σε υπολογιστές κοινής μνήμης.

Πριν την εμφάνιση του OpenMP, δεν υπήρχε τυποποιημένος τρόπος για να παραλληλοποιήσει κάποιος εύκολα σε περιβάλλον πολυεπεξεργαστών κοινή μνήμης και ήρθε για να αλλάξει την κατάσταση αυτή. Το OpenMP είναι ένα ευρέως αποδεκτό πρότυπο αφού προγράμματα που αναπτύσσονται βάση αυτού του προτύπου είναι μεταφέρσιμα σε μια μεγάλη ποικιλία συστημάτων πολυεπεξεργαστών κοινής μνήμης . Ακόμη το OpenMP δεν εξαναγκάζει τον προγραμματιστή να αλλάξει το στυλ προγραμματισμού του και η διαδικασία παραλληλοποιήσεις είναι καθοδηγούμενη από τον χρήση έχοντας πλήρη έλεγχο στο τι θα παραλληλοποιήθει και πως.

Στόχος του OpenMP ήταν να γίνει ευκολότερο για τους προγραμματιστές να μετατρέψουν ένα πρόγραμμα single-threaded σε multithreaded. Οι δύο βασικές έννοιες/ κλειδιά είναι τα αποτελέσματα να είναι τα ίδια με ένα η με πολλά νήματα και η αύξηση του παραλληλισμού με το είδος του παράλληλου προγραμματισμού που θα εξελίσσει ένα σειριακό πρόγραμμα σε παράλληλο.

3.1.2 Μοντέλο εκτέλεσης

Το OpenMP χρησιμοποιεί το μοντέλο παράλληλης εκτέλεσης fork-join. Το μοντέλο αυτό επιτρέπει να γράφονται προγράμματα που εκμεταλλεύονται πολλαπλά νήματα εκτέλεσης (threads).
Προγράμματα γραμμένα με OpenMP ξεκινούν με ένα thread το master thread (thread 0) και στην αρχή της παράλληλης περιοχής το master thread δημιουργεί μια ομάδα από παράλληλα threads (FORK). Οι εντολές σε ένα παράλληλο μπλοκ εκτελούνται παράλληλα από κάθε thread. Στο τέλος της παράλληλου μπλοκ , όλα τα threads συγχρονίζονται και κάνουν join με το master thread και από εκεί και πέρα μόνο το master thread συνεχίζει να εκτελείτε μέχρι να βρεθεί επόμενη παράλληλη περιοχή. (Σχήμα 3.1)



Σχήμα 3.1 : Fork Join Μοντέλο

3.1.3 Εντολές

Το OpenMP χρησιμοποιεί τις δηλώσεις #pragma για την επέκταση του μεταφραστή της C.

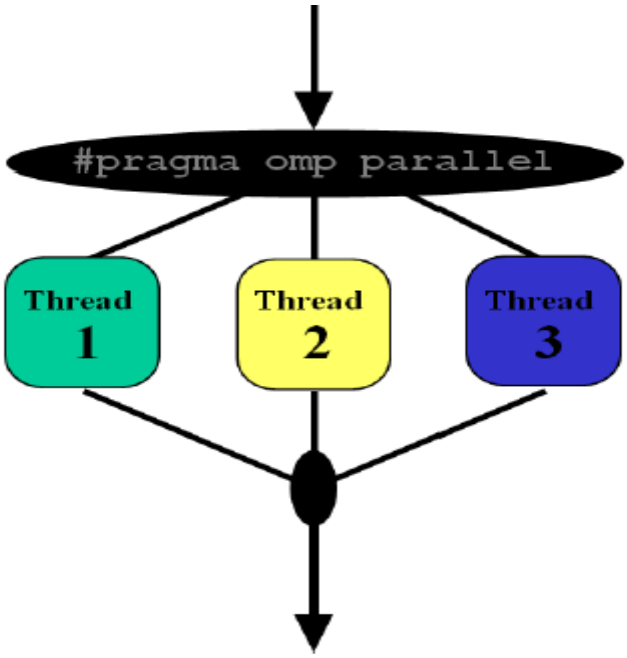
3.1.3.1 parallel

Η εντολή parallel δηλώνει μια περιοχή , η οποία εκτελείται από πολλαπλά νήματα . Αυτή είναι η βασική εντολή που εκκινεί την παράλληλη εκτέλεση δημιουργώντας τα νήματα. Η σύνταξή της εντολής είναι :

```
#pragma omp parallel [clause[ clause] …]
structured-block
```

οπού το clause μπορεί να είναι ένα από τα εξής : if (scalar-expression),private (list),firstprivate (list),default (shared j none),shared (list),reduction (operator : list).

Μόλις ένα νήμα συναντήσει μια παράλληλη περιοχή (δηλαδή μόλις εκτελεστή το #pragma omp parallel), δημιουργείτε μια ομάδα από νήματα. Τα νήματα αυτά εκτελούνται μαζί με το κυρίως νήμα και σταματάνε με το τέλος της παράλληλης περιοχής. (υπάρχει ένας έμμεσος barrier).(Σχήμα 3.2)

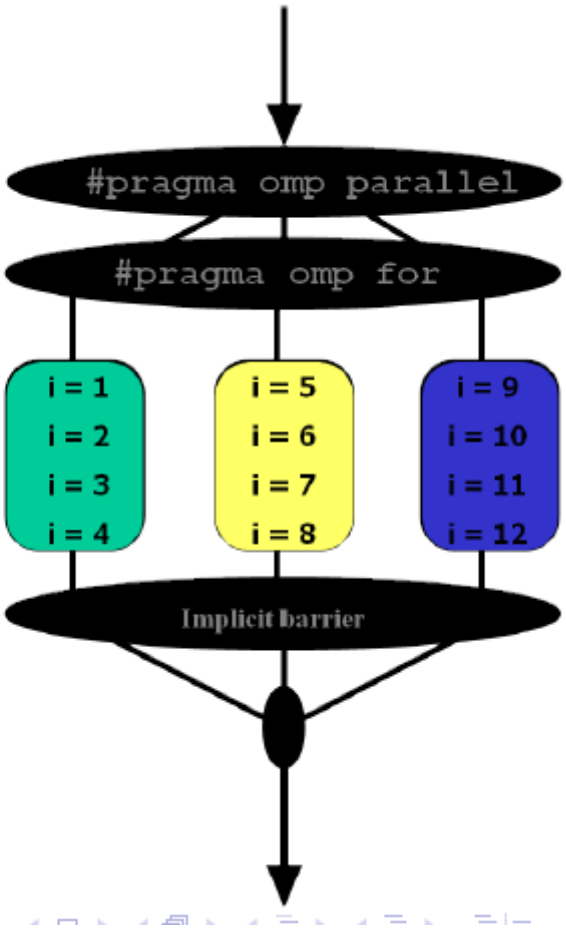


Σχήμα 3.2: Λειτουργία του #pragma omp parallel

3.1.3.2 for

Η εντολή for υποδηλώνει μια περιοχή η οποία περιλαμβάνει έναν βρόγχο του οποίου οι εντολές εκτελούνται παράλληλα. Αυτό επιτυγχάνετε με το να προσθέσουμε το #pragma omp for κατω από το #pragma omp parallel η μπορώντας να συνδυάσουμε τα 2 αυτά και να έχουμε #pragma omp parallel for πάνω από ένα for loop.

Το συγκεκριμένο pragma χρησιμοποιείται για να αναθέσει σε κάθε νήμα ένα ανεξάρτητο σετ από της επαναλήψεις του for loop και εδώ επίσης θα πρέπει τα νήματα να περιμένουν με το τέλος του for loop μιας και υπάρχει νοητά ένας barrier.(Σχήμα 3.3)



Σχήμα 33 : Λειτουργία #pragma omp parallel ακολουθούμενο από #pragma omp for

Η σύνταξή της εντολής είναι:

```
#pragma omp for [clause[ clause] …]
for-loop
```

Το clause μπορεί να είναι ένα από τα ακόλουθα:

private (list), firstprivate (list),lastprivate (list),reduction (operator : list),schedule (kind[; chunk size])

3.1.3.3 sections

Η εντολή sections περιγράφει ένα σύνολο από ενότητες οι οποίες θα διαμοιραστούν ανάμεσα στα νήματα. Κάθε ενότητα εκτελείται μόνο μια φορά από ένα τυχαίο νήμα.

Η σύνταξή της εντολής είναι :

#pragma omp sections [clause[clause] ….]
{
[#pragma omp section]
structured-block
[#pragma omp section]
structured-block
.
.
.]
}

Σε αυτή την περίπτωση ένα νήμα αναλαμβάνει μια ενότητα μόνο και αν περισσέψουν ενότητες στο τέλος εκτελούνται οποία από τα νήματα τελειώσουν γρηγορότερα ενώ στην αντίθεση περίπτωση αν ο αριθμός των νημάτων υπερισχύσει τον ενοτήτων τότε όσα νήματα περισσέψουν χωρίς να αναλάβουν κάποια ενότητα τότε απλά περιμένουν .

3.1.3.4 single

Η εντολή single καθορίζει ότι η σχετική περιοχή εκτελείται μόνο από ένα νήμα της ομάδας , τυχαίο και όχι απαραίτητα το κυρίως νήμα. Η σύνταξή είναι η ακόλουθη

```
#pragma omp single [clause[ clause] …]
    structured-block
```

3.1.4Συγχρονισμός

Το OpenMP παρέχει κάποιες εντολές που αφορούν την ατομικότητα πράξεων , εντολές συγχρονισμού και εντολές που κάνουν σειριακά κάποια κομμάτια κώδικα.

3.1.4.1 master

Η εντολή master καθορίζει ότι η σχετική περιοχή του κώδικα θα εκτελεστεί μόνο από το κυρίως νήμα και τα υπόλοιπα νήματα θα πρέπει να περιμένουν να τελειώσει την εκτέλεση του και επίσης η εντολή αυτή να είναι τοποθετημένη σε παράλληλη περιοχή. Η σύνταξή είναι :

```
#pragma omp master
  structured-block
```

3.1.4.2 critical

Η εντολή critical ορίζει μια κρίσιμη περιοχή , οπού μέσα σε αύτη μπορεί να εισέρθει ένα-ένα νήμα ανά πάσα χρονική στιγμή. Χαρακτηριστικό αναγνωριστικό των κρίσιμων περιοχών αυτών ,είναι το όνομα τους . Περιοχές οι οποίες δεν έχουν όνομα είναι by default NULL.

3.1.4.3 barrier

Η εντολή barrier συγχρονίζει τα νήματα μιας ομάδας και μόλις ένα νήμα συναντήσει την εντολή αυτή περιμένει μέχρι και τα υπόλοιπα να φτάσουν στον ίδιο σημείο. Η σύνταξη είναι :

#pragma omp barrier

3.1.4.4atomic

Η εντολή atomic χρησιμοποιείτε όταν μια συγκεκριμένη θέση της μνήμης θα πρέπει να ενημερώνεται ατομικά και αδιαίρετα αντί να εκτίθεται σε ταυτόχρονή εγγραφή από πολλά νήματα. Επίσης χρησιμοποιείτε όταν έχουμε απλές πράξεις μία από τις ακόλουθες μορφές :

x+ = expr

x + +

+ + x

x --

-- x

3.1.5 Περιβάλλον και ορατότητα δεδομένων

Μερικές εντολές όπως είδαμε πιο πάνω δέχονται παραμέτρους που έχουν σχέση με την ορατότητα των μεταβλητών και η οποία καθορίζετε από το χρήστη.

private(list)

Η δήλωσή private καθορίζει ότι όσες μεταβλητές που δηλώνονται μέσα στην συγκεκριμένη λίστα είναι ιδιωτικές σε κάθε νήμα δηλαδή υπάρχει αντίγραφο της ίδιας μεταβλητής στο προσωπικό χώρο των νημάτων και η αρχική τους τιμή δεν είναι καθορισμένη.

firstprivate(list)

Η δήλωσή αυτή έχει την ίδια λειτουργικότητα με την private με την διαφορά πως εδώ οι μεταβλητές έχουν αρχική τιμή και είναι η τιμή πριν την εντολή αυτή.

lastprivate(list)

Η δήλωση αυτή έχει την ίδια λειτουργικότητα με την private με την διαφορά πως οι private μεταβλητές αντιγράφουν την τελευταία τελική τιμή που παίρνουν στην αρχική μεταβλητή.

shared(list)

Η δήλωση shared ορίζει ότι οι μεταβλητές που δηλώνονται στην λίστα είναι κοινές και διαμοιραζόμενες και ορατές σε όλα τα νήματα.

reduction(op : list)

Η δήλωση αυτή ορίζει υποβίβαση στις μεταβλητές που εμφανίζονται στην λίστα . Οι μεταβλητές αυτές είναι κοινές και πάνω σε αυτές εφαρμόζετε ο τελεστής op. Ο τελεστής op μπορεί να είναι ένας από τους εξής : +,*,-,&,|,^,&&,||.Ακολουθεί ένα παράδειγμα :

```
x = 0;
#pragma omp parallel
#pragma omp for reduction (+:x)
for(i = 0; i < 10; i++)
x = x + 5;
```

Η μεταβλητή x ορίζετε μετά την εντολή #pragma omp for reduction (+:x) τοπική σε κάθε νήμα και έτσι το κάθε νήμα έχει το μερικό άθροισμα αναλόγως με τις πόσες επαναλήψεις εκτέλεσε από τις συνολικές.

Στο τέλος οι τοπικές μεταβλητές x από όλα τα νήματα προστίθενται στην κοινή μεταβλητή x και έτσι στο τέλος θα έχει την σωστή τιμή .

3.2.1 Προγραμματιστικό Μοντέλο DDM

Το Data-Driven Multithreading (DDM) είναι ένα μοντέλο εκτέλεσης με το οποίο επιτυγχάνει την αποδοτική εκτέλεση παράλληλων εφαρμογών με την χρήση dataflow δρομολόγησης (dataflow scheduling). Τα DDM προγράμματα αποτελούνται από μικρότερα μη επικαλυπτόμενα κομμάτια τα οποία ονομάζονται Data-Driven Threads (DThreads) τα οποία μπορούν να έχουν οποιοδήποτε μέγεθός .

Τα DThreads μπορούν να είναι ανεξάρτητα μεταξύ τους , είτε να εξαρτώνται μεταξύ τους λόγω κοινών δεδομένων , γεγονός που τους κάνει να έχουν σχέσεις producer/consumer. Οι εξαρτήσεις μεταξύ των DThreads σε ένα πρόγραμμα εκφράζονται από το γράφο συγχρονισμού που δημιουργείτε (synchronization graph), στο οποίο γράφο οι κόμβοι του αντιστοιχούν στα DThreads και οι ακμές στις εξαρτήσεις των DThreads που στην ουσία είναι στις εξαρτήσεις δεδομένων. Ο γράφος αυτός φορτώνετε σε μια μονάδα , το thread scheduler το οποίο έχοντας σαν βάση του και δεδομένο το γράφο αυτό , μπορεί να δρομολόγηση την εκτέλεση του κάθε DThread με την σωστή σειρά.

Το scheduling ενός DThread γίνετε με δυναμικό data-driven τρόπο , οπού ένα DThread θα αρχίσει την εκτέλεση του μόνο όταν όλοι producers του έχούν ολοκληρώσουν την εκτέλεση τους. Αυτό επιτυγχάνετε με ένα μετρητή που διατηρεί ο thread scheduler για κάθε DThread , ο οποίος ονομάζετε ReadyCounter , και ο οποίος αντιπροσωπεύει τον αριθμό όλων των παραγωγών του DThread . Μόνο όταν φτάσει η τιμή του μετρητή αυτού στην θέση 0 θα μπορέσει το DThread και κάθε DThread που γίνετε ίση με 0 η τιμή στον μετρητή του , τότε είναι έτοιμο για εκτέλεση και μόλις ένας επεξεργαστής γίνει διαθέσιμος , η εκτέλεση του έτοιμου DThread θα ξεκινήσει.

Στην πλατφόρμα TFlux, το ρόλο του thread scheduler αναλαμβάνει η μονάδα συγχρονισμού του TFlux , το TSU. Όταν ένα DThread ολοκληρώσει την εκτέλεση του ειδοποιεί το TSU, το οποίο με την σειρά του μειώνει τις τιμές των ReadyCounter όλων των consumers του. Όταν ένα επεξεργαστής αναζητά το επόμενο διαθέσιμο προς εκτέλεση DThread , θα το ζητήσει από το TSU το οποίο θα του απαντήσει με ένα αριθμό που αντιπροσωπεύει το αναγνωριστικό του DThread , αλλιώς αν κανένα DThread ο επεξεργαστής θα περιμένει. Οι εντολές που βρίσκονται σε ένα DThread εκτελούνται με control-flow τρόπο και μπορούν να δεχθούν οποιεσδήποτε βελτιστοποιήσεις , είτε από τον επεξεργαστή δυναμικά, είτε στατικά από τον μεταγλωττιστή.

Με τη data-driven δρομολόγηση , μπορεί να μειώνονται οι καθυστερήσεις για υπολογισμούς συγχρονισμού , αλλά ταυτόχρονα δημιουργούνται πολύπλοκα μοτίβα προσβάσεις στην μνήμη ,

αφού ο τρόπος με τον οποίο θα φορτωθούν τα δεδομένα στην μνήμη δεν είναι στατικά γνωστός. Για την επίλυση ενός τέτοιου προβλήματος , το οποίο θα κοστίσει πολλά memory misses, το μοντέλο DDM χρησιμοποίει την πολιτική CacheFlow , κατά την οποία όλα τα δεδομένα που θα χρειαστεί ένα DThread για την εκτέλεση του φορτώνονται στην μνήμη λίγο πριν το DThread δρομολογηθεί για εκτέλεση.

Τα DDM προγράμματα στις περιπτώσεις που είναι αρκετά μεγάλα μπορούν να χωριστούν σε DDM Blocks, το οποίο κάθε Block μπορεί να περιέχει ένα η περισσότερα DThreads και το μέγιστο μέγεθός καθορίζετε από το μέγεθός του TSU. Στην ουσία τα ένα DDM Block περιέχει ένα υποσύνολο DThreads του αρχικού προγράμματος. Πέρα από τα DThreads που καθορίζει ο προγραμματιστής σε ένα κάθε ένα DDM Block υπάρχουν δύο επιπρόσθετα DThreads , το Inlet και το Outlet. Ο σκοπός του Inlet DThread είναι να εγγράψει στο TSU όλα τα δεδομένα των DThreads που ανήκουν στο αντίστοιχο DDM Block που αναφέρονται, ενώ ο σκοπός του Outlet DThread είναι να αποδεσμεύσει όλους τους πόρους που δεσμεύτηκαν σε ένα DDM Block . Καθώς εκτελεστούν τα DThreads σε ένα DDM Block , αμέσως φορτώνετε στο TSU το Inlet DThread του επόμενου DDM Block

Σε ένα DDM πρόγραμμα τα Blocks εκτελούνται με την σειρά που τοποθετούνται στον κώδικα ενώ τα DThreads σε ένα DDM Block μπορούν να εκτελεστούν με οποιαδήποτε τρόπο ανάλογα με τις εξαρτήσεις που έχουν. Κώδικας ανάμεσα σε Blocks εκτελείτε σειριακά.[12]

3.2.2 TFluxSoft

Το TFlux είναι μια πλατφόρμα , η οποία με την εικονοποίηση που εφαρμόζει στο σύστημα που εκτελείτε πετυχαίνει την αποδοτική εκτέλεση DDM εφαρμογών σε οποιαδήποτε αρχιτεκτονική ανεξαρτήτως υλικού και λογισμικού.

Στην περίπτωση του TFlux το TSU είναι υλοποιημένο σαν μια οντότητα λογισμικού , που ονομάζετε TFluxSoft **.**Η υλοποίηση TFluxSoft στοχεύει σε συνηθισμένους πολύ-πυρήνες με κοινή μνήμη και περιλαμβάνει cache-coherency για την on-chip μνήμη.

Ο κώδικας από τον οποίο αναπτύχθηκε από τον προγραμματιστή, δηλαδή ο ήδη υπάρχον κώδικας μαζί με όσα TFlux directives (pragma) πρόσθεσε αφού περάσει από τον TFlux Preprocessor και λάβει υπόψη του στον τι αντιστοιχεί στο κάθε pragma μεταφράζει τον αντίστοιχο κώδικα σε παράλληλο κώδικα C .

Ακολούθως μεταγλωττίζετε ο κώδικας που παράχθηκε πιο πάνω με ένα συνηθισμένο μεταγλωττιστή και παράγετε το εκτελέσιμο του παράλληλού αρχείου, το οποίο μπορεί να εκτελεστή πάνω από οποιαδήποτε αρχιτεκτονική αγνοώντας τα χαρακτηριστικά του συστήματος και κρύβοντας όλες τις λεπτομέρειες σχετικά με το DDM **.**

Αυτό οφείλετε κυρίως στο Runtime Support του TFlux που περιλαμβάνετε στο εκτελέσιμο και το οποίο σε επικοινωνία με την μονάδα συγχρονισμού TSU , καταφέρνει να εκτελεί την εφαρμογή με data-driven τρόπο . Το Runtime Support τρέχει πάνω από ένα Λειτουργικό Σύστημα , το οποίο δεν έχει τροποποιηθεί με οποιοδήποτε τρόπο, και επιτρέπει την εκτέλεση τόσο TFlux προγραμμάτων και τόσο και άλλων συμβατικών προγραμμάτων. Για να είναι εφικτή η χρήση ενός συνηθισμένου Λειτουργικού Συστήματος για DDM εκτελέσεις , το Runtime Support θα πρέπει να πληροί κάποιες προϋποθέσεις . Η πρώτη προϋπόθεση είναι όταν μια εφαρμογή εκτελείτε παράλληλα είτα σε κοινή η διαμοιραζόμενη μνήμη πολυεπεξεργαστή , το Runtime Support θα πρέπει να παρέχει ένα τρόπο DThreads να έχουν προσβάσεις στις κοινές μεταβλητές που στην ουσία είναι μεταβλητές που αφορούν τις εξαρτήσεις που έχουν μεταξύ τους.

Για την τήρηση των πιο πάνων προϋποθέσεων σχεδιάστηκε μια απλή οντότητα η TFlux Kernel και έτσι συνεπώς κάθε TFlux εκτελέσιμο δεν έχει ανάγκη για κανένα επιπρόσθετο λογισμικό γιατί περιέχει όλα όσα χρειάζεται κατά την μεταγλώττιση του.
Κατά την εκτέλεσης μιας TFlux εφαρμογής το πρώτο πράγμα που καθόρίζει ο προγραμματιστής και επομένως και το πρώτο πράγμα που δημιουργείτε είναι ο αριθμός των kernels που θα χρησιμοποιηθούν από την εν λόγω εφαρμογή. Ένας TFlux kernel είναι ένα POSIX thread , το οποίο αν είναι δυνατό για σκοπούς βελτιστοποίηση της απόδοσης εκτελείτε σε διαφορετικό CPU .Μετά την δημιουργία των kernels και με την έναρξη της εκτέλεσης ο κάθε kernel φορτώνει στο TSU το Inlet DThread του πρώτου DDM Block που πρόκειται να εκτελεστεί και ο TSU με την σειρά του , ενημερώνει τον kernel ότι αυτό είναι το πρώτο DThread που πρέπει να εκτελεστεί. Με την εκτέλεση αυτού του DThread **(**Inlet DThread**)** όλες οι πληροφορίες των DThreads του πρώτου DDM Block θα φορτωθούν στο TSU και αυτός με την σειρά του θα επιτρέψει την εκτέλεση τους. Η όλη διαδικασία αυτή επαναλαμβάνετε σε κάθε DDM Block . Ένας kernel ολοκληρώνει την εκτέλεση του μόνο όταν εκτελέσει το Outlet DThread του τελευταίου DDM Block της εφαρμογής. Η εφαρμογή τερματίζει μόνο όταν όλοι οι TFlux kernels ολοκληρώσουν την εκτέλεση τους [11,12,13].

3. 2.3 Thread Synchronization Unit

Το Thread Synchronization Unit (TSU) είναι η μονάδα που είναι υπεύθυνή για την διατήρηση πληροφοριών που παρέχονται από τον γράφο συγχρονισμού του προγράμματος με TFlux καθώς και την δυναμική τους ενημέρωση τους μέσω διαφορών συναρτήσεων.
Σε κάθε TFlux kernel αντιστοιχεί το δικό του προσωπικό TSU στο οποίο εγγράφονται όλες οι πληροφορίες για τα DThreads για τα οποία θα εκτελέσει ο συγκεκριμένος TFlux kernel . Απαραίτητη είναι η ύπαρξη κοινών δομών για όλα τα TSUs ούτως ώστε να υπάρχει επικοινωνία μεταξύ τους για σκοπούς των καταναλωτικών (consumers) DThreads μετά την εκτέλεση ενός από τα παραγωγικά (producer) DThreads τους. Με αυτό τον τρόπο επιτυγχάνετε η εύκολη εσωτερική επικοινωνία μεταξύ των kernels της TFlux εφαρμογής χωρίς να χρειάζεται η ανάμειξη κάποιας άλλης μονάδας.

Στην προηγούμενη υλοποίηση του DDM , το D^2NOW, κάθε επεξεργαστής έπρεπε να είχε το δικό του προσωπικό TSU καθώς οι μονάδες εκτέλεσης του ήταν ανεξάρτητες μηχανές . Στο TFlux τα TSUs είναι ομαδοποιημένα σε μια μονάδα που ονομάζετε TSU Group, όπου οι πόροι του είναι χωρισμένοι σε 2 κατηγορίες ,αυτοί που εξυπηρετούν τον επεξεργαστή στον οποίο αντιστοιχεί ένα TSU και αυτοί που είναι κοινοί για όλους τους επεξεργαστές.

Η μονάδα συγχρονισμού TSU είναι ένα από τα κυριότερα συστατικά της πλατφόρμας TFlux αφού με την χρήση αυτής της μονάδας πετυχαίνετε η data-driven εκτέλεση της εφαρμογής.
Βασικό ρόλο στην λειτουργία του TSU εκτελεί το TFlux Emulator (συνάρτηση emulateTSU) το οποίο εκτελεί το κυρίως thread της ήδη υπάρχουσας εφαρμογής , δηλαδή της main , και σκοπός του είναι να διατηρεί το TSU σε λειτουργία μέχρις να ολοκληρώσουν όλοι οι TFlux kernels την εκτέλεση τους.

Βασικές συναρτήσεις του TSU είναι οι loadThread , threadCompletedExecution , updateThreadInstances , οπού στην πρώτη όπως είναι εμφανές από τα περιγραφικά ονόματα τους φορτώνει στο TSU τις πληροφορίες των DThreads στα οποία είναι έτοιμο να εκτελέσει, στη δεύτερη αφού ολοκληρωθεί ένα DThread την εκτέλεση του ενημερώνει το TSU και η τρίτη συνάρτηση ενημερώνει την τιμή του μετρητή ReadyCounter .

Λόγω του ότι το TSU είναι υλοποιημένο σε software, υπάρχει η ευελιξία να μπορούμε να χρησιμοποιήσουμε το TSU ως μόνο μια υπολογιστική οντότητα αφιερώνοντας αποκλειστικά ένα πυρήνα από τον πολυπύρηνο επεξεργαστή μας. [11]

3. 2.4 Βασικά συστατικά εκτέλεσης DDM εφαρμογής

3. 2.4.1 DDM Threads

Ένα βασικό συστατικό ενός προγράμματος TFlux είναι το DThread. Κάθε DThread αντιστοιχεί σε ένα υποσύνολο κώδικα του αρχικού και μπορεί να έχει οποιοδήποτε μέγεθος. Αυτός ο κώδικας μπορεί να περιέχει οποιοδήποτε τύπο προγραμματιστικής δομής όπως βρόχους, λειτουργίες ελέγχου ροής ακόμη και κλήση σε συνάρτηση. Τα DThreads μπορούν να τρέχουν είτε παράλληλα είτε με κάποια εξάρτηση μεταξύ τους και αυτό εξαρτάτε από τις εξαρτήσεις που θα έχουν το ένα με το άλλο, και οι εξαρτήσεις αυτές μπορούν να εκφραστούν με την χρήση κατάλληλων pragma directives . Μέσα σε ένα DThread οι εντολές που περιέχει εκτελούνται με control-flow τρόπο.

3. 2.4.2 DDM For loops

Οι βρόχοι επαναλήψεών είναι ένα συστατικό το οποίο παρουσιάζετε συχνά σε μεγάλο αριθμό επιστημονικών εφαρμογών , και η πλατφόρμα TFlux προσφέρει ειδική λειτουργικότητα για την εκτέλεση τέτοιων βρόχων και είναι αυτό που λέμε παράλληλο βρόχος επανάληψης. Αυτοί οι βρόγχοι εκτελούνται με την ανάθεση διαφορετικού υποσυνόλου επαναλήψεων σε κάθε kernel και αυτό σημαίνει ότι κάθε επανάληψη του βρόγχου εκτελείτε από ένα μόνο TFlux kernel κάθε φορα. Τα TFlux loops μπορούν να εξαρτούνται είναι από απλά DThreads είτε από άλλα TFlux loops. Και στις 2 περιπτώσεις καμία επανάληψη του TFlux loop δεν μπορεί να ξεκινήσει αν δεν τελειώσει το DThread/s ή TFlux loop/s στα οποία εξαρτάτε.

3. 2.4.3 DDM Blocks

Με την χρήση DDM Blocks επιτυγχάνετε δυναμική διαχείριση των μετά-δεδομένων και ο λόγος είναι ότι με την χρήση τους επιτυγχάνετε δυναμική εγγραφή και διαγραφή των μετά-δεδομένων που εγγράφονται στο TSU μιας και η χωρητικότητας αυτής της μονάδας είναι περιορισμένη.

Ένα DDM Block περιέχει ένα αριθμό από DThreads , μιας και όλα τα DThreads θα πρέπει να βρίσκονται μέσα σε DDM Blocks και όχι εκτός αυτών. Ένα πρόγραμμα TFlux πρέπει να αποτελείται το λιγότερο από ένα DDM Block. Εκτός των DThreads που περιέχουν τα DDM Blocks κάθε DDM Block περιέχει επιπρόσθετα ακόμα δύο DThreads , το Inlet και Outlet DThread.

Η κυρίως διαφορά των DDM Blocks με τα DDM Threads είναι ότι ενώ τα DDM Threads μπορούνε να εκτελούνται με data-driven τρόπο , τα DDM Blocks εκτελούνται σειριακά και κανένα DDM Block δεν μπορεί να αρχίσει να εκτελείται αν δεν τελειώσουν όλα τα DDM Threads του προηγούμενου DDM Block.

Κεφάλαιο 4

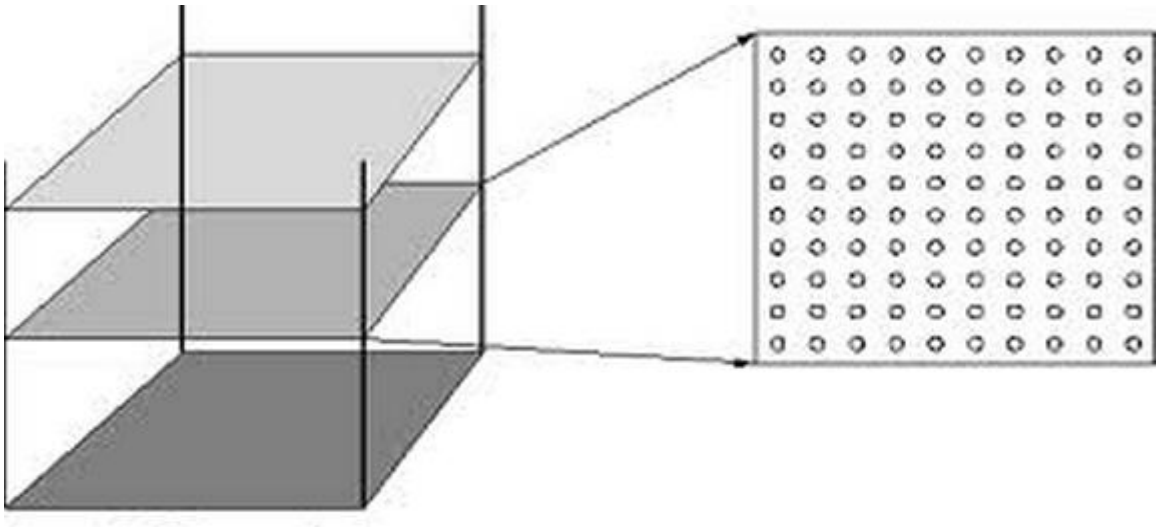
Η εφαρμογή OCEAN

4.1 Επιστημονική Περιγραφή Εφαρμογής OCEAN

Για να μοντελοποιήσουμε το κλίμα της γης, είναι σημαντικό να κατανοήσουμε πώς η ατμόσφαιρα αλληλεπιδρά με τους ωκεανούς που καταλαμβάνουν τα τρία τέταρτα της επιφάνειας της γης. Η προσομοίωση των ωκεάνιων ρευμάτων προσομοιώνει την κίνηση των ρευμάτων του νερού στον ωκεανό. Αυτά τα ρεύματα αναπτύσσεται και να εξελίσσεται κάτω από την επίδραση διαφόρων φυσικών δυνάμεων, περιλαμβανομένων των ατμοσφαιρικών επιπτώσεων, τον άνεμο, και η τριβή με το πυθμένα του ωκεανού. Κοντά στα τείχη των ωκεανών υπάρχει η επιπρόσθετη "κάθετές" τριβές καθώς, το οποίο οδηγεί στην ανάπτυξη του ρευμάτων γνωστά ως eddy. Ένα eddy ρεύμα είναι ένα βρόχος του ρεύματος που έχει αποκοπεί από το κύριο ρεύμα ή ένα μικρό περιστρεφόμενο ρεύμα. Ο στόχος της συγκεκριμένης εφαρμογής είναι να προσομοιώσει αυτά τα eddy ρεύματα με την πάροδο του χρόνου, και να κατανοηθούν οι αλληλεπιδράσεις τους με τη μέση ροή του ωκεανού.

Τα υψηλής ακρίβειας μοντέλα για τη συμπεριφορά των ωκεανών πολύπλοκα. Η πρόβλεψη της κατάστασης του ωκεανού σε οποιαδήποτε στιγμή απαιτεί τη λύση των πολύπλοκων συστημάτων εξισώσεων, τα οποία μπορούν να εκτελούνται αριθμητικά μόνο με υπολογιστή. Το πραγματικό φυσικό πρόβλημα είναι συνεχής τόσο στο χώρο (στην βάση του ωκεανού) τόσο και στον χρόνο , αλλά για να μπορέσει να προσομοιωθεί σε υπολογιστή διακριτικοποιείται και στις δύο διαστάσεις. Για να διακριτικοποιήσουμε το χώρο, μοντελοποιείτε η βάση του ωκεανού με ένα πλέγμα από στοιχεία που απέχουν ίσα μεταξύ τους. Κάθε σημαντική μεταβλητή-όπως

πίεση, ταχύτητα, και τα διαφόρων ρευμάτων-έχει μία τιμή σε κάθε σημείο του πλέγματος σε αυτήν διακριτοποίηση. Αυτή η εφαρμογή δεν χρησιμοποιεί ένα τρισδιάστατο πλέγμα, αλλά ένα σύνολο των δύο διαστάσεων, οριζόντιες διατομές που η κάθε μια εκπροσωπείται από ένα δισδιάστατο πλέγμα των σημείων (Σχήμα 3.1).



Σχήμα 4.1: *(Αριστερά) cross-sections και (δεξιά) Spatial discretization of a cross-section.*

Για λόγους απλότητας, ο ωκεανός μοντελοποιείται ως ένα ορθογώνιο σχήμα και τα σημεία του πλέγματος θεωρείται ότι απέχουν ίσα. Κάθε μεταβλητή συνεπώς αντιπροσωπεύεται από ένα ξεχωριστό πίνακα δύο διαστάσεων για κάθε οριζόντια διατομή του ωκεανού. Οι εξισώσεις της κίνησης επιλύετε σε όλα τα σημεία πλέγματος σε ένα χρονικό βήμα ,η κατάσταση των μεταβλητών ενημερώνεται ως αποτέλεσμα, και οι εξισώσεις κίνησης επιλύονται πάλι για το επόμενο χρονικό βήμα, και ούτω καθεξής επανειλημμένα. Κάθε φορά το ίδιο βήμα αποτελείται από διάφορές υπολογιστικά φάσεις. Πολλές από αυτές χρησιμοποιούνται για να πάρουν τιμές διάφορες μεταβλητές σε όλα τα σημεία πλέγματος χρησιμοποιώντας τα αποτελέσματα από την προηγούμενο χρονικό βήμα. Όλες οι φάσεις, σαρώνουν όλα τα σημεία των σχετικών πινάκων και χειραγωγούν τις τιμές τους.

Για έναν ωκεανό όπως ο Ατλαντικός, με περίπου 2.000 χιλιομέτρων-x-2.000 χιλιομέτρων έκταση , χρησιμοποιώντας ένα πλέγμα 100-x-100 σημεία, συνεπάγεται μια απόσταση 20χλμ μεταξύ των σημείων σε κάθε διάσταση. Αυτό δεν είναι μια πολύ υψηλή ανάλυση, και θα ήταν πιο ακριβές να χρησιμοποιούνταν πολλά περισσότερα σημεία . Ομοίως, μικρότερη σωματική διαστήματα μεταξύ των χρονικών-βημάτων οδηγούν στην μεγαλύτερη ακρίβεια προσομοίωσης. Για παράδειγμα, για την προσομοίωση πέντε ετών την κίνηση των ωκεανών και ενημέρωση της κατάστασης κάθε οκτώ ώρες θα χρειαστεί περίπου 5500 χρονικά βήματα.

Οι υπολογιστικές απαιτήσεις για υψηλή ακρίβεια είναι μεγάλες αλλά η εφαρμογή παρέχει επίσης εφόδια ταυτοχρονισμού: πολλά από τα ενημερώσεις των φάσεων σε ένα χρονικό βήμα είναι ανεξάρτητές η μία από την άλλη και ως εκ τούτου μπορεί να γίνει παράλληλα, καθώς

επίσης και η επεξεργασία των διαφόρων σημείων του δικτύου σε κάθε φάση ή τον υπολογισμό πλέγματος μπορεί η ίδια να γίνει παράλληλα.

Για παράδειγμα, θα μπορούσαμε να εκχωρήσετε διαφορετικά μέρη του ωκεανού σε διαφορετικούς επεξεργαστές [20].

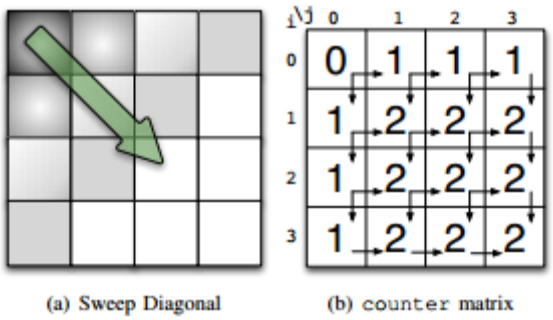
4.2 Η απλή προσέγγιση της εφαρμογής OCEAN

Η έμφαση που δόθηκε κατά την διάρκεια της όλης της μελέτης είναι στην απλή προσέγγιση του αλγόριθμου και ιδιαίτερα στις εξαρτήσεις του αλγορίθμου πέρα από την πολυπλοκότητα που κρύβει .Είναι γνωστό ότι σε χιλιάδες εργαστήρια σε παγκόσμιο εύρος ασχολούνται πιο επιστημονικά με το θέμα αυτό μιας και προσημειώνουν με κάθε ακρίβεια ωκεανούς λαμβάνοντας υπόψη τους ανέμους και οποιοδήποτε άλλο καιρικό παράγοντα.

Η προσημείωση των σημείων του Ocean στηρίζετε στο ότι για να υπολογιστεί η τιμή σε κάθε σημείο απαραίτητη προϋπόθεση είναι να στηριχθεί στις τιμές των τεσσάρων γειτόνων του. Δηλαδή θα χρειαστεί την τιμή του ακριβώς πάνω γείτονα του που θα βρίσκετε με βάση το πιο κάτω σχήμα στο πρώτο άνω στρώμα, από τον ακριβώς κάτω γείτονα που βρίσκετε στο τρίτο στρώμα και από τους γείτονες του αριστερά και δεξιά του που βρίσκονται στο ίδιο στρώμα με το σημείο που υπολογίζετε και στο σχήμα θα είναι το δεύτερο στρώμα.

Η εφαρμογή Ocean είναι ένας αλγόριθμός βασισμένος στο πρότυπο Wave-front[28] . Wavefront είναι ένα μοτίβο προγραμματισμού που εμφανίζετε σε επιστημονικές εφαρμογές και σε αυτό το μοτίβο , τα στοιχεία των δεδομένων κατανέμονται σε πολυδιάστατα πλέγματα που αντιπροσωπεύουν ένα λογικό διάστημα. Τα στοιχεία θα πρέπει να υπολογίζονται σε σειρά επειδή υπάρχουν εξαρτήσεις μεταξύ τους. Οι Wavefront[34,35] τεχνικές χρησιμοποιούνται σε αλγορίθμους οι οποίοι έχουν επανεμφανίσεις με το σπάσιμο του υπολογισμού σε τμήματα και εφαρμόζοντας pipelining στα τμήματα αυτά μέσω πολλαπλών επεξεργαστών. Περιγράφτηκε για πρώτη φορά αυτό το πρότυπο ως "hyperplane" και βρίσκει εφαρμογή σε διάφορους σημαντικούς τομείς συμπεριλαμβανομένων προσημειώσεις σωματιδίων φυσικής, παράλληλους επαναληπτικούς solvers και παράλληλη λύση τριγωνικών συστημάτων των γραμμικών εξισώσεων. Αν και απλό μοντέλο εντούτοις είναι περίπλοκο να μοντελοποιήσουμε πολλαπλά wavefronts που υπάρχουν ταυτόχρονα λόγο της πιθανής επικάλυψης των υπολογισμών και επικοινωνιών ή επικάλυψη διαφορετικών επικοινωνιών ή πράξεις υπολογισμού ξεχωριστά. Η δυσδιάστατη επανάληψη μπορεί μερικώς να εξαλειφθεί λόγο των λύσεων με θέσεων σε ένα πίνακα με διαγώνιο τρόπο , ο οποίος επιτρέπει οι θέσεις στην κάθε διαγώνιο να είναι ανεξάρτητές. Το πέρασμα των υπολογισμών γίνετε από subdomain σε subdomain με μία καθορισμένη γωνιακή κατεύθυνση. Οι εξωτερικές επιφάνειες ενός subdomain υπολογίζονται από τις επιφάνειες των ακριβώς προηγούμενων subdomain που υπολογίστηκαν και κάνουν

share τις τιμές τους. Αν και οι εξαρτήσεις που υπάρχουν , συνεπάγει σειριακή εκτέλεση μεταξύ τους, είναι γνωστό ότι οι wavefront υπολογισμοί δέχονται αποδοτική και παράλληλη υλοποίηση μέσω του pipelining , δηλαδή οι επεξεργαστές υπολογίζουν μόνο μερικά τοπικά δεδομένα πριν την αποστολή των δεδομένων στους εξαρτώμενούς γείτωνές[35]. Άλλοι αλγόριθμοι που βασίζονται στο Wavefront πρότυπο εκτός από τον Ocean είναι Heating transfer ,LCSS, DNA matching κ.λ.π.



Σχήμα 4.2 *Τυπικά wavefront διάσχιση και εξαρτήσεις μεταφράζονται σε μετρητές.*

Στο σχήμα 4.2 παρουσιάζετε ένα 2D wavefront όπου εδώ οι υπολογισμοί ξεκινάνε από μια γωνία του πίνακα και το πέρασμα του θα προχωρήσει στο επίπεδο μέχρι την απέναντι γωνία ακολουθώντας την διαγώνια πορεία . Κάθε διαγώνιος αντιπροσωπεύει τον αριθμό των υπολογισμών ή των στοιχείων που θα μπορούσαν να εκτελεστούν παράλληλα χωρίς εξαρτήσεις μεταξύ τους.

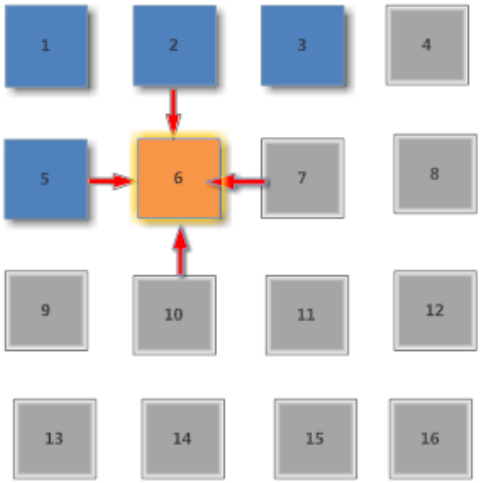
4.2.1 Αλγορίθμου με τις ακριβείς εξαρτήσεις

Ο επόμενος εναλλακτικός τρόπος επίλυσης της εφαρμογής OCEAN είναι να υπολογίζεται η κάθε θέση με τις σωστές τιμές των γειτόνων της και αυτό μόνο με το να υπάρχουν ακριβές εξαρτήσεις. Οι εξαρτήσεις αυτές θα ρυθμίζουν στο πότε θα ξεκινά να υπολογίζετε και να ανανεώνετε μια τιμή και θα υποδηλώνουν κατά πόσο οι τιμές των γειτόνων μιας θέσης είναι έτοιμες και έγκυρες , έτσι ώστε να υπολογιστή η συγκεκριμένη θέση.

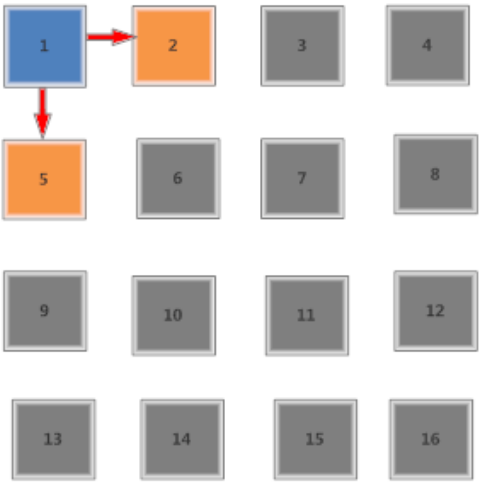
Η λογική αυτή είναι η ίδια με την οποία δουλεύει η εφαρμογή OCEAN . Το μοντέλο αυτό εφαρμόζει τις εξαρτήσεις με βάση το ποτέ τα δεδομένα που χρειάζεται και στα οποία εξαρτάτε είναι έτοιμα.

Για παράδειγμα στο πιο κάτω πλέγμα έχει 14 κόμβους συνολικά. Οι κόμβοι 2 και 5 έχουν εξάρτηση από τον κόμβο 1(Σχήμα 4.3) , ο κόμβος 3 έχει εξάρτηση από τον 2 , ο 6 από 2 και 5(Σχήμα 4.4) και όλοι οι κόμβοι έχουν εξάρτηση από τον κόμβο ακριβώς από πάνω τους και τον κόμβο στα αριστερά τους αν υπάρχουν. Οι εξαρτήσεις υποδηλώνονται με τα κόκκινα βέλη .Φυσικά για τον υπολογισμό του κάθε κόμβου χρειάζεται να υπολογιστεί με βάση τους τέσσερις

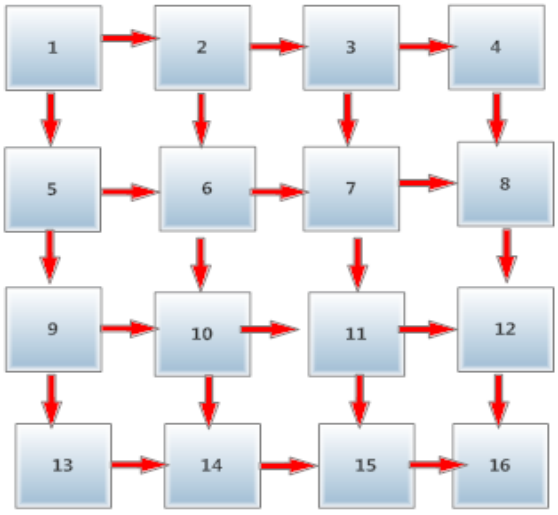
γείτονες του αν υπάρχουν αλλά οι εξαρτήσεις φανερώνουν στα δεδομένα που πρέπει να υπολογιστούν και μετά να υπολογιστούν θέσεις που εξαρτούνται από τα δεδομένα αυτά.



Σχήμα 4.3 : Οι εξαρτήσεις δεδομένων του κόμβου 6

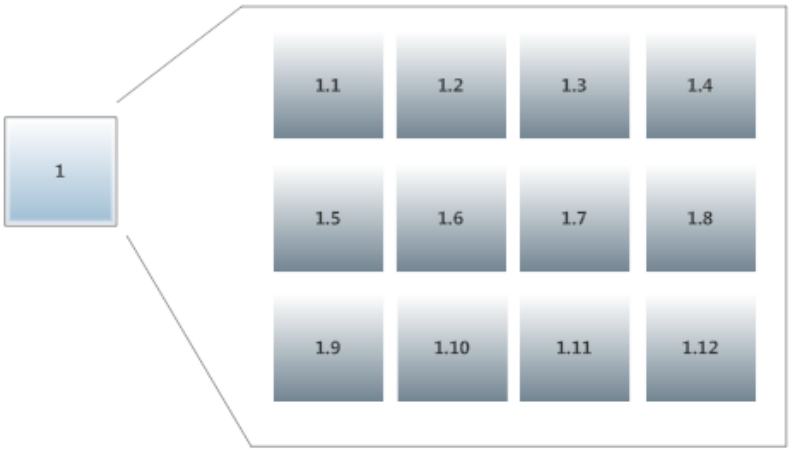


Σχήμα 4.4: Οι εξαρτήσεις δεδομένων των κόμβων 2 και 5



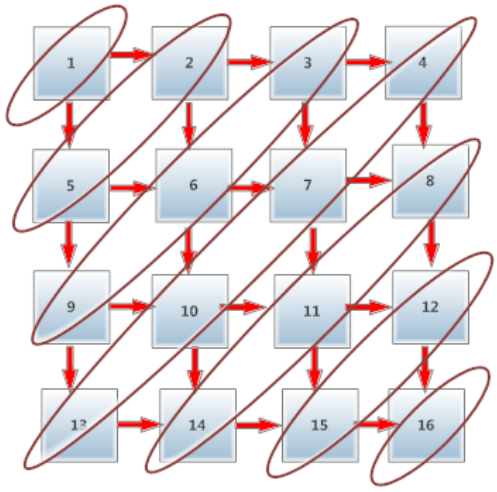
Σχήμα 4.5: *Οι εξαρτήσεις δεδομένων όλων των κόμβων*

Οι πιο πάνω θέσεις στο σχήμα 4.5 μπορεί να είτε θέσεις σε πίνακα ή σημεία η ένα μπλοκ στο οποίο μέσα να έχει ένα υποσύνολο σημείων από το συνολικό εύρος σημείων από το πλέγμα (Σχήμα 4.6). Το μέγεθος του μπλοκ μπορεί να είναι οποιοδήποτε , αλλά απαραίτητη προϋπόθεση για τον παράγοντα για blocking είναι να είναι πολλαπλάσιο της διάστασης του πλέγματος.



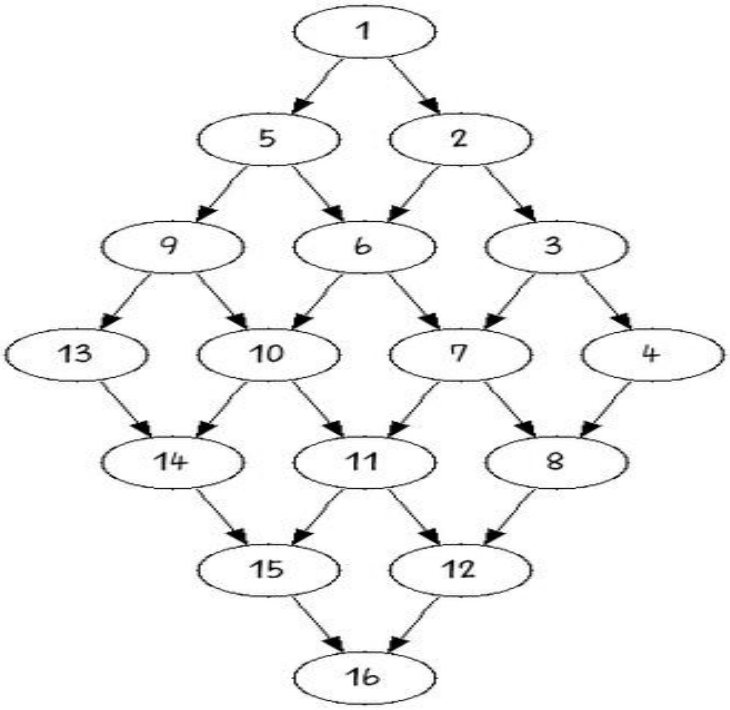
Σχήμα 4.6 *:Υποσύνολο σημείων ενός μπλοκ*

Στο πλέγμα [illegible] ήσεις μπορεί να έχει μέγιστο βαθμό παραλληλισμού μέχρι και τέσσερα διότι αναλύοντας τις εξαρτήσεις καλύτερα θα συμπεράνουμε πως η κάθε διαγώνιος εξαρτάτε από την αμέσως προηγούμενη διαγώνιο, αλλά παράλληλα οι κόμβοι που συμπεριλαμβάνει μια διαγώνιος μπορούν να εκτελούνται παράλληλα.(Σχήμα 4.7)



Σχήμα 4.7 *Βαθμός παραλληλισμού σε κάθε διαγώνιο*

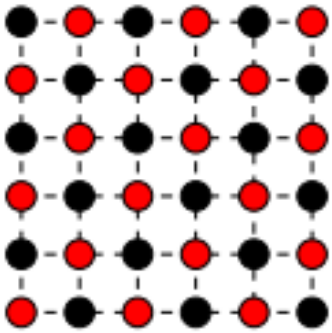
Στο πιο κάτω σχήμα 4.8 φαίνετε ο γράφος εξαρτήσεων του πιο πάνω σχήματος με τις εξαρτήσεις των κόμβων .Ανάλογα με τις εξαρτήσεις , ο πιο κάτω γράφος με τα βέλη του, εκτός του ότι αναπαριστά τις εξαρτήσεις των κόμβων μεταξύ τους ,δείχνει ξεκάθαρα ότι ένας κόμβος μπορεί να εξαρτάται και από 2 κόμβους. Αυτό σημαίνει πως θα πρέπει να περιμένει να ολοκληρωθεί η εκτέλεση και στους δύο κόμβους για να έρθει η σειρά του. Κομβόι που δεν δέχονται κανένα βέλος είναι οι κόμβοι οι οποίοι εκτελούνται χωρίς να περιμένουν κάποιο άλλο κόμβο δηλαδή δεν έχουν καμία εξάρτηση. Στην προκειμένη περίπτωση ο κόμβος με αριθμό 1 δεν διαθέτει καμία εξάρτηση και άρα μπορεί να αρχίσει να εκτελείτε πρώτο.



Σχήμα 4.8 *:Γράφος Εξαρτήσεων*

4.2.2 Αλγόριθμός Red-Black Gauss-Seidel

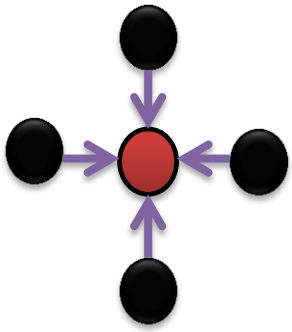
Ο αλγόριθμος Red-Black Gauss-Seidel αποτελεί ένας από τους δυνατούς τρόπους επίλυσης του OCEAN αλγορίθμου. Είναι μια επαναληπτική προσέγγιση και παράλληλα μια λύση στο να αυξήσει τον παραλληλισμό του οποιουδήποτε προβλήματος που λύνετε με αυτό τον τρόπο και στην συγκεκριμένη περίπτωση του OCEAN [22]. Η ιδέα βασίζετε στο να κατανέμουμε τα grids του συνολικού πλέγματός σε δύο ομάδες/κατηγορίες. Η μια κατηγορία είναι η κόκκινη αλλιώς "red" και η άλλη είναι η μαύρη αλλιώς « black» και αφού δεν υπάρχει data-dependencies μεταξύ των δύο ομάδων , μπορούν να ανανεώνονται ανεξάρτητα [26]. Πιο κάτω φαίνετε η εικόνα που χαρακτηρίζει τον αλγόριθμό αυτό :



Σχήμα 4.9: *Red-Black Gauss-Seidel[23]*

Η λογική του αλγορίθμου είναι να υπολογίζονται οι τιμές των κόκκινων σημείων με βάση τις τιμές των μαύρων σημείων (οι οποίες τιμές είναι ανανεωμένες με την ακριβώς προηγούμενη επανάληψη) και μετά το ίδιο τα μαύρα σημεία να υπολογίζονται και να ανανεώνουν την τιμή τους με βάση τις τιμές των κόκκινων σημείων.(Σχήμα 4.10)

Ο αλγόριθμος αυτός είναι επαναληπτικός διότι επαναλαμβάνετε η όλη διαδικασία δηλαδή να υπολογιστούν οι τιμές των κόκκινων σημείων και μετά να υπολογιστούν οι τιμές των μαύρων σημείων μέχρι οι τιμές να συγκλίνουν σε κάποιο όριο. Σε αντίθεση με τις άμεσες μεθόδους , οι επαναληπτικοί μέθοδοι δεν παράγουν την ακριβή απάντηση μετά από ένα πεπερασμένο αριθμό βημάτων αλλά μειώνουν το σφάλμα με κάποιο ποσοστό μετά από κάθε βήμα. Η επανάληψη σταματά όταν το σφάλμα είναι μικρότερο από ένα όριο που παρέχει ο χρήστης [26].Μπορεί οι τιμές που λαμβάνονται υπόψη για την ανανέωση άλλων τιμών να μην είναι οι ακριβές αλλά λόγω του ότι επαναλαμβάνετε όλο αυτό στο τέλος οι τιμές συγκλίνουν.



Σχήμα 4.10:*Λογική Red-Black Gauss-Seidel αλγορίθμου*

Ο κώδικας με τον οποίο λειτουργεί ο αλγόριθμος αυτός είναι ο εξής [23] :

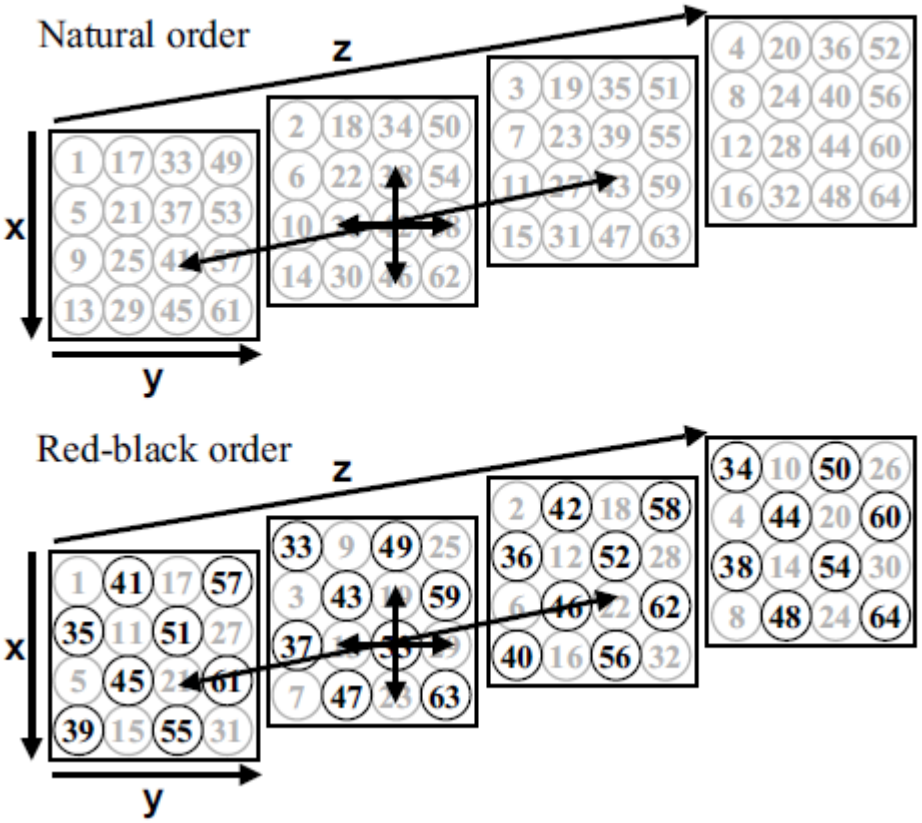
```
for i = 2:n-1
        for j = 2:n-1
                if mod(i+j,2) == 0
                        u(i,j) = ...
                end
        end
end
for i = 2:n-1
        for j = 2:n-1
                if mod(i+j,2) == 1,
                        u(i,j) = ...
                end
        end
end
```

Τυπικά ο αλγόριθμος αυτός περιγράφετε από δύο for loops , από τα οποία το πρώτο for loop θα υπολογίσει τις τιμές όλων των θέσεων οι οποίες ικανοποιούν την συνθήκη αυτή mod(i+j,2) == 0 όπου το j υποδηλώνει τον αριθμό στήλης και το i υποδηλώνει τον αριθμό της γραμμής. Ενώ το δεύτερο for loop δουλεύει όταν ισχύει η συνθήκη mod(i+j,2) == 1. Οι ακριανές θέσεις παραλείπονται , για αυτό και αρχίζει το index του κάθε for loop από το 2 και τελειώνει στο μέγεθος του πλέγματος -1 έτσι ώστε να μην συμπεριληφθούν στους υπολογισμούς η πρώτη στήλη και γραμμή καθώς και η τελευταία γραμμή και στήλη. Το ένα for loop είναι για τον υπολογισμό των κόκκινων σημείων και το άλλο για τον υπολογισμό των μαύρων σημείων.

Η μαθηματική απλή πράξη που εκτελείτε σε κάθε σημείο είναι το άθροισμα των τεσσάρων γειτόνων και ακολούθως ο μέσος όρος του. Αυτή είναι η βάση της πράξης, που άλλοτε μπορούμε να την τροποποιήσουμε και να πολλαπλασιάσουμε εν επιθυμούμε για παράδειγμα με ένα βάρος .Συνήθως πολλαπλασιάζετε με 0.2.Υπάρχουν παραλλαγές στην πράξη αυτή συνήθως με τα πόσα σημεία χρησιμοποιούνται .Υπάρχουν εφαρμογές που χρησιμοποιούν την πράξη αυτή με εφτα(7) σημεία στα οποία να στηρίζετε για τον υπολογισμό ενός σημείου [26]. Στο πιο κάτω σχήμα φαίνετε ξεκάθαρα πως είναι η κανονική εκτέλεση και πως με Red-Black Gauss-Seidel σε 3D μορφή και η εξίσωση που ακλουθείτε για το πιο κάτω σχήμα για την κανονική εκτέλεση είναι η

$$u_{(i,j,k)} = Su_{(i,j,k)} \equiv \frac{1}{6}\Big(u_{(i+1,j,k)} + u_{(i-1,j,k)} + u_{(i,j+1,k)} + u_{(i,j-1,k)} + u_{(i,j,k+1)} + u_{(i,j,k-1)} + h^2 f_{(i,j,k)}\Big).$$

Σχήμα 4.11: *family of Gauss-Seidel smoothers [26]*



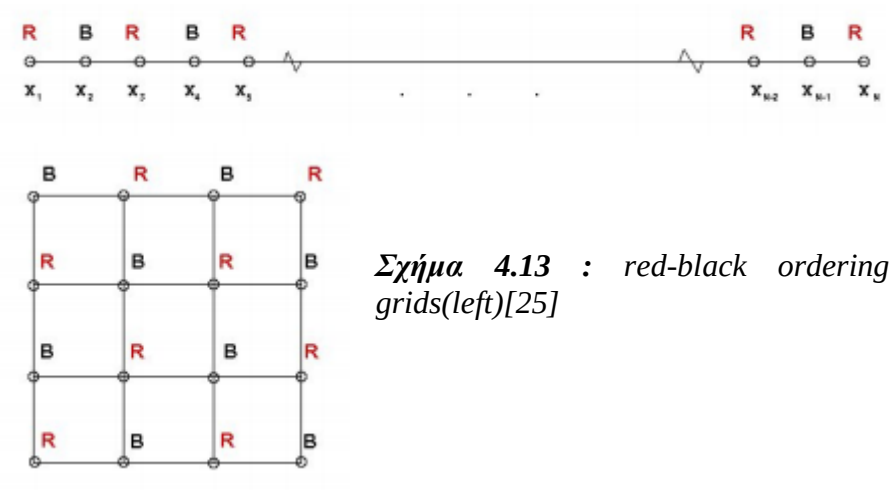
Σχήμα4.12 : *. Natural and Red-black orders in a 3D problem. The 7-point stencil accesses data in all three directions.[26]*

Στο σχήμα3.5 βλέπουμε πως στο υπολογισμό του σημείου 42 στην κανονική μορφή θα χρειαστεί το σημείο 38(πάνω),46(κάτω),26(αριστερά),58(δεξια), 41(σημέιο της ίδια θέσης αλλά προηγούμενου επιπέδου), 43(σημείο της ίδιας θέσης αλλά επόμενου επιπέδου).

Εκ φύσεως του ο αλγόριθμος αυτός είναι χρήσιμός όταν χρησιμοποιείτε σε παράλληλες εφαρμογές. Για αυτό εξίσου και από το όνομα του , χρησιμοποιεί τον χρωματισμό των σημείων

σε αποχρώσεις του κόκκινου και του μαύρου για να φτιάξει στην ουσία ένα κόκκινο-μαύρο πίνακα οπού σημεία ζυγού αριθμού να υπολογίζονται ως κόκκινα σημεία και τα υπόλοιπα σημεία μονού αριθμού να υπολογίζονται ως μαύρα.

Ο πιο κάτω γράφος δείχνει red-black σειρά για 1D και 2D σημείων.



Σχήμα 4.13 : *red-black ordering for 1D(top) and 2D grids(left)[25]*

Ο αλγόριθμος Red-Black Gauss-Seidel απέδειξε πως εκτός από τον παραλληλισμό που προσφέρει δίνει επίσης ένα καλύτερο αποτέλεσμα εξομάλυνσης τιμών στην προσπάθεια του να υπολογίσει τις τιμές των σημείων στο πλέγμα [25].

4.3Υλοποιησεις εφαρμογής Ocean

4.3.1 OpenMp

Στην υλοποίηση αυτή, πήραμε τη σειριακή μορφή του κώδικα όπως και στην πλοηγούμενη υλοποίηση και την μετατρέψαμε εδώ σε μια OpenMp εφαρμογή. Εφαρμόσαμε στο κώδικα το κατάλληλο directive και επίσης δηλώσαμε ποίες μεταβλητές θα θέλαμε να είναι ιδιωτικές έτσι ώστε να υπάρχουν οι μεταβλητές αυτές όπως και κάθε private μεταβλητή , ένα αντίγραφο για το κάθε ένα νήμα .

Πιο κάτω φαίνετε ο ψευδοκώδικας με το φωλιασμένο for loop μέσα σε άλλο for loop το οποίο έχει σαν αποτέλεσμα να διασχίζει το πλέγμα με τα σημεία στα οποία θα υπολογίσουμε την τιμή

τους με διαγώνιο τρόπο. Το εξωτερικό for loop κάθε επανάληψη ενημερώνει την μεταβλητή points με την τιμή των σημείων που βρίσκονται στην τρέχουσα διαγώνιο και με βάση αυτή την τιμή εκτελείτε το δεύτερο φωλιασμένο for loop και ενημερώνει τις τιμές αυτών των σημείων.

```
#pragma omp parallel for private(j, points) firstprivate(row,col)
for (i=0;i<2*N-1;i++)
{
    if(i<N)
    {
        points=i;
        col=0;
        row = i;
    }
    else
    {
        points=(2*N)-2 -i;
        col= i-N+1;
        row=N-1;
    }

    for(j=0;j<=points;j++)
    {

        table_4x4[col][row] =  (table_4x4[col][row]
                    + table_4x4[col+1][row] + table_4x4[col-1][row]
                    + table_4x4[col][row+1] + table_4x4[col][row-1])/5;
        col++;
        row--;
    }

}
```

Από την στιγμή που ενσωματώσουμε στο κώδικα μας το directive αυτό αυτόματα το μεγάλο for loop γίνετε παράλληλο αφού ανάλογα με τον αριθμό των νημάτων που θα δηλώσουμε θα μοιράσει την δουλεία ανάμεσα τους και έτσι θα μειωθεί ο χρόνος αφού όλα τα νήματα θα εκτελούνται την ίδια στιγμή παράλληλα.

Για το λόγο αυτό χρειάζεται να δηλώσουμε private μεταβλητές έτσι ώστε να διατηρήσουμε την συνοχή των μεταβλητών που επεξεργάζονται από όλα τα νήματα για να μην υπάρχουν συγκρούσεις και φυσικά να έχουν σωστές τιμές.

4.3.2 TFlux με διάφορες παραλλαγές του blocking

Στην υλοποίηση αυτή η εφαρμογή Ocean μετατράπηκε σε μια TFlux εφαρμογή αφού τέθηκε κάτω από την πλατφόρμα της.

Η ιδέα είναι να δημιουργήσουμε παραλλαγές του ίδιου κώδικα με το κάθε thread να αναλαμβάνει ένα υποσύνολο σημείων από το αρχικό πλέγμα για υπολογισμό, με βάση διαφορετικά μεγέθη του blocking. Η ιδέα του blocking πάρθηκε από το πρότυπο στο οποίο

είναι βασισμένο η εφαρμογή Ocean το οποίο είναι το Wavefront[29]. Ουσιαστικά το blocking είναι για να έχουμε blocks από σημεία και οι εξαρτήσεις να είναι με βάση τα blocks μεταξύ τους και όχι ανάμεσα στα σημεία. Μια αφορμή για το blocking, στάθηκε όπως ήταν αναμενόμενο για να βρούμε το πιο αποδοτικό μέσα από την μελέτη τους και σύγκριση τους.

Αφού κατανεμηθεί το συνολικό μέγεθός του πλέγματος στα διάφορα threads εφαρμόζουμε μεταξύ τους data-dependencies , έτσι ώστε όταν τελειώσει ο υπολογισμός των σημείων στο οποίο είναι υπεύθυνο ένα νήμα να ξεκινούν αμέσως μετά να εκτελούνται τα νήματα που εξαρτούνται από το συγκεκριμένο που τελείωσε , που στην ουσία εξαρτούνται από τις τιμές των θέσεων στο οποίο υπολόγισε και ανανέωσε το προηγούμενο νήμα.

Οι παραλλαγές στο μέγεθός του μπλοκ δεν είναι τίποτε άλλο από την ίδια ακριβώς φιλοσοφία όπως προαναφέραμε με την διαφορά ότι άλλοτε το νήμα θα είναι υπεύθυνο για ένα μικρότερο υποσύνολο σημείων και άλλοτε ένα μεγαλύτερο και αυτό για να δούμε στο τέλος την διαφορά στο να εφαρμόζετε το data-dependencies με διαφορά μεγέθη των μπλοκ.

Χωρίς να έχουμε οποιεσδήποτε συγχρονισμό ή κλειδαριές , το μοντέλο αυτό όπως προαναφέραμε δουλεύει όταν τα δεδομένα που περίμενε να εκτελεστούν , εκτελέστηκαν και είναι ήδη έτοιμα. Η εκτέλεση της εφαρμογής είναι βασισμένη στο Data-Driven μοντέλο, μοντέλο το οποίο στηρίζετε η πλατφόρμα TFlux.

Το μόνο που απαιτεί το μοντέλο είναι να καθορίσουμε σωστά τις εξαρτήσεις μεταξύ των νημάτων για να δημιουργήσει σωστά τον γράφο εξαρτήσεων , το οποίο θα είναι η ροη εκτέλεσης που θα ακολουθήσει αλλά και για σωστά αποτελέσματα.

Ανάλογα με τον αριθμό των kernels που θα δηλώσουμε τόσα νήματα έχουν την δυνατότητα να τρέχουν παράλληλα την ίδια στιγμή αν και φυσικά τους το επιτρέπει οι εξαρτήσεις. Δηλαδή αν για παράδειγμα έχουμε τρία νήματα που εξαρτούνται από το ίδιο νήμα, όταν αυτό το νήμα στο οποίο εξαρτούνται εκτελεστεί και ο αριθμός των επεξεργαστών είναι μεγαλύτερος η ίσος του 3 , τότε αυτά τα τρία νήματα θα μπορούν να εκτελεστούν παράλληλα.

```
#pragma ddm block 1

#pragma ddm thread 1 kernel 1
block_execution(0,0);
#pragma ddm endthread


#pragma ddm thread 2 kernel 1 depends(1)
block_execution(0,3000);
#pragma ddm endthread

#pragma ddm thread 3 kernel 1 depends(2)
block_execution(0,6000);
#pragma ddm endthread


#pragma ddm thread 4 kernel 1 depends(3)
block_execution(0,9000);
#pragma ddm endthread


#pragma ddm thread 5 kernel 1  depends(4)
block_execution(0,12000);
#pragma ddm endthread
```

Στο σχήμα παρουσιάζετε ένα μέρος του κώδικα οπού φαίνετε πως το κάθε thread αναλαμβάνει ένα μέρος του πλέγματος, ένα block με το καλέσει την συνάρτηση block_execution και να της δώσει σαν παραμέτρους τον αριθμό της γραμμής και τον αριθμό στήλη στο οποίο να αρχίσει η εκτέλεση. Στην συνάρτηση αυτή θα λάβει αυτές τις τιμές και ο αλγόριθμος θα εκτελείτε με βάση τα σημεία που περιλαμβάνει το block , όπως και κάθε άλλο block. Επίσης είναι εμφανές οι εξαρτήσεις που υπάρχουν μεταξύ των threads . Αυτές οι εξαρτήσεις υποδηλώνουν ποτέ ένα block θα αρχίσει να εκτελείτε στα σημεία του που περιέχει και το πότε το καθορίζουν οι εξαρτήσεις.

Κεφάλαιο 5

Ιδέα Υλοποίησης

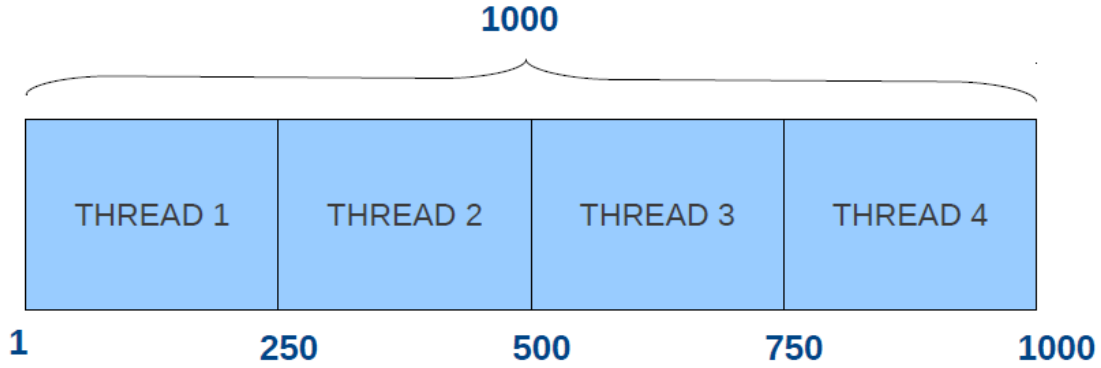
5.1 Υλοποίηση σε OpenMP

Η ιδέα και η λογική που υπάρχει με την χρησιμοποίηση του συγκεκριμένου μοντέλου στην εφαρμογή αυτή είναι να δημιουργήσει ένα σύνολο από threads και να μοιράζει την δουλεία του κομματιού στα ίσα ,και έτσι το κάθε thread να έχει την ίδια δουλεία με τα υπόλοιπα.
Το κομμάτι του κώδικα που εστιαστήκαμε είναι το μεγάλο for loop το οποίο είχε φωλιασμένο άλλο for loop . Το πρώτο εξωτερικό for loop εκτελείτε τόσες φορές όσες είναι και οι διαγώνιες γραμμές , ενώ το εσωτερικό for loop εκτελείτε όσα είναι τα σημεία σε κάθε διαγώνια γραμμή που θα βρίσκετε. Ο τρόπος που λειτουργεί το συγκεκριμένο τμήμα του κώδικα είναι να επισκέπτεται το πλέγμα και να εφαρμόζει την μαθηματική φόρμουλα , που υπολογίζει τον μέσο όρων των τεσσάρων γειτόνων διασχίζοντας το πλέγμα διαγώνια έτσι ώστε να διατηρείτε η ορθότητα και η εγκυρότητα των τελικών τιμών τα οποία θα έχουν τα σημεία.

Πιο συγκεκριμένα σε κάθε επανάληψη του εξωτερικού for loop παίρνουν τιμές οι μεταβλητές points, col και row που αναπαριστούν την σημασία των μεταβλητών αυτών μέσα από τα χαρακτηριστικά τους ονόματα. Η μεταβλητή points θα φυλάει κάθε φορά των αριθμό των σημείων που υπάρχουν στην κάθε διαγώνια γραμμή που βρίσκετε η εκτέλεση και με βάση αυτήν την τιμή εξαρτάτε οι επαναλήψεις του εσωτερικού for loop. Η μεταβλητές col και row δηλώνουν την στήλη και την γραμμή που θα αρχίσει η εκτέλεση στο εσωτερικού for loop. Μέσα στο εσωτερικού for loop προτού να εφαρμοστεί η μαθηματική απλή φόρμουλα που δημιουργεί τον μέσο όρο των τεσσάρων(4) γειτόνων για το κάθε σημείο , ορίζονται κάποιοι έλεγχοι για σημεία του πλέγματός που βρίσκονται στις τέσσερις άκρες , ή στην πρώτη γραμμή/στήλη η ακόμη στην τελευταία γραμμή/στήλη. Στην κάθε περίπτωση η μαθηματική φόρμουλα που θα υπολογίζεται στα σημεία αυτά δεν θα κάνει τον μέσο όρο των τεσσάρων

γειτόνων αλλά αντιθέτως θα κάνει αυτό με όσους γείτονες είναι εφικτοί. Για παράδειγμα το σημείο στην θέση [0][0] θα εφαρμόσει τον μέσο όρο των γειτόνων παίρνοντας την τιμή του κάτω γείτονα του ([1][0]) και του δεξιού γείτονα του ([0][1]).

Το πρόβλημα αυτό όπως το περιγράψαμε πιο πάνω θα παραλληλοποίηθει με το να τοποθετήσουμε πάνω από το εξωτερικό for loop το ειδικό OpenMP directive και έτσι θα διαμοιράσει τις επαναλήψεις στατικά στα threads όπως φαίνετε στο σχήμα 5.1 .



Σχήμα 5.1 : *Εφαρμογή του στατικού schedule στο διαμοιρασμό.*

Στο σχήμα 5.1 παρουσίαζετε ένα for loop με αριθμό επαναλήψεων ίσο με χίλιες φορές. Με την εφαρμογή του στατικού schedule τα τέσσερα threads θα πάρουν συνεχόμενα διακόσιες πενήντα(250) επαναλήψεις το κάθε ένα για να εκτελέσει. Τα threads στην υλοποίηση αυτή θα λειτουργούν όλα μαζί μέχρι το τέλος του παράλληλου κομματιού στο οποίο εμείς καθορίσαμε ποιες μεταβλητές θα πρέπει να γίνουν ιδιωτικές και ποιες όχι για την ορθή εκτέλεση ,μιας και μια ιδιωτική μεταβλητή σημαίνει ότι η συγκεκριμένη μεταβλητή θα υπάρχει αντίγραφο σε κάθε thread .

Επίσης για τον λόγο ότι εφαρμόστηκε στατικος schedule , τα threads θα πάρουν ίσο αριθμό επαναλήψεων να εκτελέσουν αλλά αυτό δεν θα σημαίνει ότι θα έχουν το ίδιο φόρτο εργασίας διότι το εσωτερικό for loop δεν εκτελείτε πάντα με τον ίδιο αριθμό επαναλήψεων αφού αυξομειώνετε ανάλογα μα σημεία που βρίσκονται στις διαγώνιες γραμμές. Για την συγκεκριμένη υλοποίηση επιλέξαμε τα τρία μεγέθη προβλήματος, δηλαδή small,medium και large.

Σκοπός της υλοποίησης αυτής είναι εκτός του ότι να παρατηρήσουμε το πώς συμπεριφέρεται το μοντέλο αυτό, θα το συγκρίνουμε με την υλοποίηση TFlux την οποία θα περιγράφετε αμέσως μετά .

5.2 Υλοποίηση σε TFlux

Η υλοποίηση της εφαρμογής ocean είναι ίσως το πιο ενδιαφέρον κομμάτι αυτού του έργου μιας και είναι ο πρότερος σκοπός της διπλωματικής αυτής. Η ιδέα στηρίχτηκε στο να μπορέσουμε να υπολογίζουμε το κάθε σημείο ενός πλέγματος με την βοήθεια των εξαρτήσεων που είναι τύπου data-dependencies. Οι εξαρτήσεις αυτές είναι οι ακριβές εξαρτήσεις στην συγκεκριμένη εφαρμογή ocean μιας και η μαθηματική φόρμουλα που ακολουθάτε για το κάθε σημείο πάνω στο πλέγμα , είναι ο μέρος όρος του αθροίσματος των τεσσάρων γειτονικών σημείων (πάνω, κάτω , δεξιά , αριστερά).

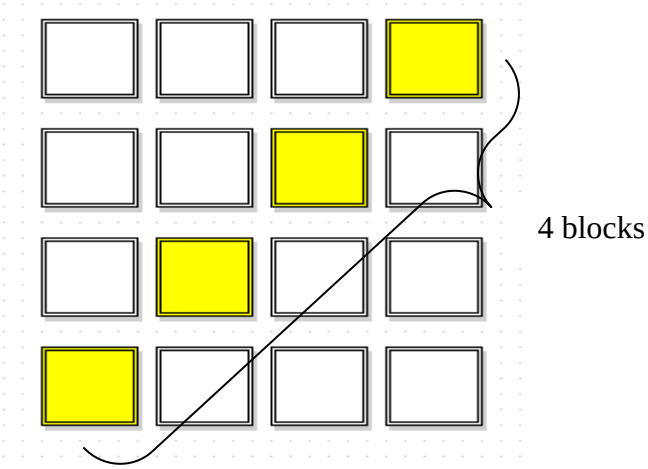
Για την συγκεκριμένη υλοποίηση , επιλέχθηκε blocking με παράγοντες του τέσσερα(4), οκτώ(8), δώδεκα(12) και δεκαέξι (16) , όλα αυτά συνδυασμένα με τρία διαφορετικά size του προβλήματός που αυτό συνεπάγετε ως τρία διαφορετικά size του πλέγματος ως small,medium και large. Η ιδέα του blocking πάρθηκε όπως προαναφέρθηκε από το ότι η εφαρμογή ocean είναι βασισμένη στο πρότυπο Wavefront. Το Wavefront είναι ένα μοτίβο προγραμματισμού που εμφανίζετε σε επιστημονικές εφαρμογές και σε αυτό το μοτίβο , τα στοιχεία των δεδομένων κατανέμονται σε πολυδιάστατα πλέγματα που αντιπροσωπεύουν ένα λογικό διάστημα. Τα στοιχεία θα πρέπει να υπολογίζονται σε σειρά επειδή υπάρχουν εξαρτήσεις μεταξύ τους. Έτσι παίρνοντας την ιδέα του πρότυπο Wavefront , χωρίστηκε το πλέγμα με τα σημεία σε blocks, και έτσι από ένα μεγάλο πλέγμα το οποίο ήτανε το αρχικό, σαν αποτέλεσμα αποχτηθήκαν αρκετά πολυδιάστατα πλέγματα.

Κάθε μπλοκ , το οποίο περιλάμβανε ένα υποσύνολο του αρχικού συνόλου του πλέγματός περιβάλλετε από ένα DThread στο οποίο εκτός από το μοναδικό αναγνωριστικό του , περιλαμβάνει το σε πιο πυρήνα θα εκτελεστεί με την έκφραση kernel #kernel_id θα και από ποιο/α DThread/s εξαρτάτε με την εξίσου έκφραση depends(#lists_of_ids_threads).Το κάθε thread ήταν υπεύθυνο για ένα ξεχωριστό block. Μέσα στο κάθε μπλοκ αφού λάμβανε ως παραμέτρους το αριθμό της στήλης και της γραμμή , που αντιπροσωπεύει την αρχή του κάθε μπλοκ, άρχιζει ο αλγόριθμος να εκτελείτε και να προσημειώνει τα σημεία που αντιστοιχούν στο συγκεκριμένο block. Η σειρά εκτέλεσης που υπολογίζει τις τιμές των σημείων η υλοποίηση αυτή είναι με τον διαγώνιο τρόπο , ο ίδιος που χρησιμοποιείτε στην υλοποίηση με OpenMP.

5.2.1 Blocking : 4x4Blocks

Στην υλοποίηση αυτή δημιουργήσαμε 4blocks στην κατεύθυνση x και 4blocks στην κατεύθυνση y αρά συνολικά υπάρχουν 16 blocks που καλύπτουν ολόκληρο το πλέγμα των

σημείων. Το κάθε block αναλαμβάνει να υπολογίσει τις τιμές των σημείων του 1/16 του ολόκληρού πλέγματος.

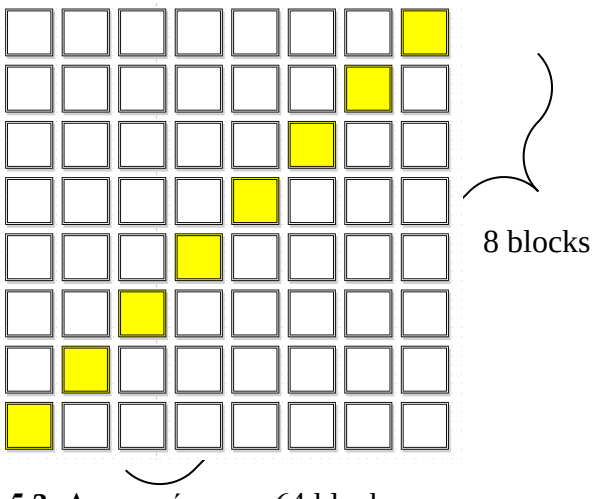


Σχήμα 5.2: Αναπαράσταση 16 blocks

Αφού υπάρχουν διαθέσιμα 16 blocks, που το κάθε ένα είναι υπεύθυνο για το 1/16 του συνολικού πλέγματος όπως αναφέραμε και πιο πάνω , αν το δούμε γραφικά στο σχήμα 5.2 , ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι τέσσερα , η κυρίως διαγώνιος που αποτελείτε από τέσσερα blocks.

5.2.2 Blocking : 8x8Blocks

Στην υλοποίηση αυτή δημιουργήσαμε 8blocks στην κατεύθυνση x και 8blocks στην κατεύθυνση y αρά συνολικά υπάρχουν 64 blocks που καλύπτουν ολόκληρο το πλέγμα των σημείων.Το κάθε block αναλαμβάνει να υπολογίσει τις τιμές των σημείων του 1/64 του ολόκληρού πλέγματος.

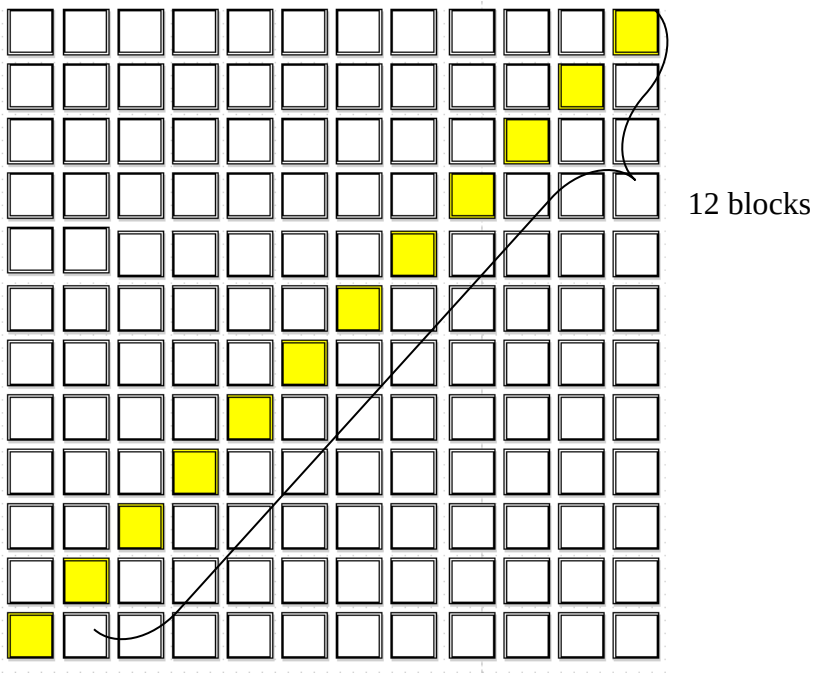


Σχήμα 5.3: Αναπαράσταση 64 blocks

Αφού υπάρχουν διαθέσιμα 64 blocks, που το κάθε ένα είναι υπεύθυνο για το 1/64 του συνολικού πλέγματος όπως αναφέραμε και πιο πάνω , αν το δούμε γραφικά στο σχήμα 5.3, ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι οκτώ , η κυρίως διαγώνιος που αποτελείτε από οκτώ blocks.

5.2.3 Blocking : 12x12 Blocks

Στην υλοποίηση αυτή δημιουργήσαμε 12blocks στην κατεύθυνση x και 12blocks στην κατεύθυνση y αρά συνολικά υπάρχουν 144blocks που καλύπτουν ολόκληρο το πλέγμα των σημείων. Το κάθε block αναλαμβάνει να υπολογίσει τις τιμές των σημείων του 1/144 του ολόκληρού πλέγματος.

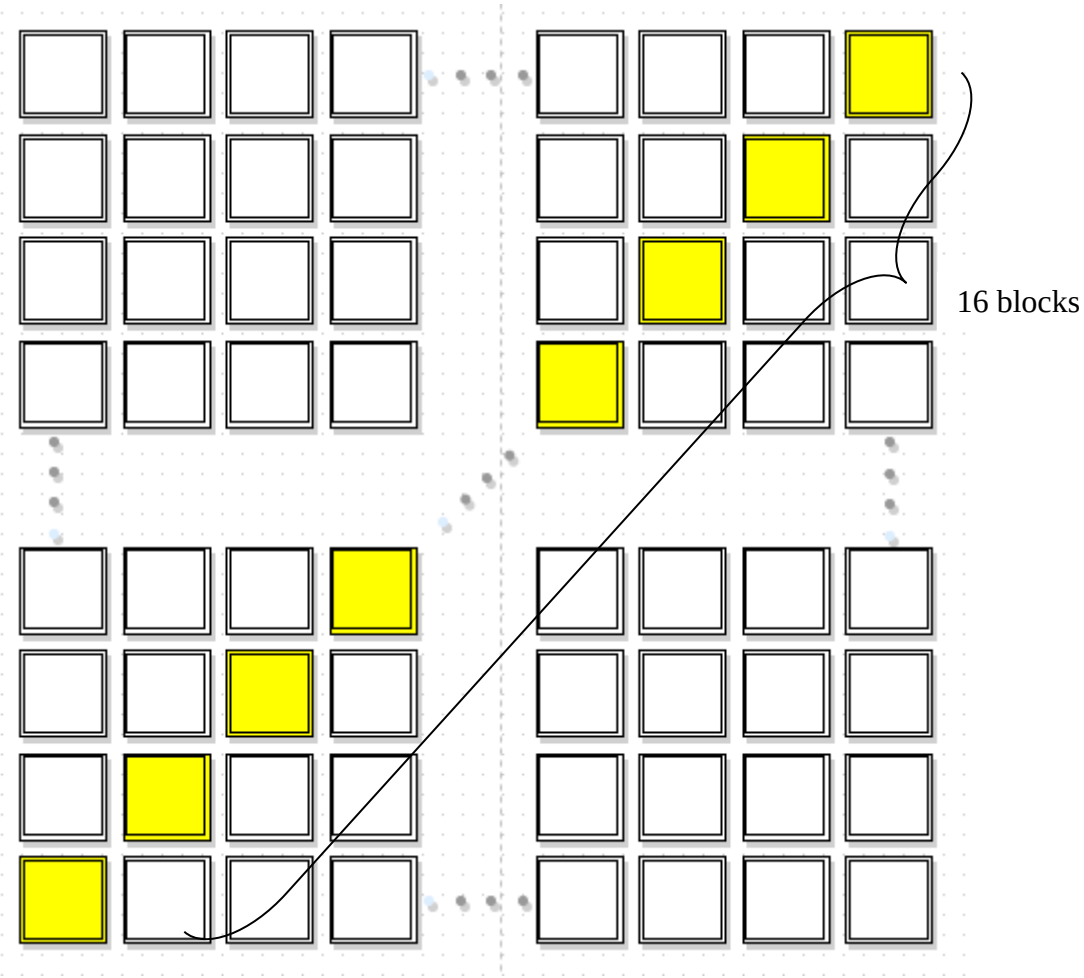


Σχήμα 5.4: Αναπαράσταση 144 blocks

Αφού υπάρχουν διαθέσιμα 144 blocks, που το κάθε ένα είναι υπεύθυνο για το 1/144 του συνολικού πλέγματος όπως αναφέραμε και πιο πάνω , αν το δούμε γραφικά στο σχήμα 5.4 , ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι δώδεκα , η κυρίως διαγώνιος που αποτελείτε από δώδεκα blocks.

5.2.4 Blocking : 16x16 Blocks

Στην υλοποίηση αυτή δημιουργήσαμε 16blocks στην κατεύθυνση x και 16blocks στην κατεύθυνση y αρά συνολικά υπάρχουν 256blocks που καλύπτουν ολόκληρο το πλέγμα των σημείων. Το κάθε block αναλαμβάνει να υπολογίσει τις τιμές των σημείων του 1/256 του ολόκληρού πλέγματος.



Σχήμα 5.5: Αναπαράσταση 256 blocks

Αφού υπάρχουν διαθέσιμα 256 blocks, που το κάθε ένα είναι υπεύθυνο για το 1/256 του συνολικού πλέγματος όπως αναφέραμε και πιο πάνω , αν το δούμε γραφικά , ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι δεκαέξι , η κυρίως διαγώνιος που αποτελείτε από δεκαέξι blocks.

Ο τρόπος που μετατραπήκανε σε παράλληλες υλοποιήσεις και οι τέσσερις(4) υλοποιήσεις TFlux που τις περιγράψαμε πιο πάνω ήτανε σε όλες κοινός. Αναθέταμε την πρώτη σειρά στην οριζόντια κατεύθυνση στο Kernel #1 , την δεύτερη στον Kernel #2 και ούτω κάθε εξής. Το ίδιο θα ήταν αν αναθέταμε σειρές στην κάθετη κατεύθυνση. Με αυτό τον τρόπο επιτυγχάνετε , τα blocks που αποτελούν κάθε διαγώνιο να τρέχουν σε διαφορικό kernel εφαρμόζοντας ένας είδος pipeline. Πρακτικά αυτή η διαδικασία γινόταν με το να αναθέταμε τα threads που ήτανε υπεύθυνα για τα blocks στους κατάλληλους kernels. Επίσης σκοπός της υλοποίησης αυτής είναι τα κομμάτια που μπορούν να εκτελούνται παράλληλα αφού καθοριστούν οι ακριβές συναρτήσεις να κατανέμονται σε διαφορετικούς πυρήνες για μεγαλύτερη απόδοση.

Κεφάλαιο 6

Αξιολόγηση

6.1 Διαδικασία Πειραματικής Διάταξής .....

Ο κύριος στόχος της εργασίας μας ήταν η μετατροπή της εφαρμογής OCEAN σε μια data-driven εφαρμογή με την βοήθεια της πλατφόρμας TFlux έτσι ώστε η εφαρμογή αυτή να είναι υλοποιημένη με τέτοιο τρόπο έτσι ώστε να υπάρχουν εξαρτήσεις δεδομένων ,και για να είναι εφικτό να υπολογιστεί η τιμή ενός σημείου του πλέγματος ,απαραίτητη προϋπόθεση να είναι έτοιμες και ανανεωμένες τιμές των θέσεων στις οποίες εξαρτάτε.
Για την αξιολόγηση του έργου μας μετατρέψαμε την ddm εφαρμογή που προέκυψε από την προσθήκη των directives σε διαφορές παραλλαγές με διαφορετικό παράγοντα για blocking και με διαφορετικά size του προβλήματος.

Όσο αφορά τη μελέτη της επίδοσης της υλοποίησης μας , για κάθε DDM εφαρμογή από το σύνολο που δημιουργήσαμε από τους πιο πάνω συνδυασμούς ,μετρήσαμε τους μέσους χρόνους εκτέλεσης σε διαφορετικό αριθμό πυρήνων και τους συγκρίναμε μεταξύ τους. Για την εξαγωγή του μέσου χρόνου εκτέλεσης μιας DDM εφαρμογής X σε ένα συγκεκριμένο πλήθος πυρήνων K ακλουθήσαμε τη συνηθισμένη διαδικασία εύρεσης μέσου όρου. Τρέξαμε την X σε K πυρήνες 10 φορές λαμβάνοντας χρόνο εκτέλεσης για την καθεμία ξεχωριστά . Ακολούθως αφαιρέσαμε τον μεγαλύτερο και τον μικρότερο χρόνο εκτέλεσης και από τους υπόλοιπους οκτώ χρόνους υπολογίσαμε τον μέσο όρο. Το αποτέλεσμα είναι ο μέσος χρόνος εκτέλεσης της X εφαρμογής σε K πυρήνες. Για την κάθε εφαρμογή θα συγκρίνουμε τους μέσους χρόνους εκτέλεσης καθώς αυξάνατε ο αριθμός των πυρήνων με σκοπό αν όντως πετυχαίνεται βελτίωση της επίδοσης. Επίσης συγκρίναμε τους χρόνους αυτούς με το χρόνο εκτέλεσης της αντίστοιχης σειριακής εφαρμογής σε C κώδικα που δεν κάνει καθόλου χρήση της πλατφόρμας TFlux. Μέσω αυτών των συγκρίσεων εξάγαμε γραφικές παραστάσεις σχετικά με την βελτίωση της επίδοσης

(speedup) καθώς αυξάνονται οι πυρήνες , που ουσιαστικά σημαίνει την αύξηση του βαθμού παραλληλισμού.

6.2 Παραλλαγές Πειραμάτων & Πειραματική Διάταξη

Για την αξιολόγηση της υλοποίηση μας, χρησιμοποιήσαμε για την ίδια εφαρμογή μας , την Ocean σε δυο διαφορετικές υλοποιήσεις της σε TFlux και OpenMP. Οι δύο υλοποιήσεις αυτές επιλεχθήκανε για το λόγο ότι μοιάζουνε αρκετά τα directives που υπάρχουν διαθέσιμα και για τα δύο μοντέλα και επίσης στο ότι και τα δύο αυτά μοντέλα βασίζονται σε pthreads (threads). Παρόλα αυτά, έχουν διαφορές τα μοντέλα όπως έχουν περιγραφεί σε προηγούμενα κεφάλαια.

Οι δύο αυτές υλοποιήσεις εκτελέστηκαν στην μηχανή cs6473 .Η συγκεκριμένη μηχανή περιέχει 12 επεξεργαστές με λειτουργικό σύστημα Ubuntu 12.04.2 LTS 12.04. Οι επεξεργαστές είναι ίδιοι μεταξύ τους με κοινά χαρακτηρίστηκα. Μερικά από τα χαρακρηριτκά που διαθέτουν είναι ότι το μοντέλο τους είναι το νούμερο 8 με όνομα Six-Core AMD Opteron(tm) Processor 2427, η συχνότητα του cpu τους είναι 2194.612 MHz, το μέγεθός της cache είναι 512 KB και όσο αφορά τις διευθύνεις τους τόσο σε φυσικές όσο και τις εικονικές έχουν μέγεθος 48 bits. Επίσης η έκδοση του compiler που χρησιμοποιείτε είναι gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) και με thread μοντέλο το posix.

Στο Πίνακα (Σχήμα 6.1) παρουσιάζονται οι δυο αυτές υλοποιήσεις με όλα τα με τα μεγέθη που χρησιμοποιήθηκαν για το πλέγμα των σημείων και επίσης τις παραλλαγές του blocking που εφαρμόστηκε. Οι διάφορες μετρήσεις έγινε λαμβάνοντας υπόψη μόνο το χρόνο εκτέλεσης του παράλληλου μέρους της κάθε υλοποίησης αφού αυτό είναι το κομμάτι που μας ενδιαφέρει. Οποιαδήποτε σειριακά μέρη , όπως επαληθεύεις και εκτυπώσεις αποτελεσμάτων , έχουν αγνοηθεί.

		Input Sizes		
OpenMP	OpenMP	24000	48000	72000
TFlux	TFlux με 4x4blocks (144 blocks)	4 blocks μεγέθους 6000	4 blocks μεγέθους 12000	4 blocks μεγέθους 18000
	TFlux με 8x8 blocks (64 blocks)	8 blocks μεγέθους 3000	4 blocks μεγέθους 6000	4 blocks μεγέθους 9000
	TFlux με 12x12 blocks (144 blocks)	12 blocks μεγέθους 2000	12 blocks μεγέθους 4000	12 blocks μεγέθους 6000
	TFlux με 16x16blocks (256 blocks)	16 blocks μεγέθους 1500	16 blocks μεγέθους 3000	16 blocks μεγέθους 4500

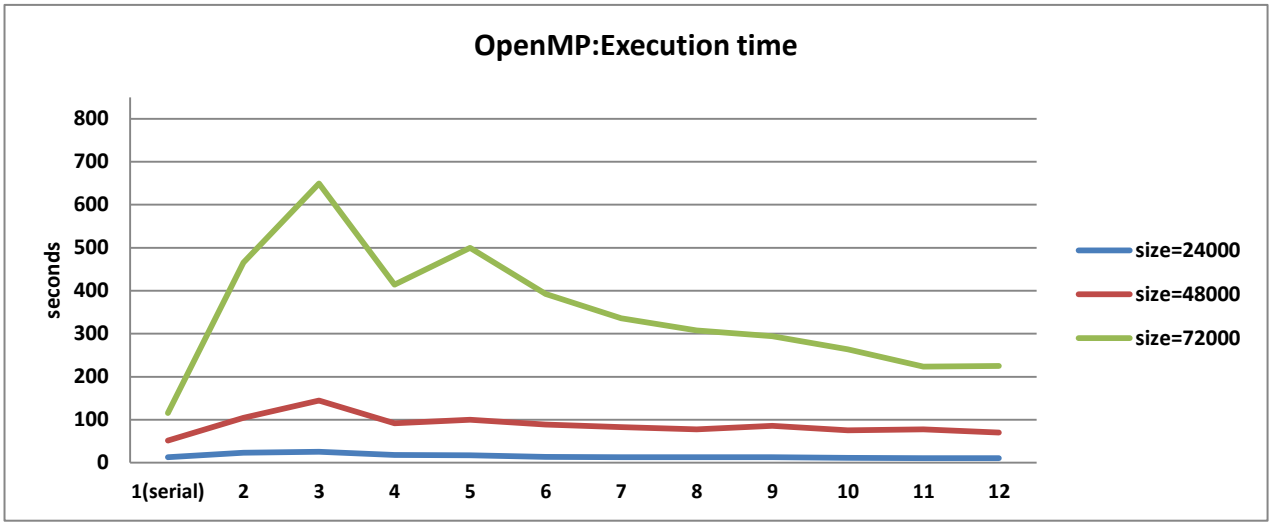
Σχήμα 6.1*: Μεγέθη πλέγματος*

6.3 Παρουσίαση και Ανάλυση Αποτελεσμάτων

Στο σχήμα 6.3(α) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με OpenMP για την σειριακή εκτέλεση σε σύγκριση με το να αλλάζετε κάθε φορά ο αριθμός των threads.

Με μπλε χρώμα αντιπροσωπεύεται ο χρόνος εκτέλεσης με μέγεθος 24000. Αντίστοιχα το κόκκινο χρώμα αντιπροσωπεύει το χρόνο εκτέλεσης με μέγεθος 48000, ενώ το πράσινο για μέγεθος 72000.Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα του OpenMP σε σχέση με τα threads που χρησιμοποιούνται και την επίδραση του μοντέλου σε σύγκριση με την σειριακή εκτέλεση.

Παρατηρούμε ότι ο χρόνος έχει αυξηθεί σε σύγκριση με το καλύτερο σειριακό που είχαμε σαν baseline. Εντονότερα αυτή η αύξηση φαίνετε στο μεγάλο μέγεθος (large) . Παρατηρούμε επίσης ότι καθώς χρεισιμοποίηουμε περισσότερα threads παρατηρείτε μείωση αλλά δεν παύει να είναι μικρότερη του σειριακού χρόνου. Αιτία του γεγονότος του γεγονότος αυτού είναι ότι το καλύτερο σειριακό που έχουμε σαν baseline εφαρμόζει row major καθώς διασχίσει το πίνακα , πράγμα που επιτυγχάνει γρηγορότερη πρόσβαση στην μνήμη. Αντιθέτως στην υλοποίηση μας με OpenMP διασχίσει τον πίνακα με διαγώνιο τρόπο και έτσι υπάρχει αυξήση στην πρόσβαση μνήμης για την ενημέρωση των σημείων.

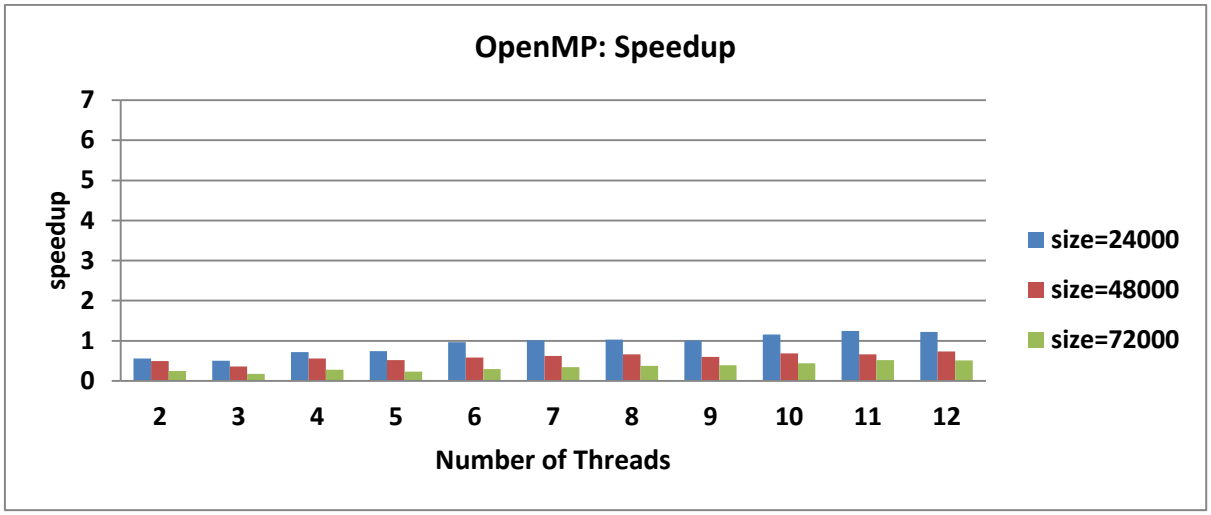


Σχήμα 6.3(α) *: Χρόνοι εκτέλεσης OpenMP υλοποίησης*

Είναι φανερό ότι σχετικά για το μικρό μέγεθος εισόδου , συγκεκριμένα 24000 , ο χρόνος εκτέλεσης όσο αυξάνουμε τα threads διατηρείτε στα ίδια επίπεδα περίπου με αυτό της σειριακής εκτέλεσης. Ο λόγος είναι ότι το κάθε thread αναλαμβάνει ένα μικρότερο κομμάτι του πλέγματος για να υπολογίζει την νέα τιμη των σημείων και έτσι η καθυστέρηση που δημιουργείτε από την πρόσβαση της μνήμης επικαλύπτετε κάπως αφού μοιράζετε το πρόβλημα στα threads.

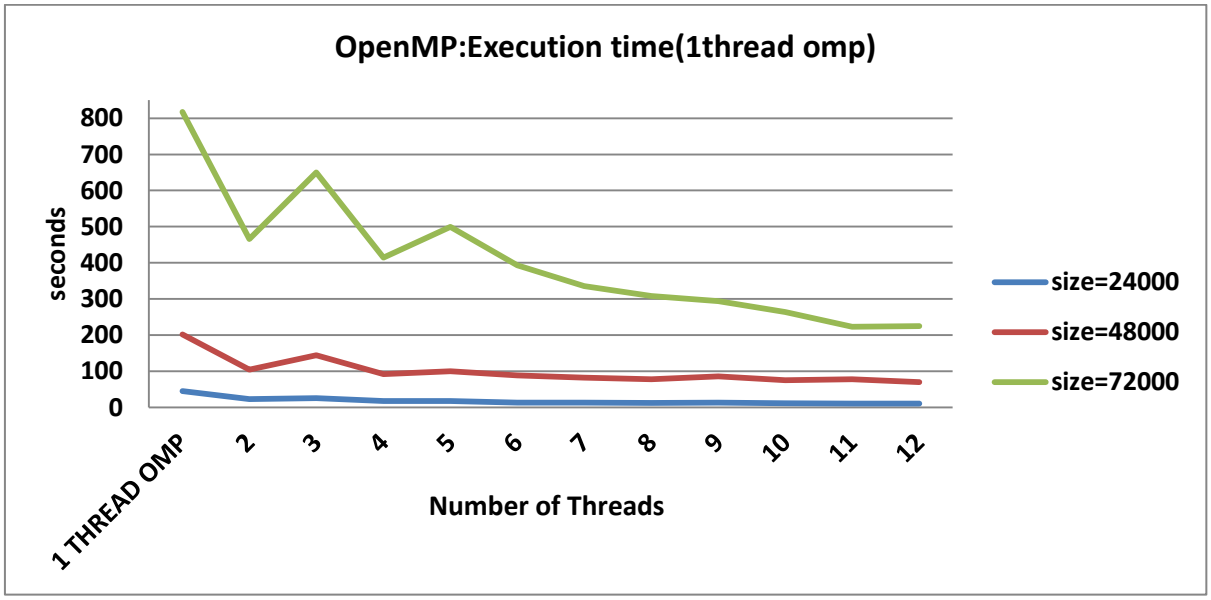
Όταν όμως ο αριθμός των threads γίνει ίσος με 3 βλέπουμε πως ο χρόνος αυξήθηκε σε σχέση με το να είχαμε 2 threads. Ο λόγος είναι ότι επειδή οι επαναλήψεις μοιράζονται στα ίσα στα 3 αυτά threads, και επιπρόσθετα η προσπέλαση στο πλέγμα των σημείων γίνετε με διαγώνιο τρόπο , τότε ένα από τα 3 threads θα εκτελέσει περισσότερη δουλεία από τα υπόλοιπα δύο και έτσι δεν θα έχουν ίση δουλειά για να τελειώσουν και τα 3 το ίδιο. Όταν όμως ο αριθμός των threads γίνει ίσος με 4 βλέπουμε πως μειώνετε ο χρόνος σε σύγκριση με το να είχαμε 3 threads και λίγο ελάχιστα με το να είχαμε 2 threads. Ο λόγος είναι ότι όπως προαναφέραμε οι επαναλήψεις μοιράζονται στα ίσα με τον αριθμό των threads, και σε αυτή την περίπτωση μοιράζονται στα ίσα με τα 4 threads. Τα δύο από αυτά θα έχουν το ίδιο φόρτο εργασίας , αφού αναφερόμαστε στα δύο threads τα οποία βρίσκονται στην αρχή και στο τέλος του πλέγματος με την φορά που εκτελείτε και έτσι έχουν λιγότερα σημεία να υπολογίσουν από τα υπόλοιπα 2 τα οποία μοιράζονται το μεγάλο χώρο πριν και μετά την κύρια διαγώνιο που σίγουρα εκεί περά υπάρχουν πολύ περισσότερα σημεία.

Στο σχήμα 6.3(β) , μας παρουσιάζετε η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής, δηλαδή με OpenMP. Όσο αυξάνουμε τον αριθμό των threads μπορούμε να πούμε πως η επίδοση βελτιώνετε ελάχιστα στο μικρότερο μέγεθός προβλήματος που αναπαριστάτε με την μπλε απόχρωση .



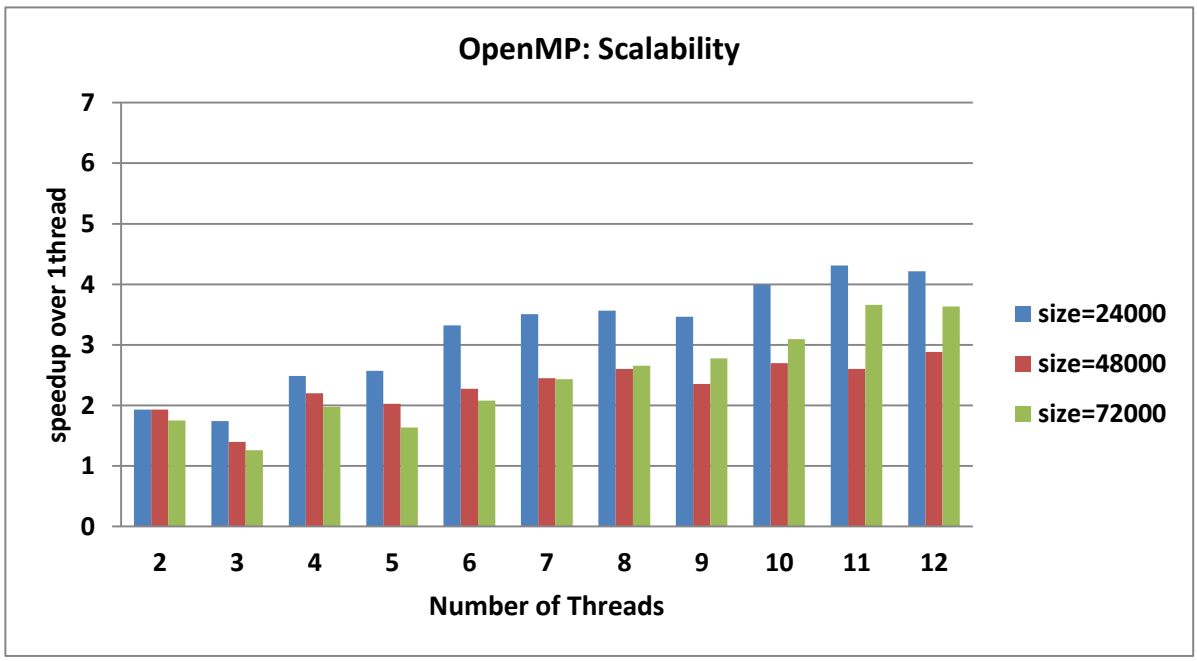
Σχήμα 6.3(β) *: Επίδοση OpenMP υλοποίησης.*

Στο σχήμα 6.3(γ) μας παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με OpenMP σε σύγκριση με την εκτέλεση της ίδιας υλοποιήσεις με ένα thread. Βλέπουμε το κόστος να εκτελεστεί η εφαρμογή αυτή με ένα thread σε OpenMP . Παρατηρούμε πώς ο χρόνος μειώνετε καθώς προσθέτουμε threads με το ίδιο τρόπο όπως και στο σχήμα 6.2 (α).



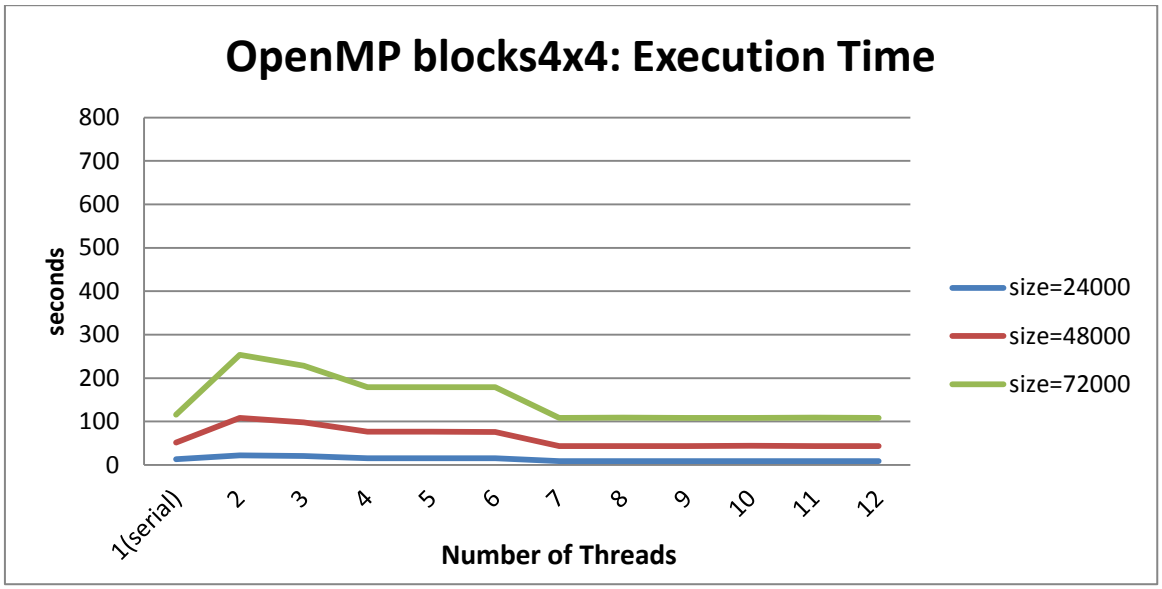
Σχήμα 6.3(γ): *Χρόνοι εκτέλεσης OpenMP υλοποίησης σε σύγκριση με την εκτέλεση σε ένα(1) thread.*

Στο σχήμα 6.3(δ) παρατηρούμε το scalability του προηγούμενος σχήματος που συγκρίνει την υλοποίηση χρησιμοποιώντας δύο μέχρι δώδεκα threads με το να χρησιμοποιείτε μόνο ένα thread. Παρατηρούμε σε αυτή την περίπτωση ότι το scalability αυξάνετε καθώς αυξάνονται τα threads και έχει μέγιστο scalability μέχρι και τέσσερις (και λίγο περισσότερο) φορές.



Σχήμα 6.2(δ): *Scalability OpenMP υλοποίησης*

Στο σχήμα 6.2(ε) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με OpenMP blocks4x4 για την σειριακή εκτέλεση σε σύγκριση με το να αυξάνετε κάθε φορά ο αριθμός των threads.

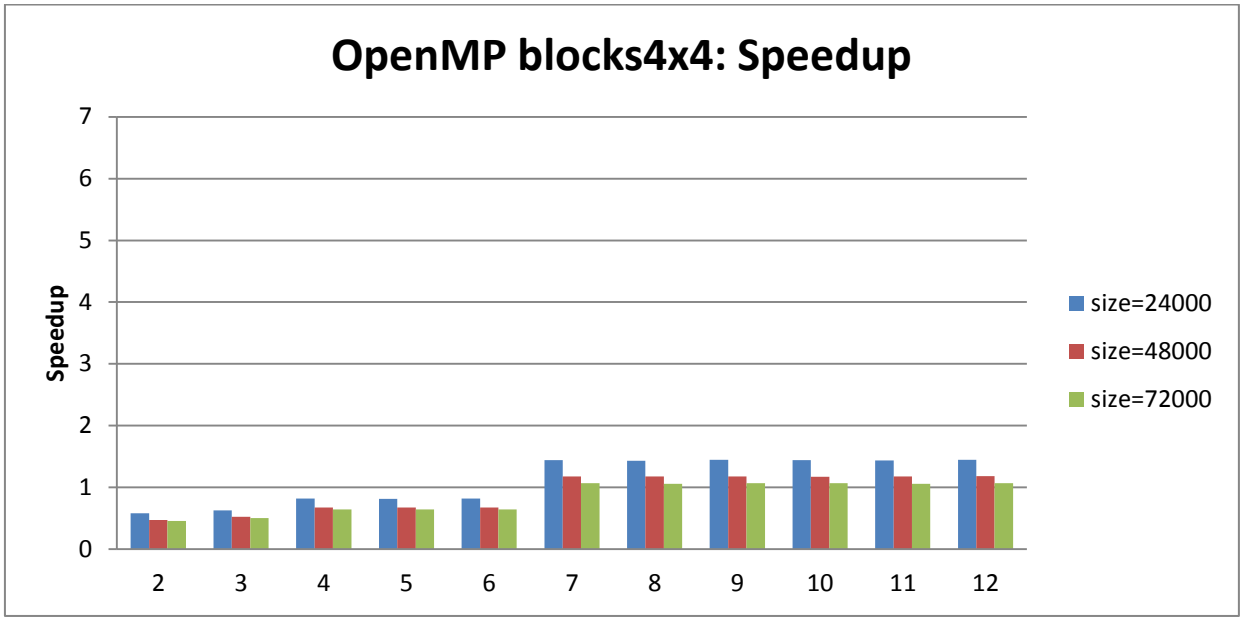


Σχήμα 6.2(ε): *Χρόνοι εκτέλεσης OpenMP υλοποίησης με blocking 4x4*

Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας OpenMP και συγκεκριμένα έχοντας χωρίσει το πλέγμα μας σε 4x4 Blocks (συνολικά 16 Blocks) σε σχέση με τα threads που χρησιμοποιούνται . Παρατηρούμε ότι ο χρόνος έχει αυξηθεί σε σύγκριση με το σειριακή εκτέλεση όταν χρησιμοποιούμε δύο threads . Ο λόγος οφείλετε στο κόστος δημιουργίας των threads και καθώς επίσης όσο λιγότερα threads τόσο περισσότερη δουλεία έχει το κάθε ένα. Επίσης παρατηρείτε ότι καθώς αυξάνονται ο αριθμός των threads μειώνετε ο χρόνος εκτέλεσης .

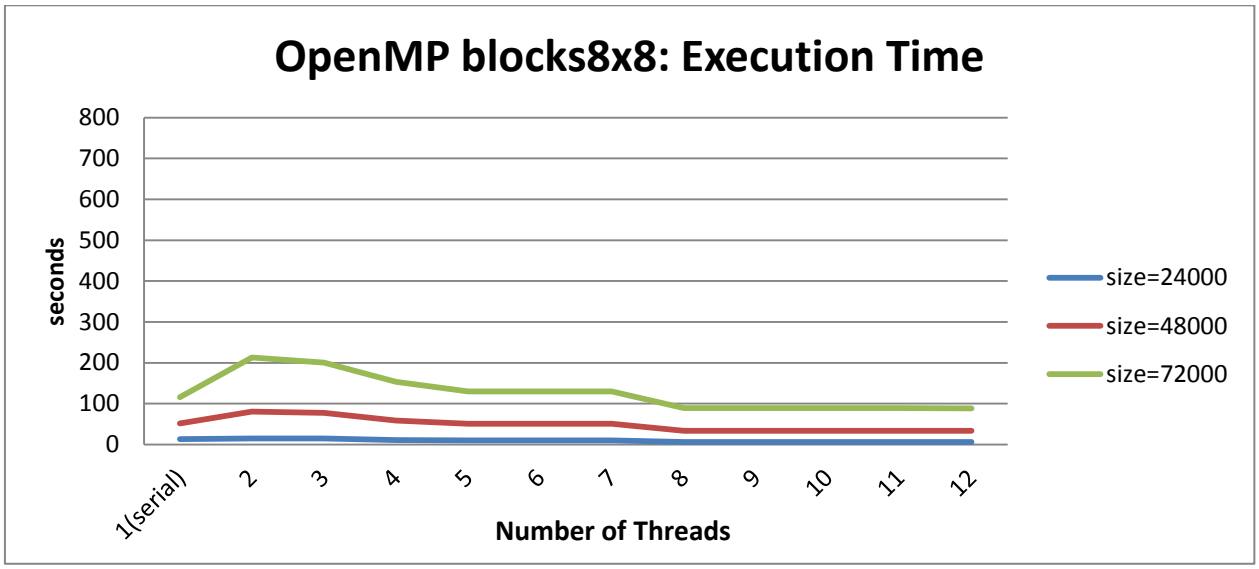
Το σχήμα 6.2(ζ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των threads άρχισε να παρατηρείτε μια μικρή βελτίωση η οποία βελτίωση ξεπερνά τη επίδοση σε βαθμό ένα(1) όταν εκτελούνται εφτά threads και περισσότερα. Ο λόγος είναι ότι οι διαγώνιες γραμμές που

υπάρχουν εδώ είναι εφτα(7) και με αριθμό threads ίσο με εφτά επιτυγχάνετε στο σημείο αυτό και πέρα επίδοση η οποία μένη σταθερή και δεν επηρεάζετε περεταίρω με καθώς αυξάνονται ο αριθμός των threads. Η μέγιστη επίδοση που επιτυγχάνετε είναι 1,44≈1.



Σχήμα 6.2(ζ): *Επίδοση OpenMP υλοποίησης με blocking 4x4*

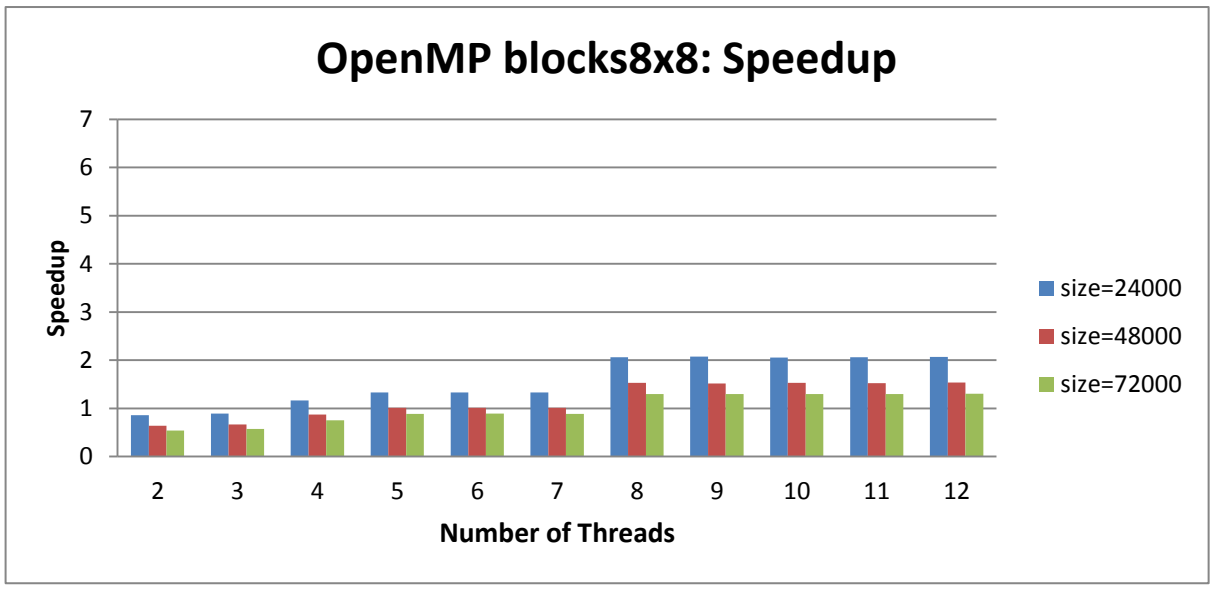
Στο σχήμα 6.2(η) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με OpenMP blocks8x8 για την σειριακή εκτέλεση σε σύγκριση με το να αυξάνετε κάθε φορά ο αριθμός των threads.



Σχήμα 6.2(η): *Χρόνοι εκτέλεσης OpenMP υλοποίησης με blocking 8x8*

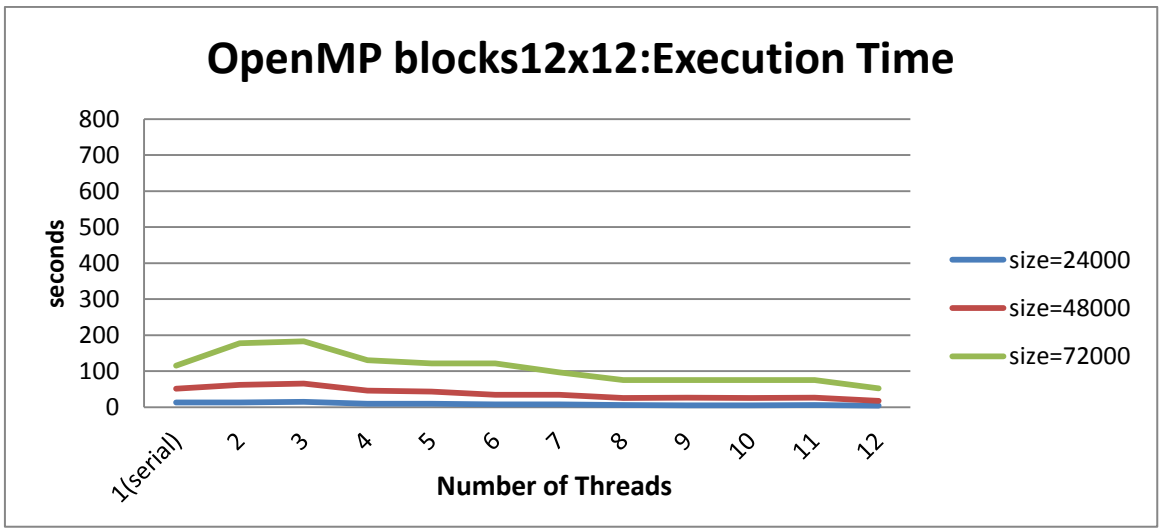
Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας OpenMP και συγκεκριμένα έχοντας χωρίσει το πλέγμα μας σε 8x8 Blocks (συνολικά 64 Blocks) σε σχέση με τα threads που χρησιμοποιούνται . Παρατηρούμε ότι ο χρόνος έχει αυξηθεί σε σύγκριση με το σειριακή εκτέλεση όταν χρησιμοποιούμε δύο threads . Ο λόγος οφείλετε στο κόστος δημιουργίας των threads και καθώς επίσης όσο λιγότερα threads τόσο περισσότερη δουλεία έχει το κάθε ένα. Επίσης παρατηρείτε ότι καθώς αυξάνονται ο αριθμός των threads μειώνετε ο χρόνος εκτέλεσης .

Το σχήμα 6.2(θ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των threads άρχισε να παρατηρείτε μια μικρή βελτίωση η οποία βελτίωση ξεπερνά τη επίδοση σε βαθμό ένα(1) όταν εκτελούνται τέσσερα threads και περισσότερα. Ο λόγος είναι ότι οι διαγώνιες γραμμές που υπάρχουν εδώ είναι δεκαπέντε(15) και έτσι μαζί τέσσερα threads μπορούν να τρέχουν σε περισσότερες διαγώνιες γραμμές από πριν. Η μέγιστη επίδοση η οποία είναι 2,05≈2 επιτυγχάνεται όταν χρησιμοποιούνται οκτώ(8) threads. Ο λόγος είναι ότι ο αριθμός των blocks που υπάρχουν στην κύρια διαγώνιο είναι οκτώ και από εκεί και πέρα η επίδοση διατηρείται στο ίδιο σημείο καθώς αυξάνετε ο αριθμός των threads.



Σχήμα 6.2(θ): *Επίδοση OpenMP υλοποίησης με blocking 8x8*

Στο σχήμα 6.2(ι) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με OpenMP blocks12x12 για την σειριακή εκτέλεση σε σύγκριση με το να αυξάνετε κάθε φορά ο αριθμός των threads.

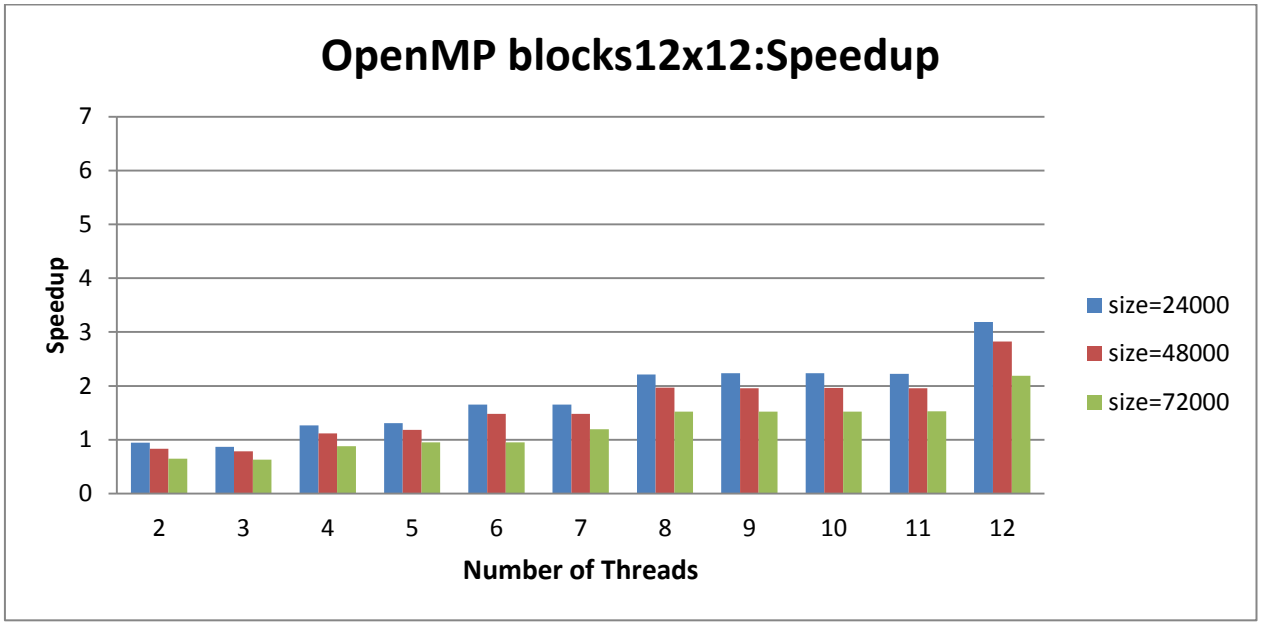


Σχήμα 6.2(ι): *Χρόνοι εκτέλεσης OpenMP υλοποίησης με blocking 12x12*

Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας OpenMP και συγκεκριμένα έχοντας χωρίσει το πλέγμα μας σε 12x12 Blocks (συνολικά 144 Blocks) σε σχέση με τα threads που χρησιμοποιούνται . Παρατηρούμε ότι ο χρόνος έχει

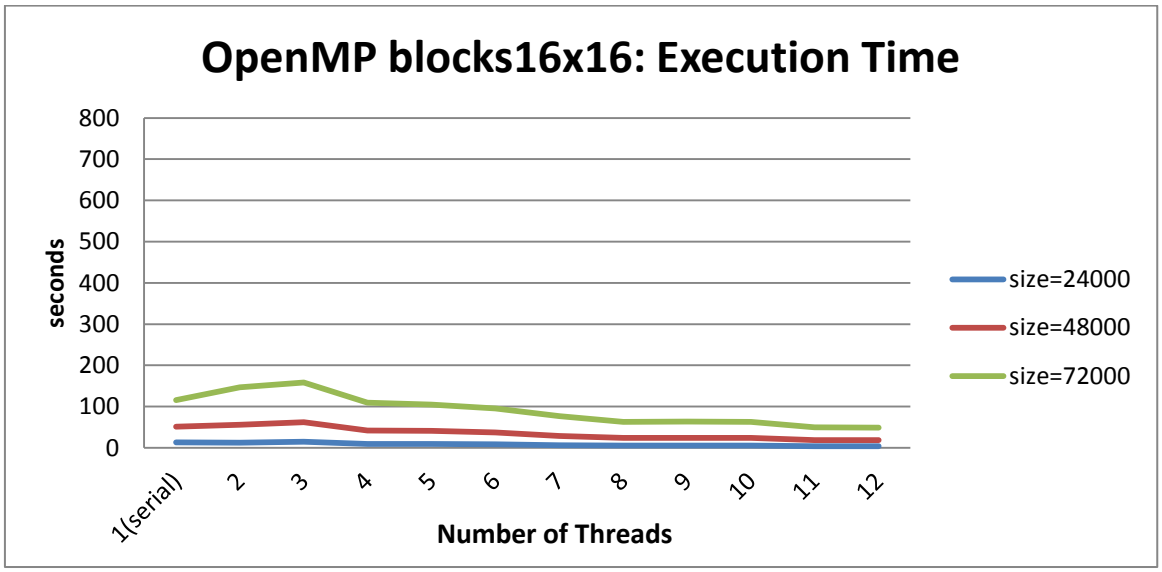
αυξηθεί σε σύγκριση με το σειριακή εκτέλεση όταν χρησιμοποιούμε δύο threads . Ο λόγος οφείλετε στο κόστος δημιουργίας των threads και καθώς επίσης όσο λιγότερα threads τόσο περισσότερη δουλεία έχει το κάθε ένα. Επίσης παρατηρείτε ότι καθώς αυξάνονται ο αριθμός των threads μειώνετε ο χρόνος εκτέλεσης .

Το σχήμα 6.2(κ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των threads άρχισε να παρατηρείτε μια μικρή βελτίωση η οποία βελτίωση ξεπερνά τη επίδοση σε βαθμό ένα(1) όταν εκτελούνται τέσσερα threads και περισσότερα. Ο λόγος είναι ότι οι διαγώνιες γραμμές που υπάρχουν εδώ είναι εκατοσαραντατρία(143) και έτσι μαζί τέσσερα threads μπορούν να τρέχουν σε περισσότερες διαγώνιες γραμμές από πριν. Η μέγιστη επίδοση η οποία είναι 3,18≈3 επιτυγχάνετε όταν χρησιμοποιούνται δώδεκα(12) threads. Ο λόγος είναι ότι ο αριθμός των blocks που υπάρχουν στην κύρια διαγώνιο είναι δώδεκα .



Σχήμα 6.2(κ): *Επίδοση OpenMP υλοποίησης με blocking 12x12*

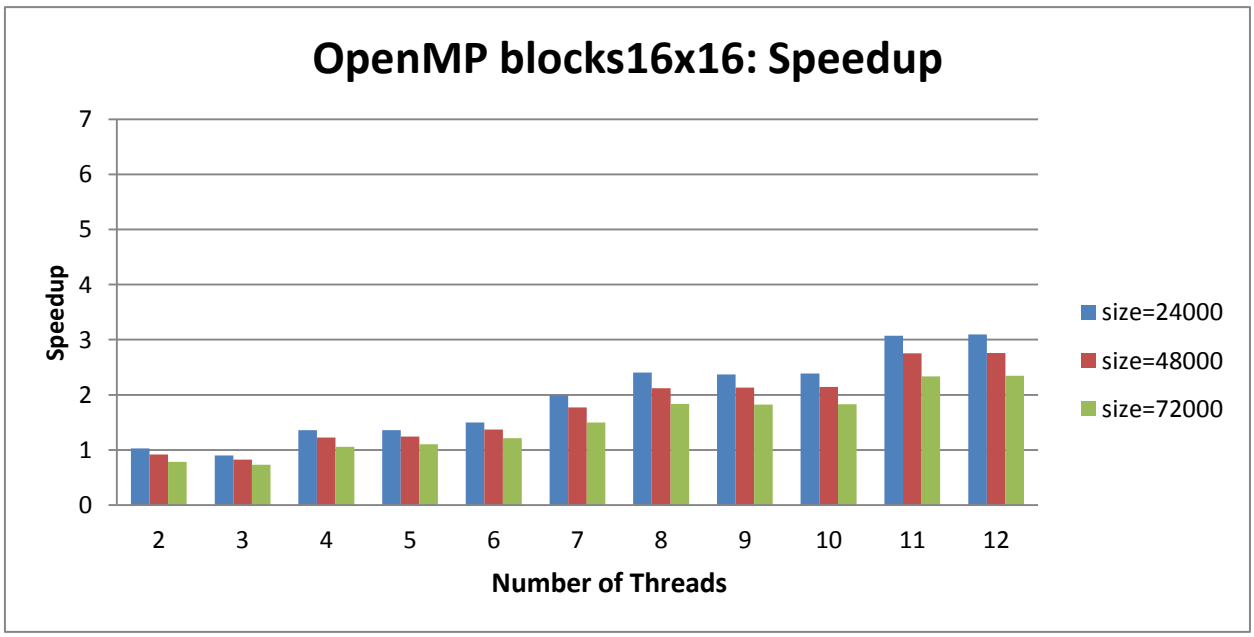
Στο σχήμα 6.2(ι) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με OpenMP blocks16x16 για την σειριακή εκτέλεση σε σύγκριση με το να αυξάνετε κάθε φορά ο αριθμός των threads.



Σχήμα 6.2(λ): *Χρόνοι εκτέλεσης OpenMP υλοποίησης με blocking 16x16*

Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας OpenMP και συγκεκριμένα έχοντας χωρίσει το πλέγμα μας σε 16x16 Blocks (συνολικά 256 Blocks) σε σχέση με τα threads που χρησιμοποιούνται . Παρατηρούμε ότι ο χρόνος έχει αυξηθεί σε σύγκριση με το σειριακή εκτέλεση όταν χρησιμοποιούμε δύο-τρία threads . Ο λόγος οφείλετε στο κόστος δημιουργίας των threads και καθώς επίσης όσο λιγότερα threads τόσο περισσότερη δουλεία έχει το κάθε ένα. Επίσης παρατηρείτε ότι καθώς αυξάνονται ο αριθμός των threads μειώνετε ο χρόνος εκτέλεσης .

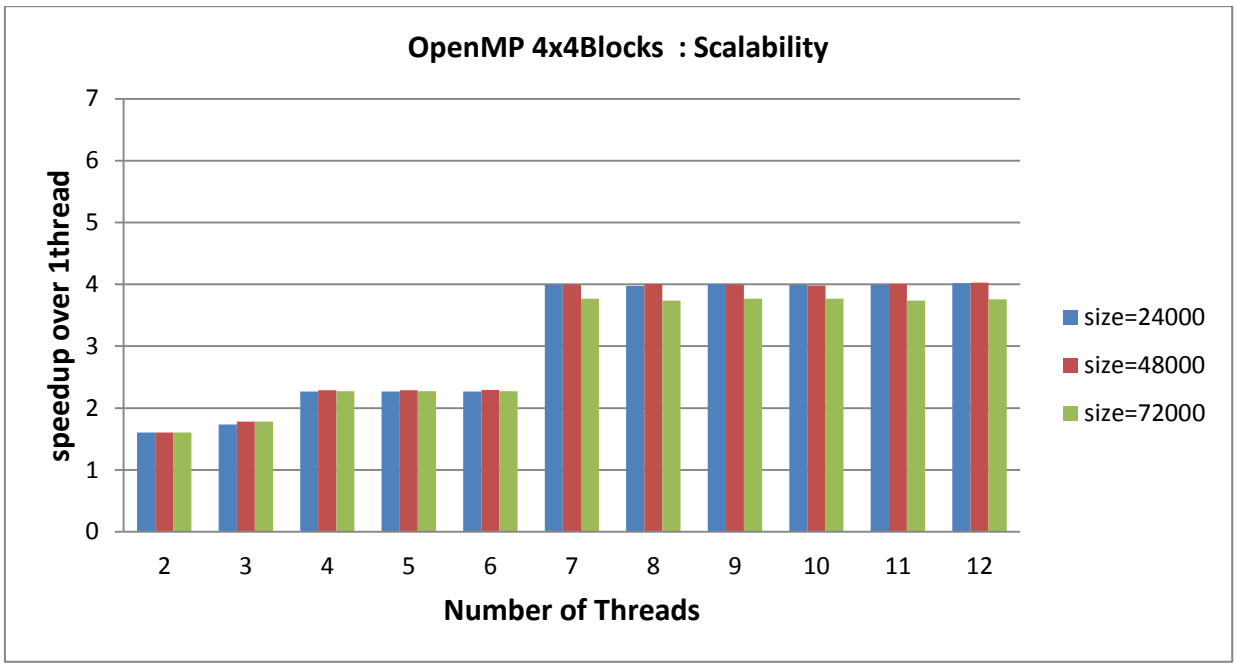
Το σχήμα 6.2(μ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των threads άρχισε να παρατηρείτε μια μικρή βελτίωση η οποία βελτίωση ξεπερνά τη επίδοση σε βαθμό ένα(1) όταν εκτελούνται τέσσερα threads και περισσότερα. Ο λόγος είναι ότι οι διαγώνιες γραμμές που υπάρχουν εδώ είναι διακοσιαπενήνταπέντε(255) και έτσι μαζί τέσσερα threads μπορούν να τρέχουν σε περισσότερες διαγώνιες γραμμές από πριν. Η μέγιστη επίδοση η οποία είναι 3,09≈3 επιτυγχάνετε όταν χρησιμοποιούνται έντεκα και δώδεκα threads. Ο λόγος είναι ότι ο αριθμός των blocks που υπάρχουν στην κύρια διαγώνιο είναι δεκαέξι πράγμα που επιτρέπει όταν χρησιμοποιούνται έντεκα και δώδεκα threads να τρέχουν σε περισσότερες διαγώνιες και όχι μόνο στην κυρία διαγώνιο . Επίσης παρατηρείτε ότι με τρία threads έχει λίγη λιγότερη επίδοση από ότι υπάρχει με το να χρησιμοποιούνται δύο threads. Ο λόγος είναι ότι στις μόνες περιπτώσεις που χρησιμοποιούνται ακριβώς και τα τρία threads στην περίπτωση που δηλώσουμε για αριθμό threads ίσο με 3 είναι σε διαγώνιες γραμμές που έχει σημεία για προσημείωση πολλαπλάσιο του τρία, δηλαδή μόνο στις διαγώνιους οι οποίοι έχουν 3,6,9 και 12 σημεία. Σε οποιεσδήποτε άλλες διαγώνιες γραμμές τα σημεία που υπολείπονται θα εκτελεστούν όταν θα τελειώσουν από τα άλλα σημεία που ανέλαβαν τα threads.



Σχήμα 6.2(μ): *Επίδοση OpenMP υλοποίησης με blocking 16x16*

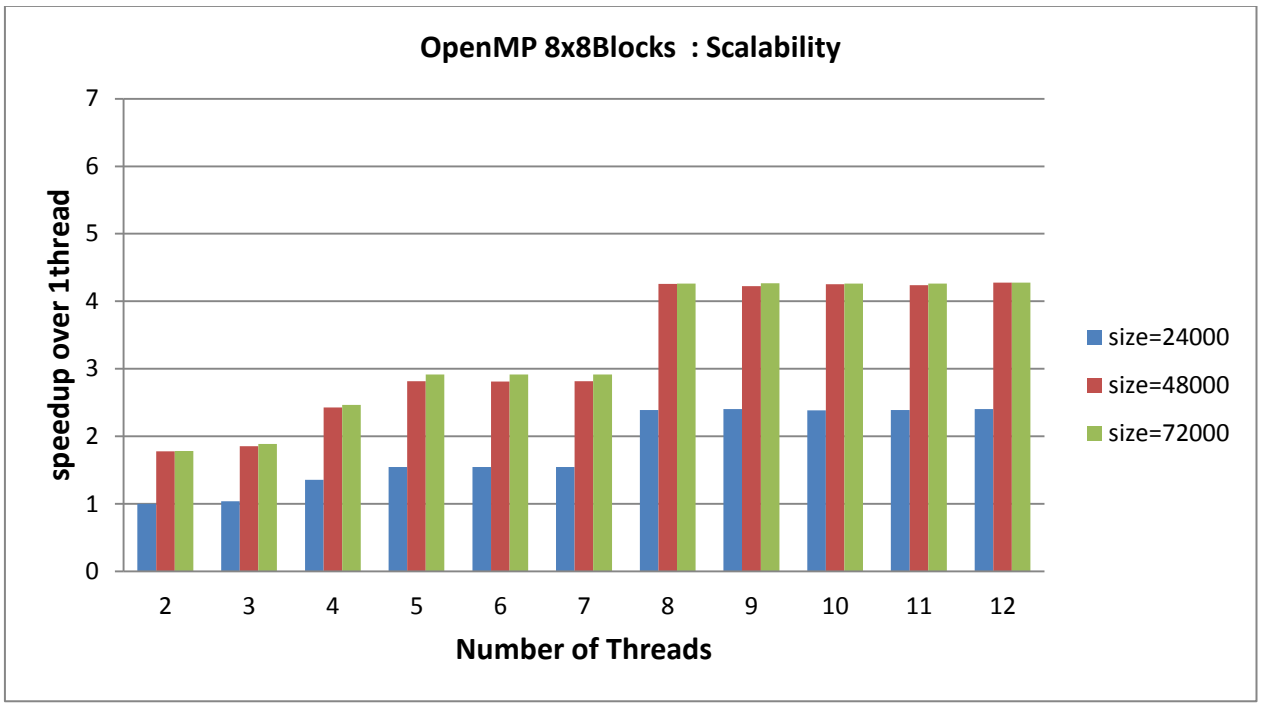
Οι πιο κάτω γραφικές που ακολουθούν παρουσιάζουν το scalability της υλοποίησεις με OpenMP με κάθε παραλλαγή του blocking που δημιούργήθηκε.(σχήμα 6.2(ν)-(π)).

Στο σχήμα 6.2(ν) παρουσίαζετε το speedup over one(1) thread με 4x4 Blocks και η μέγιστη δυνατότητα του μοντέλου με 16 Blocks φτάνει μέχρι το τέσσερα.



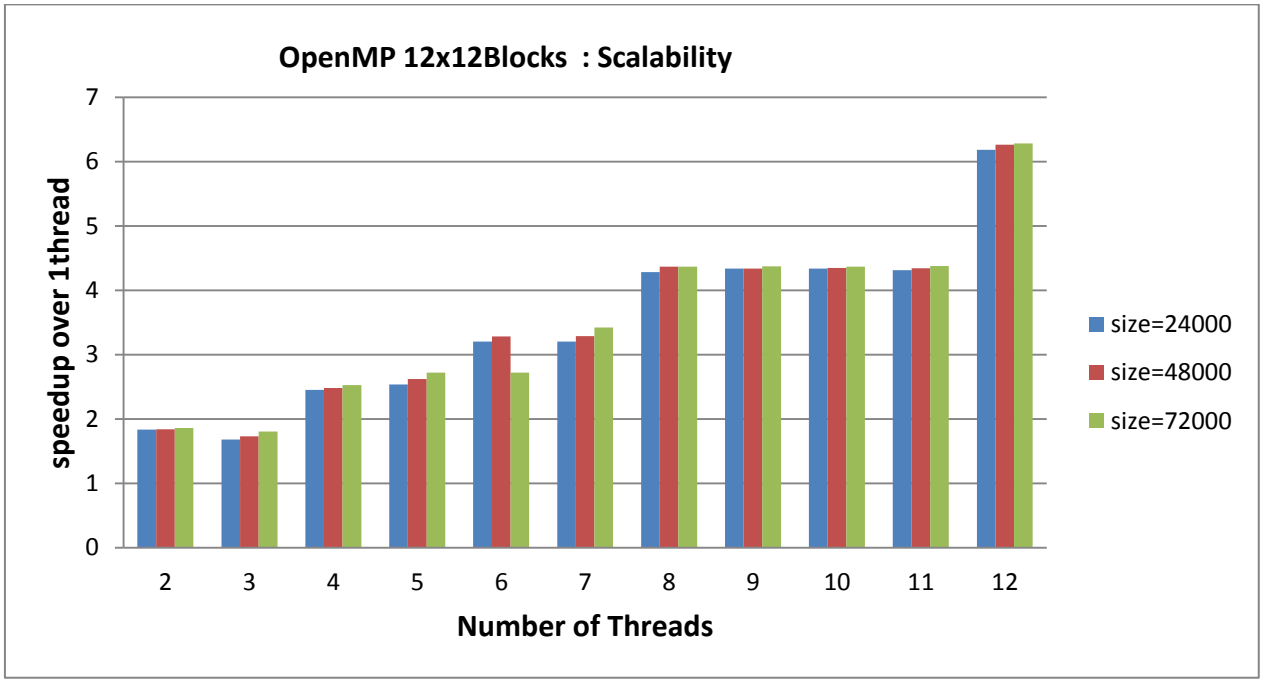
Σχήμα 6.2(ν): *Scalability OpenMP 4x4blocks υλοποίησης*

Στο σχήμα 6.2(ξ) παρουσίαζετε το speedup over one(1) thread με 8x8 Blocks και η μέγιστη δυνατότητα του μοντέλου με 64 Blocks φτάνει μέχρι το τέσσερα(4,26).



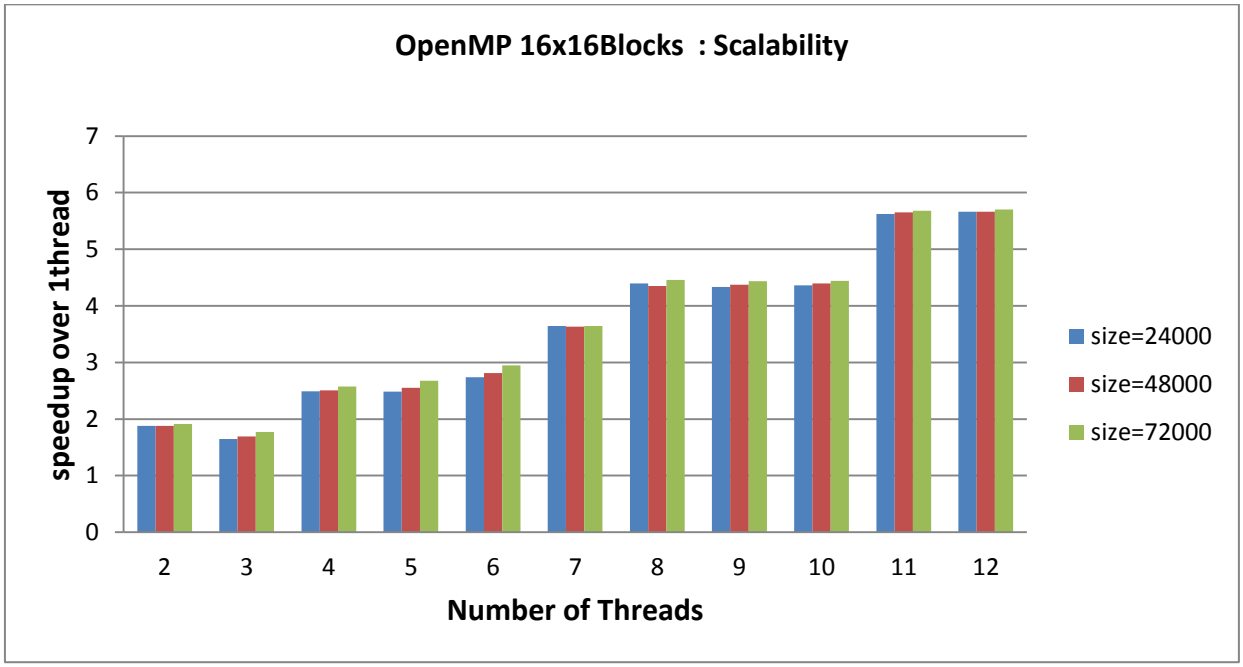
Σχήμα 6.2(ξ): *Scalability OpenMP 8x8blocks υλοποίησης*

Στο σχήμα 6.2(ο) παρουσίαζετε το speedup over one(1) thread με 12x12 Blocks και η μέγιστη δυνατότητα του μοντέλου με 144 Blocks φτάνει μέχρι το έξι(6,28).



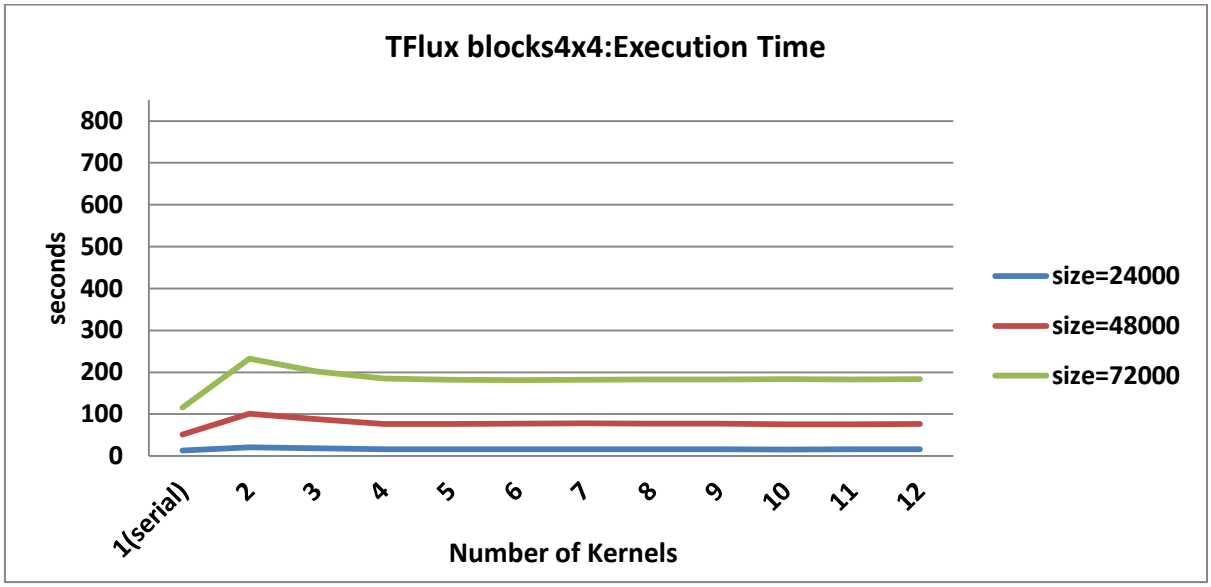
Σχήμα 6.2(ο): *Scalability OpenMP 12x12blocks υλοποίησης*

Στο σχήμα 6.2(π) παρουσίαζετε το speedup over one(1) thread με 16x16 Blocks και η μέγιστη δυνατότητα του μοντέλου με 256 Blocks φτάνει μέχρι το πέντε(5,7).



Σχήμα 6.2(π): *Scalability OpenMP 16x16 blocks υλοποίησης*

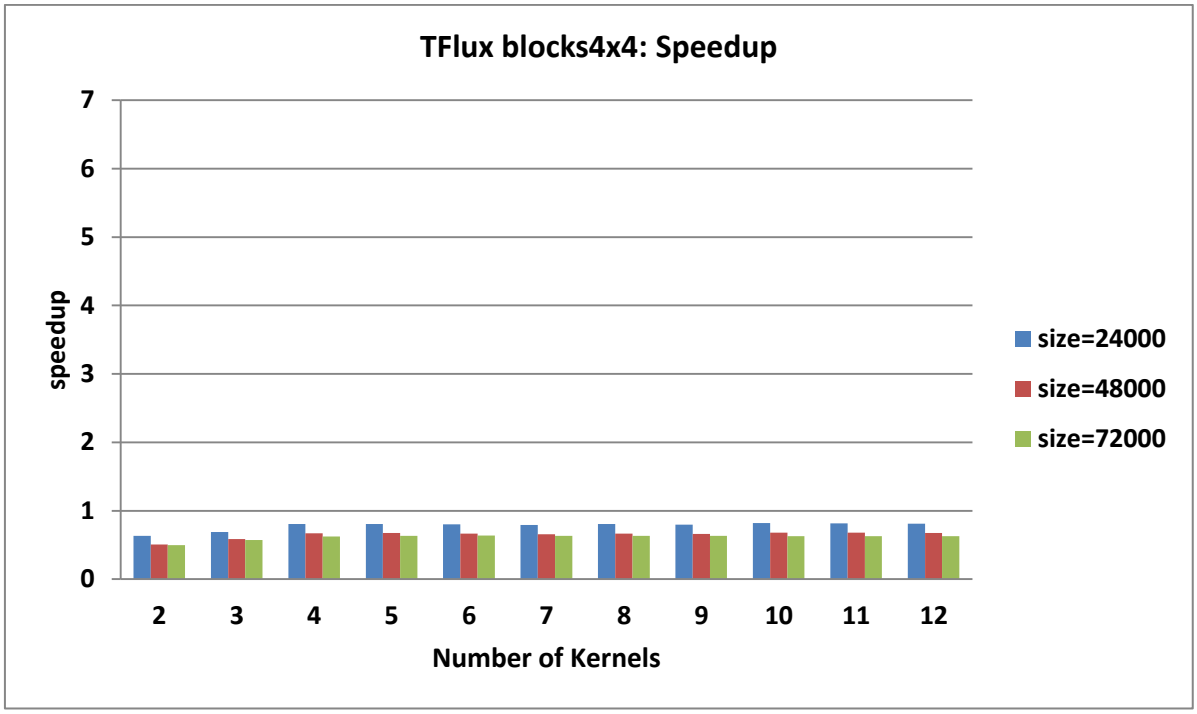
Στο σχήμα 6.3(ε) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux blocks4x4 για την σειριακή εκτέλεση σε σύγκριση με το να αυξάνετε κάθε φορά ο αριθμός των kernels.



Σχήμα 6.3(ε) *: Χρόνοι εκτέλεσης TFlux υλοποίησης με blocking 4x4*

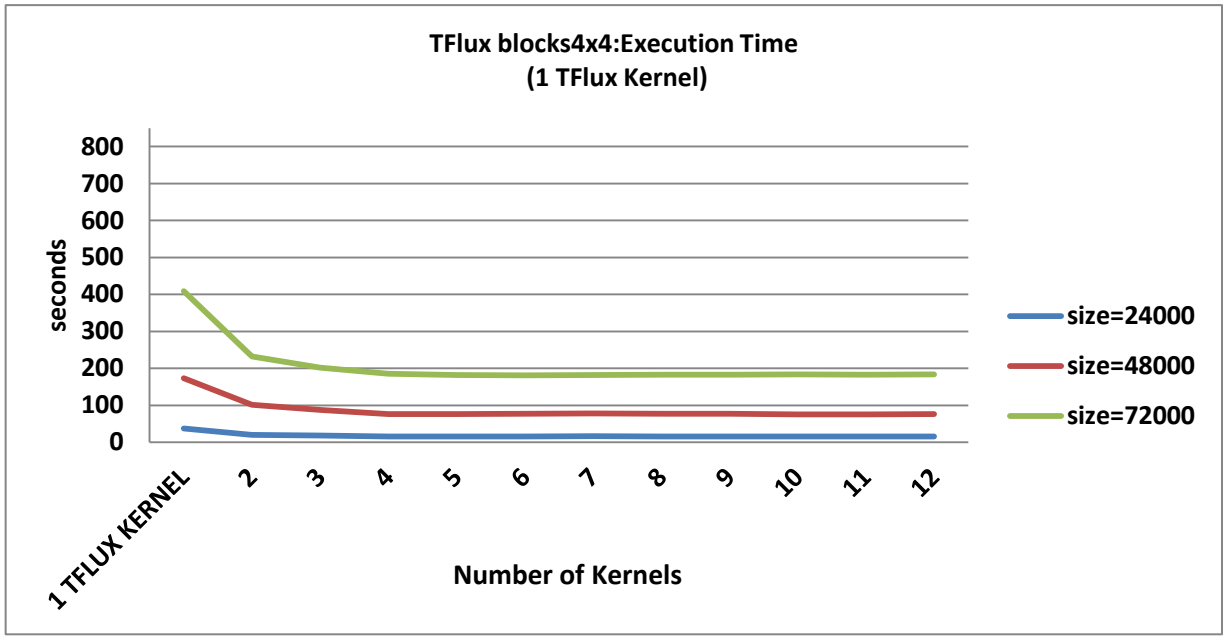
Με μπλε χρώμα αντιπροσωπεύεται ο χρόνος εκτέλεσης με μέγεθος 24000. Αντίστοιχα το κόκκινο χρώμα αντιπροσωπεύει το χρόνο εκτέλεσης με μέγεθος 48000, ενώ το πράσινο για μέγεθος 72000.Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας TFlux και συγκεκριμένα έχοντας χωρίσει το πλέγμα μας σε 4x4 Blocks (συνολικά 16 Blocks) σε σχέση με τους kernels που χρησιμοποιούνται δια μέσο των threads, αφού στο κάθε thread ορίζουμε σε ποιο kernel θα τρέξει και την επίδραση του μοντέλου σε σύγκριση με την σειριακή εκτέλεση. Παρατηρούμε ότι ο χρόνος έχει αυξηθεί σε σύγκριση με το σειριακή εκτέλεση . Αιτία του γεγονότος αυτού είναι το runtime κόστος που δημιουργείτε από την πλατφόρμα TFlux καθώς δεν μπορεί να επικαλυφθεί το κόστος αυτό . Και ο λόγος είναι ότι έχουμε λίγα blocks , στο σύνολο 16 blocks ,καθώς ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι τέσσερα , η κυρίως διαγώνιος που αποτελείτε από τέσσερα blocks.

Το σχήμα 6.3(ζ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των kernels μπορούμε να πούμε πως η επίδοση δεν αυξάνετε ,αλλά αντιθέτως μειώνετε . Ο λόγος όπως προαναφέρθηκε είναι ότι δεν γίνετε καλύτερη εκμετάλλευση του παραλληλισμού λόγω του μικρού αριθμού των blocks και αυτό έχει σαν συνεπακόλουθο να μην μπορεί να καλύψει το runtime κόστος που δημιουργείτε από την πλατφόρμα TFlux. Με τα δεδομένα αυτά μπορούμε να πούμε πως η υλοποίηση του προβλήματος μας με TFlux έχοντας 4x4blocks δεν είναι ικανοποιητική για οποιοδήποτε αριθμό kernels χρησιμοποιήσουμε.



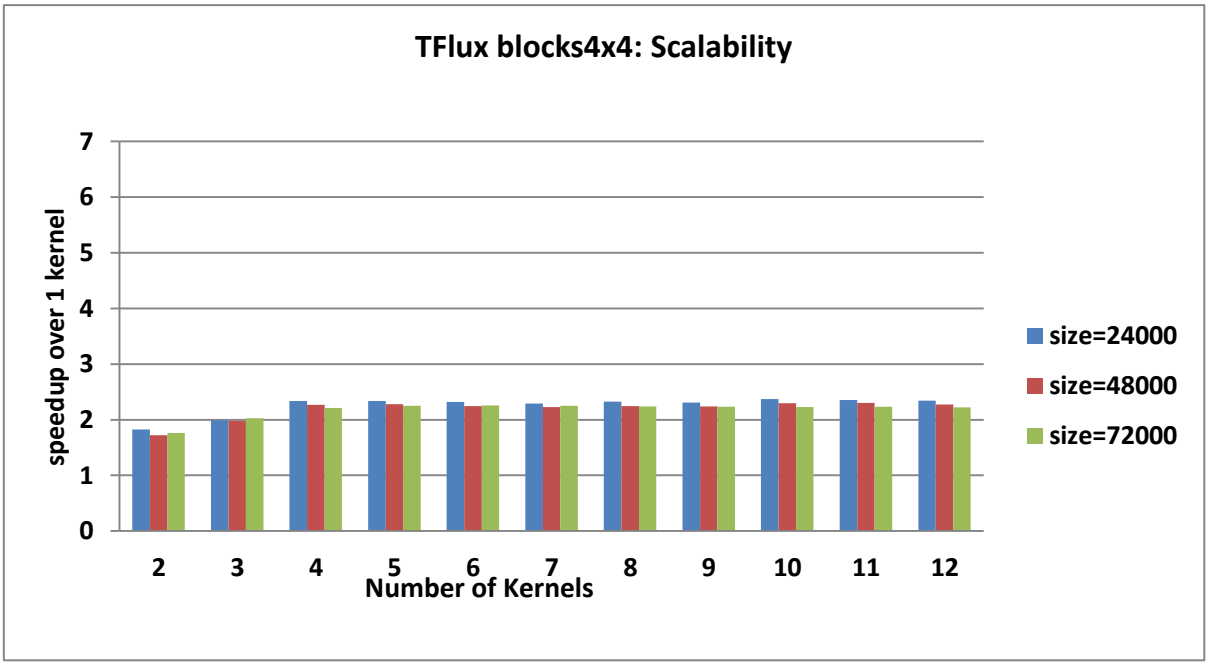
Σχήμα 6.3 (ζ): *Επίδοση TFlux υλοποίησης με blocking 4x4*

Στο σχήμα 6.3(η) μας παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 4x4blocks με το να αυξάνετε κάθε φορά ο αριθμός των kernels σε σύγκριση με την εκτέλεση της ίδιας υλοποιήσεις με ένα kernel . Βλέπουμε το κόστος να εκτελεστεί η εφαρμογή αυτή με ένα kernel σε TFlux . Παρατηρούμε πώς ο χρόνος μειώνετε καθώς προσθέτουμε kernels μέχρι τους τέσσερις kernels . Μετά τους τέσσερις kernels δεν παρατηρείτε καμία μείωση στο χρόνο εκτέλεσης και αυτό γιατι ο μέγιστος αριθμός blocks που θα μπορούσαν να τρέξουν παράλληλα τα είναι τέσσερα , αρά και θα χρησιμοποιούνταν για αυτό το σκοπό τέσσερις kernels .



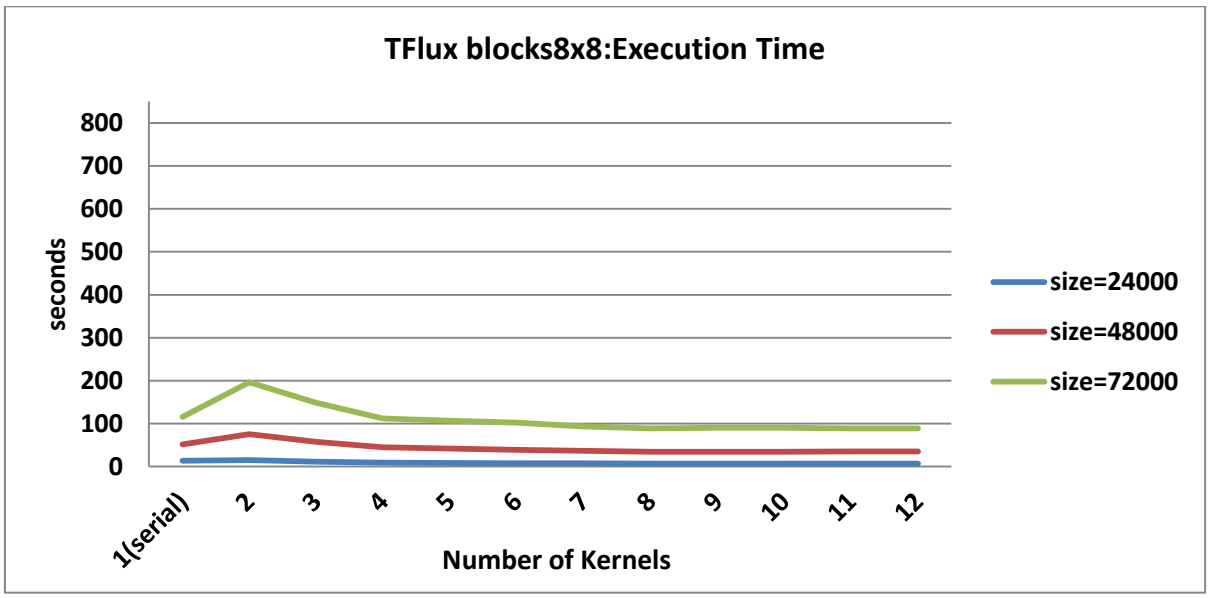
Σχήμα 6.3(η): *Χρόνοι εκτέλεσης TFlux 4x4blocks υλοποίησης σε σύγκριση με την εκτέλεση σε ένα(1) kernel.*

Στο σχήμα 6.3(ι) παρατηρούμε το scalability του προηγούμενος σχήματος που συγκρίνει την υλοποίηση χρησιμοποιώντας δύο μέχρι δώδεκα kernels με το να χρησιμοποιείτε μόνο ένα kernel . Παρατηρούμε σε αυτή την περίπτωση ότι το scalability αυξάνετε καθώς αυξάνονται οι kernels αλλά αυτό μέχρι τους τέσσερις kernels για τον λόγο που εξηγήσαμε πιο πάνω και έχει μέγιστο scalability μέχρι και δύο (και λίγο περισσότερο) φορές.



Σχήμα 6.3(ι): *Scalability TFlux 4x4blocks υλοποίησης*

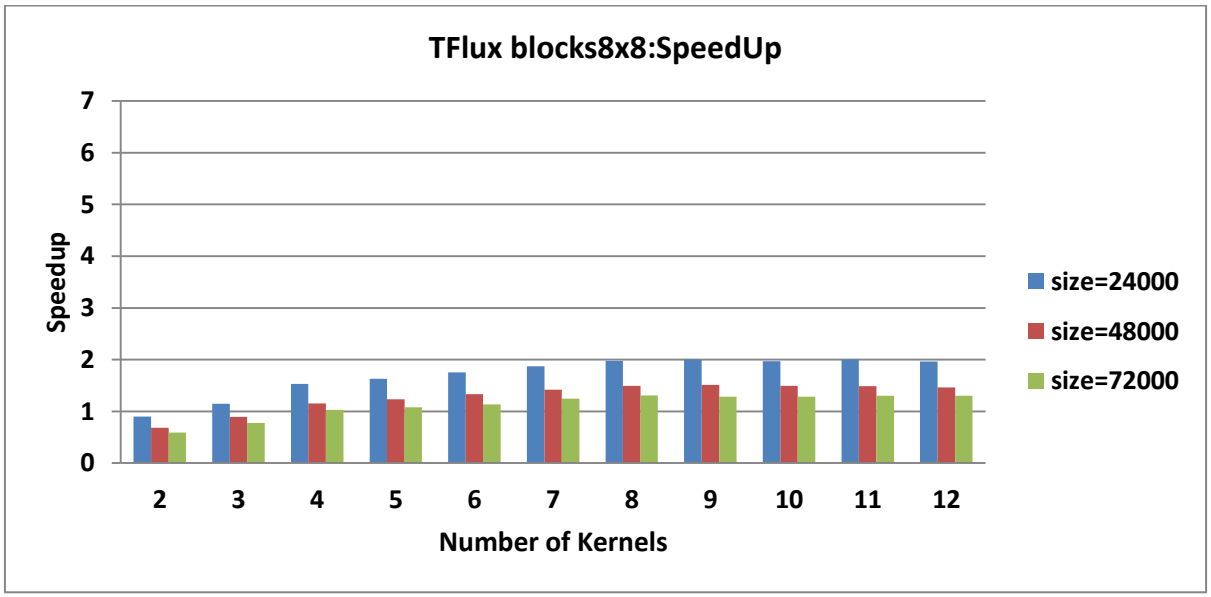
Στο σχήμα 6.3(κ) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 8x8 blocks για την σειριακή εκτέλεση σε σύγκριση με το να αλλάζετε κάθε φορά ο αριθμός των kernels.



Σχήμα 6.3(κ) : *Χρόνοι εκτέλεσης TFlux υλοποίησης με blocking 8x8*

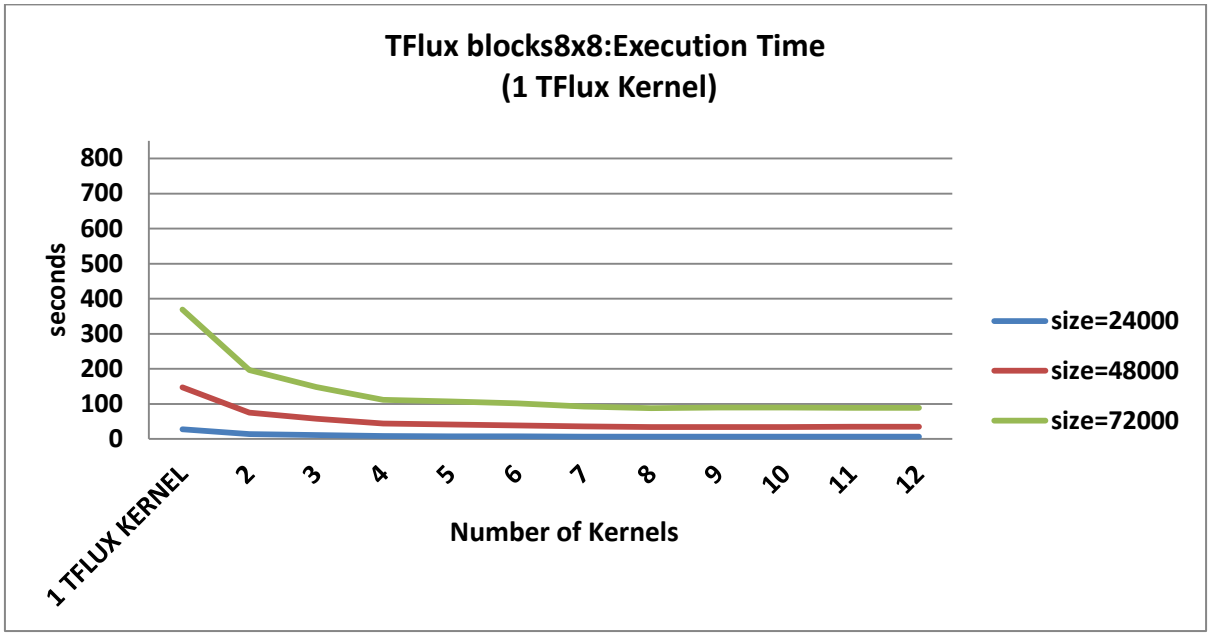
Και εδώ, τα χρώματα συμβολίζουν ότι συμβολίζανε και προηγουμένως. Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας TFlux και συγκεκριμένα έχοντας χωρίσει το πλέγμα των σημείων μας σε 8x8 blocks (συνολικά 64 blocks) σε σχέση με τους kernels που χρησιμοποιούνται δια μέσω των threads, αφού στο κάθε thread ορίζουμε σε ποιο kernel θα τρέξει και την επίδραση του μοντέλου σε σύγκριση με την σειριακή εκτέλεση. Παρατηρούμε ότι ο χρόνος έχει μειωθεί λίγο στα τρία μεγέθη. Είναι φανερό ότι σχετικά ότι ο χρόνος εκτέλεσης όσο αυξάνουμε τους kernels μειώνετε αλλά αυτό μέχρι τους οκτώ kernels. Μετά τους οκτώ kernels μέχρι τους δώδεκα μένει περίπου ίδιος ο χρόνος. Αιτία του γεγονότος αυτού είναι ότι αφού υπάρχουν διαθέσιμα 64 blocks, το κάθε ένα είναι υπεύθυνο για το1/64 του συνολικού πλέγματος, αν το δούμε γραφικά , ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι οκτώ , η κυρίως διαγώνιος ,που αποτελείτε από οκτώ blocks.

Το σχήμα 6.3(λ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των kernels μπορούμε να πούμε πως η επίδοση βελτιώνετε αλλά πέρα των οκτώ kernels παραμένει σταθερή η επίδοση χωρίς καμία άλλη αύξηση. Επίσης παρατηρούμε πώς η απόδοση φτάνει μέχρι και 2 φορές καλύτερη για το μεγάλο(large) μέγεθός ενώ για τα άλλα δυο μεγέθη small και medium έχουν μέχρι 1,5 καλύτερη επίδοση .Παρατηρούμε σε σύγκριση με την υλοποίηση TFlux 4x4 blocks, στη υλοποίηση TFlux 8x8 blocks παρατηρείτε μια αύξηση στην απόδοση διότι υπάρχουν περισσότερα blocks , και έτσι το να εκτελούνται παράλληλα περισσότερα blocks επιτυγχάνετε να γίνετε μεγαλύτερη επικάλυψη runtime κόστος που δημιουργείτε από την πλατφόρμα TFlux. .Με τα δεδομένα μας αυτά μπορούμε να πούμε πως η υλοποίηση του προβλήματος μας με TFlux 8x8 blocks είναι αρκετά πιο ικανοποιητική σε σχέση με την TFlux 4x4 blocks μιας και η απόδοση εδώ αυξάνετε μέχρι τους οκτώ kernels σε σύγκριση με την προηγουμένη που η απόδοση αυξανόταν μέχρι τους τέσσερις kernels. Αρά εκμεταλλεύεται καλύτερα τους kernels η τελευταία υλοποίηση που μόλις παρατηρήσαμε.



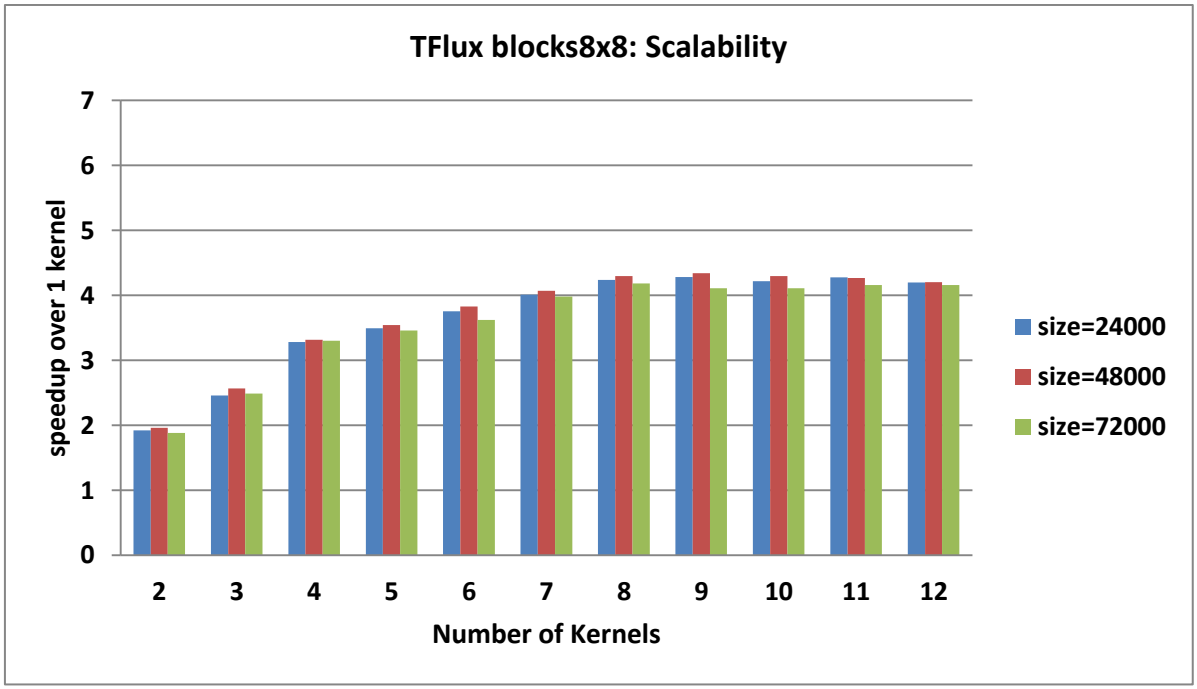
Σχήμα 6.3(λ) *: Επίδοση TFlux υλοποίησης με blocking 8x8*

Στο σχήμα 6.3(μ) μας παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 8x8 blocksμε το να αυξάνετε κάθε φορά ο αριθμός των kernels σε σύγκριση με την εκτέλεση της ίδιας υλοποιήσεις με ένα kernel . Βλέπουμε το κόστος να εκτελεστεί η εφαρμογή αυτή με ένα kernel σε TFlux . Παρατηρούμε πώς ο χρόνος μειώνετε καθώς προσθέτουμε kernels μέχρι τους οκτώ kernels . Μετά τους οκτώ kernels δεν παρατηρείτε καμία μείωση στο χρόνο εκτέλεσης και αυτό γιατί ο μέγιστος αριθμός blocks που θα μπορούσαν να τρέξουν παράλληλα τα είναι οκτώ , αρά και θα χρησιμοποιούνταν για αυτό το σκοπό οκτώ kernels .



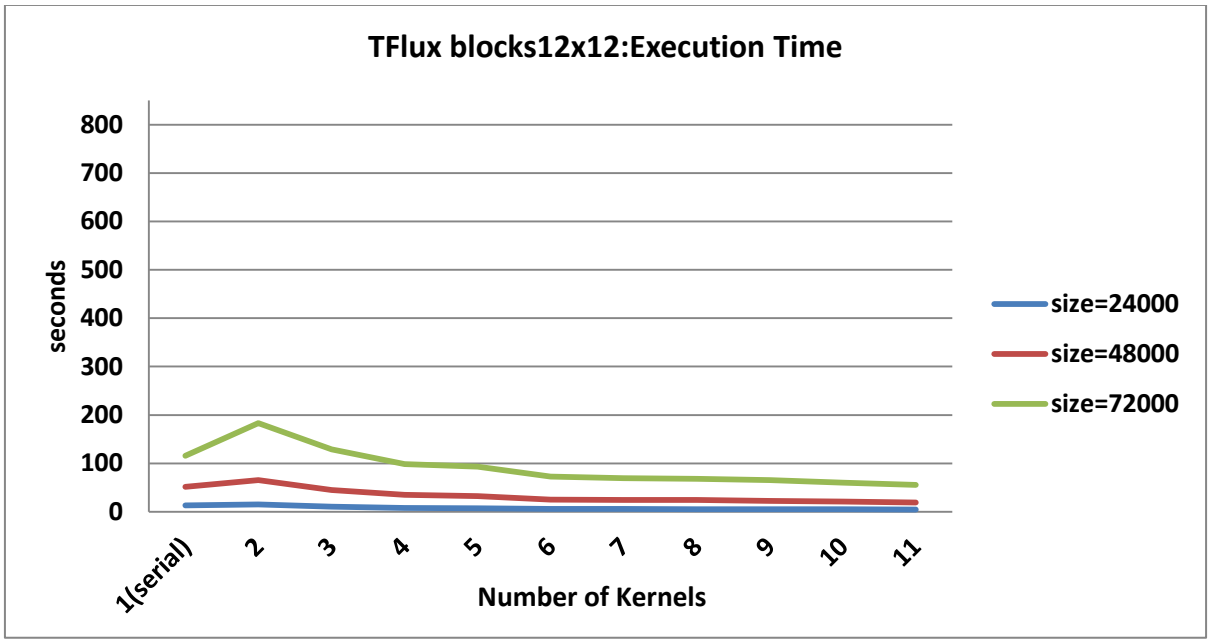
Σχήμα 6.3(μ) *: Χρόνοι εκτέλεσης TFlux 8x8blocks υλοποίησης σε σύγκριση με την εκτέλεση σε ένα(1) kernel.*

Στο σχήμα 6.3(ν) παρατηρούμε το scalability του προηγούμενος σχήματος που συγκρίνει την υλοποίηση χρησιμοποιώντας δύο μέχρι δώδεκα kernels με το να χρησιμοποιείτε μόνο ένα kernel. Παρατηρούμε σε αυτή την περίπτωση ότι το scalability αυξάνετε καθώς αυξάνονται οι kernels αλλά αυτό μέχρι τους οκτώ kernels για τον λόγο που εξηγήσαμε πιο πάνω και έχει μέγιστο scalability μέχρι και τέσσερις (και λίγο περισσότερο) φορές



Σχήμα 6.3(ν): *Scalability TFlux 8x8blocks υλοποίησης*

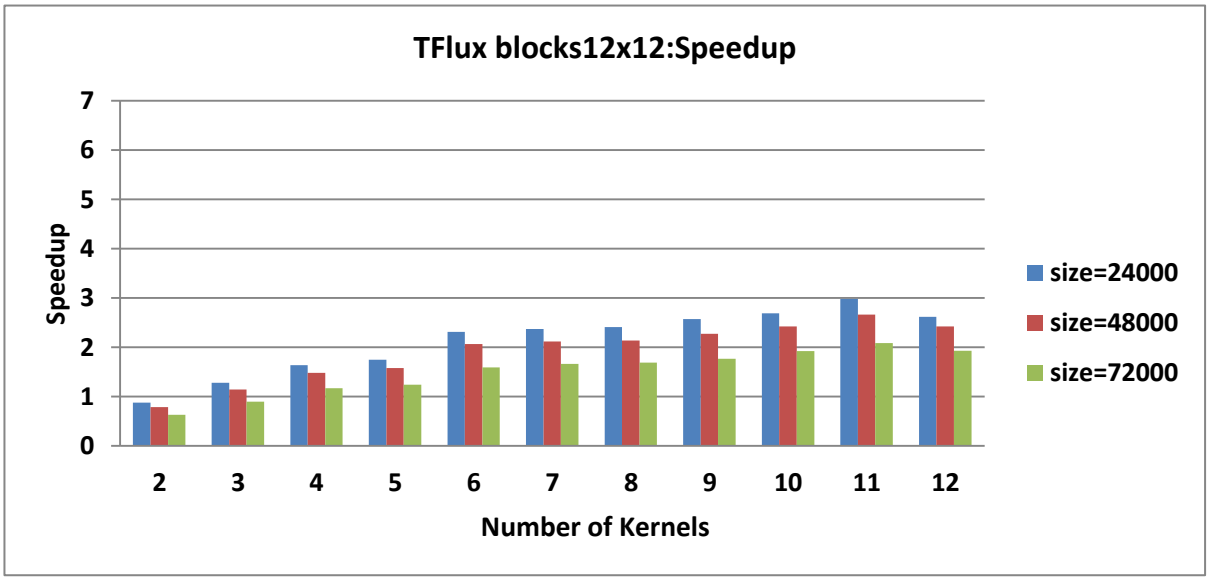
Στο σχήμα 6.3(ξ) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 12x12 blocks(συνολικά 144 Blocks) για την σειριακή εκτέλεση σε σύγκριση με το να αλλάζετε κάθε φορά ο αριθμός των kernels.



Σχήμα 6.3(ξ) *: Χρόνοι εκτέλεσης TFlux υλοποίησης με blocking 12x12*

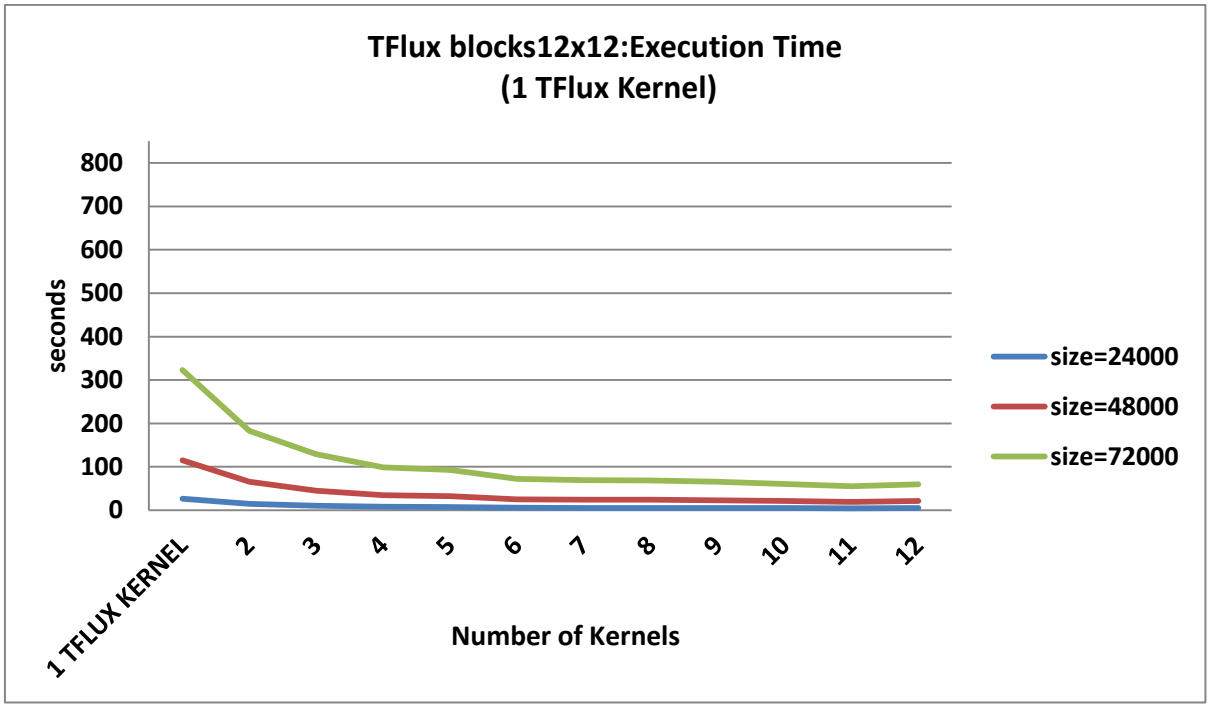
Ακριβώς και όπως και πριν και εδώ, τα χρώματα συμβολίζουν ότι συμβόλιζαν και προηγουμένως . Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας TFlux και συγκεκριμένα έχοντας χωρίσει το πλέγμα των σημείων μας τώρα σε 12x12blocks (συνολικά 144 Blocks) σε σχέση με τους kernels που χρησιμοποιούνται δια μέσω των threads, αφού στο κάθε thread ορίζουμε σε ποιο kernel θα τρέξει και την επίδραση του μοντέλου σε σύγκριση με την σειριακή εκτέλεση. Παρατηρούμε ότι ο χρόνος έχει μειωθεί σε μεγαλύτερο βαθμό σε σύγκριση με τις προηγούμενες δύο υλοποιήσεις και έχει κατεβεί και στα τρία μεγέθη ο χρόνος, και αυτό παρατηρείτε πιο έντονα στο small μέγεθός . Είναι φανερό ότι σχετικά ότι ο χρόνος εκτέλεσης όσο αυξάνουμε τους kernels μειώνετε και τώρα αυτό ισχύει μέχρι τους kernels. Αιτία του γεγονότος αυτού είναι ότι αφού υπάρχουν διαθέσιμα 144 blocks, το κάθε ένα είναι υπεύθυνο για το1/144 του συνολικού πλέγματος, αν το δούμε γραφικά , ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι δώδεκα , η κυρίως διαγώνιος ,που αποτελείτε από δώδεκα blocks.

Το σχήμα 6.3(ο) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των kernels μπορούμε να πούμε πως η επίδοση βελτιώνετε και στην συγκεκριμένη εκτέλεση βελτιώνετε πέρα των τεσσάρων και οκτώ kernels όπως παρατηρήσαμε στις προηγούμενες εκτελέσεις. Τώρα εδώ συνεχίζει να υπάρχει βελτίωση μέχρι τους έντεκα kernels συγκεκριμένα. Ο λόγος που δεν συνεχίζετε η βελτίωση μέχρι τους δώδεκα kernels είναι ότι , ο ένας kernel από αυτούς που δηλώνουμε κάθε φορά στην υλοποίηση μας δεσμεύετε από τη μονάδα συγχρόνισμου TSU.Με τα δεδομένα μας αυτά μπορούμε να πούμε πως η υλοποίηση του προβλήματος μας με TFlux 12x12blocks είναι η καλύτερη και αρκετά πιο ικανοποιητική σε σχέση με την TFlux 4x4 blocks και TFlux 8x8 blocks μιας και η απόδοση εδώ αυξάνετε μέχρι τους έντεκα kernels. Αρά εκμεταλλεύεται καλύτερα τους όλους kernels η τελευταία υλοποίηση που μόλις παρατηρήσαμε σε σύγκριση με τις προηγούμενες άλλες δύο.



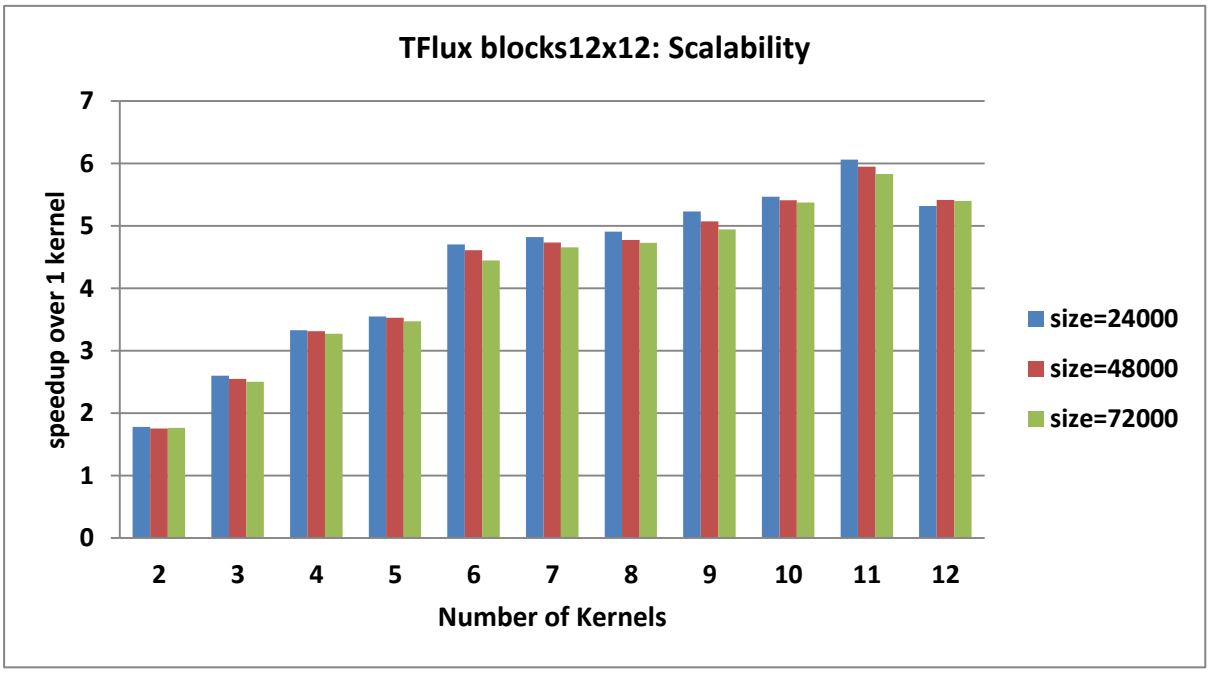
Σχήμα 6.3(ο): *Επίδοση TFlux υλοποίησης με blocking 12x12.*

Στο σχήμα 6.3(π) μας παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 12x12 blocksμε το να αυξάνετε κάθε φορά ο αριθμός των kernels σε σύγκριση με την εκτέλεση της ίδιας υλοποιήσεις με ένα kernel . Βλέπουμε το κόστος να εκτελεστεί η εφαρμογή αυτή με ένα kernel σε TFlux . Παρατηρούμε πώς ο χρόνος μειώνετε καθώς προσθέτουμε kernels μέχρι τους έντεκα kernels . Μετά τους έντεκα kernels δεν παρατηρείτε καμία μείωση στο χρόνο εκτέλεσης και αυτό γιατί ο μέγιστος αριθμός blocks που θα μπορούσαν να τρέξουν παράλληλα τα είναι δώδεκα , αρά και θα χρησιμοποιούνταν για αυτό το σκοπό δώδεκα kernels .



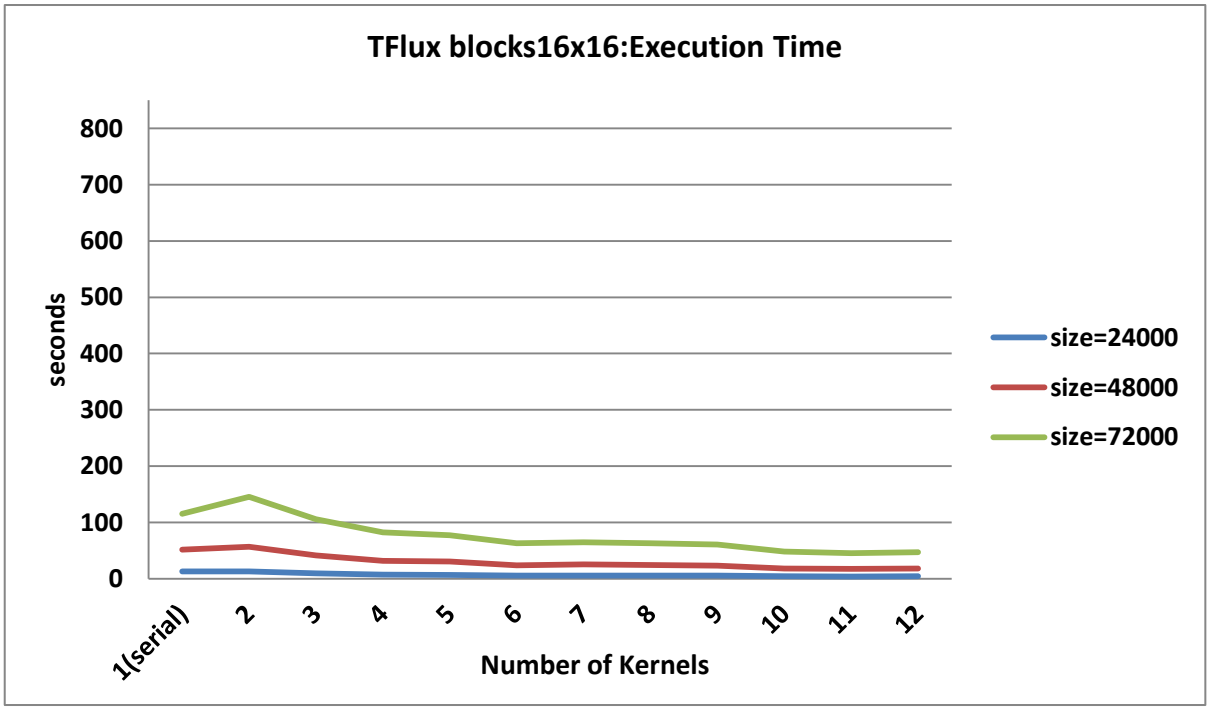
Σχήμα 6.3(π): *Χρόνοι εκτέλεσης TFlux 12x12blocks υλοποίησης σε σύγκριση με την εκτέλεση σε ένα(1) kernel.*

Στο σχήμα 6.3(ρ) παρατηρούμε το scalability του προηγούμενος σχήματος που συγκρίνει την υλοποίηση χρησιμοποιώντας δύο μέχρι δώδεκα kernels με το να χρησιμοποιείτε μόνο ένα kernel . Παρατηρούμε σε αυτή την περίπτωση ότι το scalability αυξάνετε καθώς αυξάνονται οι kernels αλλά αυτό μέχρι τους οκτώ kernels για τον λόγο που εξηγήσαμε πιο πάνω και έχει μέγιστο scalability μέχρι και τέσσερις (και λίγο περισσότερο) φορές



Σχήμα 6.3(ρ): *Scalability TFlux 12x12blocks υλοποίησης*

Στο σχήμα 6.3(σ) παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 16x16 blocks(συνολικά 256 Blocks) για την σειριακή εκτέλεση σε σύγκριση με το να αλλάζετε κάθε φορά ο αριθμός των kernels.

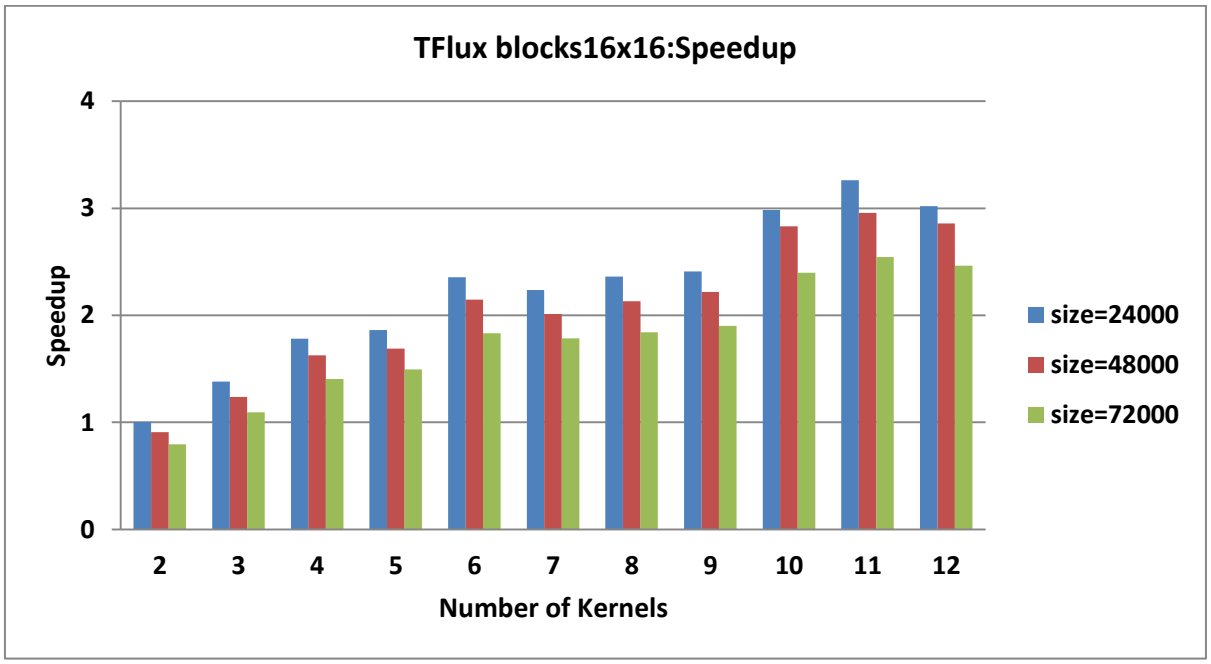


Σχήμα 6.3(σ) : *Χρόνοι εκτέλεσης TFlux υλοποίησης με blocking 16x16*

Ακριβώς και όπως και πριν και εδώ, τα χρώματα συμβολίζουν ότι συμβόλιζαν και προηγουμένως . Σκοπός του γραφήματος αυτού είναι να παρουσιάσουμε την δυνατότητα της πλατφόρμας TFlux και συγκεκριμένα έχοντας χωρίσει το πλέγμα των σημείων μας τώρα σε 16x16blocks (συνολικά 256 Blocks) σε σχέση με τους kernels που χρησιμοποιούνται δια μέσω των threads, αφού στο κάθε thread ορίζουμε σε ποιο kernel θα τρέξει και την επίδραση του μοντέλου σε σύγκριση με την σειριακή εκτέλεση. Παρατηρούμε ότι ο χρόνος έχει μειωθεί σε ακόμη μεγαλύτερο βαθμό σε σύγκριση με τις προηγούμενές τρεις υλοποιήσεις και έχει κατεβεί και στα τρία μεγέθη ο χρόνος. Είναι φανερό ότι σχετικά ότι ο χρόνος εκτέλεσης όσο αυξάνουμε τους kernels μειώνετε και τώρα αυτό ισχύει για όλους τους kernels. Αιτία του γεγονότος αυτού είναι ότι αφού υπάρχουν διαθέσιμα 256 blocks, το κάθε ένα είναι υπεύθυνο για το1/256 του συνολικού πλέγματος, αν το δούμε γραφικά , ο μέγιστος αριθμός blocks που μπορούν να τρέχουν παράλληλα είναι δώδεκα , η κυρίως διαγώνιος ,που αποτελείτε από δώδεκα blocks.

Το σχήμα 6.3(τ) μας παρουσιάζει η επίδοση η οποία έχει υπολογιστεί με βάση τον χρόνο της σειριακής εκτέλεσης της υλοποίησης αυτής. Όσο αυξάνουμε τον αριθμό των kernels μπορούμε να πούμε πως η επίδοση βελτιώνετε και στην συγκεκριμένη εκτέλεση βελτιώνετε πέρα των τεσσάρων οκτώ και δώδεκα kernels όπως παρατηρήσαμε στις προηγούμενες εκτελέσεις. Τώρα εδώ συνεχίζει να υπάρχει βελτίωση μέχρι τους έντεκα kernels συγκεκριμένα για τον ίδιο λόγο

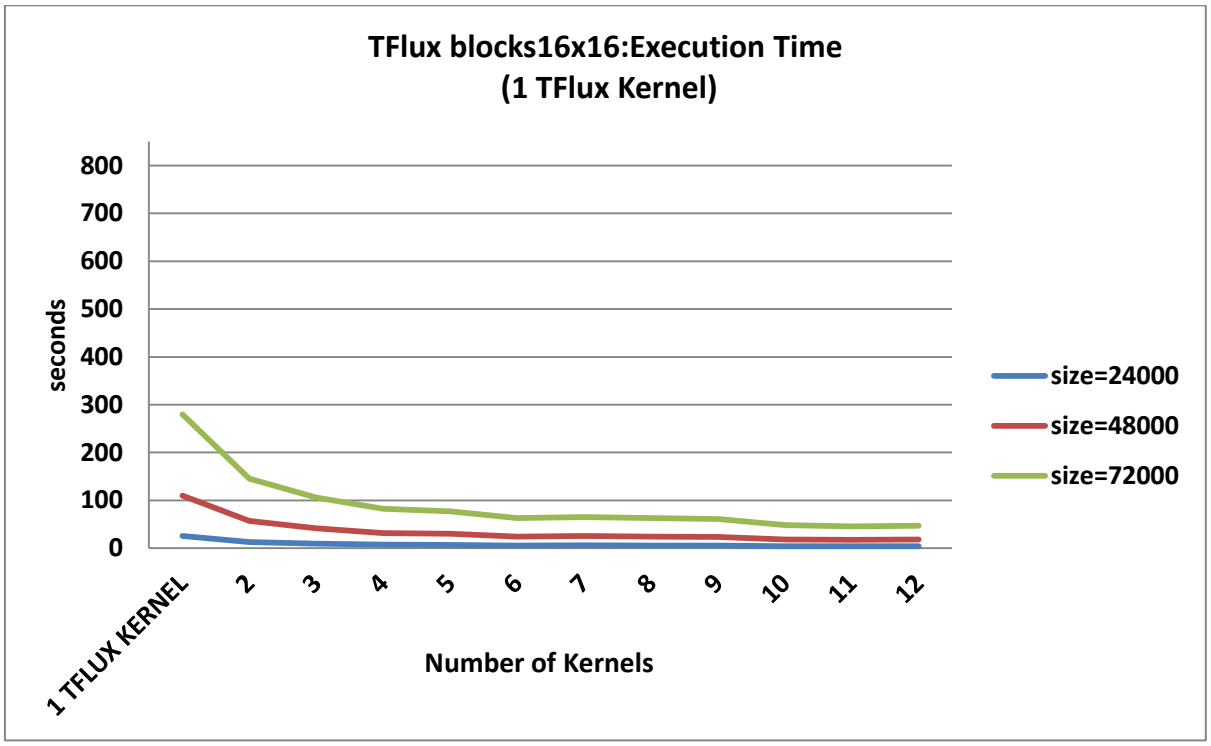
όπως αναφέραμε στο σχήμα 6.3(ζ). Όπως είναι λογικό η επίδοση εδώ θα εμφανιστεί πριν από την κύρια διαγώνιο σε σύγκριση με την προηγούμενη υλοποίηση με 12x12 Blocks διότι ήδη θα μπορούν να εκτελούνται δώδεκα παράλληλα Blocks πριν από την κύρια διαγώνιο. Παρατηρούμε επίσης πως η επίδοση εδώ είναι πιο αυξημένη αλλά παράλληλα η αύξηση δεν γίνετε κάθε φόρα με την προσθήκη ενός επιπροσθέτου kernel. Ο λόγος είναι ότι τα 256 DThreads που δημιουργηθήκαν για να διαχειρίζονται το κάθε ένα ξεχωριστά από ένα Block από τα συνολικά 256 Blocks δεν χωρούσανε όλα σε ένα DDM Block. Για αυτό χρηστήκανε 3 DDM Blocks. Αυτός είναι ο λόγος που η επίδοση γινόταν σταδιακά λόγω του κόστους δημιουργίας DDM Blocks μιας και για να εκτελεστεί ένα DDM Block θα έπρεπε να τελείωνε την εκτέλεση του το ακριβώς αμέσως DDM Block. Με τα δεδομένα μας αυτά μπορούμε να πούμε πως η υλοποίηση του προβλήματος μας με TFlux έχοντας blocks μεγέθους 16x16 είναι η καλύτερη και αρκετά πιο ικανοποιητική σε σχέση με την TFlux 4x4 blocks , TFlux 8x8 blocks και TFlux 12x12 blocks μιας και η απόδοση εδώ αυξάνετε μέχρι τους έντεκα kernels και η επίδοση γίνετε λίγο μεγαλύτερη του τρία, δηλαδή περίπου 3,5 φορές γρηγορότερη. Αρά εκμεταλλεύεται καλύτερα τους όλους kernels η τελευταία υλοποίηση που μόλις παρατηρήσαμε σε σύγκριση με τις προηγούμενες άλλες δύο.



Σχήμα 6.3(τ): *Επίδοση TFlux υλοποίησης με blocking 16x16*

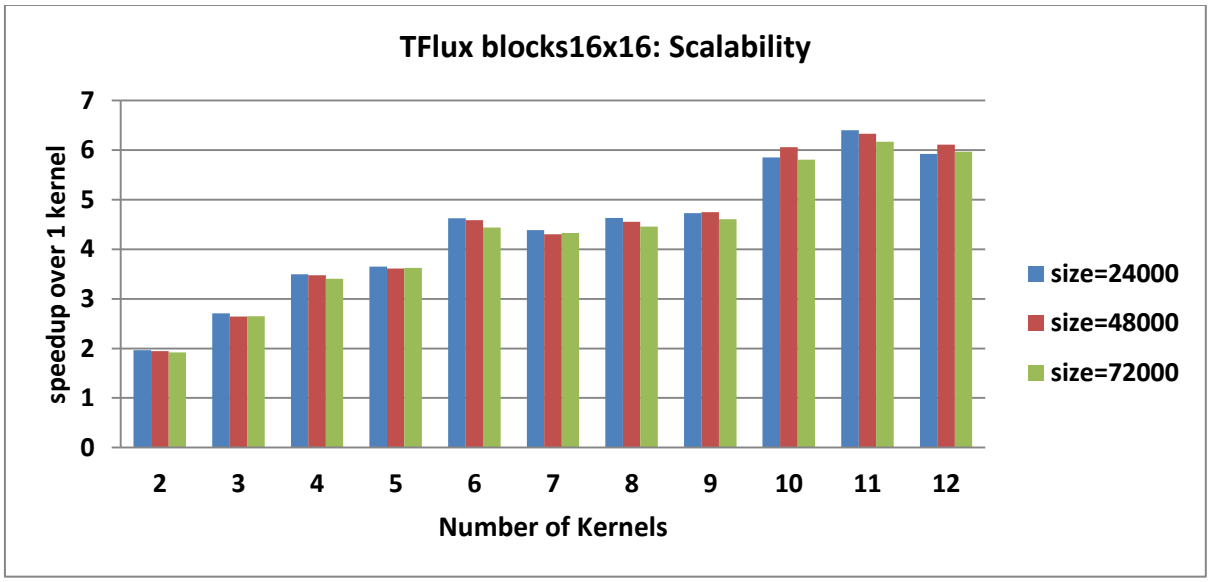
Στο σχήμα 6.3(υ) μας παρουσιάζονται οι χρόνοι εκτέλεσης της υλοποίησης με TFlux 16x16 blocksμε το να αυξάνετε κάθε φορά ο αριθμός των kernels σε σύγκριση με την εκτέλεση της ίδιας υλοποιήσεις με ένα kernel . Βλέπουμε το κόστος να εκτελεστεί η εφαρμογή αυτή με ένα kernel σε TFlux . Παρατηρούμε πώς ο χρόνος μειώνετε καθώς προσθέτουμε kernels μέχρι τους έντεκα kernels . Μετά τους έντεκα kernels δεν παρατηρείτε καμία μείωση στο χρόνο εκτέλεσης

και αυτό γιατί ο μέγιστος αριθμός blocks που θα μπορούσαν να τρέξουν παράλληλα τα είναι δώδεκα , αρά και θα χρησιμοποιούνταν για αυτό το σκοπό δώδεκα kernels .



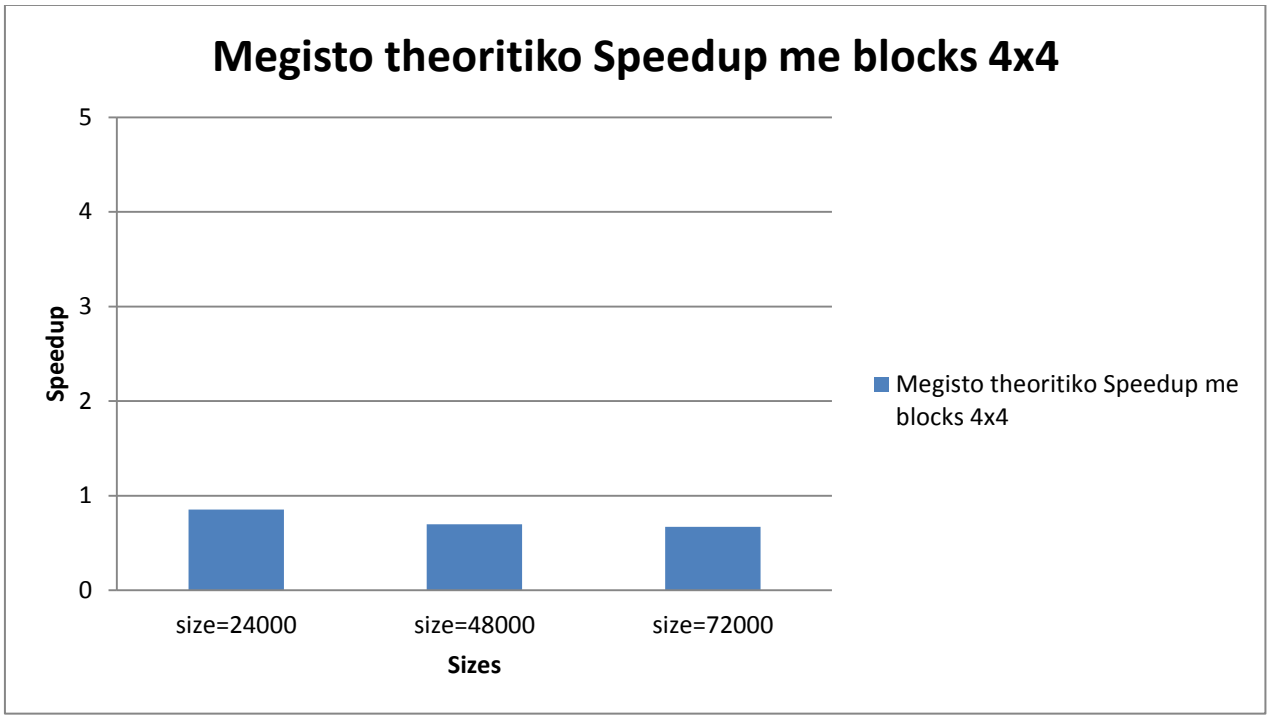
Σχήμα 6.3(υ): *Χρόνοι εκτέλεσης TFlux 16x16blocks υλοποίησης σε σύγκριση με την εκτέλεση σε ένα(1) kernel.*

Στο σχήμα 6.3(φ) παρατηρούμε το scalability του προηγούμενος σχήματος που συγκρίνει την υλοποίηση χρησιμοποιώντας δύο μέχρι δώδεκα kernels με το να χρησιμοποιείτε μόνο ένα kernel. Παρατηρούμε σε αυτή την περίπτωση ότι το scalability αυξάνετε καθώς αυξάνονται οι kernels αλλά αυτό μέχρι τους έντεκα kernels γιατί όπως προαναφέραμε ένας kernel από τους δώδεκα που διαθέτει η μηχανή στην οποία προσομοιώθηκε είναι δεσμευμένος για την μονάδα συγχρονισμού .

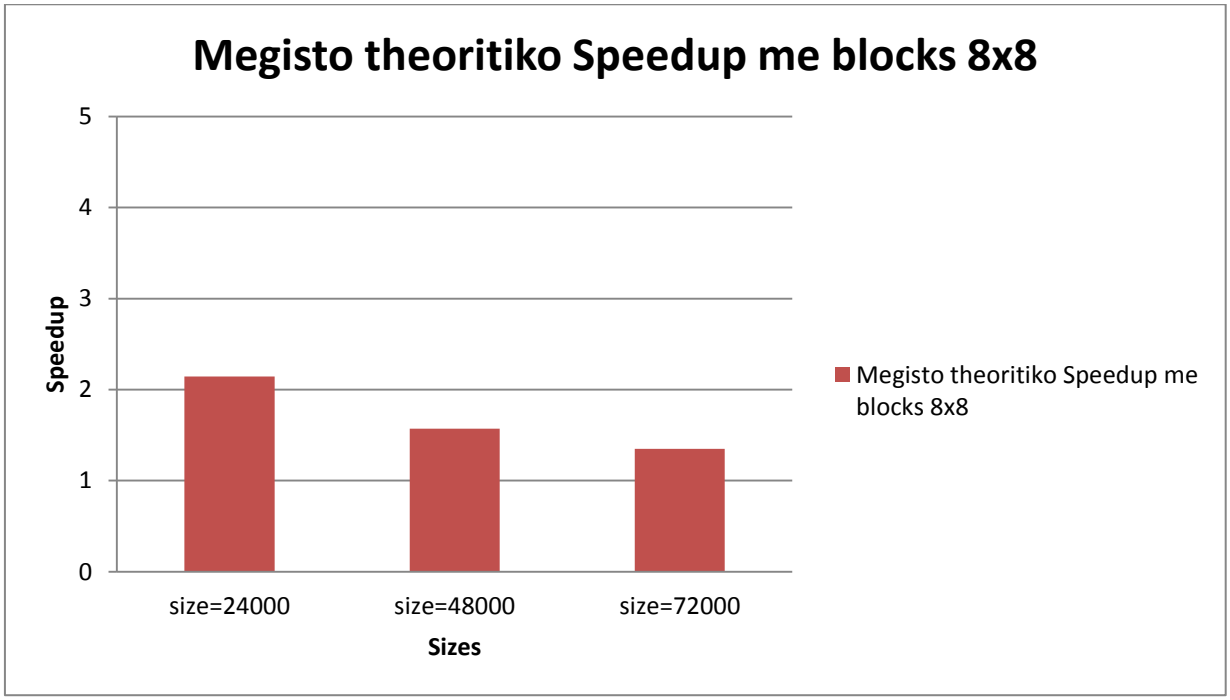


Σχήμα 6.3(φ): *Scalability TFlux 16x16blocks υλοποίησης*

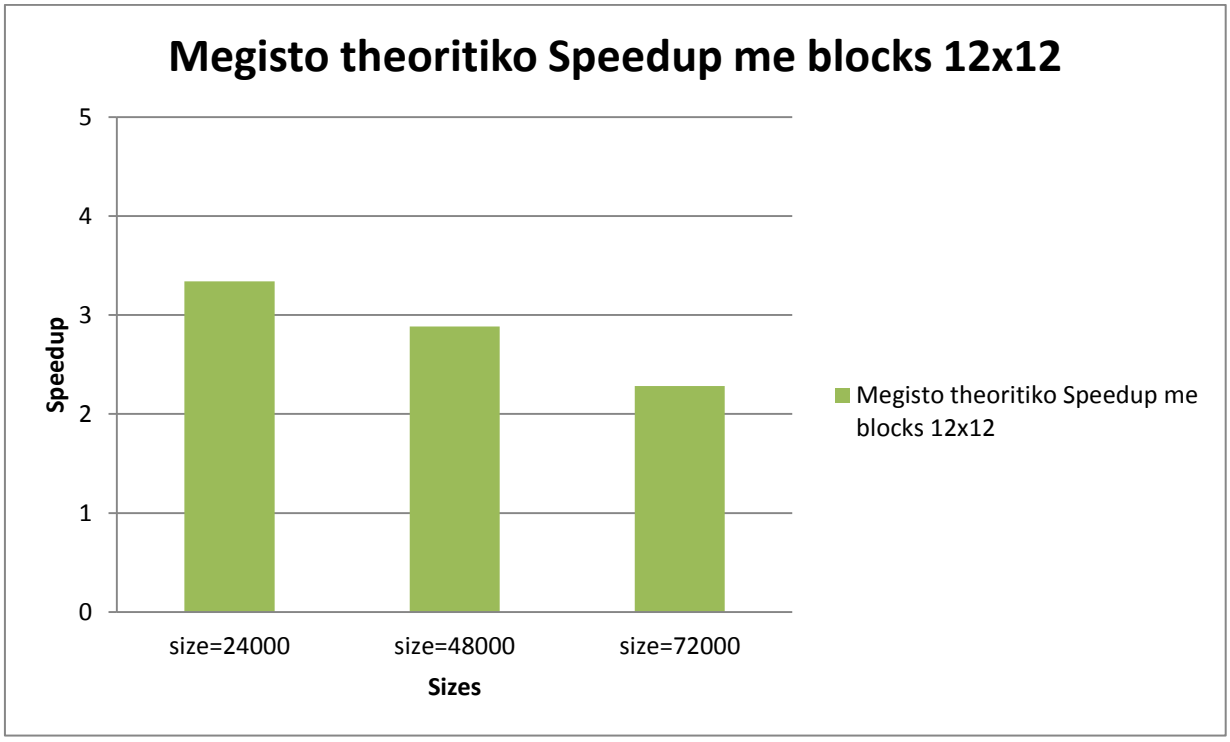
Οι γραφικές παραστάσεις που ακολουθούνε παρουσιαζούν το μεγιστο θεωρητικό Speedup που υπάρχει αναλόγα με το σε πόσα blocks διασπάτε η υπολογισμός ολόκληρού του πλεγματός και αυτό σε κάθε μέγεθός του προβήματος .(σχήμα 6.4(α)-(δ))



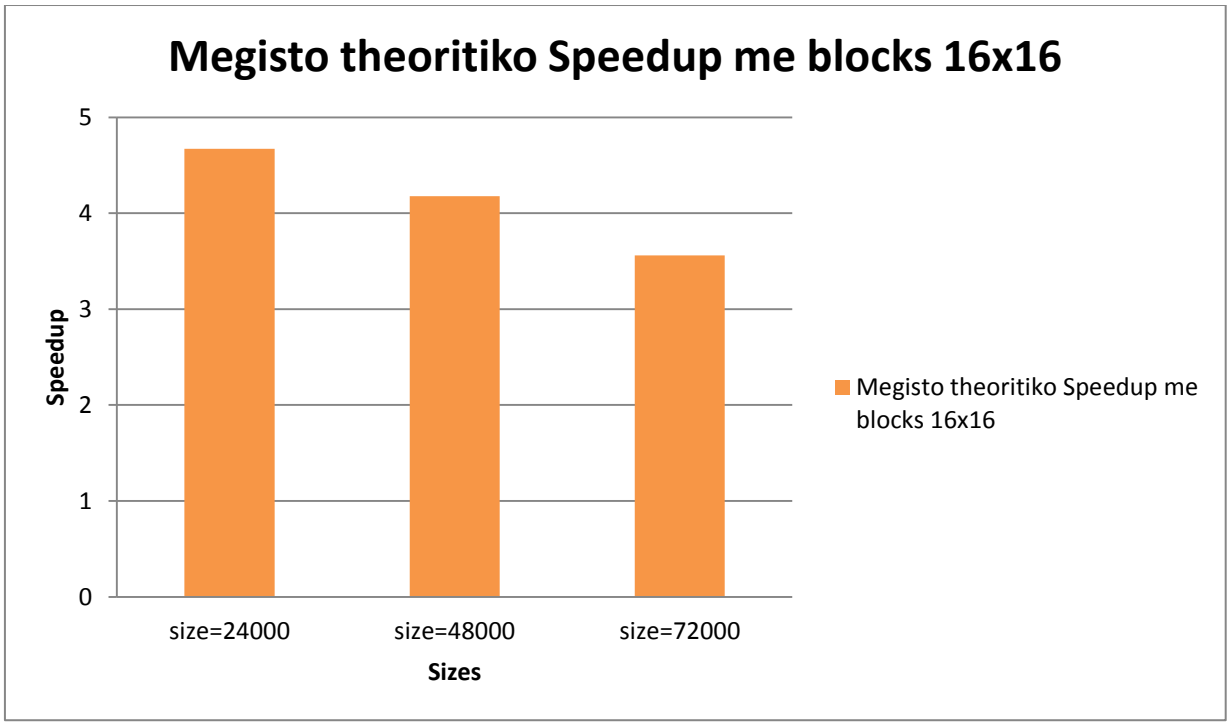
Σχήμα 6.4(α):*Μέγιστο θεωρητικό Speedup με 4x4 blocks(Συνολικά 16 blocks)*



Σχήμα 6.4(β): *Μέγιστο θεωρητικό Speedup με 8x8 blocks(Συνολικά 64 blocks)*



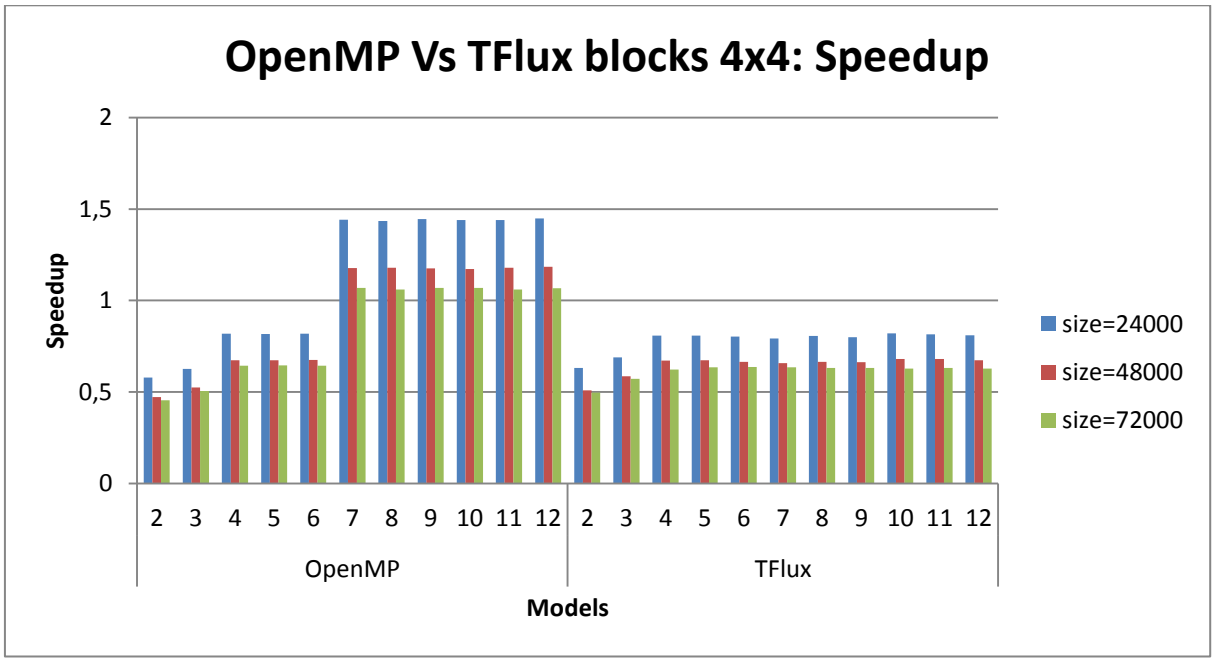
Σχήμα 6.4(γ): *Μέγιστο θεωρητικό Speedup με 12x12 blocks(Συνολικά 144 blocks)*



Σχήμα 6.4(δ): *Μέγιστο θεωρητικό Speedup με 16x16 blocks(Συνολικά 256 blocks)*

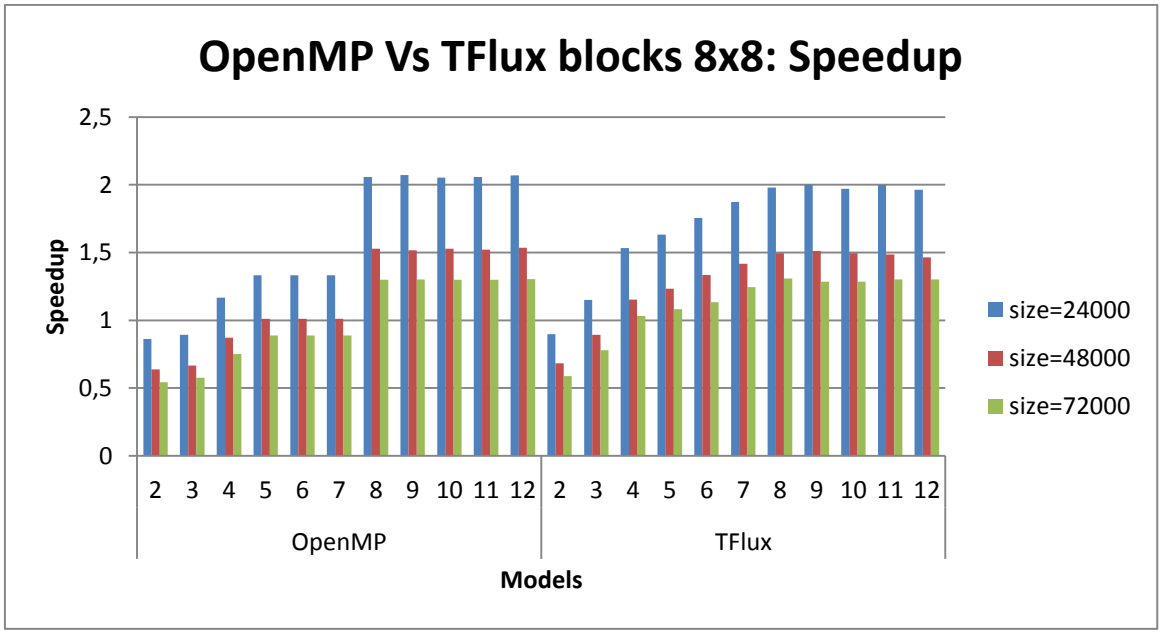
Οι πιο κάτω γραφικές που ακολουθούν παρουσιάζουν για κάθε δοκιμή από διαφορετικό αριθμό blocks την σύγκριση επίδοσης μεταξύ OpenMP και TFlux υλοποίησης.

Στο σχήμα 6.5(α) φαίνετε η σύγκριση επίδοσης μεταξύ OpenMP και TFlux υλοποίησης για 4x4blocks(16 blocks). Είναι εμφανές ότι στην TFlux υλοποίηση είναι μικρότερη η επίδοση επειδή δεν υπάρχουν αρκετά blocks να τρέχουν παράλληλα αφού ο μέγιστος αριθμός των blocks που μπορούν να τρέχουν παράλληλα είναι τέσσερα στην κύρια διαγώνιο. Έτσι με το μικρό αριθμό blocks τα οποία μπορούν να εκτελεστούν παράλληλα δεν επικαλύπτουν το runtime overhead της πλατφόρμας TFlux με αποτέλεσμα να έχει μειωμένη επίδοση με την υλοποίηση σε OpenMP.



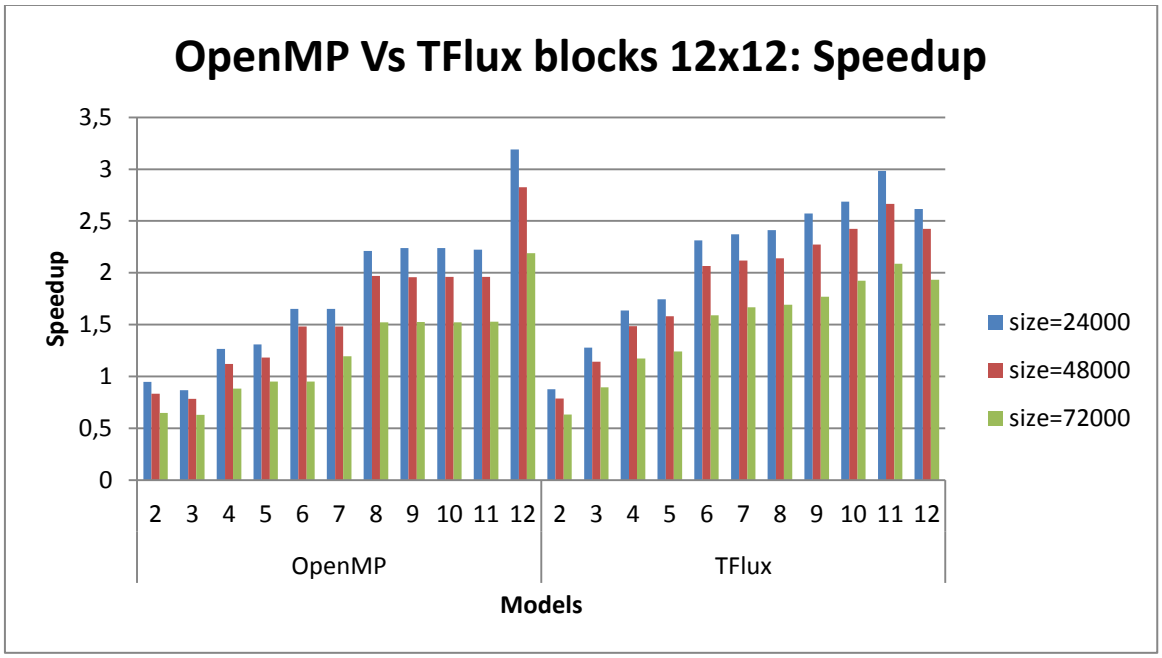
Σχήμα 6.5(α): *Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 4x4*

Στο σχήμα 6.5(β) φαίνετε η σύγκριση επίδοσης μεταξύ OpenMP και TFlux υλοποίησης για 8x8blocks(64blocks). Ο μέγιστος αριθμός των blocks που μπορούν να τρέχουν παράλληλα είναι οκτώ στην κύρια διαγώνιο. Έτσι τώρα με μεγαλύτερο αριθμό blocks τα οποία μπορούν να εκτελεστούν παράλληλα επικαλύπτουν περισσότερα το runtime overhead της πλατφόρμας TFlux και έχει σχεδόν την ίδια επίδοση σε OpenMP με μερικές εξαιρέσεις όπως για παράδειγμα στην OpenMP από τα οκτώ threads και μετά η επίδοση ναι μεν μένει σταθερή αλλά λίγο πιο αυξημένη σε σχέση με τα αντίστοιχα threads/kernels ενώ με την πλατφόρμα TFlux φαίνετε πώς ξεκίνα εξαρχής με πιο ψηλή επίδοση.



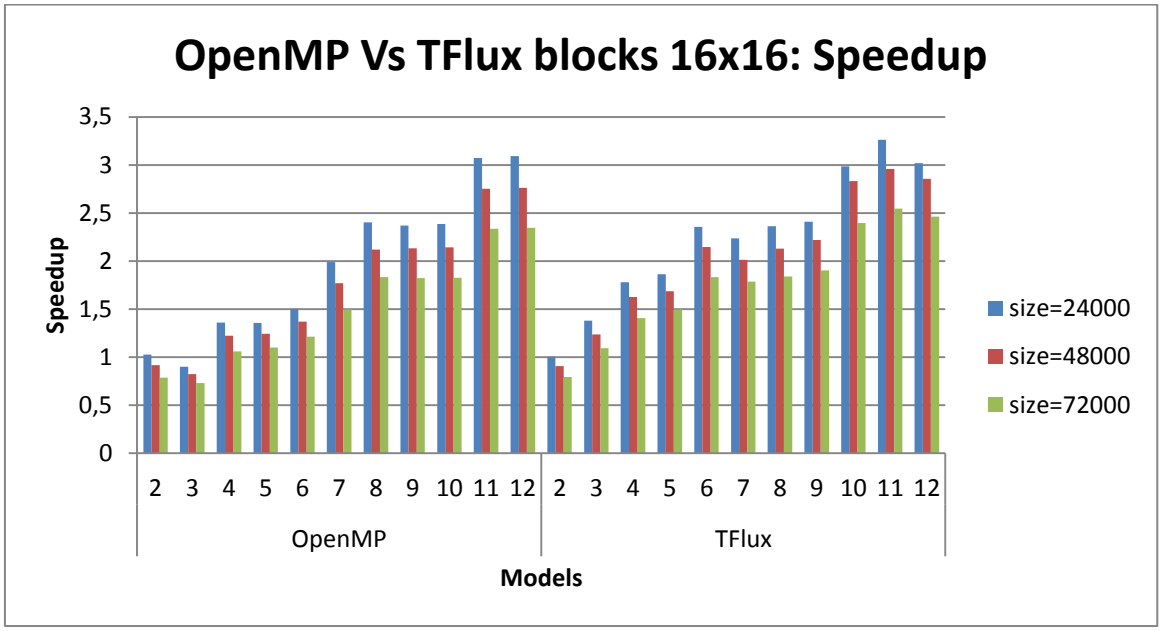
Σχήμα 6.5(β): *Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 8x8*

Στο σχήμα 6.5(γ) φαίνετε η σύγκριση επίδοσης μεταξύ OpenMP και TFlux υλοποίησης για 12x12blocks(144blocks). Ο μέγιστος αριθμός των blocks που μπορούν να τρέχουν παράλληλα είναι δώδεκα στην κύρια διαγώνιο. Έτσι τώρα με μεγαλύτερο αριθμό blocks τα οποία μπορούν να εκτελεστούν παράλληλα επικαλύπτουν περισσότερα το runtime overhead της πλατφόρμας TFlux και είναι εμφανές τώρα ότι έχει λίγη καλύτερη επίδοση από την υλοποίηση με OpenMP. Η αιτία για το φαινόμενο αυτό είναι ο αριθμός των blocks αφού όπως παρατηρείτε με βάση τα προηγούμενα μεγέθη , καθώς αυξάνετε ο αριθμός των blocks αυξάνετε και η επίδοση της υλοποίησης με την χρήση της πλατφόρμας TFlux καθώς εφαρμόζετε πάνω στα blocks dataflow δρομολόγηση με ακριβές εξαρτήσεις που δείχνει ότι εκμεταλλεύεται καλύτερα τον παραλληλισμό . Στο σημείο οπού μειώνετε η επίδοσης της είναι στους 12 Kernels(Threads) αφού δηλώνει με την βάση το σχεδιασμός της πλατφόρμας 13 Threads γεγονός που επιφέρει μεγαλύτερο κόστος.



Σχήμα 6.5(γ): *Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 12x12*

Στο σχήμα 6.5(δ) φαίνετε η σύγκριση επίδοσης μεταξύ OpenMP και TFlux υλοποίησης για 16x16blocks(256blocks). Ο μέγιστος αριθμός των blocks που μπορούν να τρέχουν παράλληλα είναι δεκαέξι στην κύρια διαγώνιο. Έτσι τώρα με μεγαλύτερο αριθμό blocks τα οποία μπορούν να εκτελεστούν παράλληλα επικαλύπτουν περισσότερα το runtime overhead της πλατφόρμας TFlux και είναι εμφανές τώρα ότι έχει λίγη καλύτερη επίδοση από την υλοποίηση με OpenMP. Η αιτία για το φαινόμενο αυτό είναι ο αριθμός των blocks αφού όπως παρατηρείτε με βάση τα προηγούμενα μεγέθη , καθώς αυξάνετε ο αριθμός των blocks αυξάνετε και η επίδοση της υλοποίησης με την χρήση της πλατφόρμας TFlux καθώς εφαρμόζετε πάνω στα blocks dataflow δρομολόγηση με ακριβές εξαρτήσεις που δείχνει ότι εκμεταλλεύεται καλύτερα τον παραλληλισμό .Η μέγιστη επίδοση με την οποία επιτυγχάνετε με την πλατφόρμα TFlux είναι 3,26 ενώ με OpenMP είναι 3,09. Η σχετική διαφορά που υπάρχει μπορεί να είναι μικρή αλλά δείχνει έμπρακτα πως το blocking που εφαρμόζετε και στα δύο μοντέλα εκμεταλλεύεται καλύτερα λόγω του dataflow μοντέλου που στηρίζετε η πλατφόρμα TFlux. Στο σημείο οπού μειώνετε η επίδοσης της είναι στους 12 Kernels(Threads) αφού δηλώνει με την βάση το σχεδιασμός της πλατφόρμας 13 Threads γεγονός που επιφέρει μεγαλύτερο κόστος.



Σχήμα 6.5(δ): *Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 16x16*

6.4 Περίληψη αποτελεσμάτων

Αξιολογήσαμε τον αλγόριθμό OCEAN με δύο διαφορετικές υλοποιήσεις , με την χρήση του μοντέλου OpenMP και της TFlux, στην οποία δημιουργήσαμε τέσσερεις παραλλαγές τις , την εκτέλεση TFlux με 4x4blocks , 8x8blocks, 12x12blocks και με 16x16 blocks και τις ίδιες ακριβώς επίσης και με το μοντέλο OpenMP. Επίσης για την αξιολόγηση μας τρέξαμε τα πιο πάνω πειράματα μας σε τρία διαφορετικά μεγέθη.

Επίσης παρουσιάστηκαν οι μέγιστες θεωρητικές επιδόσεις που μπορούν να υπάρχουν σε κάθε παραλλαγή των blocks πράγμα κατά πόσο βοηθάει να κατανοήσουμε αν τα αποτελέσματα και οι επιδόσεις είναι η όχι λογικές με βάση τις τιμές αυτές. Με βάση τις τιμές μέγιστες θεωρητικές επιδόσεις , όλες οι επιδόσεις που παρατηρούνται είναι σε λογικά πλαίσια μας και είναι κοντά και μικρότερες από αυτές και δεν παρατηρείτε τίποτα το περίεργο.

Η εκτέλεση με OpenMP φάνηκε πώς έχει καλύτερο χρόνο εκτέλεσης αρά και επίδοση από την υλοποίηση TFlux με 4x4blocks . Ο λόγος που παρατηρείτε αυτό είναι ότι η εκτέλεση με blocks4x4 μπορεί να τρέξει παράλληλα το πολύ τέσσερα blocks , ενώ με την εκτέλεση OpenMP και λόγω του μικρού μεγέθους , με το που προσθέτουμε threads μοιράζετε στα ίσα η δουλεία μεταξύ τους με αποτέλεσμα να μειώνετε ο χρόνος εκτέλεσης . Ασφαλέστατε όσο και να μειώνετε ο χρόνος της εκτέλεσης της OpenMP δεν μπορεί σε καμία περίπτωση από όλες τις περιπτώσεις που εξετάσαμε προηγμένος να μειωθεί στο βαθμό που μειώνετε με τις εκτελέσεις 8x8blocks ,12x12blocks και 16x16blocks γιατί εκτενέστερα καθώς αυξάνουμε τον αριθμό φαίνετε ότι στην υλοποίηση με την πλατφόρμα TFlux γίνετε μια επικάλυψει του runtime χρόνου . Ο λόγος είναι ότι όσο μεγαλώνουμε τον παράγοντα για το blocking τότε περισσότερα blocks μπορούν να τρέχουν παράλληλα αφού έχουμε στην διάθεση μας δώδεκα kernels.

Ακόμη ένας παράγοντας που είχαμε ως σκοπό να μελετήσουμε τις υλοποιήσεις που δημιουργήσαμε και τα σχήματα τους που μελετήσαμε προηγουμένως είναι το blocking . Με όλα όσα είδαμε και παρατηρήσαμε πιο πάνω, είναι εμφανές ότι όσο μεγαλύτερο blocking έχουμε τόση μεγαλύτερη επίδοση έχουμε. Δηλαδή η υλοποίηση με blocks8x8 έχει καλύτερη επίδοση από την υλοποίηση 4x4blocks, η υλοποίηση με 12x12blocks έχει πολύ καλύτερη επίδοση από την υλοποίηση 8x8 blocks και τέλος η υλοποίηση με 16x16blocks έχει πολύ καλύτερη επίδοση από την υλοποίηση 12x12 blocks .

Έχοντας ως δεδομένα τα δύο πιο πάνω συμπεράσματα , μπορούμε να συμπεράνουμε ότι επιτυχαίνουμε την καλύτερη επίδοση μεταξύ όλων των γραφικών που παρατηρήσαμε όταν το μέγεθός του πίνακα φτάνει στο μικρό μέγεθος και όταν η υλοποίηση είναι με 16x16blocks και την επίδοση αυτή από την πλατφόρμα TFlux.

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα

Στην εργασία αυτή έγιναν όλες οι διαδικασίες που απαιτούνταν για την εκτέλεση της εφαρμογής OCEAN στο μοντέλο OpenMP και στην πλατφόρμα TFlux. Δημιουργήσαμε παραλλαγές στην πλατφόρμα TFlux έτσι ώστε να δούμε την συμπεριφορά της υλοποίησης αυτής με το να εισάγουμε blocking. Έτσι δημιουργήσαμε τρεις υλοποιήσεις της πλατφόρμα TFlux με 4x4blocks, 8x8blocks, 12x12blocks και 16x16blocks .Το ίδιο εφαρμόσαμε και στην υλοποίηση με το μοντέλο OpenMP Σκοπός ήταν να μελετήσουμε γενικότερα την συμπεριφορά των υλοποιήσεων TFlux μεταξύ της υλοποίησης με OpenMP και μετά να συγκρίνουμε την συμπεριφορά των των υλοποιήσεων TFlux μεταξύ τους παρατηρώντας την συμπεριφορά που είχε η κάθε μια με διαφορετικό blocking size.

Κατά την αξιολόγηση των υλοποιήσεων μας χρησιμοποιήσαμε τρία μεγέθη του προβλήματος ονομασμένα ως small(24000) medium (48000) και large(72000). Τα συμπέρασμα που εξάγουν τα αποτελέσματα είναι ότι η υλοποίηση με την πλατφόρμα TFlux έχει σχετικά λίγο καλύτερη επίδοση από το μοντέλο OpenMP , και επίσης καλυτερεύει και ακόμη πιο πολύ με την αύξηση του αριθμού των blocks . Ο λόγος είναι ότι με την αύξηση του αριθμού των blocks που χρησιμοποιείτε στην υλοποίηση , εκμεταλλεύεται καλύτερα τον παραλληλισμό αφού περισσότερα blocks έχουν την ευκαιρία να τρέχουν παράλληλα.
Μόνο στην υλοποίηση blocking 4x4 η πλατφόρμα TFlux έχει χειρότερη και μικρότερη επίδοση από την υλοποίηση με το μοντέλο OpenMP καθώς ο αριθμός των blocks είναι μικρό και έτσι δεν γίνετε επικάλυψη του runtime χρόνου της πλατφόρμας.
Στην υλοποίηση με blocking 8x8 αυξάνετε η επίδοση της υλοποίηση με πλατφόρμα TFlux μέχρι τους οκτώ kernels και όσοι kernels και να προστεθούν από εκεί και πέρα δεν κάνει καμία διαφορά στο αποτέλεσμα. Ενώ στην υλοποίηση με blocking 12x12 αυξάνετε η επίδοση καθ’

όλη την διάρκεια του προστίθενται kernels μέχρι όμως τους έντεκα kernels λόγω του ότι όπως προαναφέραμε ο ένας kernel από τους διαθέσιμους δώδεκα δεσμεύετε από την μονάδα συγχρονισμού TSU, και έτσι σε αυτή την περίπτωση εδώ χρησιμοποιούνται εποικοδομητικά και οι δώδεκα kernels που υποστήριζε το hardware .

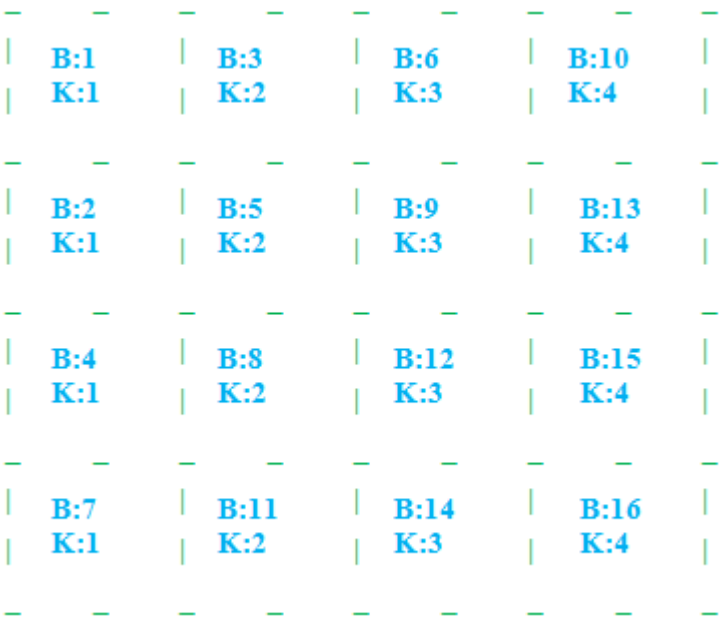
Αν έπρεπε να διαλέγαμε την υλοποίηση με την καλύτερη επίδοση που επετεύχθητε με αυτά τα δεδομένα, θα διαλέγαμε την υλοποίηση με blocking 16x16 αφού γίνετε μέχρι και 1,05 φορές γρηγορότερη σε σύγκριση με την OpenMP .

7.2 Μελλοντική Εργασία

Σε γενικές γραμμές , είμαστε αρκετά ικανοποιημένοι από την υλοποίηση μας αφού ο στόχος για την υλοποίηση του OCEAN αλγορίθμου στην πλατφόρμα TFlux έχει επιτευχθεί, ενώ παράλληλα κάναμε την ίδια υλοποίηση σε OpenMP για να μπορούσαμε να συγκρίναμε τα αποτελέσματα της πλατφόρμα TFlux με αποτέλεσμα ενός άλλου μοντέλου , το οποίο έχει κάποιες ομοιότητες του όσο αφορά την σύνταξη. Θα μπορούσε η σύγκριση αυτή να γίνει και με άλλα μοντέλα όπως BBT ή UPC και είναι μια καλή μελλοντική επέκταση , έτσι ώστε η πλατφόρμα TFlux να συγκριθεί με αρκετά γνωστά μοντέλα και γλώσσες και να παίρναμε μια πιο πλήρης εικόνα για την συμπεριφορά της.

Μελετώντας την υλοποίηση και τις γραφικές οι οποίες προέκυψαν και μελετήσαμε σε προηγούμενα κεφάλαια , βρήκαμε πως την ψηλότερη επίδοση επιτεύχθηκε από την υλοποίηση με blocking 16x16. Ό λόγος που είχε την μεγαλύτερη επίδοση , όπως προαναφέραμε, ήταν ότι με αυτό το blocking που εφαρμόστηκε υπήρξε η δυνατότητα να τρέξουν παράλληλα δώδεκα blocks μιας και το σύστημα στο οποίο βρισκόμασταν διέθετε δώδεκα πυρήνες. Μια άλλη ιδέα βελτίωσης της επίδοσης είναι να μεγαλώσουμε το μέγεθός του blocking και να δοκιμαστεί η υλοποίηση αυτή και σε μηχανή με περισσότερους επεξεργαστές.
Μια άλλη ακόμη ιδέα της βελτίωσης είναι να υλοποιηθεί μια επιπρόσθετη λειτουργία στην πλατφόρμα TFlux. Η ιδέα για αυτή την επέκταση είναι στο ότι όταν θα θέλουμε να χρησιμοποιούμε blocking, παρά να γίνετε αυτό manual και να πρέπει να είμαστε προσεκτικοί με την θέση στήλη και γραμμή που δίνουμε σαν παραμέτρους , να φτιάξουμε ένα pragma directive που να κάνει όλη αυτή την δουλειά που γινόταν με manual τρόπο σε αυτόματη δουλεία. Θα παίρνει σαν παραμέτρους το blocking το οποίο θα θέλει να εφαρμοστεί και τις διαστάσεις του πλέγματος. Αν η διαίρεση πλέγματος με το blocking δεν είναι ακριβές και αφήνει υπόλοιπο να βγάζει μήνυμα λάθους. Η κυρίως λειτουργία του macro είναι να δημιουργεί τα blocks με διαγώνιο τρόπο και το κάθε ένα θα έχει μοναδικό αναγνωριστικό ένα αύξων αριθμό που θα αρχίζει με ένα και επίσης σε κάθε διαγώνιο να αναθέτουμε το κάθε block σε διαφορετικό kernel αρχίζοντας από τον kernel 1. Αφού το κάθε block πάρει τις δύο αυτές τιμές θα μπορούν

να δημιουργούνται τα DThreads αφού το id και ο αριθμός του kernel σε κάθε directive θα είναι υπολογισμένες από τις δύο αυτές τιμές που προαναφέραμε. Θα ήταν ακόμη καλύτερο να παρουσιάζεται αυτό με τον ίδιο γραφικό τρόπο που παρουσιάζει η πλατφόρμα TFlux τις πληροφορίες για το κάθε threads όταν γίνει compile.



Σχήμα 7 : Γραφικό περιβάλλον παρουσιάσεις τις αντιστοιχείς Block με Kernel

Το σχήμα 7 παρουσιάζει μια ιδέα το πώς θα μπορούσε να ήταν το γραφικό παραβάλλον. Απλό και κατανοητό και να παρουσιάζετε στον χρήστη την στιγμή που κάνει compile για να βλέπει και ο ίδιος οι kernels θα αναλάβουν ποια blocks συγκεκριμένα και να γίνετε και μια επαλήθευση του .

Βιβλιογραφία

[1] Stavrou K., Nikolaides M., Pavlou D., Arandi S., Evripidou P., Trancoso P. "TFlux: A Portable Platform for Data-Driven Multithreading on Commodity Multicore Systems," Parallel Processing, 2008. ICPP '08. 37th International Conference on, vol., no., pp.25-34, 9-12 Sept. 2008

[2] Stavrou K., Pavlou D., Nikolaides M., Petrides P., Evripidou P., Trancoso P., Popovic Z., Giorgi R. "Programming Abstractions and Toolchain for Dataflow Multithreading Architectures," Parallel and Distributed Computing, 2009. ISPDC '09. Eighth International Symposium on, vol., no., pp.107-114, June 30 2009-July 4 2009

[3]Trancoso, Pedro, Kyriakos Stavrou, and Paraskevas Evripidou. "DDMCPP: The data-driven multithreading C pre-processor." The 11th Workshop on Interaction between Compilers and Computer Architectures. 2007.

[4] Kyriacou, Costas, Paraskevas Evripidou, and Pedro Trancoso. "Data-driven multithreading using conventional microprocessors." Parallel and Distributed Systems, IEEE Transactions on 17.10 (2006): 1176-1188.

[5] Stavrou, Kyriakos, Paraskevas Evripidou, and Pedro Trancoso. "DDM-CMP: data-driven multithreading on a chip multiprocessor." Embedded Computer Systems: Architectures, Modeling, and Simulation. Springer Berlin Heidelberg, 2005. 364-373.

[6] Arandi, Samer, and Paraskevas Evripidou. "Programming multi-core architectures using data-flow techniques." Embedded Computer Systems (SAMOS), 2010 International Conference on. IEEE, 2010.

[7] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. 1992. SPLASH: Stanford parallel applications for shared-memory. SIGARCH Comput. Archit. News 20, 1 (March 1992), 5-44. DOI=10.1145/130823.130824 http://doi.acm.org/10.1145/130823.130824

[8] Woo, Steven Cameron, et al. "The SPLASH-2 programs: Characterization and methodological considerations." ACM SIGARCH Computer Architecture News. Vol. 23. No. 2. ACM, 1995..

[9] Singh, Jaswinder Pal, and John L. Hennessy. "Finding and exploiting parallelism in an ocean simulation program: Experience, results, and implications." Journal of Parallel and Distributed Computing 15.1 (1992): 27-48.

[10] Kersken, H-P., et al. "Parallelization of large scale ocean models by data decomposition." High-Performance Computing and Networking. Springer Berlin Heidelberg, 1994.

[11] UCY CS Dep. (2008) TFlux C Preprocessor TFluxCpp, Technical Manual, July 3rd 2008, Univarsity Of Cyprus

[12] Nikolaides Marios “Preprocessor και Benchmarks για Πλατφορμα TFlux-Soft”, Bachelor Thesis, May 2008, University Of Cyprus CS Department

[13]Giannos Stylianou “Data-Driven Multithreading για Many-Core Αρχιτεκτονικές: TFluxSCC”, Bachelor Thesis, May 2012, University Of Cyprus CS Department

[14] Bob Kuhn, Paul Petersen ,Kuck & Associates, Inc, Eamonn O’Toole Compaq Computer Corporation "OpenMP versus Threading in C/C++"

[15] OpenMP Application,Program Interface,Version 3.0 May 2008

[16] Marcela Madridi, Mahidhar Tatineni, TeraGrid Outreach Workshop, "MPI & OpenMP Examples"

[17] Edward Valeev, Department of Chemistry Virginia Tech Blacksburg, VA , " Introduction to OpenMP"

[18] Mattson, Tim. "Introduction to openmp." Proceedings of the 2006 ACM/IEEE conference on Supercomputing. ACM, 2006.

[19]Colignon, David. "Introduction to Parallel Programming in OpenMP." CÉCI-Consortium des Équipements de Calcul Intensif, http://www. cecihpc. be (2009).

[20]Kiessling, Alina. "An Introduction to Parallel Programming with OpenMP." (2009).

[21] Tim Mattson,Principal Engineer Intel Corporation ,Larry Meadows,Principal EngineerIntel Corporation , “A “Hands-on” Introduction to OpenMP* ”

[22] Matloff, Norm. "Programming on parallel machines." University of Calfornia, Davis (2011).”

[23] Culler, David E., Jaswinder Pal Singh, and Anoop Gupta. "Parallel computer architecture." Harcoust Asia and Morgan Kaufmann (1999).

[24] Jun Zhang, Laboratory for High Performance Scientific Computing & Computer Simulation, and Laboratory for Computational Medical Imaging & Data Analysis, CS 623 - Parallel Iterative Computing (http://www.cs.uky.edu/courses/cs623)

[25] Preclik, Tobias, and Dominik Bartuschat. "Gauss Seidel Parallelization." (2009).

[26] Wallin, Dan, et al. "Multigrid and Gauss-Seidel smoothers revisited: parallelization on chip multiprocessors." Proceedings of the 20th annual international conference on Supercomputing. ACM, 2006.

[27] Demmel, James W. Applied numerical linear algebra. Society for Industrial and Applied Mathematics, 1997.

[28] Dios, Antonio J., et al. A case study of the task-based parallel wavefront pattern. Technical Report at http://www. ac. uma. es/~ asenjo/research, 2011.

[29] Heinrich, Mark, and Mainak Chaudhuri. "Ocean warning: avoid drowning." ACM SIGARCH Computer Architecture News 31.3 (2003): 30-32.

[30] Bienia, Christian, Sanjeev Kumar, and Kai Li. "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors." Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on. IEEE, 2008.

[31] Manjikian, Naraig. "Multiprocessor enhancements of the SimpleScalar tool set." ACM SIGARCH Computer Architecture News 29.1 (2001): 8-15.

[32] Matteo Monchiero, Jung Ho Ahn, Ayose Falc\&\#243;n, Daniel Ortega, and Paolo Faraboschi. 2009. How to simulate 1000 cores. SIGARCH Comput. Archit. News 37, 2 (July 2009), 10-19. DOI=10.1145/1577129.1577133 http://doi.acm.org/10.1145/1577129.1577133

[32] Michelle L. Goodstein, Evangelos Vlachos, Shimin Chen, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2010. Butterfly analysis: adapting dataflow analysis to dynamic

parallel monitoring. SIGPLAN Not. 45, 3 (March 2010), 257-270. DOI=10.1145/1735971.1736050 http://doi.acm.org/10.1145/1735971.1736050

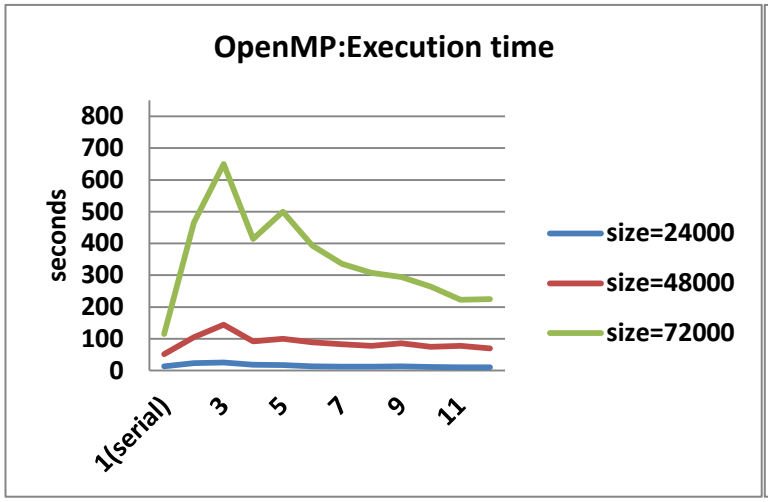
[33] Hoisie, Adolfy, Olaf Lubeck, and Harvey Wasserman. "Performance analysis of wavefront algorithms on very-large scale distributed systems." Workshop on wide area networks and high performance computing. Springer London, 1999.

[34] Wavefront Pattern, University of Illinois at Urbana-Champaign. College of Engineering Department of Computer Science. [Online]. Available: http://www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf

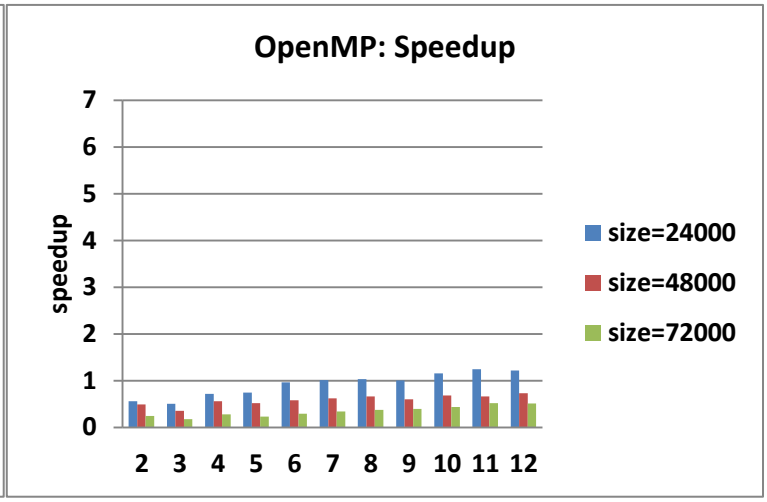
[35] Lewis, E. Christopher, and Lawrence Snyder. "Pipelining wavefront computations: Experiences and performance." Parallel and Distributed Processing. Springer Berlin Heidelberg, 2000. 261-268.

Παράρτημα Α

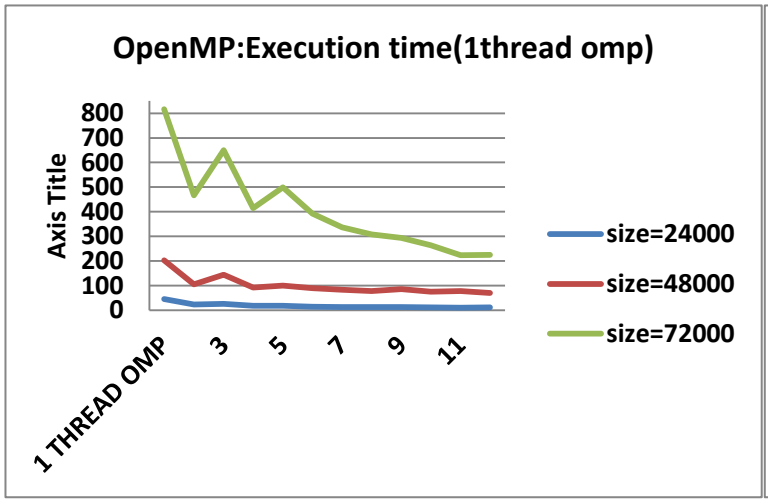
Το παράρτημα αυτό αναφέρεται στο Κεφάλαιο 5 , στο οποίο παρουσιάζονται οι γραφικές παραστάσεις με τα αποτελέσματα που πήραμε από τις εκτελέσεις των εφαρμογών.



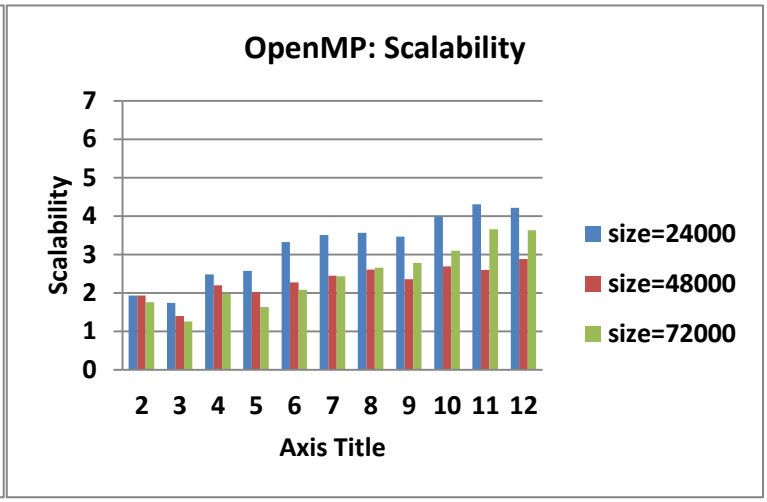
Χρόνοι Εκτέλεσης OpenMP



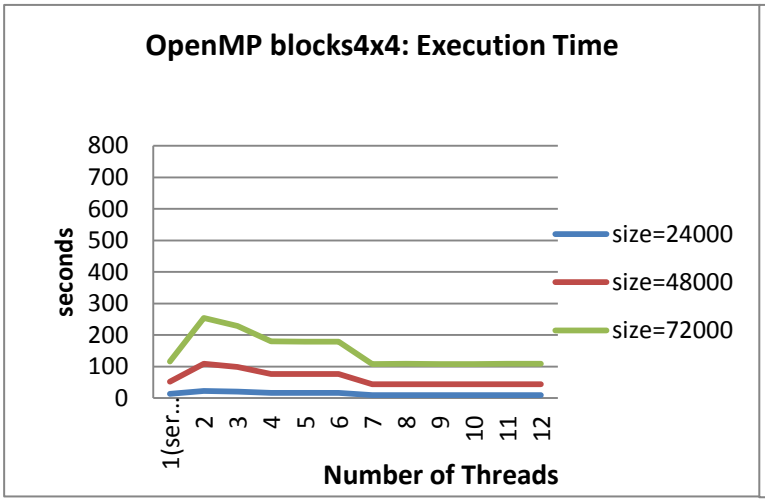
Επίδοση OpenMP



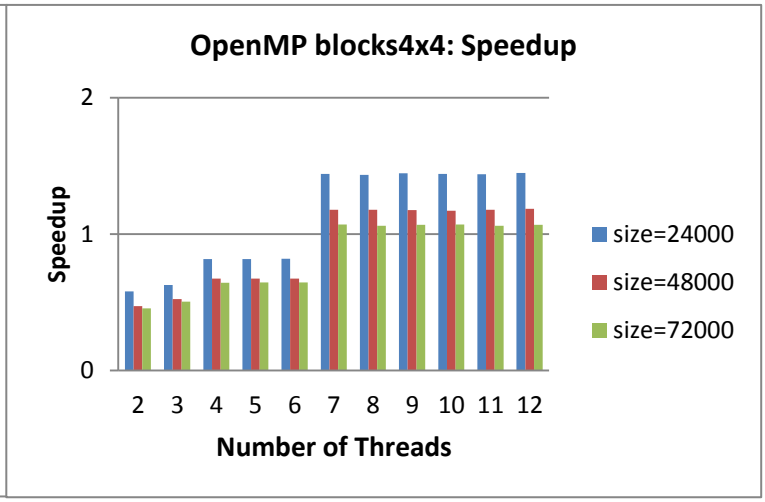
Χρόνοι Εκτέλεσης OpenMP



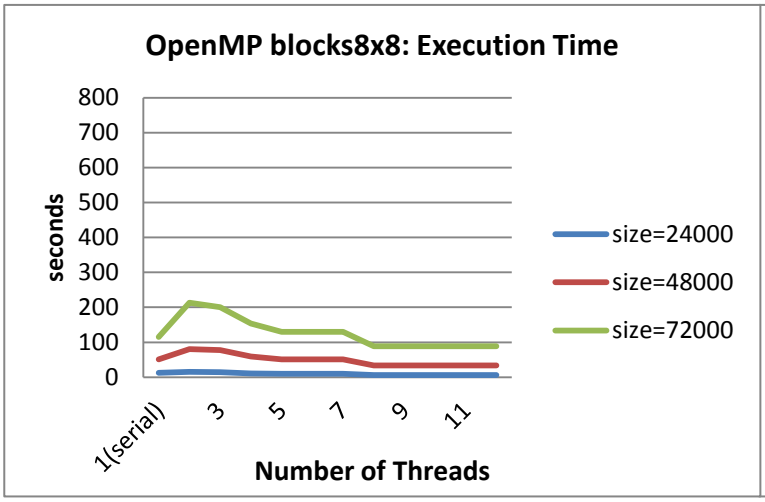
Scalability OpenMP



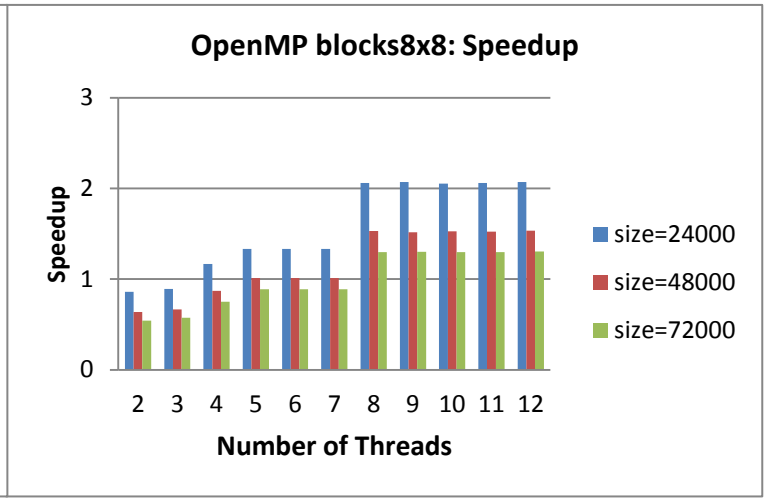
Χρόνοι Εκτέλεσης OpenMP 4x4blocks



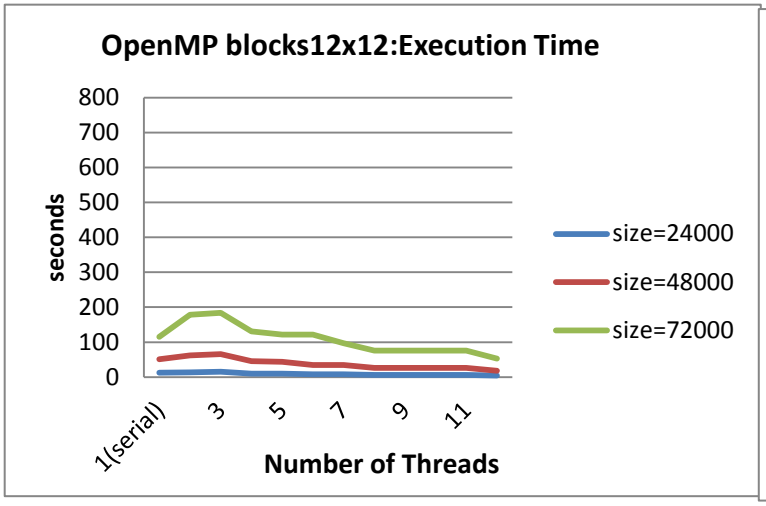
Επίδοση OpenMP 4x4blocks



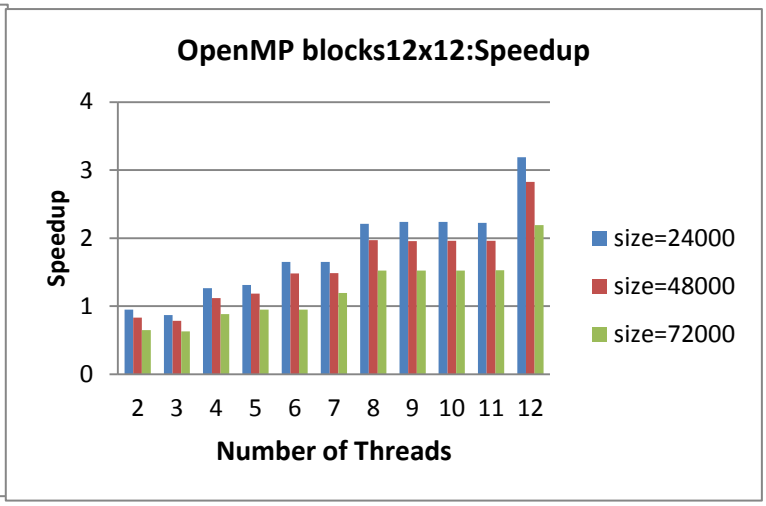
Χρόνοι Εκτέλεσης OpenMP 8x8blocks



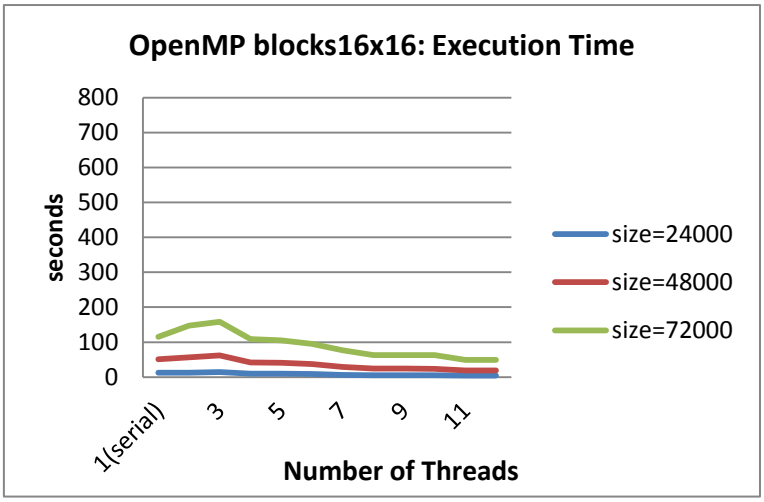
Επίδοση OpenMP 8x8blocks



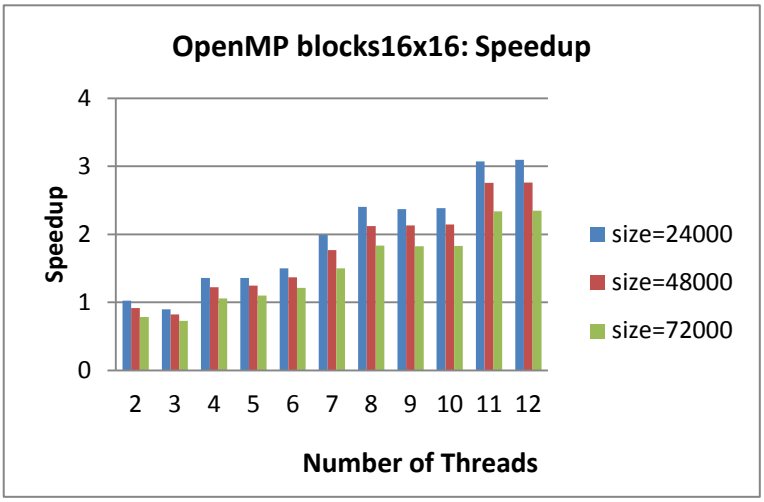
Χρόνοι Εκτέλεσης OpenMP 12x12blocks



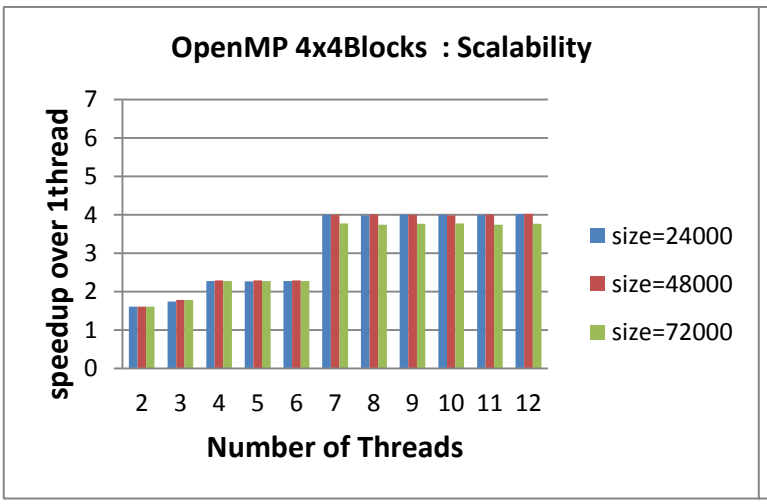
Επίδοση OpenMP 12x12blocks



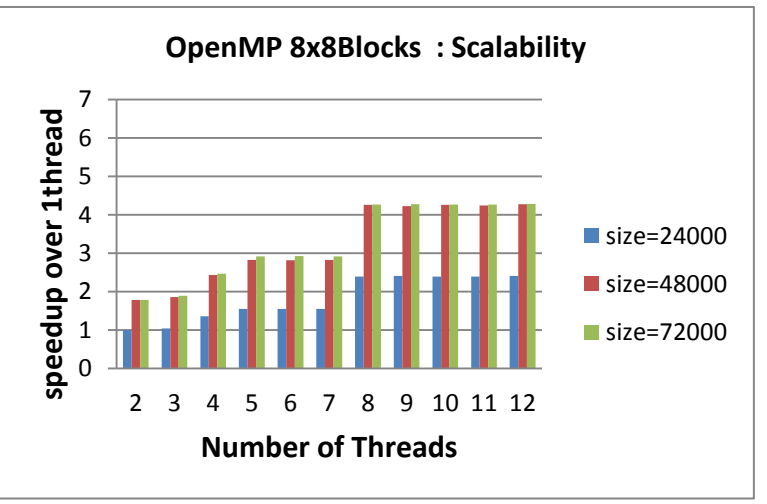
Χρόνοι Εκτέλεσης OpenMP 16x16blocks



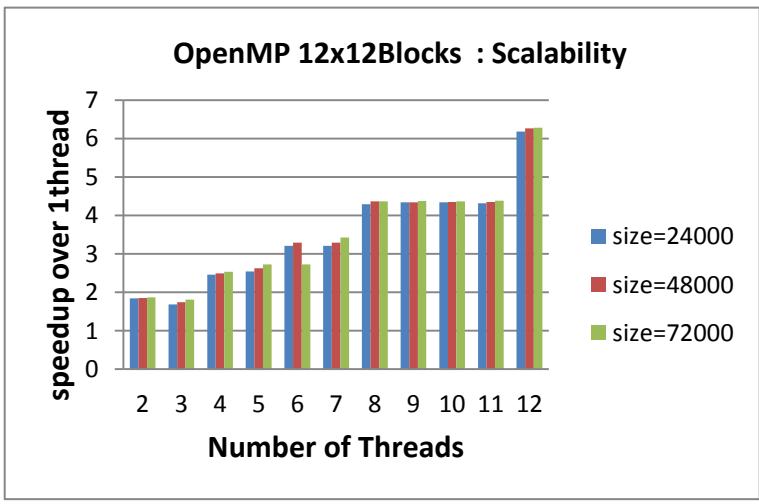
Επίδοση OpenMP 16x16blocks



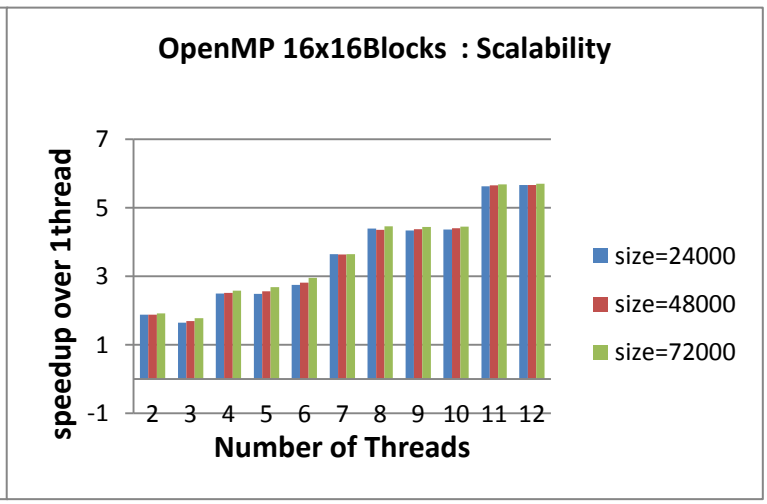
Scalability OpenMp 4x4blocks



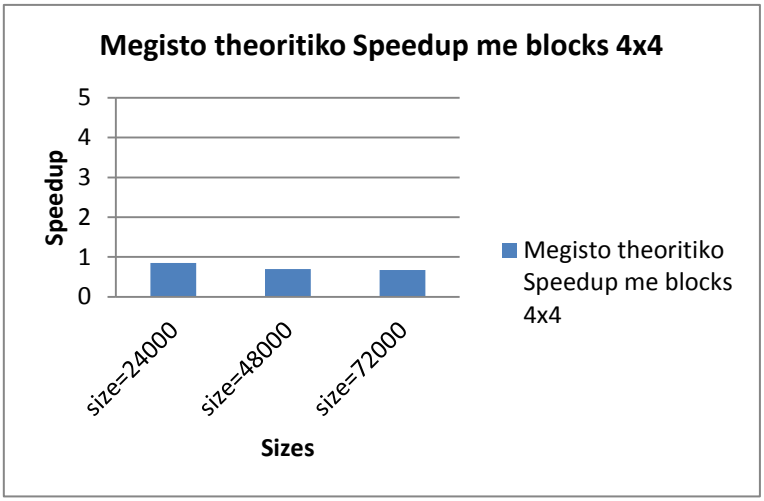
Scalability OpenMp 8x8 blocks



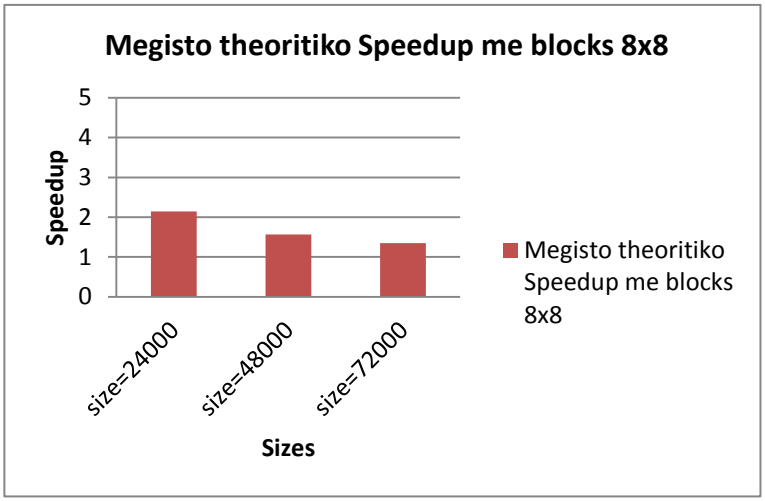
Scalability OpenMp 12x12blocks



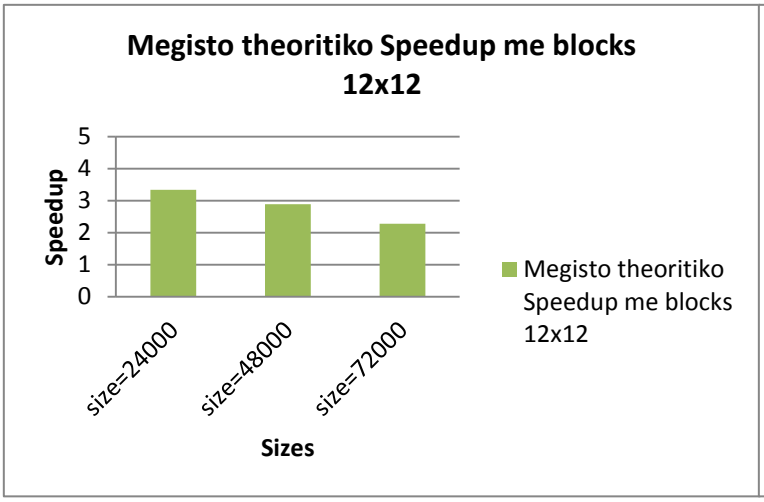
Scalability OpenMp 16x16 blocks



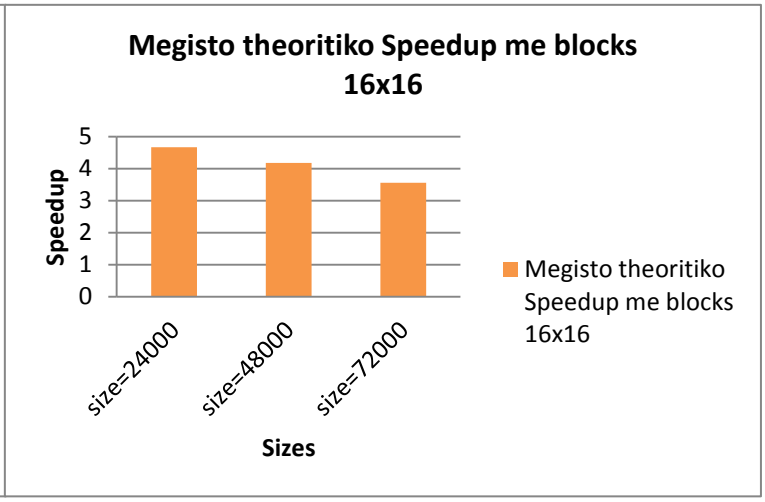
Μέγιστο θεωρητικό Speedup με 4x4 blocks



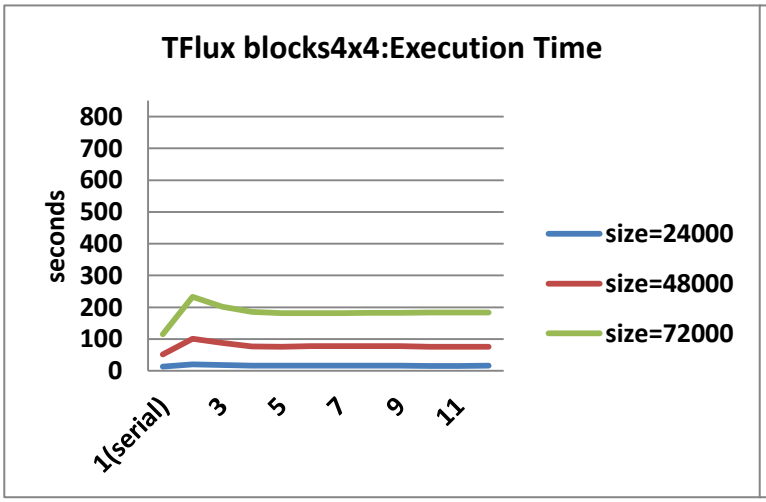
Μέγιστο θεωρητικό Speedup με 8x8 blocks



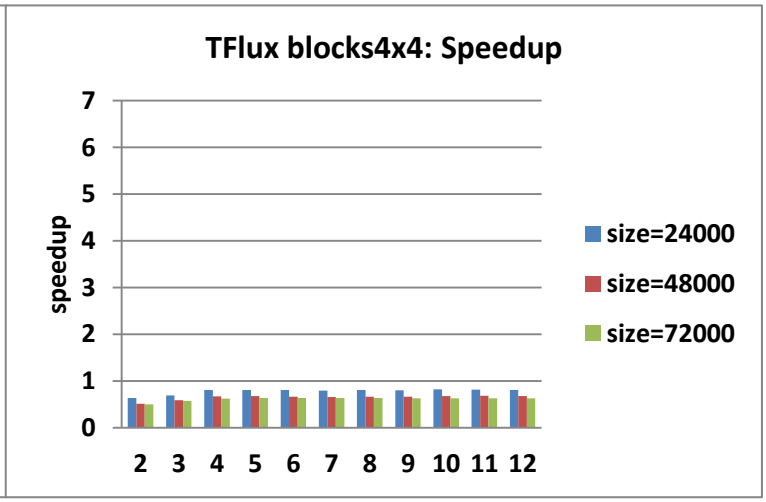
Μέγιστο θεωρητικό Speedup με 12x12 blocks



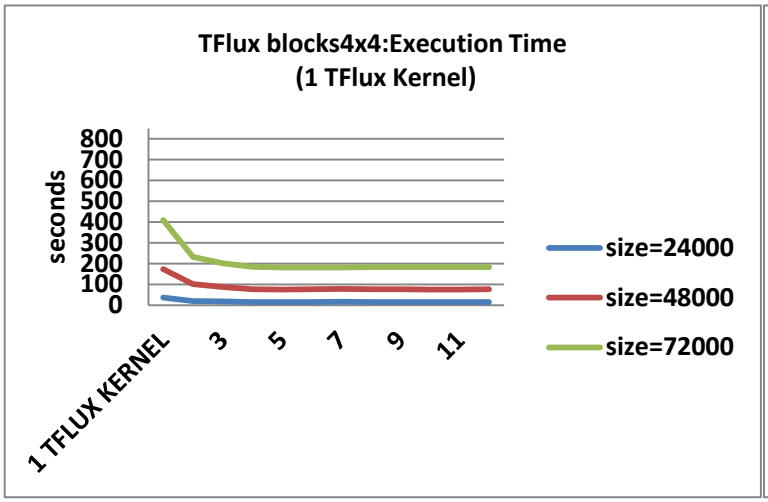
Μέγιστο θεωρητικό Speedup με 16x16blocks



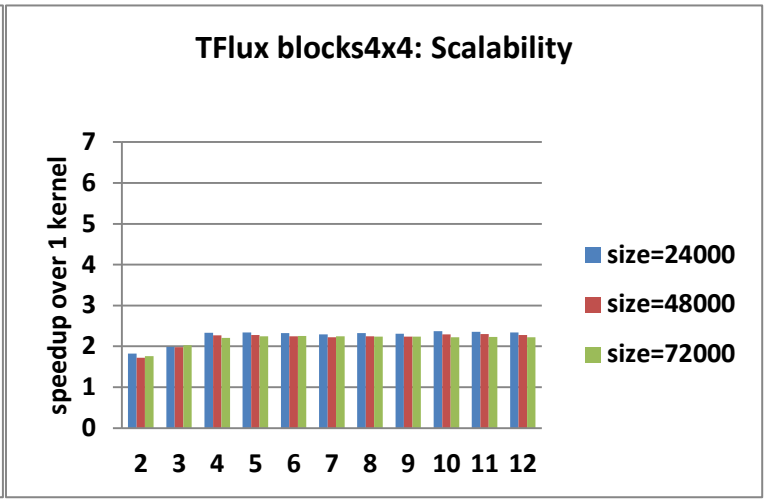
Χρόνοι Εκτέλεσης TFlux 4x4blocks



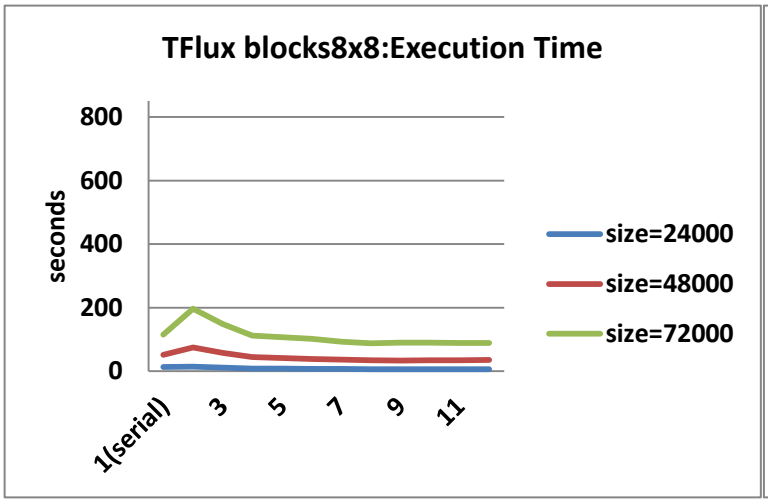
Επίδοση TFlux 4x4blocks



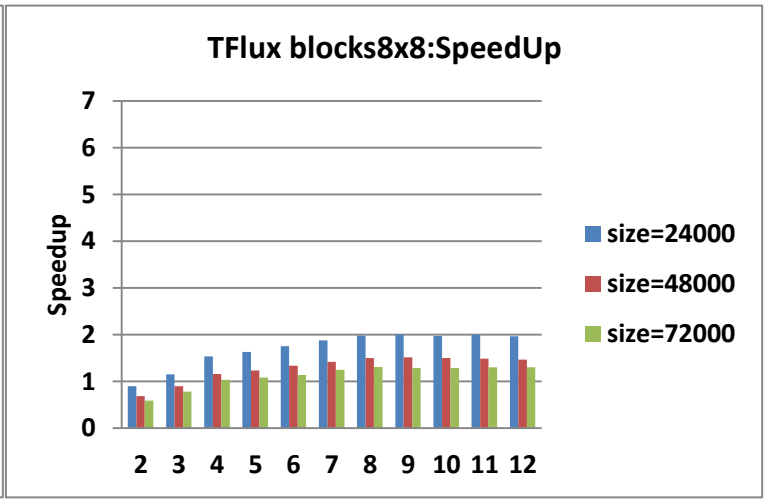
Χρόνοι Εκτέλεσης TFlux 4x4blocks



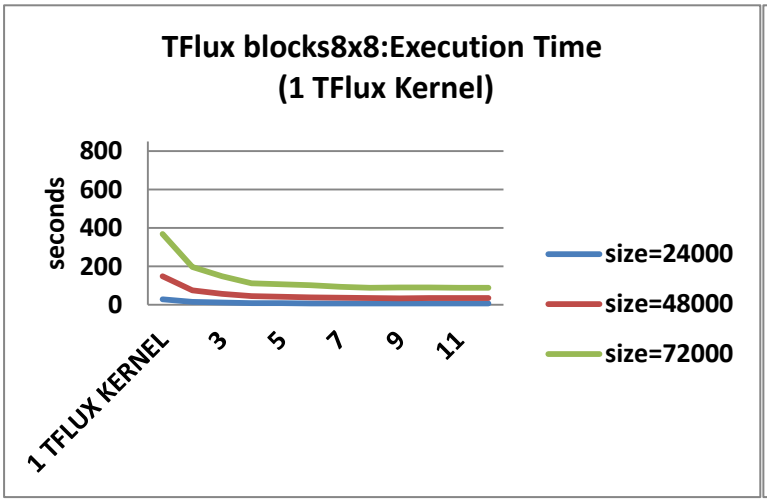
Scalability TFlux 4x4blocks



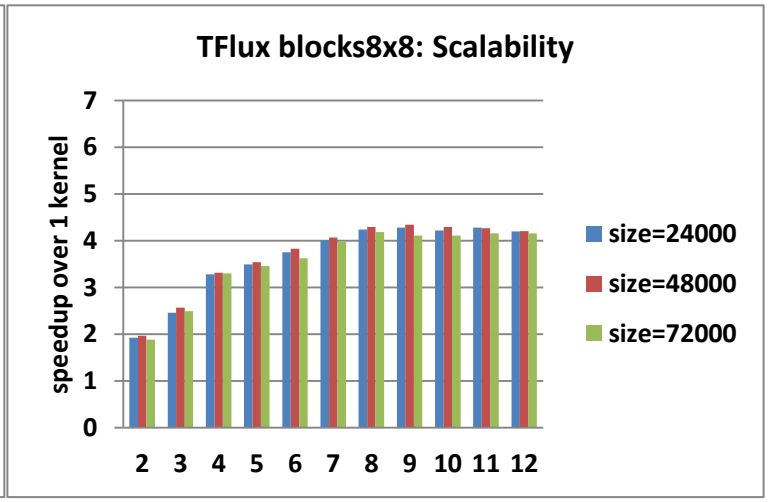
Χρόνοι Εκτέλεσης TFlux 8x8blocks



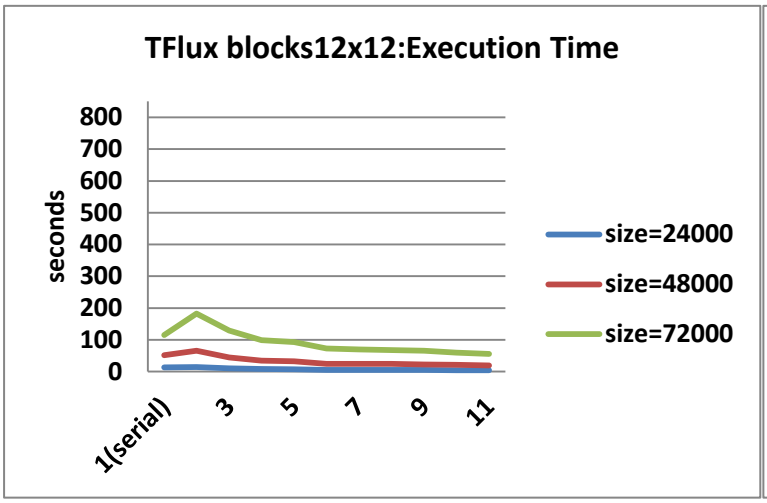
Επίδοση TFlux 8x8blocks



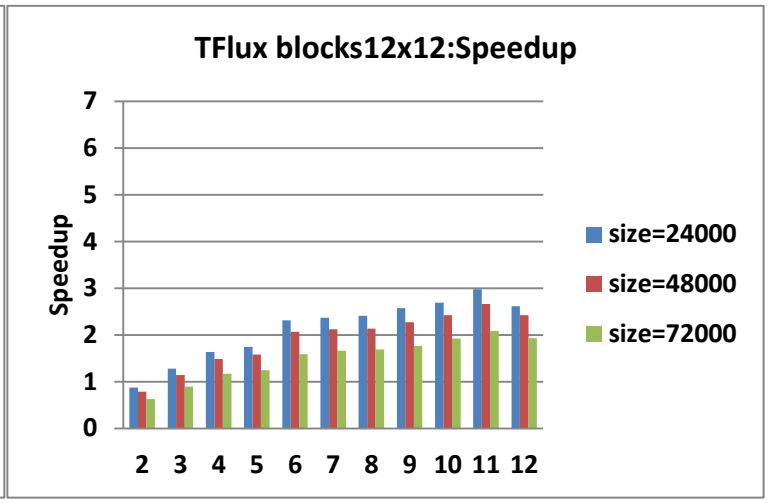
Χρόνοι Εκτέλεσης TFlux 8x8blocks



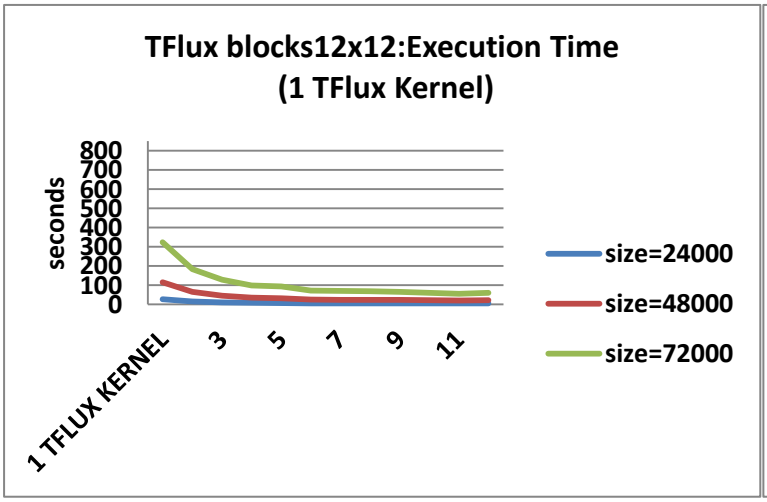
Scalability TFlux 8x8blocks



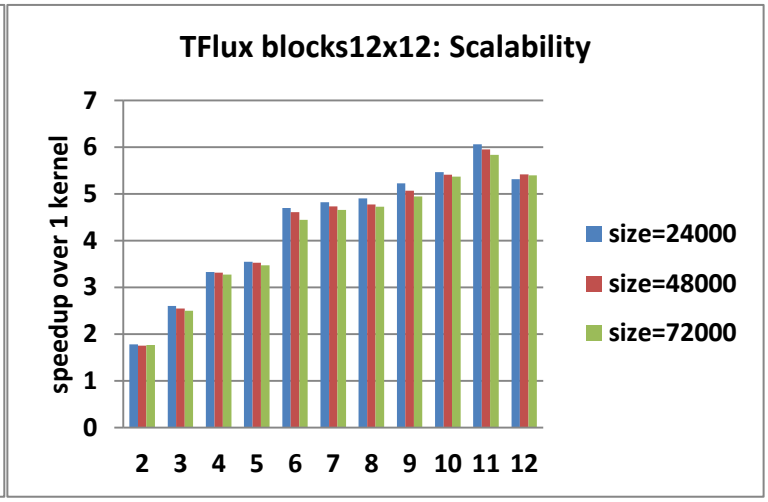
Χρόνοι Εκτέλεσης TFlux 12x12blocks



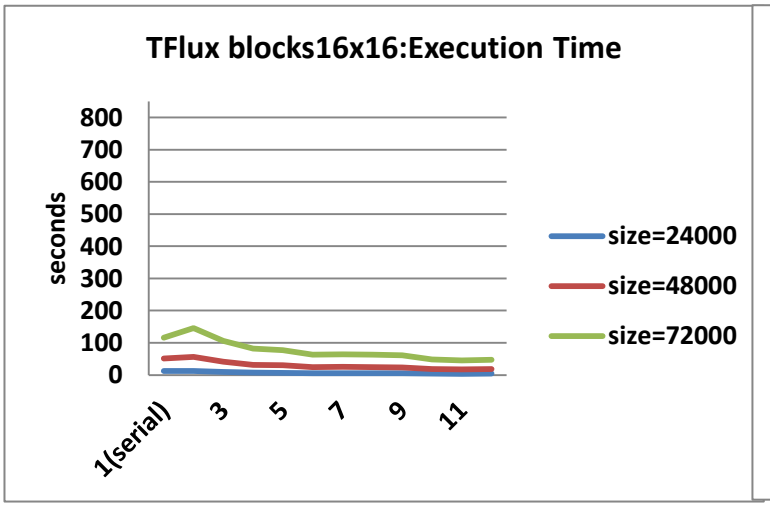
Επίδοση TFlux 12x12blocks



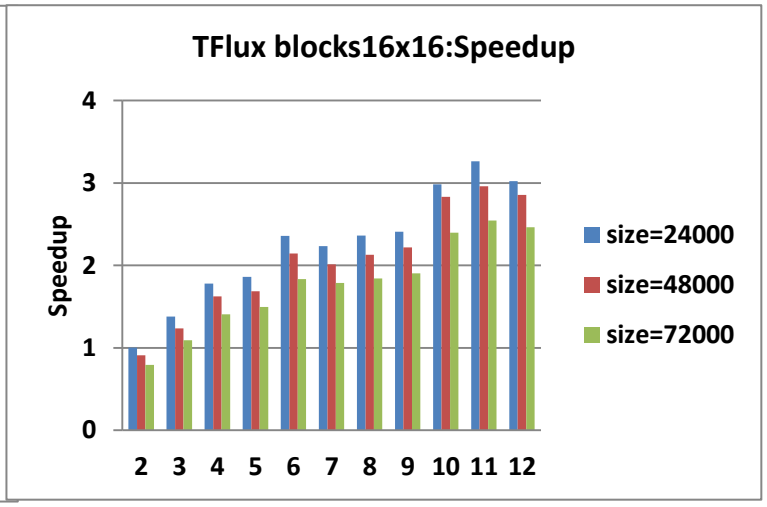
Χρόνοι Εκτέλεσης TFlux 12x12blocks



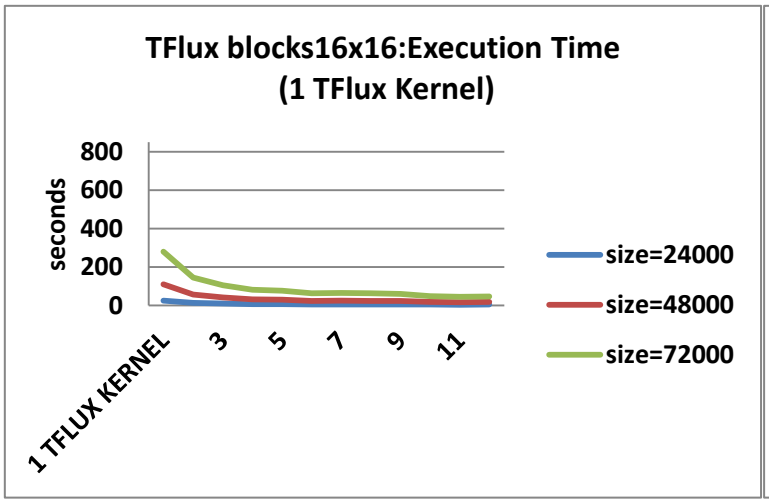
Scalability TFlux 12x12blocks



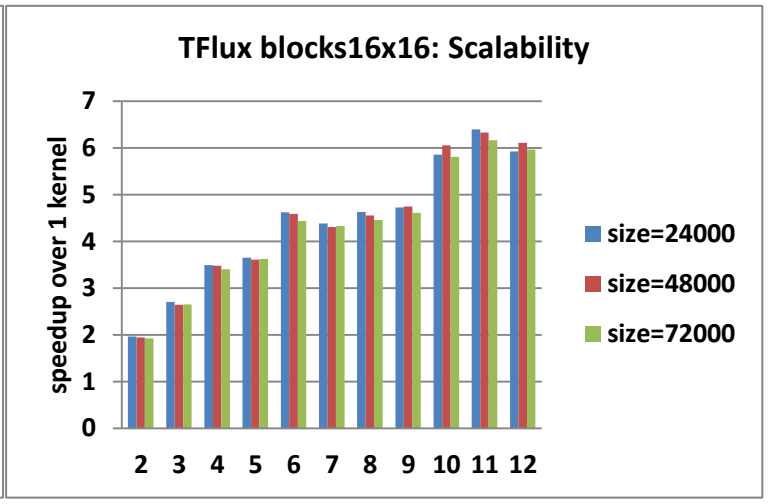
Χρόνοι Εκτέλεσης TFlux 16x16blocks



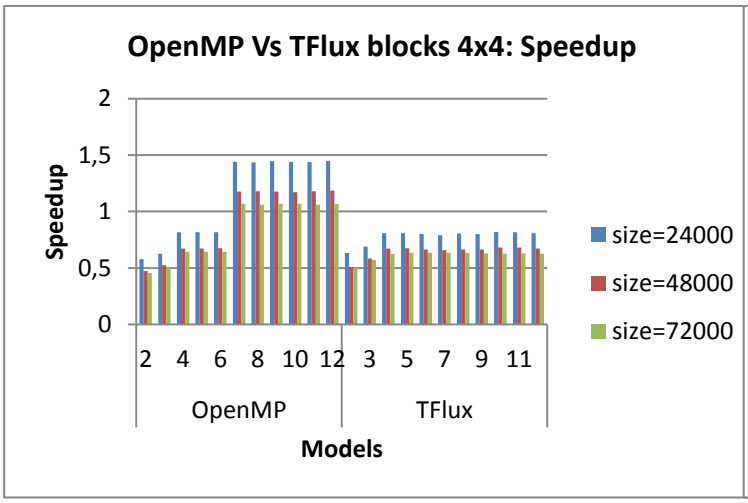
Επίδοση TFlux 16x16blocks



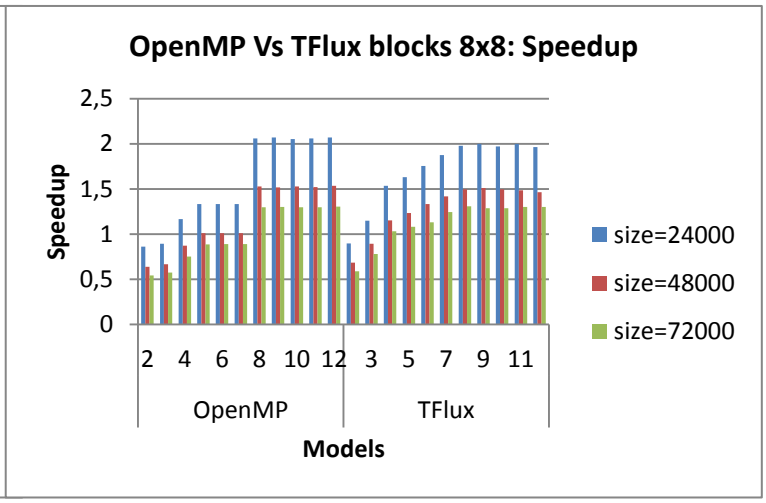
Χρόνοι Εκτέλεσης TFlux 16x16blocks



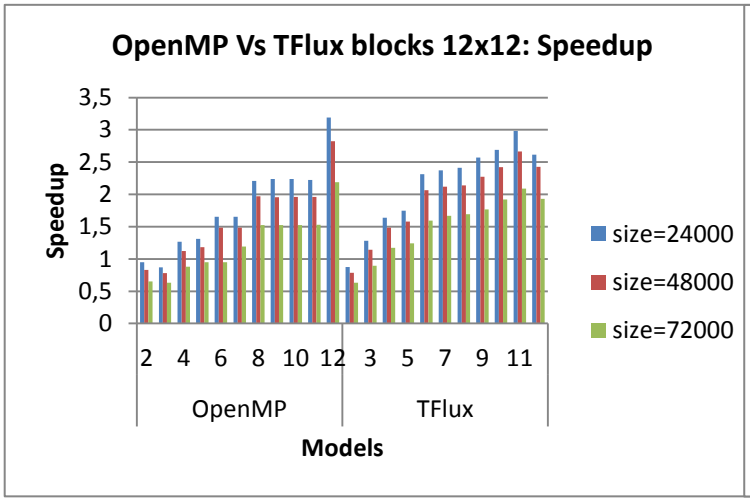
Scalability TFlux 16x16blocks



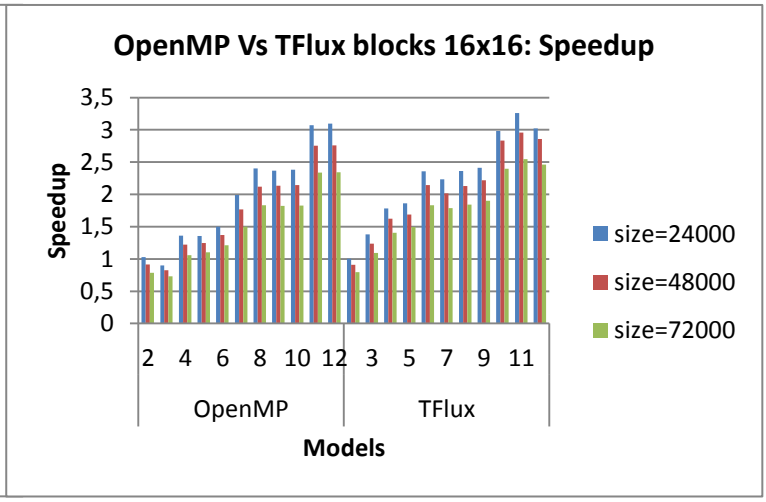
Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 4x4



Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 8x8



Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 12x12



Επίδοση TFlux και OpenMP υλοποιήσεων με blocking 16x16

Παράρτημα Β

Το παράρτημα αυτό αναφέρετε στο Κεφάλαιο 3 και συγκεκριμένα στο τμήμα 3.1.1. όπου αναφέρετε το μοντέλο προγραμματισμού OpenMP. Πιο κάτω παρουσιάζονται μερικές διαθέσιμές συναρτήσεις στο OpenMP.

void omp set num threads(int num threads);
Καθορίζει τον αριθμό των νημάτων που θα χρησιμοποιηθούν στις παράλληλες περιοχές που ακολουθούν.

int omp get num threads(void);
Επιστρέφει τον αριθμό των νημάτων

int omp get thread num(void);
Επιστρέφει τον αριθμό του νήματος που εκτελεί την συνάρτηση και η τιμή βρίσκετε ανάμεσα στο 0 και στο omp get num threads()-1 .Για το main thread θα επιστρέψει 0.

int omp get num procs(void);
Επιστρέφει το μέγιστο αριθμό επεξεργαστών που θα μπορούσε να χρησιμοποίηση ένα πρόγραμμα.

int omp in parallel(void);
Η συνάρτηση αυτή αν κληθεί μέσα από μια παράλληλη περιοχή επιστρέφει μια μη-μηδενική τιμή αλλιώς επιστρέφει 0.

void omp set dynamic(int dynamic threads);
Ενεργοποιεί/απενεργοποιεί το δυναμικό καθορισμό του αριθμού νημάτων στις παράλληλες περιοχές.

int omp get dynamic(void);
Η συνάρτηση αυτή επιστρέφει μη-μηδενική τιμή αν ο δυναμικός καθορισμός των νημάτων είναι ενεργοποιημένος αλλιώς 0.

void omp set nested(int nested);
Ενεργοποιεί/απενεργοποιεί το φωλιασμένο παραλληλισμό.

void omp get nested(void);
Η συνάρτηση αυτή επιστρέφει μη μηδενική τιμή εάν είναι ενεργοποιημένος ο φωλιασμένος παραλληλισμός διαφορετικά 0.

void omp init lock(omp lock t *lock);
void omp init nest lock(omp nest lock t *lock);
Αρχικοποιεί την κλειδαριά lock/ nest lock αντίστοιχα.

void omp destroy lock(omp lock t *lock);
void omp destroy nest lock(omp nest lock t *lock);
Απελευθερώνει τους πόρους που κατέλαβε η κλειδαριά lock/ nest lock αντίστοιχα.

void omp set lock(omp lock t *lock);
void omp set nest lock(omp nest lock t *lock);
Η συνάρτηση αυτή παγώνει την εκτέλεση του νήματος που καλεί την συνάρτηση αυτή και αυτό μέχρι η κλειδαριά να γίνει διαθέσιμή και έπειτα να μπορεί να την κλειδώσει.

void omp unset lock(omp lock t *lock);
void omp unset nest lock(omp nest lock t *lock);
Αφαιρεί την κλειδαριά από το νήμα.

void omp test lock(omp lock t *lock);
void omp test nest lock(omp nest lock t *lock);
Η συνάρτηση αυτή προσπαθεί να καταλάβει την κλειδαριά και να κλειδώσει χωρίς να παγώσει την εκτέλεση του νήματος.