Ατομική Διπλωματική Εργασία

ΒΙΒΛΙΟΘΗΚΗ ΠΑΡΑΛΛΗΛΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ ΓΙΑ ΤΗΝ ΓΛΩΣΣΑ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ JAVA ΚΑΙ ΕΦΑΡΜΟΓΗ ΑΥΤΟΜΑΤΗΣ ΜΕΤΑΤΡΟΠΗΣ ΣΕΙΡΙΑΚΟΥ ΚΩΔΙΚΑ JAVA ΣΕ ΠΑΡΑΛΛΗΛΟ

Χρίστος Κυριάκου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Βιβλιοθήκη παράλληλου προγραμματισμού για την γλώσσα προγραμματισμού Java και εφαρμογή αυτόματης μετατροπής σειριακού κώδικα Java σε παράλληλο

Χρίστος Κυριάκου

Επιβλέπων Καθηγητής
Ανδρέας Ανδρέου

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

Μάιος 2010

Ευχαριστίες

Θα ήθελα να ευχαριστήσω πάνω απ’ όλα το Θεο που μου έδωσε δύναμη να προχωρίσω και τους γονείς μου για την στήριξη και συμπαράσταση τους όλα τα χρόνια φοίτησης μου στο Τμήμα Πληροφορικής του Πανεπιστημίου Κύπρου. Ήταν πάντοτε δίπλα μου στην δύσκολη και επίπονη πορεία για την κατάκτηση αυτού του πτυχίου.

Θα ήθελα επίσης να ευχαριστήσω τον Δρ. Ανδρέα Ανδρέου και τον Δρ. Αναστάση Σοφοκλέους για την κατανόηση, καθοδήγηση και βοήθειά τους στις δυσκολίες που αντιμετώπισα για να φέρω εις πέρας αυτή την εργασία.

Περίληψη

Οδηγούμενοι από την ανάγκη δημιουργίας υψηλού επιπέδου επίδοσης για μια ποικιλία εφαρμογών που πρέπει να καλύπτουν τόσο τις ανάγκες των απλών καταναλωτών αλλά και των επιχειρήσεων και του επιστημονικού τομέα περάσαμε στη γενιά των παράλληλων υπολογιστών και του παράλληλου υπολογισμού. Ο παραλληλισμός είναι μια γνωστή έννοια που προκύπτει συχνά στη καθημερινή μας ζωή. Το σημαντικό όμως είναι ότι ο παραλληλισμός μπορεί να συμβάλει από πολλές απόψεις στη σταθερή βελτίωση της απόδοσης των υπολογιστών. Με βάση αυτό, νέες αρχιτεκτονικές και καινούργια προγραμματιστικά μοντέλα άρχισαν να δημιουργούνται ανοίγοντας έτσι καινούργιους ορίζοντες στο χώρο της πληροφορικής.

Παρουσιάστηκε η ανάγκη για νέους τρόπους προσέγγισης του παράλληλου υπολογισμού. Οι προσεγγίσεις αυτές είναι ο παράλληλος προγραμματισμός και ο αυτόματος παραλληλισμός. Κάποιος που θέλει να εμπλακεί με τον παράλληλο υπολογισμό εξετάζοντάς τις προσεγγίσεις αυτές με βάση κάποια κριτήρια είναι σε θέση να αποφασίσει πια ταιριάζει κάθε φορά στις ανάγκες του.

Ο στόχοι αυτής της διπλωματικής εργασίας είναι η δημιουργία μιας παράλληλης βιβλιοθήκης σε γλώσσα Java καθώς και ενός ήμι-αυτόματου μεταφραστή σειριακού κώδικα Java σε παράλληλο με βάση την προηγούμενη βιβλιοθήκη. Με τις δυο υλοποιήσεις αυτές θα είμαστε σε θέση να καλύψουμε και τους δυο τρόπους προσέγγισης του παράλληλου υπολογισμού, παράλληλο προγραμματισμό και ο αυτόματο παραλληλισμό. Κύριος στόχος όμως είναι να αφαιρέσουμε από τον προγραμματιστή όσο το δυνατόν περισσότερο την ανάγκη να ασχολείται με τις χαμηλού επιπέδου λεπτομέρειες για την δημιουργία μιας σωστής και αποδοτικής παράλληλης εφαρμογής

Σε γενικές γραμμές, θα προσδιορίσουμε το πλαίσιο του προβλήματος που οδήγησε κατά κάποιον τρόπο στην εποχή των παράλληλων υπολογιστών με πολλαπλούς πυρήνες και του παράλληλου υπολογισμού. Θα προσπαθήσουμε να εξηγήσουμε τις προσεγγίσεις και τα κριτήρια του παραλληλισμού αλλά και τα πραγματικά γεγονότα (πλεονεκτήματα – μειονεκτήματα / ευκολίες – δυσκολίες) που ισχύουν γύρω από τον

παράλληλο υπολογισμό σήμερα. Θα γίνει μια μικρή αναφορά γύρο από τα κυριότερα παράλληλα προγραμματιστικά μοντέλα αλλά και ολοκληρωμένες εφαρμογές αυτόματου παραλληλισμού που υπάρχουν και χρησιμοποιούνται σήμερα. Θα παρουσιάσουμε την όλη μεθοδολογία και προτεινόμενη λύση, καθώς και μερικές πειραματικές μετρήσεις/αποτελέσματα της επίδοσης διαφόρων προγραμμάτων που μπορούν να παραλληλιστούν με τα εργαλεία που έχουμε δημιουργήσει. Τελειώνοντας, θα μιλήσουμε για τα συμπεράσματα μας, καθώς και για τη μελλοντική εργασία που σίγουρα προκύπτει πάνω σε αυτού του είδους την μελέτη.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

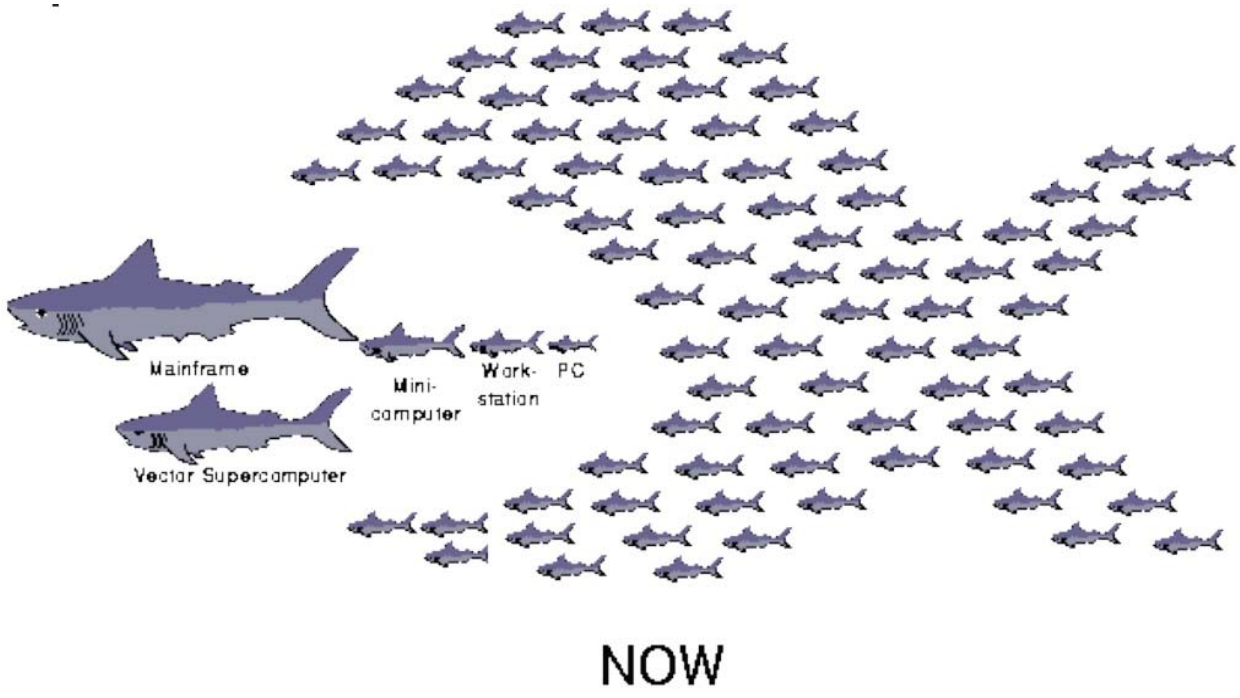
1.1 Κίνητρο

Ο παραλληλισμός είναι μια γνωστή έννοια που προκύπτει συχνά στη καθημερινή μας ζωή. Το σημαντικό όμως είναι ότι ο παραλληλισμός έχει συμβάλει από πολλές απόψεις στη σταθερή βελτίωση της απόδοσης των υπολογιστών κατά τη διάρκεια τουλάχιστον των προηγούμενων τριών δεκαετιών.

Σημαντικό όμως πρώτα είναι να διευκρινιστεί η διαφορά μεταξύ της σειριακής εκτέλεσης και της παράλληλης εκτέλεσης [5, 6, 7] όσον αφορά τον προγραμματισμό. Στο σειριακό υπολογισμό, για μια μονάδα χρόνου έχουμε εκτέλεση μόνο μιας διαδικασίας/εντολής (όπου σε εμάς είναι γνωστό για κάθε μονάδα του χρόνου ποιά εντολή είναι αυτή που εκτελείται) για τον πολύ απλό λόγο, στο τέλος της εκτέλεσης ολόκληρου του προγράμματος έχουμε την εγγύηση ότι το αποτέλεσμα είναι το αναμενόμενο. Στον παράλληλο υπολογισμό, για μια μονάδα χρόνου έχουμε εκτέλεση πολλαπλών διαδικασιών/εντολών (χωρίς εμείς να έχουμε την επιλογή να γνωρίζουμε ποιά εντολή εκτελείται πότε) θυσιάζοντας ίσως την ακρίβεια του αποτελέσματός μας για την επίδοση του συστήματος. Κατά συνέπεια, η ανάγκη το αποτέλεσμα στο

τέλος μιας παράλληλης εκτέλεσης να είναι το σωστό και το αναμενόμενο τροποποιεί τον τρόπο σκέψης και την προσέγγιση προγραμματισμού μας.

Οι καταναλωτές και οι εταιρίες λογισμικού δεν μπορούν να επιτύχουν τις περαιτέρω βελτιώσεις στην επίδοση των συστημάτων τους χωρίς τον παραλληλισμό. Οδηγούμενοι από την ανάγκη δημιουργίας υψηλού επιπέδου επίδοσης για μια ποικιλία εφαρμογών που πρέπει να καλύπτουν τόσο τις ανάγκες των απλών καταναλωτών αλλά και των επιχειρήσεων και του επιστημονικού τομέα, οι ερευνητές έχουν αναπτύξει παράλληλες αρχιτεκτονικές και μεταγλωττιστές βασισμένοι στις τεχνικές του παραλληλισμού εδώ και περισσότερο από τρεις δεκαετίες.



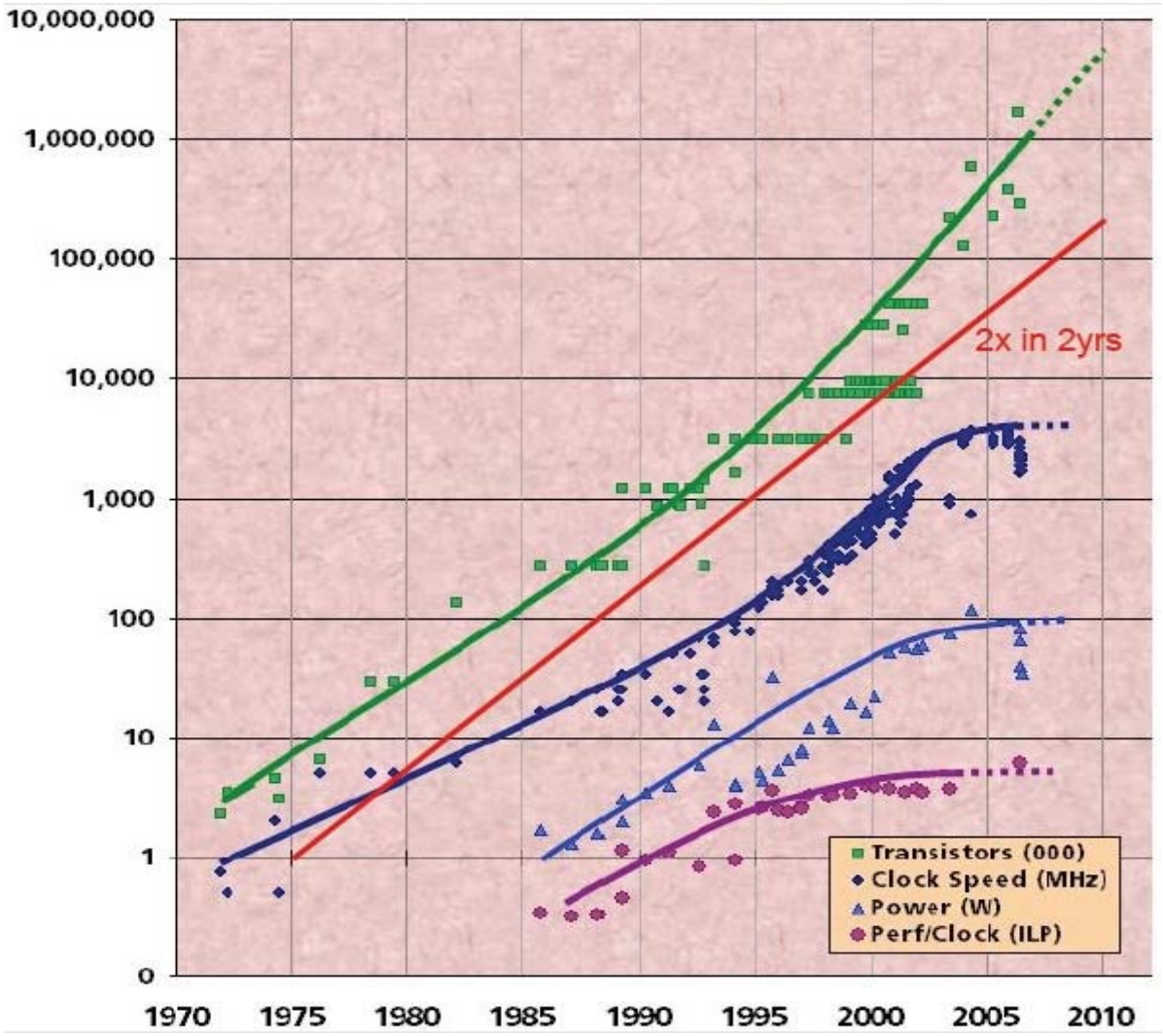
Εικόνα 1.1 : Οι υψηλής απόδοσης μικροεπεξεργαστές θα στηρίζονταν εφεξής στους πολλαπλούς επεξεργαστές ή πυρήνες. Αυτό ήταν η έναρξη για τη δημιουργία μιας νέας γενιάς επεξεργαστών, των πολυ-επεξεργαστών (multi-cores).

Η βιομηχανία υπολογιστών άλλαξε πορεία πλεύσης το 2005 όταν η Intel, ακολουθώντας τους πρωτοπόρους IBM με το Power 4 και Sun Microsystems με τον

επεξεργαστή Niagara, ανάγγειλε ότι οι υψηλής απόδοσης μικροεπεξεργαστές της θα στηρίζονταν εφεξής στους πολλαπλούς επεξεργαστές ή πυρήνες. Αυτό ήταν η έναρξη για τη δημιουργία μιας νέας γενιάς επεξεργαστών, των πολύ-επεξεργαστών (multi-cores) [Εικόνα 1.1]. Σε κάθε chip θα υπάρχουν 2, 4, 8 κοκ. επεξεργαστικές μονάδες/επεξεργαστές (cores) που θα μπορούν να εκτελούν παράλληλα ανεξάρτητες εργασίες/εντολές με αποτέλεσμα να αυξάνεται η υπολογιστική ισχύς χωρίς κατ' ανάγκη να χρειάζεται αύξηση της συχνότητας ρολογιού του επεξεργαστή. Ο στόχος του παράλληλου υπολογισμού είναι η αύξηση της επίδοσης μιας εφαρμογής με την εκτέλεσή της στους πολλαπλούς επεξεργαστές. Ενώ παραδοσιακά ο παράλληλος υπολογισμός έχει συνδεθεί με την κοινότητα του επιστημονικού τομέα για υψηλή υπολογιστική επίδοση [12] (HPC-High Performance Computing Community), με την παρουσία προϊόντων multi-core αρχιτεκτονικής γίνεται όλο και πιο διαδεδομένος για να καλύψει όλο το φάσμα χρηστών υπολογιστών.

Η multi-core αρχιτεκτονική, και σύντομα η many-core αρχιτεκτονική, είναι ένα νέο παράδειγμα που δείχνει ότι συνεχίζουμε να συμβαδίζουμε με το νόμο του Moore [Εικόνα 1.2] ο οποίος ισχύει απόλυτα και για τις πιο πάνω αρχιτεκτονικές επεξεργαστών. Ο νόμος του Moore [6,7] βασικά αναφέρει ότι ο αριθμός των transistors που μπορούν να τοποθετηθούν σε ένα ολοκληρωμένο κύκλωμα τείνει να διπλασιάζεται κάθε 2 χρόνια. Η τάση αυτή έχει συνεχιστεί για περισσότερο από μισό αιώνα και οι ερευνητές δεν ανάμεναν να σταματήσει τουλάχιστον μέχρι το 2015. Δεν υπάρχει όμως πλέον ούτε το κίνητρο αλλά ούτε και η δυνατότητα για περεταίρω αύξηση της συχνότητας του επεξεργαστή και αυτό λόγω του ότι φτάσαμε στο φυσικό όριο μεγέθους για τα transistors (άρα πρέπει να αλλάξουμε υλικό για να πετύχουμε μικρότερα μεγέθη), και η κατανάλωση ισχύος φτάνει σε μεγέθη που δημιουργούν σοβαρά προβλήματα εκπομπής θερμότητας.

Συνεπώς, αναμένεται ότι οι μελλοντικές γενιές των εφαρμογών θα πρέπει να εκμεταλλευτούν στο έπακρο τον παραλληλισμό που προσφέρεται από τη multi-core αρχιτεκτονική.



Εικόνα 1.2 : Γραφική αναπαράσταση του νόμου του Moore για τα τελευταία 40 χρόνια.

1.2 Σκοπός

Σκοπός αυτής της διπλωματικής εργασίας είναι να προταθεί μια μεθοδολογία με στόχο τη δημιουργία μιας παράλληλης βιβλιοθήκης σε γλώσσα Java καθώς και ενός ημι-αυτόματου μεταφραστή σειριακού κώδικα Java σε παράλληλο με βάση την προηγούμενη βιβλιοθήκη. Με τις δυο υλοποιήσεις αυτές θα είμαστε σε θέση να καλύψουμε και τους δυο τρόπους προσέγγισης του παράλληλου υπολογισμού, παράλληλο προγραμματισμό και ο αυτόματο παραλληλισμό. Κύριος στόχος όμως είναι να αφαιρέσουμε από τον προγραμματιστή όσο το δυνατόν περισσότερο την ανάγκη να

ασχολείται με τις χαμηλού επιπέδου λεπτομέρειες για τη δημιουργία μιας σωστής και αποδοτικής παράλληλης εφαρμογής

1.3 Ανασκόπηση Κεφαλαίων

Στο Κεφάλαιο 2 δίδεται το τεχνικό υπόβαθρο γύρω από το τι οδήγησε στον παράλληλο υπολογισμό και παραθέτουμε τις προσεγγίσεις και τα κριτήρια του παραλληλισμού όπως αυτά προέκυψαν μέσα από πολύχρονες έρευνες και μελέτες πάνω στο θέμα αυτό. Τέλος παρουσιάζουμε το πώς φαίνονται πραγματικά τα γεγονότα γύρω από τον παράλληλο υπολογισμό σήμερα με όλα τα θετικά και αρνητικά στοιχεία που προκύπτουν.

Στο Κεφάλαιο 3 παρουσιάζουμε περιληπτικά τα κυριότερα προγραμματιστικά μοντέλα που χρησιμοποιούνται ευρέως σήμερα διαχωρίζοντάς τα στις δυο κυριότερες αρχιτεκτονικές τους αφού η κάθε μια βασίζεται σε διαφορετικό μοντέλο μνήμης, την αρχιτεκτονική κοινής μνήμης (shared memory) και κατανεμημένης μνήμης (distributed memory) αντίστοιχα. Επιπλέον γίνεται μια σύντομη αναφορά στους βασικότερους παράλληλους μεταγλωττιστές που έχουν δημιουργηθεί μέχρι σήμερα παρουσιάζοντας τι ακριβώς προσφέρουν από πλευράς παραλληλισμού αλλά και ποιες γλώσσες προγραμματισμού υποστηρίζουν.

Στο Κεφάλαιο 4 παρουσιάζουμε την όλη μεθοδολογία και προτεινόμενη λύση. Περιγράφουμε τον τρόπο λειτουργίας της παράλληλης βιβλιοθήκης σε γλώσσα Java που έχει υλοποιηθεί και αναφερόμαστε λεπτομερώς στις κύριες μεθόδους από τις οποίες αποτελείται. Στην συνέχεια βλέπουμε πώς ακριβώς λειτουργεί η εφαρμογή του ήμι-αυτόματου μεταφραστή σειριακού κώδικα Java σε παράλληλο με βάση την προηγούμενη βιβλιοθήκη. Τέλος παρουσιάζουμε και αναλύουμε μερικές πειραματικές μετρήσεις/αποτελέσματα της επίδοσης διαφόρων προγραμμάτων που μπορούν να παραλληλιστούν με τα εργαλεία που έχουμε δημιουργήσει.

Στο κεφάλαιο 5 γίνεται μια ανακεφαλαίωση των όλων όσων είδαμε και παρουσιάσαμε μέσα σε αυτή την μελέτη και αναφέρονται κάποια γενικά συμπεράσματα από την όλη εργασία που έχει γίνει. Τέλος, δίδονται επίσης μερικές προτάσεις/εισηγήσεις για

μελλοντική εργασία πάνω στο θέμα της διπλωματικής εργασίας αυτής που σκοπό έχουν την περεταίρω ανάπτυξη, εμπλουτισμό και βελτιστοποίηση των εργαλείων που έχουμε αναπτύξει.

Κεφάλαιο 2

Τεχνικό Υπόβαθρο

2.1 Τι οδήγησε στον παράλληλο υπολογισμό

Η εξέλιξη της αρχιτεκτονικής μικροεπεξεργαστών εξαρτάται από τις μεταβαλλόμενες πτυχές της τεχνολογίας. Η αύξηση της πυκνότητας και της ταχύτητας των transistors ανά επεξεργαστή, η μνήμη και η συμπεριφορά προγράμματος γίνονται όλο και περισσότερο σημαντικά στην βιομηχανία αρχιτεκτονικής Η/Υ. Ενώ η τεχνολογία επιτρέπει περισσότερο σύνθετες σχεδιάσεις επεξεργαστών, υπάρχουν φυσικά και προγραμματιστικά όρια στη χρησιμότητα αυτής της πολυπλοκότητας. Τα φυσικά όρια περιλαμβάνουν τα όρια συσκευών/επεξεργαστών καθώς επίσης και τα πρακτικά όρια που αφορούν τη δύναμη και το κόστος. Τα όρια συμπεριφοράς προγράμματος προκύπτουν από απρόβλεπτα γεγονότα που εμφανίζονται κατά τη διάρκεια μιας εκτέλεσης. Οι αρχιτεκτονικές και οι εφαρμογές που επεκτείνουν αυτά τα όρια είναι ζωτικής σημασίας στη συνεχή εξέλιξη των επεξεργαστών.

Στον πίνακα που ακολουθεί [Πίνακας 2.1] παρατίθενται όλα αυτά τα όρια/περιορισμοί και κατευθυντήριες αρχές που επεξηγούν ακριβώς πώς όλα αλλάζουν στον υπολογισμό σε σχέση με το τι ίσχυε στο παρελθόν και το τι ισχύει σήμερα [17].

#	Παρελθόν	Σήμερα	Συμπέρασμα
1	Η ενέργεια είναι φτηνή αλλά τα transistors είναι ακριβά	Η ενέργεια είναι "ακριβή" αλλά τα transistors είναι πολύ φτηνά	Τείχος ενέργειας – Power Wall : μπορούμε να βάλουμε περισσότερα transistors σε ένα chip σε σχέση με το πόση ενέργεια μπορούμε να δώσουμε για να τα ενεργοποιήσουμε
2	Εάν ανησυχείς για την ενέργεια τότε πρέπει να λάβεις υπόψη σου μόνο την δυναμική ενέργεια (μέση ενέργεια που καταναλώνεται ανά κύκλο του ρολογιού)	Εάν ανησυχείς για την ενέργεια τότε πρέπει να λάβεις σοβαρά υπόψη σου και την στατική ενέργεια (μέση ενέργεια που χρειάζεται για να είναι ενεργοποιημένα τα transistors)	Τείχος ενέργειας – Power Wall : στα desktop και server συστήματα η απώλεια μόνο της στατικής ενέργειας υπολογίζεται στο 40% της συνολικής ενέργειας που καταναλώνει ολόκληρο το σύστημα
3	Τα μονολιθικά Chip επεξεργαστών από σιλικόνη είναι αξιόπιστα εσωτερικά με σφάλματα να παρουσιάζονται μόνο στα pins	Η τεχνολογία ολοκλήρωσης των chip έχει ξεπεράσει τα 65nm με αποτέλεσμα ο μέσος ρυθμός εμφάνισης σφαλμάτων να αυξάνεται τόσο στο υλικό όσο και στο λογισμικό	Τόσο η Intel όσο και η nVidia τα τελευταία χρόνια αναγκάστηκαν να ξοδέψουν εκατοντάδες εκατομμύρια η κάθε μια για αντικατάσταση ελαττωματικών chips που κατασκεύασαν

Πίνακας 2.1 : Όρια-περιορισμοί από το παρελθόν στο παρόν.

4	Μπορούμε να βελτιώσουμε το επίπεδο αφαιρετικότητας και σαν αποτέλεσμα και το μέγεθος σχεδίου του υλικού χωρίς να έχουμε ιδιαίτερη αύξηση στην πολυπλοκότητα της κατασκευής	Καθυστέρηση καλωδίων, θόρυβος, διαγώνια σύζευξη (χωρητική και επαγωγική), μεταβλητότητα κατασκευής, αξιοπιστία (βλ. #3), παραμόρφωση χρονισμού του ρολογιού, εκπομπή θερμότητας	Τείχος πολυπλοκότητας – Complexity Wall : Αυξάνεται δραματικά ο χρόνος επικύρωσης των σχεδίων για chip που ολοκληρώνονται σε πολύ μικρά μεγέθη καθώς επίσης και το κόστος ελέγχου τους
5	Οι ερευνητές παρουσιάζουν τις νέες ιδέες αρχιτεκτονικής με την δημιουργία πρωτοτύπων των chip	Μεγάλο κόστος σχεδίων για ολοκληρώματα μεγέθους 65nm και κάτω καθώς επίσης και των σχεδιαστικών λογισμικών που πρέπει να χρησιμοποιηθούν	Τείχος πολυπλοκότητας – Complexity Wall : οι ερευνητές δεν μπορούν να πιστεύουν πλέον στη δημιουργία πρωτοτύπων και κατά συνέπεια, μια εναλλακτική προσέγγιση στην αξιολόγηση των αρχιτεκτονικών πρέπει να αναπτυχθεί
6	Οι βελτιώσεις απόδοσης έχουν σαν αποτέλεσμα μικρότερο εύρος λαθών και μεγαλύτερο εύρος ζώνης μετάδοσης (bandwidth)	Σε πολλές τεχνολογίες, το εύρος ζώνης μετάδοσης (bandwidth) βελτιώνεται τουλάχιστον στο τετράγωνο της βελτίωσης στο εύρος λαθών	Όπως και πιο πριν (βλ. #4) ο χρόνος/κόστος επικύρωσης/ ελέγχου αυξάνεται δραματικά

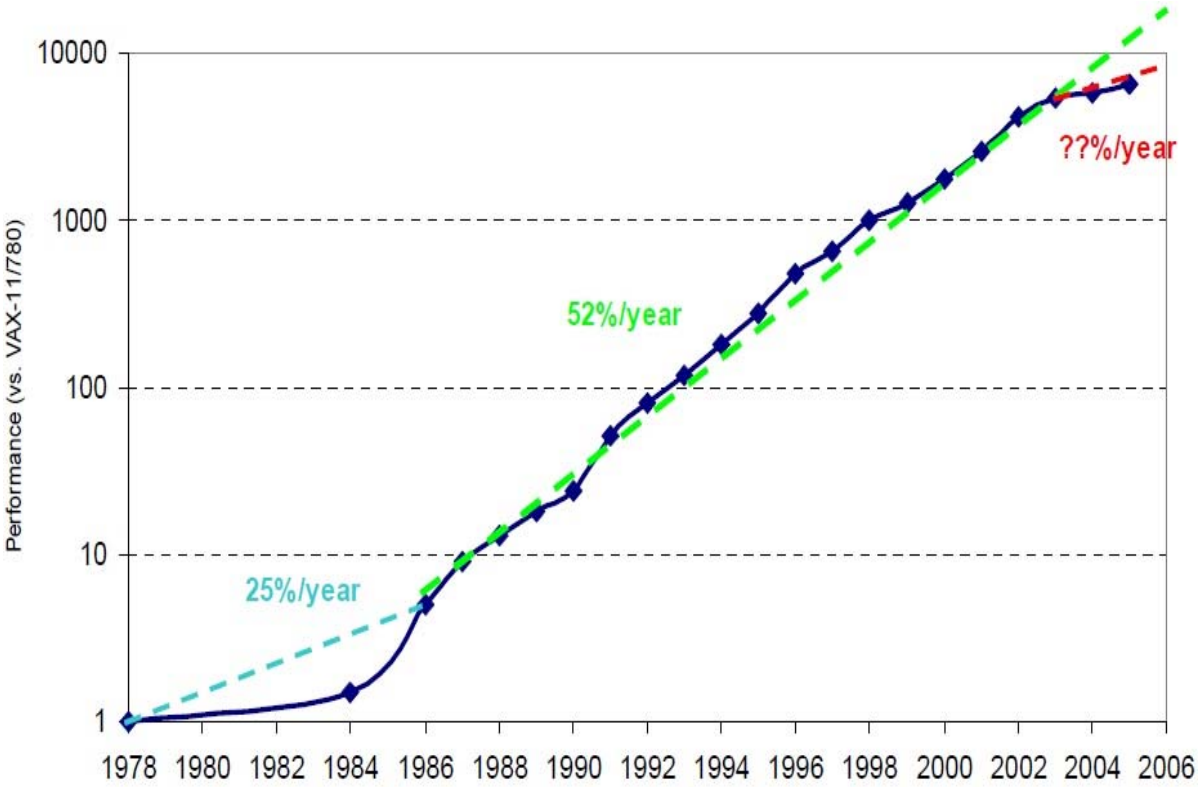
Πίνακας 2.1 : Όρια-περιορισμοί από το παρελθόν στο παρόν (συνέχεια).

7	Οι πράξεις πολλαπλασιασμού και διαίρεσης έχουν μεγαλύτερο χρόνο εκτέλεσης απ' ότι η ανάκτηση και αποθήκευση δεδομένων	Οι πράξεις πολλαπλασιασμού και διαίρεσης έχουν πολύ μικρότερο χρόνο εκτέλεσης απ' ότι η ανάκτηση και αποθήκευση δεδομένων	Τείχος μνήμης – Memory Wall : οι σύγχρονοι μικροεπεξεργαστές μπορεί να χρειαστούν μέχρι και 200 κύκλους ρολογιού για να έχουν πρόσβαση στην δυναμική μνήμη τυχαίας προσπέλασης (DRAM) αλλά για πράξεις κινητής υποδιαστολής 5 κύκλοι αρκούν
8	Μπορούμε να πάρουμε περισσότερο παραλληλισμό των προς εκτέλεση εντολών μέσω μεταγλωττιστών και καινοτομιών στην αρχιτεκτονική του συστήματος	Οι μεταγλωττιστές έχουν φτάσει σχεδόν τα όρια τους όσον αφορά παραλληλισμό των προς εκτέλεση εντολών (branch prediction, out-of-order execution, speculation)	Τείχος παραλληλισμού των εντολών – ILP Wall (Instruction level parallelism) : καινοτομίες στην αρχιτεκτονική του συστήματος δημιουργούν περαιτέρω προβλήματα καθυστερήσεων λόγω πολυπλοκότητας (βλ. #6)
9	Οι επιδώσεις των μονοπύρηνων επεξεργαστών διπλασιάζονται (2x) κάθε 18 μήνες	Μετά το 2006 η αύξηση της επίδοσης των επεξεργαστών είναι της τάξης του 0.3 (0.3x) για κάθε 18 μήνες [Εικόνα 2.1]	Power Wall + Complexity Wall + Memory Wall + ILP Wall = Brick Wall : ο συνδυασμός όλων των πιο πάνω έφερε την επίδοση των επεξεργαστών στα όριά της, ο διπλασιασμός της επίδοσης των μονοπύρηνων επεξεργαστών μπορεί να πάρει τώρα 5 χρόνια

Πίνακας 2.1 : Όρια-περιορισμοί από το παρελθόν στο παρόν (συνέχεια).

10	Γιατί να σκοτίζεσαι να παραλληλίσεις την εφαρμογή σου αφού μπορείς να περιμένεις λίγο ακόμα για να την τρέξεις σε έναν πιο γρήγορο σειριακό επεξεργαστή	Θα πρέπει να περιμένεις αρκετό καιρό μέχρι να βγει ένας πιο γρήγορος σειριακός επεξεργαστής που θα καλύπτει της ανάγκες επίδοσης της εφαρμογής σου (βλ. #9)	Αν και τα πιο πάνω δημιουργούν μια αρνητική εικόνα για το χώρο της πληροφορικής το απόλυτο θετικό είναι ότι σύντομα θα έχουμε μέσα σε ένα chip χιλιάδες απλούς πυρήνες (many-cores). Ο παραλληλισμός έτσι θα αρχίσει να γίνεται αναπόσπαστο κομμάτι κάθε προγραμματιστή ή εταιρίας ανάπτυξης λογισμικού γιατί μόνο μέσω αυτού θα έχουμε άμεσα αποτελέσματα βελτίωσης του χρόνου εκτέλεσης των εφαρμογών
11	Η αύξηση της συχνότητας του ρολογιού είναι η κύρια μέθοδος για βελτίωση της επίδοσης του επεξεργαστή	Η αύξηση του παραλληλισμού είναι η κύρια μέθοδος για βελτίωση της επίδοσης του συστήματος	
12	Οτιδήποτε λιγότερο από τη γραμμική αύξηση της επίδοσης μιας εφαρμογής θεωρείται αποτυχία	Οποιαδήποτε αύξηση της επίδοσης μιας εφαρμογής μέσω παραλληλισμού της θεωρείται επιτυχία	

Πίνακας 2.1 : Όρια-περιορισμοί από το παρελθόν στο παρόν (συνέχεια).



Εικόνα 2.1 : Βελτίωση της επίδοσης των επεξεργαστών μεταξύ 1978 και 2006.

Η τεχνολογία RISC (Reduced Instruction Set Computer) [18] έδωσε τεράστια ώθηση στην αύξηση της επίδοσης, της τάξης του 52% ανά χρόνο, στο διάστημα μεταξύ 1978 και 2002. Μετά το 2002 η αύξηση της επίδοσης των επεξεργαστών ήταν της τάξης του 20% ανά χρόνο. Από το 2002 μέχρι το 2006 η αύξηση της επίδοσης των επεξεργαστών ήταν τρεις φορές πιο αργή σε σχέση με το 52% του προηγούμενου διαστήματος (1978 - 2002).

Όλα όσα είδαμε πιο πάνω δημιουργούν ένα μαύρο πέπλο γύρω από το χώρο της Πληροφορικής, όμως παρόλα αυτά αρχίζουν να διαφαίνονται και κάποια θετικά στοιχεία. Πρώτον, ο νόμος του Moore συνεχίζει να ισχύει και έτσι σύντομα θα είμαστε σε θέση να δούμε επεξεργαστές όπου σε ένα chip θα υπάρχουν μέχρι και εκατοντάδες απλοί, οικονομικοί πυρήνες (many-cores). Για παράδειγμα η CISCO ετοιμάζεται να βγάλει ένα επεξεργαστή όπου σε ένα chip θα υπάρχουν 188 cores/επεξεργαστές RISC και για το chip θα χρησιμοποιηθεί τεχνολογία ολοκλήρωσης 130nm. Δεύτερον, η

επικοινωνία μεταξύ των cores στο ίδιο chip θα έχει πολύ μικρές καθυστερήσεις και αρκετά μεγάλο εύρος ζώνης (bandwidth). Με βάση αυτό, νέες αρχιτεκτονικές και καινούργια προγραμματιστικά μοντέλα θα πρέπει να δημιουργηθούν και έτσι καινούργιοι ορίζοντες ανοίγουν στο χώρο της έρευνας. Τρίτο, το κίνημα που δημιουργήθηκε με το λογισμικό ανοικτού κώδικα και τις διαστάσεις που έχει πάρει θα δώσει περεταίρω ώθηση στην εξέλιξη του λογισμικού σε σχέση με το παρελθόν, αφού τώρα θα έχει και σύμμαχό του τον παραλληλισμό, κάτι το καινούργιο που όλοι πρέπει να εφαρμόσουν.

2.2 Προσεγγίσεις και κριτήρια του παραλληλισμού

Υπάρχουν δύο κύριες προσεγγίσεις για να παραλληλίσουμε τις εφαρμογές [14,17]: ο παράλληλος προγραμματισμός και ο αυτόματος παραλληλισμός τα οποία διαφέρουν από την άποψη του αν θέλει κάποιος να πετύχει το μέγιστο βαθμό επίδοσης (maximum speedup) της εφαρμογής του ή αν θέλει να ευνοηθεί της ευκολίας του παραλληλισμού αντίστοιχα.

Η προσέγγιση του αυτόματου παραλληλισμού, π.χ. ILP (Instruction Level Parallelism - παραλληλισμός σε επίπεδο εντολών) ή οι παράλληλοι μεταγλωττιστές, παραλληλίζουν αυτόματα τις εφαρμογές οι οποίες έχουν αναπτυχθεί χρησιμοποιώντας σειριακά πρότυπα προγραμματισμού. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι οι ήδη υπάρχουσες εφαρμογές δεν χρειάζονται να τροποποιηθούν, π.χ. οι εφαρμογές πρέπει απλά να μεταφραστούν με έναν παράλληλο μεταγλωττιστή. Επομένως, οι προγραμματιστές δεν χρειάζεται να μάθουν τα νέα πρότυπα του παράλληλου προγραμματισμού. Εντούτοις, αυτό γίνεται και ένας περιοριστικός παράγοντας στην εκμετάλλευση ενός υψηλότερου βαθμού παραλληλισμού: είναι μια εξαιρετικά μεγάλη πρόκληση ο αυτόματος μετασχηματισμός των αλγόριθμων από τη σειριακή τους φύση στην παράλληλη.

Σε αντίθεση με τον αυτόματο παραλληλισμό, στην παράλληλη προσέγγιση προγραμματισμού, οι εφαρμογές αναπτύσσονται συγκεκριμένα για να εκμεταλλευτούν όσο το δυνατόν σε μεγαλύτερο βαθμό τον παραλληλισμό με σκοπό να φτάσουν την μέγιστη απόδοση που μπορούν να πετύχουν στη συγκεκριμένη αρχιτεκτονική που θα

τρέξουν. Γενικά, η ανάπτυξη μιας παράλληλης εφαρμογής περιλαμβάνει καταμερισμό του φόρτου εργασίας σε τμήματα και ανάθεση των τμημάτων αυτών προς εκτέλεση. Γίνεται αντιληπτό ότι ο παράλληλος προγραμματισμός οδηγεί σε αποτελέσματα υψηλότερης επίδοσης σε σχέση με τον αυτόματο παραλληλισμό, αλλά με την προϋπόθεση ότι ο προγραμματιστής θα πρέπει να καταβάλει περισσότερο κόπο και προσπάθεια για να πετύχει το στόχο του.

Τόσο στον αυτόματο παραλληλισμό όσο και στην παράλληλη προσέγγιση προγραμματισμού η διαδικασία που πρέπει να ακολουθηθεί για να παραλληλιστεί μια εφαρμογή είναι κοινή και για τις δυο περιπτώσεις. Σε γενικές γραμμές η όλη διαδικασία μπορεί να χωριστεί σε 4 βήματα [Εικόνα 2.1] που είναι τα εξής:

- **Διάσπαση – Decomposition**

  Προσδιορισμός του τι μπορεί να εκτελεστεί παράλληλα και διάσπαση του σειριακού προγράμματος/υπολογισμού σε μικρότερες εργασίες/στόχους

- **Ανάθεση – Assignment**

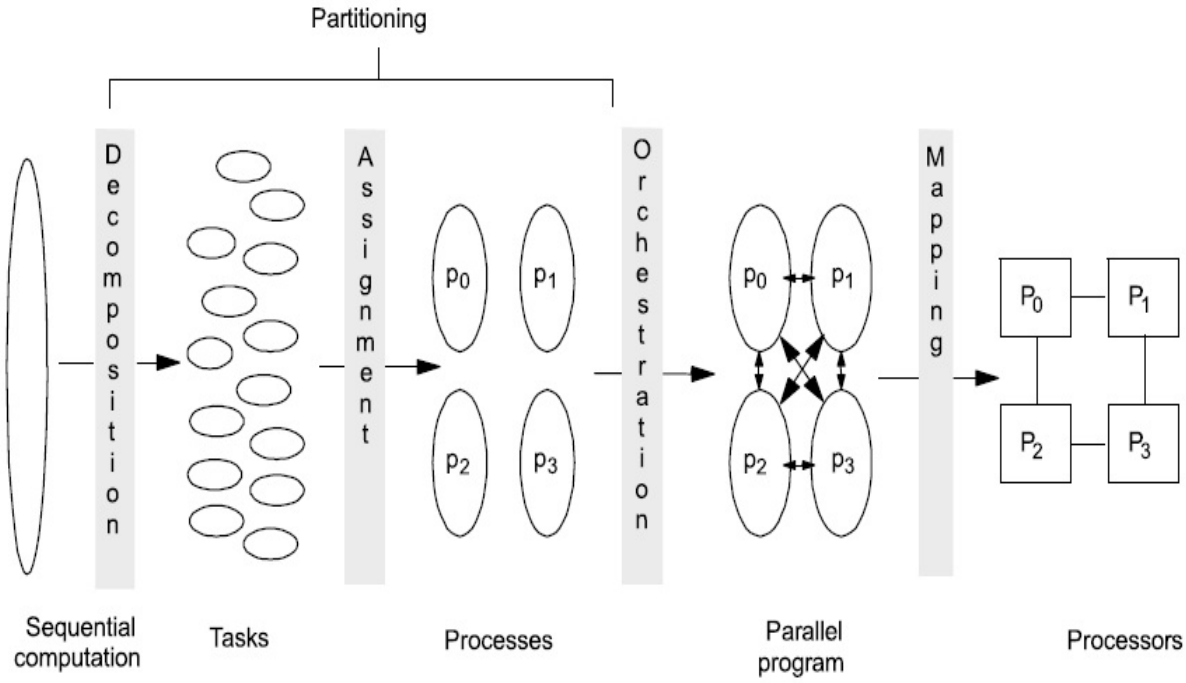
  Καταμερισμός των εργασιών/στόχων που δημιουργήθηκαν από το προηγούμενο βήμα στις διάφορες μονάδες εκτέλεσης

- **Ενορχήστρωση -Orchestration**

  Διαχείριση της πρόσβασης στα δεδομένα, της επικοινωνίας και του συγχρονισμού μεταξύ των μονάδων εργασίας

- **Δέσμευση – Mapping**

  Δέσμευση των διαδικασιών και των μονάδων εκτέλεσης στους διαθέσιμους επεξεργαστές



Εικόνα 2.1 : Τα τέσσερα στάδια δημιουργίας μιας παράλληλης εφαρμογής.

Καθ' όλη τη διάρκεια των ετών, έχει υπάρξει ένας μεγάλος αριθμός προτεινόμενων προτύπων παράλληλου προγραμματισμού. Ένας χαρακτηριστικός τρόπος εκτίμησης για την επιλογή ενός προτύπου είναι η επίδοση που προκύπτει από τη βελτιωμένη από το πρότυπο εφαρμογή. Εντούτοις, είναι εξίσου σημαντικό να εξεταστούν και οι ποιοτικές πτυχές των προτύπων. Μια τέτοια ποιοτική πτυχή είναι το επίπεδο αφαιρετικότητας του παραλληλισμού και πώς αυτός παρουσιάζεται στους υπεύθυνους για την ανάπτυξη της εφαρμογής. Για την αξιολόγηση αυτής της πτυχής προτείνεται όπως και κάθε πρότυπο τυγχάνει αξιολόγησης βασισμένης σε επτά κριτήρια [H. Kasim et. al.] [17,22,25]. Πιο κάτω ακολουθεί μια σύντομη περιγραφή των κριτηρίων αυτών:

i. Αρχιτεκτονική του συστήματος

Εξετάζουμε δύο αρχιτεκτονικές: της κοινής μνήμης (shared memory) [14,7,6] και της κατανεμημένης μνήμης (distributed memory) [14,7,6]. Η αρχιτεκτονική κοινής μνήμης αναφέρεται στα συστήματα ως ένας κόμβος SMP/MPP (Symmetric Multi-Processing/Massively Parallel Processing) με τον οποίο όλοι οι επεξεργαστές μοιράζονται ένα ενιαίο διάστημα διευθύνσεων για να μπορούν να επικοινωνούν μεταξύ τους. Με τέτοια πρότυπα, οι εφαρμογές μπορούν να

τρέξουν και να χρησιμοποιήσουν μόνο τους επεξεργαστές μέσα σε έναν ενιαίο κόμβο. Από την άλλη, η κατανεμημένη αρχιτεκτονική μνήμης αναφέρεται στα συστήματα ως μια συστάδα από υπολογιστικούς κόμβους SMP/MPP όπου υπάρχει ένα μοναδικό διάστημα διευθύνσεων ανά κόμβο και οι κόμβοι επικοινωνούν μεταξύ τους συνήθως μέσω μηνυμάτων.

ii. Μεθοδολογίες προγραμματισμού

Εξετάζουμε το πώς οι ευκαιρίες παραλληλισμού εκτίθενται στους προγραμματιστές. Παραδείγματα αποτελούν η διεπαφή προγραμματισμού εφαρμογής (API), οι πρόσθετες οδηγίες (special directives), οι νέες γλωσσικές προδιαγραφές τις οποίες πρέπει να μάθει ο προγραμματιστής, κλπ.

iii. Διαχείριση μονάδων εκτέλεσης

Αυτό το κριτήριο εξετάζει τη δημιουργία μονάδας εκτέλεσης ενός συνόλου εντολών, των νημάτων (threads) ή των διεργασιών (processes). Η διαχείριση αυτών των μονάδων υπονοείται όταν δεν υπάρχει καμία ανάγκη για τους προγραμματιστές να διαχειριστούν τη διάρκεια ζωής των διεργασιών. Συνήθως, πρέπει να διευκρινίσουν μόνο τον αριθμό των μονάδων που απαιτούνται ή το τμήμα του κώδικα για να μπορούν να οργανωθούν παράλληλα οι διεργασίες. Στη λεπτομερή προσέγγιση όμως, ο προγραμματιστής πρέπει να κωδικοποιήσει τη δημιουργία και καταστροφή των μονάδων αυτών καθώς και το τμήμα του κώδικα που πρέπει να εκτελέσουν.

iv. Διαχωρισμός φόρτου εργασίας στις μονάδες

Ο διαχωρισμός σε μονάδες εκτέλεσης καθορίζει πώς ο φόρτος εργασίας διαιρείται σε μικρότερα κομμάτια τα οποία καλούνται στόχοι. Όταν ο διαχωρισμός του φόρτου εργασίας υπονοείται, οι προγραμματιστές πρέπει μόνο να διευκρινίσουν αν ολόκληρο το φόρτο εργασίας που μπορεί να τύχει παράλληλης επεξεργασίας, στον οποίο δεν πρέπει να υπάρχουν εξαρτήσεις για κανένα σημείο του κώδικα προς εκτέλεση (no dependencies). Το πώς όμως ο φόρτος εργασίας διαχωρίζεται πραγματικά σε στόχους δεν χρειάζεται να ρυθμιστεί με λεπτομέρεια από τους προγραμματιστές. Αντίθετα, στη λεπτομερή

προσέγγιση, υπάρχει η ανάγκη οι προγραμματιστές να αποφασίσουν (με το χέρι – manually) πώς ο φόρτος εργασίας μοιράζεται.

v. Διαμοιρασμός στόχων στις μονάδες εκτέλεσης

Ο διαμοιρασμός στόχων στις μονάδες εκτέλεσης καθορίζει πώς οι στόχοι κατανέμονται στις διάφορες μονάδες εκτέλεσης. Όταν ο διαμοιρασμός στόχων στις μονάδες εκτέλεσης υπονοείται, οι προγραμματιστές δε χρειάζεται να προσδιορίσουν ποιά μονάδα εκτέλεσης είναι υπεύθυνη για κάποιον συγκεκριμένο στόχο. Η λεπτομερής προσέγγιση όμως, απαιτεί όπως ο προγραμματιστής είναι υπεύθυνος να διαχειριστεί το πώς οι στόχοι ορίζονται ανά μονάδα εκτέλεσης.

vi. Συγχρονισμός

Ο συγχρονισμός καθορίζει τη χρονική ιεράρχηση με την οποία οι μονάδες εκτέλεσης έχουν πρόσβαση στα κοινά στοιχεία στη μνήμη. Ο συγχρονισμός υπονοείται όταν καταβάλλεται πολύ μικρή προσπάθεια από τους προγραμματιστές: είτε κανένας συγχρονισμός δεν απαιτείται, λόγω του ότι κάθε μια μονάδα εκτέλεσης έχει πρόσβαση μόνο σε δικές της μεταβλητές, είτε είναι ικανοποιητικό μόνο να διευκρινιστεί ότι ένας συγχρονισμός απαιτείται. Στο ρητό συγχρονισμό, οι προγραμματιστές πρέπει να διαχειριστούν με λεπτομέρεια την πρόσβαση της κάθε μονάδας εκτέλεσης σε κοινές μεταβλητές και κοινά τμήματα μνήμης.

vii. Μοντέλο επικοινωνίας

Το μοντέλο επικοινωνίας καθορίζει τον τρόπο με τον οποίο οι κόμβοι και οι επεξεργαστές που τους αποτελούν μπορούν να επικοινωνούν μεταξύ τους ανταλλάζοντας δεδομένα και πληροφορίες. Τα πιο διαδεδομένα μοντέλα επικοινωνίας είναι μέσω της κοινής μνήμης (shared memory) και της ανταλλαγής μηνυμάτων (message passing)

2.3 Τα πραγματικά γεγονότα στον παραλληλισμό σήμερα

Ο καθένας μας έχει διδαχτεί κάποτε τον νόμο του Amdahl [5, 10, 19], αλλά όλοι μας σύντομα τον ξεχνούμε...

"Everyone is taught Amdahl's Law in school, but they quickly forget it"
T. R. Puzak, IBM, 2007

Ο νόμος του Amdahl χρησιμοποιείται για να βρεθεί η μέγιστη αναμενόμενη βελτίωση σε ενός συστήματος όταν βελτιώνεται μόνο κάποιο μέρος του. Χρησιμοποιείται συχνά στον παράλληλο υπολογισμό για να προβλέψει το θεωρητικό μέγιστο speedup (βελτίωση του χρόνου εκτέλεσης σε σχέση με τον σειριακό) για ένα παράλληλο σύστημα με πολλαπλούς επεξεργαστές. Το speedup ορίζεται ως ο λόγος του χρόνου εκτέλεσης του σειριακού προγράμματος προς το χρόνο εκτέλεσης του παράλληλου προγράμματος. Ο νόμος του Amdahl για τα παράλληλα συστήματα έχει ως εξής:

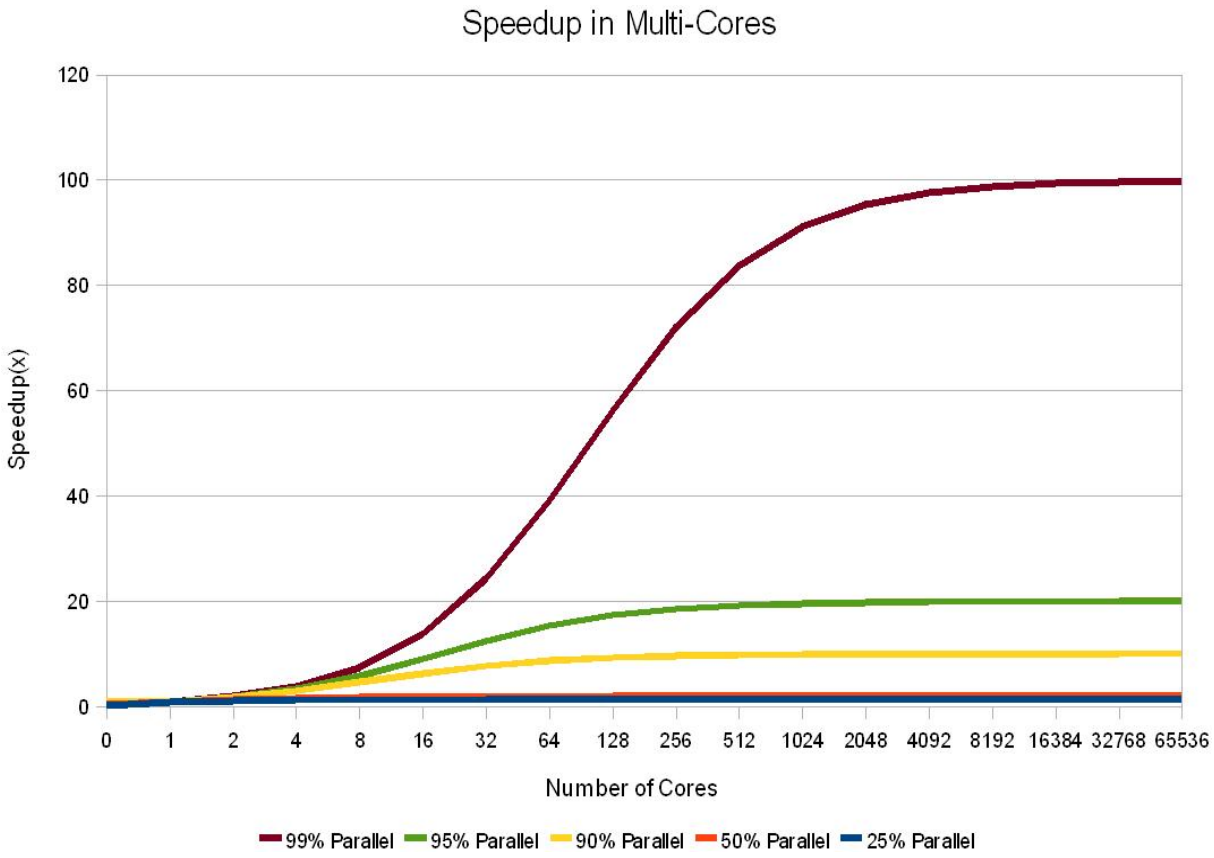
$$\text{speedup} = \frac{1}{S + \frac{(1 - S)}{N}}$$

Εικόνα 2.2 : Ο ορισμός του speedup.

όπου το S υποδηλώνει το ποσοστό του προγράμματος που είναι σειριακό, το 1-S υποδηλώνει το ποσοστό του προγράμματος που είναι παράλληλο και το N υποδηλώνει τον αριθμό των επεξεργαστών που έχει το σύστημα. Αν το S είναι μηδέν, δηλαδή η εφαρμογή δεν έχει καθόλου σειριακό κομμάτι μέσα τότε το speedup θεωρείται ότι είναι άπειρο. Αν η τιμή του speedup είναι μικρότερη από 1 τότε έχουμε αρνητική βελτίωση (speedup-down). Όπως διαφαίνεται όμως από τον πιο πάνω τύπο, το speedup που μπορούμε να πάρουμε από μια εφαρμογή που χρησιμοποιεί πολλαπλούς επεξεργαστές σε έναν παράλληλο υπολογισμό περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει. [Εικόνα 2.3]

Το speedup που μπορούμε να πάρουμε από μια εφαρμογή που χρησιμοποιεί πολλαπλούς επεξεργαστές σε έναν παράλληλο υπολογισμό περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει. Για παράδειγμα αν η εφαρμογή μας

παραλληλίζεται σε ένα ποσοστό του 95% τότε το θεωρητικό μέγιστο speedup που μπορεί να έχει η εφαρμογή μας για ένα παράλληλο σύστημα με πολλαπλούς επεξεργαστές δεν μπορεί να ξεπερνά το 20x, ασχέτως πόσοι επεξεργαστές χρησιμοποιούνται. Αντίστοιχα αν η εφαρμογή μας παραλληλίζεται σε ποσοστό 99% τότε το θεωρητικό μέγιστο speedup που μπορεί να έχει δεν μπορεί να ξεπερνά το 100x, ασχέτως πόσοι επεξεργαστές χρησιμοποιούνται.



Εικόνα 2.3 : Γραφική παράσταση που απεικονίζει το speedup που μπορούμε να πάρουμε από μια εφαρμογή που χρησιμοποιεί πολλαπλούς επεξεργαστές.

Η βάση για την ανίχνευση του παραλληλισμού είναι η ανάλυση εξαρτήσεων (dependence analysis). Τα αποτελέσματα της ανάλυσης επιτρέπουν τόσο στο μεταγλωττιστή όσο και στον προγραμματιστή να προσδιορίσουν τα τμήματα κώδικα που μπορούν να εκτελεσθούν παράλληλα. Κατά συνέπεια, πολλή προσπάθεια έχει γίνει για την ανάπτυξη αλγορίθμων για την ανάλυση και έλεγχο των εξαρτήσεων με βάση την ακρίβεια του αποτελέσματος και της πολυπλοκότητας της εφαρμογής, που

χρησιμοποιείται σαν είσοδος για να αναλυθεί. Κανένας όμως αλγόριθμος μέχρι σήμερα δεν είναι αρκετά αποδοτικός λόγο του ότι έχουν να αντιμετωπίσουν αρκετά πολύπλοκες εφαρμογές οι οποίες δύναται να τρέχουν και σε αρκετά πολύπλοκες αρχιτεκτονικές. Οι αναλύσεις εξαρτήσεων χρησιμοποιούνται ευρέως από τους μεταγλωττιστές αλλά είναι συντηρητικής φύσεως, δηλ., εάν σε μια ανάλυση κάποια δοκιμή δεν μπορεί να αποδείξει την απουσία μιας εξάρτησης τότε θεωρείται ότι υπάρχει εξάρτηση. Δεδομένου ότι ο αυτόματος παραλληλισμός έχει εστιάσει κυρίως στις επιστημονικές εφαρμογές, το μεγαλύτερο μέρος της εργασίας για ανάλυση εξαρτήσεων εξετάζει προγράμματα που περιλαμβάνουν αναφορές σε πίνακες μέσα στους βρόχους. Τα αποτελέσματα των δοκιμών για ανάλυση εξαρτήσεων για τα μη-επιστημονικά προγράμματα που περιλαμβάνουν εντατικούς υπολογισμούς και εκτενή χρήση δεικτών δεν δείχνουν καθόλου ενθαρρυντικά.

Τα πιο πάνω όμως δεν αποτελούν τους μόνους περιορισμούς για την ανάπτυξη μιας παράλληλης εφαρμογής ή τη μετατροπή μιας ήδη υφιστάμενης σειριακής εφαρμογής σε παράλληλη. Εκτός αυτών τόσο οι προγραμματιστές όσο και οι μεταγλωττιστές αυτόματης παραλληλοποίησης θα πρέπει να είναι σε θέση να αντεπεξέλθουν και μια σειρά άλλων προβλημάτων που προκύπτουν μέσα σε μια παράλληλη εκτέλεση. Τα υπάρχοντα λειτουργικά συστήματα δεν κάνουν σωστή διαχείριση εργασιών (χρονοδρομολόγηση) με τρόπο που να ευνοούν την παράλληλη εκτέλεση. Στην περίπτωση όμως που το κάνουν, τότε πρέπει να αντιμετωπίσουν συγκρούσεις όταν απαιτείται ταυτόχρονη χρήση κοινών πόρων, με κυριότερο παράδειγμα αυτό της ταυτόχρονης εγγραφής στη μνήμη. Ο χρόνος ζωής των εφαρμογών και των λειτουργικών συστημάτων (software) υπολογίζεται σε δεκάδες χρόνια σε αντίθεση με τον χρόνο ζωής του υλικού (hardware) που υπολογίζεται μόλις το πολύ σε μερικά χρόνια, αν όχι μήνες. Στις σύγχρονες αρχιτεκτονικές multi-core που δημιουργήθηκαν για να υποστηρίζουν την παράλληλη εκτέλεση, αν και οι επεξεργαστές-πυρήνες βρίσκονται στο ίδιο chip, η επικοινωνία μεταξύ τούς ακόμα θεωρείται ακριβή. Ακόμα οι προγραμματιστές και οι αυτόματοι μεταφραστές πρέπει να είναι σε θέση να υπολογίσουν με ακρίβεια τον αριθμό των νημάτων (threads) και διεργασιών (process) που θα τρέχουν σε μια παράλληλη εφαρμογή καθώς ο χρόνος δημιουργίας και καταστροφής τους δημιουργεί επιπλέον καθυστερήσεις. Σημαντικό παράγοντα παίζει και ο συγχρονισμός των νημάτων (threads) και διεργασιών (processes) που τρέχουν

παράλληλα σε μια εφαρμογή έτσι ώστε το τελικό αποτέλεσμα να είναι το αναμενόμενο και σωστό. Επίσης θα πρέπει να ληφθεί υπόψη και ο φόρτος εργασίας που θα αναλογεί σε κάθε μονάδα εργασίας, είτε αυτή θεωρείται νήμα (thread) ή διεργασία (process) ή ακόμα και ολόκληρος ο πυρήνας. Πολλές είναι οι περιπτώσεις που μπορεί κάποιο νήμα ή διεργασία με μεγάλο φόρτο εργασίας να καθυστερήσει ολόκληρη την εκτέλεση ή κάποιος πυρήνας που δουλεύει περισσότερο σε σχέση με τούς άλλους να δημιουργήσει προβλήματα θερμοκρασίας σε ολόκληρο το chip. Οι πυρήνες που βρίσκονται στο ίδιο chip πρέπει να μοιράζονται το εύρος ζώνης ταχύτητας (bandwidth) για πρόσβαση στο 3ο επίπεδο της κρυφής μνήμης (L3 cache) καθώς και για πρόσβαση στην κοινή τους μνήμη (shared memory).

2.4 Ανασκόπηση επόμενου κεφαλαίου

Στο επόμενο κεφάλαιο [Κεφάλαιο 3] θα παρουσιάσουμε περιληπτικά τα κυριότερα προγραμματιστικά μοντέλα που χρησιμοποιούνται ευρέως σήμερα διαχωρίζοντας τα στις δυο κυριότερες αρχιτεκτονικές τους όπου η κάθε μια βασίζεται σε διαφορετικό μοντέλο μνήμης, την αρχιτεκτονική κοινής μνήμης (shared memory) και κατανεμημένης μνήμης (distributed memory) αντίστοιχα. Επιπλέον θα γίνει και μια σύντομη αναφορά στους βασικότερους παράλληλους μεταγλωττιστές που έχουν δημιουργηθεί μέχρι σήμερα παρουσιάζοντας τι ακριβώς προσφέρουν από πλευράς παραλληλισμού αλλά και ποιες γλώσσες προγραμματισμού υποστηρίζουν.

Κεφάλαιο 3

Θεωρητικό Υπόβαθρο

3.1 Εισαγωγή

Τα παράλληλα προγραμματιστικά μοντέλα υλοποιούνται με διάφορους τρόπους : βιβλιοθήκες οι οποίες δημιουργούνται ως επί το πλείστον από σειριακές γλώσσες, ως γλωσσικές επεκτάσεις ή σαν πλήρη νέα πρότυπα εκτέλεσης. Είναι επίσης κατά προσέγγιση ταξινομημένα σε δύο είδη συστημάτων – αρχιτεκτονικών : αρχιτεκτονική Κοινής Μνήμης (Shared Memory) και αρχιτεκτονική Κατανεμημένης Μνήμης (Distributed Memory).

Οι περισσότεροι από τους σημερινούς αυτόματους μεταφραστές για παράλληλα συστήματα κάνουν source to source μετάφραση χρησιμοποιώντας υποδείξεις – οδηγίες openMP, MPI ή κάτι παρόμοιας σύνταξης, ενώ μερικοί άλλοι μεταφράζουν άμεσα σε κώδικα μηχανών. Αυτό μπορεί να γίνει είτε εντελώς αυτόματα, είτε με τη βοήθεια του προγραμματιστή χρησιμοποιώντας ένα γραφικό εργαλείο περιβάλλοντος (Graphical User Interface – GUI). Η πιο συνηθισμένη αυτόματα παραλληλοποιήσιμη γλώσσα προγραμματισμού είναι η FORTRAN και αυτό οφείλεται κυρίως στην έλλειψη δεικτών. Διάφορες τεχνικές και προσεγγίσεις παρουσιάζονται στα διαφορετικά προγράμματα όπου το κάθε ένα από αυτά συμβάλλει στην έρευνα της αυτόματης παραλληλοποίησης είτε για αρχιτεκτονικές Κοινής Μνήμης (Shared Memory), είτε για αρχιτεκτονικές Κατανεμημένης Μνήμης (Distributed Memory).

Πιο κάτω [Πίνακας 3.1, Πίνακας 3.2] παρουσιάζονται εν συντομία τα κύρια παράλληλα προγραμματιστικά μοντέλα που χρησιμοποιούνται σήμερα για την ανάπτυξη παράλληλων εφαρμογών ενώ στην συνέχεια παρατίθενται [Πίνακας 3.3] οι πιο γνωστοί αυτόματοι μεταφραστές για παράλληλα συστήματα .

Έχει γίνει κατηγοριοποίηση των προγραμματιστικών μοντέλων σύμφωνα με τα εφτά κριτήρια από τον H. Kasim et. al. 2008 [14] στις δυο κύριες αρχιτεκτονικές που χρησιμοποιούνται ευρέως σήμερα. Οι αυτόματοι μεταφραστές κατηγοριοποιούνται σύμφωνα με τις γλώσσες προγραμματισμού που υποστηρίζουν.

Κριτήρια	MPI	UPC	Fortress
Μονάδα εκτέλεσης	Νήμα (Thread)	Νήμα (Thread)	Νήμα (Thread)
Μεθοδολογίες προγραμματισμού	API, C, Fortran	API, C, Fortran	API, Extension to C
Διαχείριση μονάδων εκτέλεσης	Ρητός καθορισμός	Υπονοείται	Υπονοείται
Χωρισμός φόρτου εργασίας	Ρητός καθορισμός	Υπονοείται	Υπονοείται
Διαμοιρασμός στόχων στις μονάδες εκτέλεσης	Ρητός καθορισμός	Υπονοείται	Υπονοείται
Συγχρονισμός	Ρητός καθορισμός	Υπονοείται	Υπονοείται
Μοντέλο επικοινωνίας	Κοινό διάστημα διευθύνσεων	Κοινό διάστημα διευθύνσεων	Κοινό διάστημα διευθύνσεων

Πίνακας 3.1 : Κατηγοριοποίηση των μοντέλων παράλληλου προγραμματισμού της αρχιτεκτονικής Κοινής Μνήμης (Shared Memory) σύμφωνα με τα εφτά κριτήρια του H. Kasim et. Al 2008.

Κριτήρια	MPI	UPC	Fortress
Μονάδα εκτέλεσης	Διεργασία (Process)	Νήμα (Thread)	Νήμα (Thread)
Μεθοδολογίες προγραμματισμού	API, C, Fortran	API, C	Νέα Γλώσσα
Διαχείριση μονάδων εκτέλεσης	Υπονοείται	Υπονοείται	Υπονοείται/Ρητός καθορισμός
Χωρισμός φόρτου εργασίας	Ρητός καθορισμός	Υπονοείται/Ρητός καθορισμός	Υπονοείται/Ρητός καθορισμός
Διαμοιρασμός στόχων στις μονάδες εκτέλεσης	Ρητός καθορισμός	Υπονοείται/Ρητός καθορισμός	Υπονοείται/Ρητός καθορισμός
Συγχρονισμός	Υπονοείται	Υπονοείται/Ρητός καθορισμός	Υπονοείται/Ρητός καθορισμός
Μοντέλο επικοινωνίας	Αποστολή μηνυμάτων	Χωρισμένο σφαιρικό διάστημα διευθύνσεων	Σφαιρικό διάστημα διευθύνσεων

Πίνακας 3.2 : Κατηγοριοποίηση των μοντέλων παράλληλου προγραμματισμού της αρχιτεκτονικής Κατανεμημένης Μνήμης (Distributed Memory) σύμφωνα με τα εφτά κριτήρια του H. Kasim et. Al 2008.

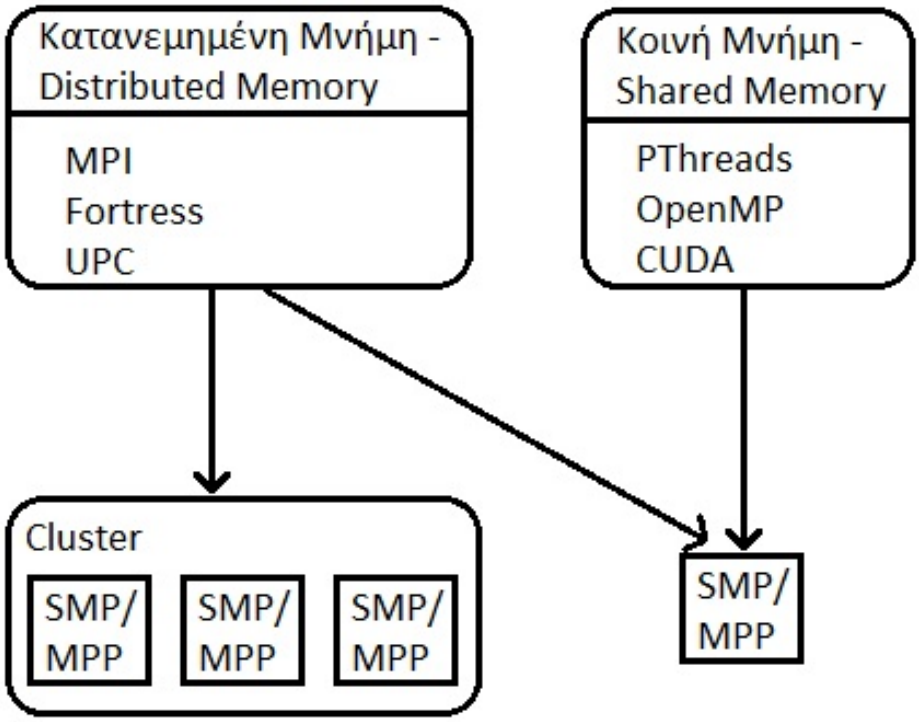
Language / Compiler	C	C++	C#	Java	FORTRAN	Other Language
VS 2010 .Net Framework 4.0		✓	✓		✓	✓
ICU-PFC Compiler					✓	
JavaB				✓		
JavaR				✓		
Paradigm Compiler					✓	
Parasol					✓	
Parawise					✓	
PGI Compiler	✓	✓			✓	
Polaris					✓	
Promis	✓	✓		✓	✓	✓
SUIF Compiler	✓	✓		✓	✓	
SunStudio SunCC	✓					
ZJava Compiler				✓		
ZPL						✓

Πίνακας 3.3 : Κατηγοριοποίηση αυτόματων μεταφραστών σύμφωνα με τις γλώσσες προγραμματισμού που υποστηρίζουν.

Οι παράλληλοι επεξεργαστές κοινής μνήμης (Shared-Memory Multiprocessors - SMPs) εκμεταλλεύονται τον παραλληλισμό με την εκτέλεση πολλαπλών νημάτων (threads) ελέγχου που εκτελούνται στους ανεξάρτητους επεξεργαστές και επικοινωνούν μεταξύ τους μέσω της κοινής μνήμης. Τέτοια συστήματα υποστηρίζοντας το μοντέλο σφαιρικής ονομασίας (global name-space) μπορούν να παρέχουν μια ομοιόμορφη άποψη της μνήμης σε όλους τους επεξεργαστές απλοποιώντας το έργο του προγραμματιστή ή αυτόματης παραλληλοποίησης μιας εφαρμογής. Τεχνικές ανάλυσης εξαρτήσεων χρησιμοποιούνται για να ανιχνεύσουν και να προγραμματίσουν παραλληλισμό σε επιπέδων βρόχων στις μηχανές κοινής μνήμης. Ο παραλληλισμός εφαρμογών που δεν χρησιμοποιούν αριθμητικούς υπολογισμούς για τις μηχανές κοινής μνήμης είναι αρκετά περιορισμένος. Για να επιτύχει κάποιος υψηλά επίπεδα της απόδοσης σε τέτοια συστήματα, εκτός από τον παραλληλισμό, είναι σημαντικό ο μεταγλωττιστής να δίνει μεγάλη προσοχή στην τοποθεσία και στην τοποθέτηση των στοιχείων (data locality). Αυτό, λόγω του ότι τέτοια συστήματα υποστηρίζουν πολλαπλά επίπεδα ιεράρχησης της μνήμης (multilevel memory hierarchies).

Σε μια προσπάθεια να αυξηθεί ο αριθμός των επεξεργαστών σε ένα παράλληλο σύστημα μεγάλης κλίμακας, έχουν αναπτυχθεί τα κατανεμημένα συστήματα μνήμης (distributed-memory systems). Ο στόχος του μεταγλωττιστή για διαμοιρασμό των πόρων στα προγράμματα βασισμένα στο μοντέλο κατανεμημένης μνήμης είναι πολύ περίπλοκος. Όχι μόνο είναι απαραίτητο να ανιχνευθεί ο παραλληλισμός και να διαχωριστεί ο υπολογισμός για την παράλληλη εκτέλεση, αλλά τα κοινά στοιχεία πρέπει επίσης να χωριστούν και να κατανεμηθούν στις μνήμες του κάθε μεμονωμένου επεξεργαστή. Η πολυπλοκότητα του πιο πάνω στόχου έχει οδηγήσει στην εισαγωγή των νέων τεχνικών σε συνδυασμό με νέες γλώσσες για την κατανομή των στοιχείων μετατοπίζοντας μερική πολυπλοκότητα από τον μεταγλωττιστή στον προγραμματιστή. Δεδομένου ότι το κόστος επικοινωνίας μεταξύ των επεξεργαστών-κόμβων του συστήματος είναι σημαντικό, ο μεταγλωττιστής πρέπει επίσης να υιοθετήσει ποικίλες τεχνικές βελτιστοποίησης έτσι ώστε να γίνουν ανεκτές οι καθυστερήσεις επικοινωνίας. Επιπλέον η κατανομή των πινάκων στις μνήμες των κόμβων απαιτεί χρήση νέων τεχνικών παραγωγής διευθύνσεων.

3.2 Παράλληλα προγραμματιστικά μοντέλα



Εικόνα 3.2.1 : Τα έξι προγραμματιστικά μοντέλα κατηγοριοποιημένα σε δυο ομάδες και τι υποστηρίζουν ως προς τις αρχιτεκτονικές κοινής και κατανεμημένης μνήμης.

3.2.1 Pthreads

Το γεννικό μοντέλο ονομάζεται Pthreads ή Portable Operating System Interface (POSIX) [14,42]. Τα νήματα (threads) είναι ένα σύνολο τύπων και κλήσεων διαδικασιών της γλώσσας προγραμματισμού C. Το μοντέλο Pthreads υλοποιείται σαν αρχείο επικεφαλίδων (header file - *pthread.h*) και η βιβλιοθήκη για τη δημιουργία και το χειρισμό κάθε μονάδας εκτέλεσης (worker) αποκαλείται *threads*.

Η διαχείριση των μονάδων εκτέλεσης σε Pthreads απαιτεί από τον προγραμματιστή να τις δημιουργήσει ρητά και να καταστρέψει τα threads με τη χρήση των συναρτήσεων *pthread_create* και *pthread_exit* αντίστοιχα. Η συνάρτηση

pthread_create απαιτεί τέσσερις παραμέτρους: (i) το thread που χρησιμοποιείται για να τρέξει, (ii) ιδιότητες εκτέλεσης του thread, (iii) η συνάρτηση που καλείται το thread να τρέξει κατά την κλήση, και (iv) οι παράμετροι της συνάρτησης. Το thread που δημιουργείται θα τρέξει τη συνάρτηση έως ότου να κληθεί η συνάρτηση pthread_exit.

Ο χωρισμός φόρτου εργασίας και η ανάθεση στόχων διευκρινίζονται ρητά από τους προγραμματιστές σαν δεδομένα εισόδου – παράμετροι της pthread_create. Ο χωρισμός φόρτου εργασίας διευκρινίζεται στην τρίτη παράμετρο της καλούμενης από το thread συνάρτησης, ενώ η ανάθεση στόχου περνιέται σαν πρώτη παράμετρος στην pthread_create. Ένα thread μπορεί να ενωθεί με άλλα threads χρησιμοποιώντας τη συνάρτηση *pthread_join*. Όταν αυτή καλείται, το thread που την καλεί θα αναστείλει την εκτέλεσή του μέχρι το κυρίως thread τελειώσει την εκτέλεσή του ακριβώς πριν ενωθεί με τα άλλα threads.

Όταν τα πολλαπλά threads έχουν πρόσβαση στα κοινά στοιχεία, οι προγραμματιστές πρέπει να είναι σε θέση να αντιμετωπίσουν συγκρούσεις ασυνέπειας στα δεδομένα (*data race*) και τυχόν αδιέξοδα (*deadlocks*). Για να προστατεύσουμε τα κρίσιμα τμήματα του κώδικα, δηλ. κομμάτι του κώδικα που περιέχει πρόσβαση στα κοινά στοιχεία, η Pthreads περιλαμβάνει αμοιβαίο αποκλεισμό (*mutex*) και σηματοφόρους (*semaphores*). Το mutex επιτρέπει μόνο σε ένα thread να μπορεί να εισέλθει σε ένα κρίσιμο τμήμα, ενώ ο σηματοφόρος επιτρέπει σε διάφορα threads να εισέλθουν ένα κρίσιμο τμήμα.

3.2.2 OpenMP

Η OpenMP [14,40] αποτελείται από ένα σύνολο οδηγιών (*directives*) για τον μεταγλωττιστή, βιβλιοθηκών συναρτήσεων οι οποίες μπορούν να κληθούν κατά την εκτέλεση και μεταβλητών περιβάλλοντος που επεκτείνουν τα προγράμματα της FORTRAN, C και C++. Η OpenMP θεωρείται φορητό μοντέλο για την αρχιτεκτονική κοινής μνήμης. Η μονάδα εκτέλεσης της OpenMP είναι τα threads.

Η διαχείριση των μονάδων εκτέλεσης υπονοείται. Οι πρόσθετες οδηγίες (special directives) χρησιμοποιούνται για να διευκρινίσουν ότι ένα τμήμα του κώδικα πρόκειται να τρέξει παράλληλα. Ο αριθμός των threads που χρησιμοποιούνται διευκρινίζεται χρησιμοποιώντας έναν εκτός ζώνης μηχανισμό, που είναι μια μεταβλητή περιβάλλοντος. Κατά συνέπεια, αντίθετα από τη Pthread, δεν υπάρχει καμία ανάγκη για τους προγραμματιστές να διαχειριστούν τη διάρκεια ζωής των threads.

Ο διαχωρισμός του φόρτου εργασίας και η και η ανάθεση στόχων απαιτούν σχετικά μικρή προσπάθεια προγραμματισμού. Οι προγραμματιστές απλά πρέπει διευκρινίσουν τις οδηγίες (directives) του μεταγλωττιστή για να υποδείξουν μια παράλληλη περιοχή (block), δηλαδή (i) *#pragma omp parallel {}* για C/C++, και (ii) *!$omp parallel* και *!$omp and parallel* FORTRAN. Η OpenMP επίσης αφαιρεί το πώς ο φόρτος εργασίας (π.χ. ένας πίνακας) διαιρείται σε στόχους (π.χ. υπο-πίνακες) και πώς οι στόχοι αυτοί ορίζονται στα threads, αν και ο προγραμματιστής αν θέλει μπορεί να τα ορίσει ρητά.

Η OpenMP επίσης ακολουθεί διάφορες τεχνικές για να υποστηρίξει τον αυτονόητο συγχρονισμό όπου οι προγραμματιστές πρέπει μόνο να διευκρινίσουν πού χρειάζεται συγχρονισμός. Δηλαδή, (i) *barrier* – επιτρέπει το συγχρονισμό όλων των threads που ανήκουν στην ίδια ομάδα, (ii) *atomic* – επιτρέπεται σε όλα τα threads να εκτελούνται, αλλά για κάθε μονάδα του χρόνου μόνο ένα επιτρέπεται να εκτελέσει εντολή ανάγνωσης ή εγγραφής, (iii) *ordered* – επιτρέπει στο κομμάτι του κώδικα να εκτελείται διαδοχικά και (iv) *flush* – εξασφαλίζει ότι όλα τα threads έχουν μια συνεπή άποψη ορισμένων αντικειμένων στη μνήμη. Ο πραγματικός όμως μηχανισμός συγχρονισμού είναι αυτός που προέρχεται από την πρόνοια των προγραμματιστών.

3.2.3 CUDA

Η CUDA (Compute Unified Device Architecture) [30,32] είναι μια αρχιτεκτονική υλικού και λογισμικού που σχεδιάστηκε για την υποστήριξη της παράλληλης

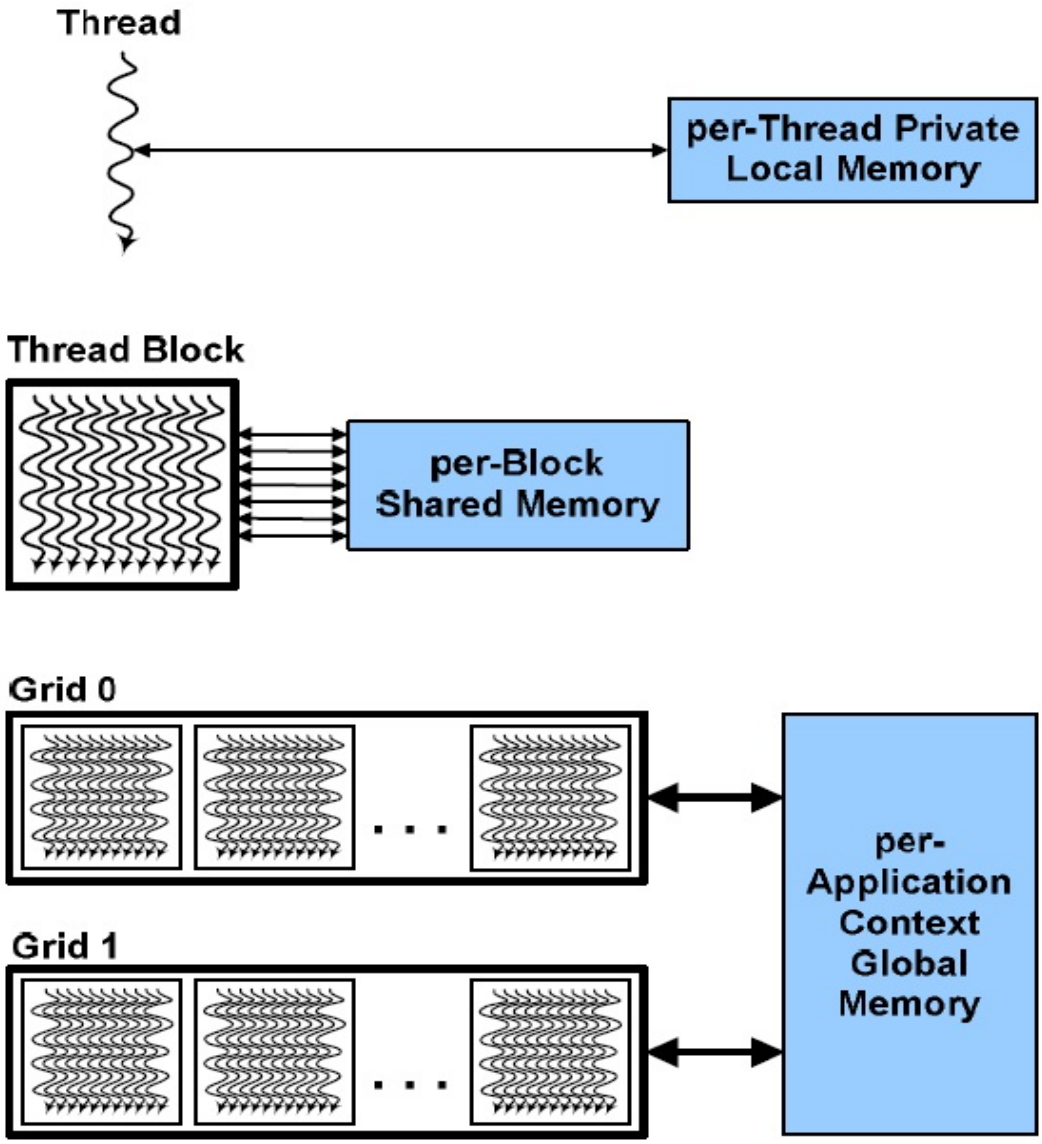
επεξεργασίας σε κάρτες γραφικών (GPU – Graphics Processing Unit) της nVidia. Αποτελεί την επέκταση του προγραμματισμού σε γλώσσες όπως η C, C++, FORTRAN, OpenCL, DirectCompute, και άλλες, και δίνει την δυνατότητα παράλληλης εκτέλεσης σε προγράμματα γραμμένα σε αυτές τις γλώσσες.

Ένα πρόγραμμα CUDA καλεί τους παράλληλους πυρήνες όπου ένας πυρήνας έχει την δυνατότητα να εκτελεί εργασίες παράλληλα μέσα από ένα σύνολο παράλληλων threads. Ο προγραμματιστής ή ο μεταγλωττιστής οργανώνει αυτά τα threads σε σύνολα και πλέγματα (grids) συνόλων threads. Το GPU δημιουργεί ένα πρόγραμμα πυρήνων για κάθε πλέγμα των παράλληλων συνόλων threads [Εικόνα .6]. Κάθε thread μέσα σε ένα σύνολο εκτελεί μια περίπτωση του πυρήνα, έχει μια ταυτότητα (thread ID), ένα μετρητή προγράμματος (program counter), καταχωριτές, ιδιωτική τοπική μνήμη (per-thread private local memory), δεδομένα εισόδου και αποτελέσματα εξόδου.

Ένα σύνολο αποτελεί μια ομάδα από threads που μπορεί να εκτελεστεί παράλληλα και να συνεργαστούν μεταξύ τους τα ίδια, μέσω της κοινής μνήμης και με τη βοήθεια συγχρονισμού. Επίσης κάθε σύνολο έχει την δική του ταυτότητα μέσα στο πλέγμα (block ID).

Ένα πλέγμα είναι ένας πίνακας συνόλων threads που εκτελούν τον ίδιο πυρήνα, διαβάζουν και γράφουν τα δεδομένα εισόδου και εξόδου, αντίστοιχα, από τη σφαιρική μνήμη (global memory) και συγχρονίζονται μεταξύ των εξαρτώμενων κλήσεων του πυρήνα. Στο παράλληλο πρότυπο προγραμματισμού της CUDA, κάθε ένα από τα threads έχει δικό του ιδιωτικό διάστημα μνήμης (per-thread private memory space) που χρησιμοποιείται για τους καταχωρητές, τις κλήσεις συναρτήσεων και για τις αυτόματες μεταβλητές πινάκων της C. Κάθε σύνολο από threads έχει κοινό διάστημα μνήμης (per-Block shared memory space) που χρησιμοποιείται για ενδοεπικοινωνία μεταξύ των threads και διαμοιρασμό δεδομένων εισόδου και εξόδου όταν αυτά τρέχουν παράλληλα. Τα πλέγματα από σύνολα threads μοιράζονται αποτελέσματα από το σφαιρικό διάστημα μνήμης (per-

Application Context Global memory space) κατόπιν συγχρονισμού όλων των πυρήνων.



Εικόνα 3.2.2 : Η ιεράρχηση των threads, συνόλων και πλεγμάτων στο μοντέλο προγραμματισμού CUDA με τα αντίστοιχα ιδιωτική τοπική μνήμη ανά thread (per-thread private local memory), κοινό διάστημα μνήμης ανά σύνολο (per-Block shared memory space) και σφαιρικό διάστημα μνήμης ανά εφαρμογή (per-Application Context Global memory space).

Η διαχείριση των μονάδων εκτέλεσης στην CUDA υπονοείται, χωρίς οι προγραμματιστές να διαχειρίζονται τη δημιουργία και την καταστροφή των threads. Πρέπει απλά να διευκρινίσουν τη διάσταση του πλέγματος και του συνόλου που απαιτούνται για να τύχει επεξεργασίας ένας συγκεκριμένος στόχος.

Ο διαχωρισμός του φόρτου εργασίας και η και η ανάθεση στόχων γίνονται ρητά. Οι προγραμματιστές πρέπει να καθορίσουν το φόρτο εργασίας που τυγχάνει επεξεργασίας παράλληλα με τη χρησιμοποίηση του *Global_Function <<<dimGrid,dimBlock>>> (Arguments)* όπου (i) *Global_Function* είναι η γενική συνάρτηση που θα τρέξει στα threads, (ii) *dimGrid* είναι η διάσταση και το μέγεθος του πλέγματος, (iii) *dimBlock* είναι η διάσταση και το μέγεθος κάθε συνόλου και (iv) *(Arguments)* αντιπροσωπεύει την παράμετρο της γενικής συνάρτησης *Global_Function*. Το *<<<dimGrid,dimBlock>>>,* που αναφέραμε πιο πάνω, αποτελεί το στόχο που θα τρέξουν οι μονάδες εκτέλεσης.

Ο συγχρονισμός για όλα τα threads στην CUDA υπονοείται μέσω της συνάρτησης *_syncthreads()*. Αυτή η συνάρτηση είναι υπεύθυνη να συντονίσει την επικοινωνία μεταξύ των threads του ιδίου συνόλου.

Η nVidia έχει ήδη ανακοινώσει την εξέλιξη του πιο πάνω μοντέλου, δηλαδή την επόμενη γενιά αρχιτεκτονικής CUDA, με το κωδικό όνομα FERMI [9], το οποίο και χαρακτηρίζει ως τη ψυχή ενός υπέρ-υπολογιστή στο σώμα ενός GPU … "*The Soul of a Supercomputer in the Body of a GPU*"

3.2.4 MPI

Η MPI (Message Passing Interface) [14,38] αποτελεί μια προδιαγραφή μοντέλου προγραμματισμού για αποστολή μηνυμάτων. Ορίζει σαν μονάδα εκτέλεσης μια διεργασία (process). Η MPI είναι προς το παρόν το κύριο standard για ανάπτυξη εφαρμογών HPC (High Performance Computing) σε αρχιτεκτονικές με κατανεμημένη μνήμη. Προσφέρει επεκτάσεις για γλώσσες προγραμματισμού όπως

C, C++ και FORTRAN. Κάποιες από τις γνωστές υλοποιήσεις της MPI είναι η OpenMPI , MVAPICH , MPICH, GridMPI και LAM/MPI.

Η διαχείριση των μονάδων εκτέλεσης στην MPI υπονοείται, έτσι δεν είναι απαραίτητο να κωδικοποιηθεί η δημιουργία, ο σχεδιασμός, ή η καταστροφή των διαδικασιών από τον προγραμματιστή. Αντιθέτως, κάποιος χρειάζεται μόνο ένα εργαλείο γραμμής εντολών, όπως το *mpirun*, για να δηλώσει ρητά στην MPI πόσες διεργασίες χρειάζονται και, προαιρετικά, την ανάθεση των διεργασιών στον επεξεργαστή. Με βάση αυτές τις πληροφορίες, η υποδομή runtime, θα πραγματοποιήσει έπειτα τη διαχείριση των μονάδων εκτέλεσης για το συμφέρον των χρηστών.

Η διαχείριση του φόρτου εργασίας και η ανάθεση των εργασιών πρέπει να γίνει από τους προγραμματιστές, όπως και στην Pthread. Οι προγραμματιστές πρέπει να ελέγχουν ποιες εργασίες-στόχοι πρέπει να υπολογίζονται από κάθε διεργασία. Για παράδειγμα, δεδομένου ενός δισδιάστατου πίνακα (σαν φόρτος εργασίας), ο προγραμματιστής μπορεί να χρησιμοποιήσει ένα αναγνωριστή διεργασίας (βαθμός ιεράρχησης – Rank), για να καθορίσει ποιό μέρος του πίνακα θα υπολογίσει η κάθε διεργασία.

Η επικοινωνία μεταξύ των διεργασιών γίνεται εφικτή με την αποστολή μηνυμάτων, όπου ο διαμοιρασμός των δεδομένων γίνεται από μια διεργασία όπου αυτή στέλνει τα δεδομένα στις άλλες.

Η MPI ταξινομεί την αποστολή μηνυμάτων σε δυο κατηγορίες, σαν από σημείο σε σημείο (*point-to-point*) και σαν συλλογικές (*collective*). Οι από σημείο σε σημείο διαδικασίες όπως τις *MPI_Send* και *MPI_Recv* διευκολύνουν την επικοινωνία μεταξύ των διεργασιών, ενώ οι συλλογικές διαδικασίες όπως το MPI_Bcast διευκολύνουν την επικοινωνία δύο ή περισσοτέρων διεργασιών.

Η *MPI_Barrier* χρησιμοποιείται για να καθορίσει ότι χρειάζεται συγχρονισμός. Η διαδικασία αυτή εμποδίζει κάθε διεργασία από το να συνεχίζει την εκτέλεσή της

μέχρις ότου όλες οι διεργασίες να μπούνε στον περιορισμό (barrier). Μια τυπική χρήση της MPI_Barrier είναι να επιβεβαιώσει ότι καθολικά δεδομένα (global data) έχουν δοθεί στις κατάλληλες διεργασίες.

3.2.5 Fortress

Η Fortress [14,34] είναι μια γλώσσα προγραμματισμού προδιαγραφών που σχεδιάστηκε για υπολογισμό υψηλής επίδοσης. Η οντότητα της μονάδας εργασίας είναι το thread (νήμα).

Στην Fortress η διαχείριση των μονάδων εργασίας και του φόρτου εργασίας καθώς και η ανάθεσή του της μονάδας εργασίας μπορεί να υπονοείται ή να καθοριστεί ρητά από τον προγραμματιστή. Στην περίπτωση που υπονοείται, η αλληλεπίδραση με τους βρόγχους είναι παράλληλη εξ' ορισμού. Ο προγραμματιστής δεν χρειάζεται να ορίσει ποια threads θα πρέπει να τρέχουν σε κάθε επανάληψη. Στην περίπτωση του ρητού καθορισμού, η δημιουργία ενός thread μπορεί να γίνει χρησιμοποιώντας την εντολή *spawn*. Για παράδειγμα στο *t = spawn Global_Function(Arguments)*,το t υποδηλώνει το thread που δημιουργήθηκε και το Global_Function υποδηλώνει τις εργασίες που θα τρέξει το t. Σημειώστε ότι εκτός από την κλήση μιας καθολικής συνάρτησης (Global Function), μια εργασία μπορεί να εκφραστεί και σαν ένας συγκεκριμένος στόχος. Για να σταματήσει την εκτέλεσή του ένα thread t, πρέπει να γίνει κλήση της συνάρτησης t.stop().

Η διαχείριση του φόρτου εργασίας και η ανάθεση των εργασιών στην περίπτωση του ρητού καθορισμού είναι παρόμοια με της CUDA. Κάποιος χρειάζεται να αποφασίσει πώς ο φόρτος εργασίας θα σπάσει σε στόχους (tasks) και πώς οι στόχοι θα ανατεθούν στα threads.

Για να αποφευχθούν συγκρούσεις ασυνέπειας στα δεδομένα (*data races*) και τυχόν αδιέξοδα (*deadlocks*) σε ένα πρόγραμμα, ο προγραμματιστής πρέπει να ορίσει ρητά τον τρόπο συγχρονισμού. Υπάρχουν δύο είδη συγχρονισμού, της μείωσης

(*reduction*) και της ατομικότητας (*atomic*). Η χρήση του reduction είναι για να αποφεύγεται η ανάγκη για συγχρονισμό, όπου εκτελεί ένα υπολογισμό όσο τοπικά γίνεται. Το atomic, χρησιμοποιείται για να γίνεται έλεγχος των δεδομένων στις παράλληλες εκτελέσεις και έχει την ίδια ακριβώς λειτουργία με το atomic στην OpenMP.

3.2.6 UPC

Η UPC (Unified Parallel C) [14,46] είναι μια παράλληλη γλώσσα προγραμματισμού για αρχιτεκτονικές κοινής αλλά και κατανεμημένης μνήμης. Άσχετα από την αρχιτεκτονική του συστήματος, η UPC υιοθετεί την έννοια της διαχωρισμένης μνήμης (partitioned memory). Με αυτήν την έννοια, οι προγραμματιστές βλέπουν το σύστημα ως ένα σφαιρικό διάστημα διευθύνσεων (global address space) το όποιο χωρίζεται σε διάφορα λογικά διαστήματα διευθύνσεων ανά-thread. Κάθε ένα thread έχει δύο τύπους προσβάσεων μνήμης: στο ιδιωτικό του διάστημα διευθύνσεών του ή διάστημα διευθύνσεων άλλων threads. Οι προσβάσεις και στους δύο τύπους διαστημάτων διευθύνσεων ανά-thread χρησιμοποιούν την ίδια σύνταξη. Για να βελτιώσει την απόδοση των προσβάσεων στη μνήμη, η UPC εισάγει την έννοια του *thread affinity* (συνάφεια νήματος). Με αυτό το χαρακτηριστικό γνώρισμα, η UPC βελτιστοποιεί την απόδοση προσβάσεων στη μνήμη μεταξύ ενός thread και του διαστήματος διευθύνσεων ανά-thread όπου έχει δεσμευθεί το thread.

Στην UPC, η διαχείριση φόρτου εργασίας υπονοείται, ενώ ο χωρισμός του φόρτου εργασίας και η ανάθεση εργασιών στα threads μπορεί είτε να υπονοείται είτε να καθορίζεται ρητά από τον προγραμματιστή. Για τη διαχείριση των μονάδων εργασίας, οι προγραμματιστές πρέπει να διευκρινίσουν τον αριθμό των threads που απαιτούνται κατά τη διάρκεια της κλήσης από το εργαλείο γραμμής εντολών, *upcrun*. Ο αυτόματος χωρισμός φόρτου εργασίας και ανάθεσης εργασιών υποστηρίζεται μέσω ενός API (application programming interface), του *upc_forall,* που είναι παρόμοιο με το for της γλώσσας προγραμματισμού C, εκτός από το ότι το περιεχόμενο της επανάληψης θα οργανωθεί για να τρέξει παράλληλα. Όταν χρησιμοποιείται το API, δεν υπάρχει καμία ανάγκη πρόσθετης προσπάθειας

προγραμματισμού από τους προγραμματιστές για ανάθεση εργασιών στα threads. Στη ρητή προσέγγιση διαχωρισμού του φόρτου εργασίας και ανάθεσης εργασιών στο UPC τα πράγματα είναι παρόμοια με της MPI, όπου οι προγραμματιστές πρέπει να διευκρινίσουν πού θα τρέξει κάθε thread.

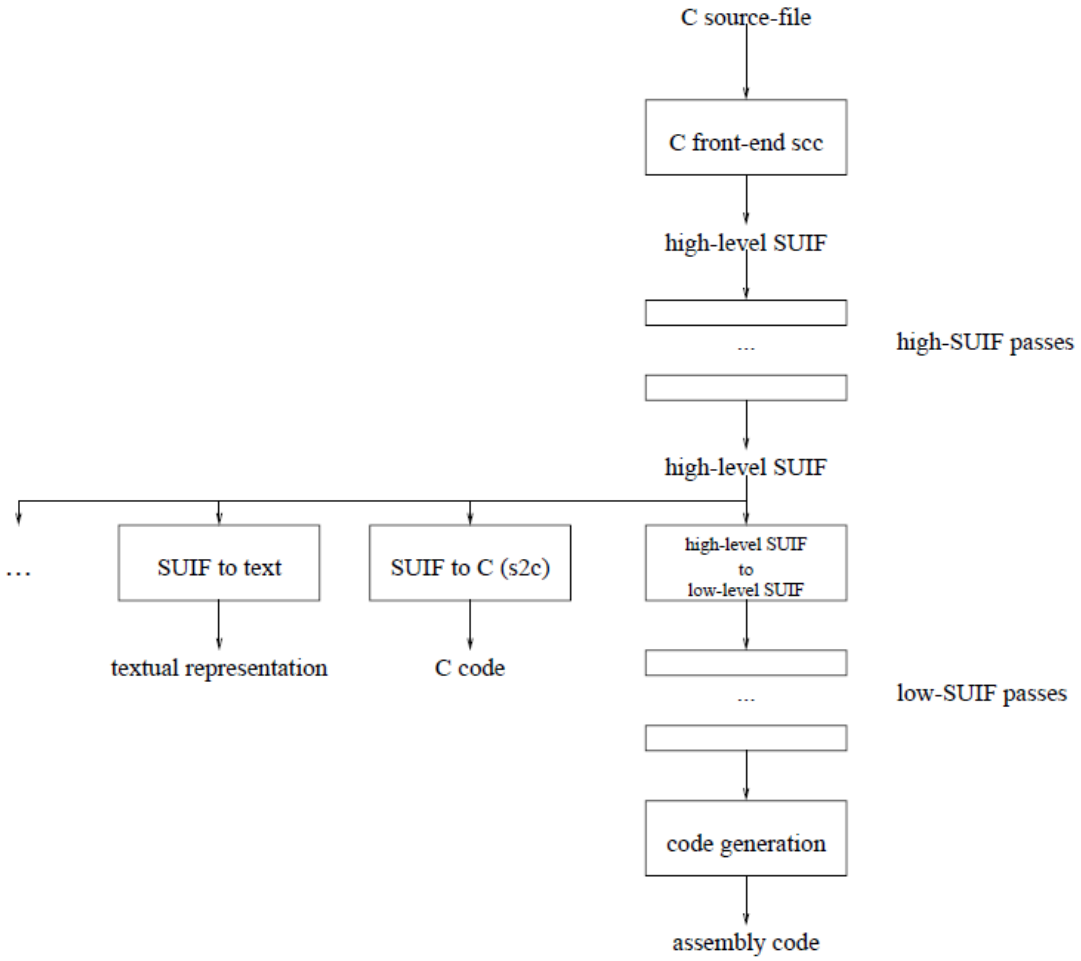
Η επικοινωνία μεταξύ των threads στην UPC υιοθετεί τη χωρισμένη σφαιρική διεύθυνση (*Partitioned Global Address Space* – PGAS) με τη χρησιμοποίηση δεικτών (*pointers*). Υπάρχουν τρεις τύποι δεικτών που χρησιμοποιούνται συνήθως στην UPC: (i) *private pointer* όπου οι ιδιωτικοί δείκτες δείχνουν στο ιδιωτικό διάστημα διευθύνσεών τους, (ii) *private pointer-to-share* όπου οι ιδιωτικοί δείκτες δείχνουν στο κοινό διάστημα διευθύνσεων και (iii) *shared pointer-to-share* όπου οι κοινοί δείκτες από ένα διάστημα διευθύνσεων δείχνουν άλλο κοινό διάστημα διευθύνσεων.

Η UPC παρέχει διάφορους μηχανισμούς συγχρονισμού οι οποίοι είναι: (i) *upc_barrier expression* όπου η λειτουργία του υπονοείται και είναι παρόμοιο με το MPI_Barrier της MPI, (ii) *upc_notify expression* και *upc_wait expression* όπου η λειτουργία τους υπονοείται και όταν αυτά καλούνται δεν εμποδίζουν κανένα thread από το να τρέξει, (iii) *upc_fence* όπου η λειτουργία του υπονοείται και διαβεβαιώνει ότι όλες οι κοινές αναφορές σε δεδομένα ολοκληρώνονται πριν αυτό καλεστεί, (iv) *upc_lock()* και *upc_unlock()* όπου η λειτουργία τους καθορίζεται ρητά και προστατεύουν κοινά δεδομένα από ταυτόχρονες προσβάσεις και (v) δυο βιβλιοθήκες, *upc_strict.h* και *upc_relaxed.h*, για έλεγχο της συνέπειας των δεδομένων στη μνήμη (memory consistency control). Η λειτουργία τους καθορίζεται ρητά, στην περίπτωση της αυστηρής συνέπειας (strict consistency) η πρόσβαση σε κοινά δεδομένα γίνεται σειριακά ενώ στην περίπτωση της χαλαρής συνέπειας (relaxed consistency) τα κοινά δεδομένα μπορούν να αλλάξουν σειρά είτε κατά τη διάρκεια της μετάφρασης είτε κατά τη διάρκεια της εκτέλεσης.

3.3 Αυτόματοι μεταφραστές για παράλληλα συστήματα

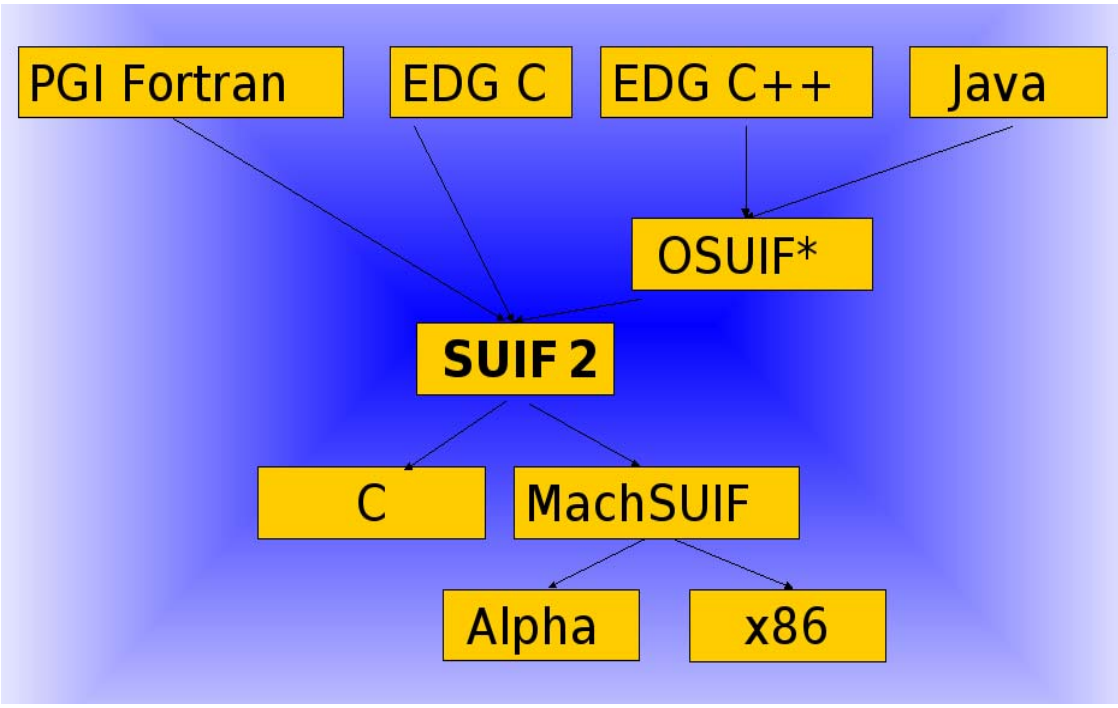
3.3.1 SUIF Compiler

Το σύστημα μεταγλωττιστών SUIF [11, 16, 21, 24, 44], που έχει αναπτυχθεί στο Πανεπιστήμιο του Stanford, υποστήριξε την πειραματική έρευνα για τις πρώτες νέες τεχνικές παράλληλων μεταγλωττιστών. Αποτελείται από τα διάφορα εργαλεία με τυποποιημένα περάσματα βελτιστοποίησης, τα οποία λειτουργούν πάνω από ένα κοινό πλαίσιο (framework) [Εικόνα 3.3.1]. Υπάρχουν δύο σημαντικές εκδόσεις του SUIF, ο SUIF 1.0 [11, 24, 44] και μια αναθεωρημένη έκδοση του, ο SUIF 2.0 [16, 21, 44]. Ο SUIF 2.0 μοιράζεται τις βασικές αρχές και λειτουργίες του SUIF 1.0, αλλά ο σχεδιασμός και ο τρόπος εφαρμογής του είναι εντελώς διαφορετικά. Και οι δύο εκδοχές είναι γραμμένες σε C++ και περιέχουν πάνω από 200.000 γραμμές κώδικα η κάθε μια τουλάχιστον. Επιπλέον, ο SUIF 2.0 έχει ένα καθαρότερο σχέδιο και μια πιο συγυρισμένη δομή.



Εικόνα 3.3.1 : Το βασικό πλαίσιο (framework) πάνω από το οποίο λειτουργεί ο παράλληλος μεταγλωττιστής SUIF.

Δυστυχώς το ενδιάμεσο πλαίσιο πάνω από το οποίο λειτουργούν οι SUIF 1.0 και SUIF 2.0 υποστηρίζουν γλώσσες όπως η FORTRAN και C χωρίς να είναι σε θέση να ανταπεξέλθουν στην αντικειμενοστρέφεια των C++ ή Java οι οποίες απαιτούν απολύτως διαφορετικές υψηλού επιπέδου πληροφορίες σε σχέση με τις προηγούμενες. Ένα πρόσθετο όμως στρώμα πάνω από τον SUIF 2.0, αποκαλούμενο ως Object SUIF (OSUIF) επεκτείνει το πεδίο που καλύπτει ο SUIF 2.0 δίνοντάς του την δυνατότητα να μπορεί να υποστηρίξει και τις αντικειμενοστραφείς γλώσσες προγραμματισμού C++ και Java [Εικόνα 3.3.2]. Το ενισχυτικό πλαίσιο του SUIF 2.0 και OSUIF, αναπτύσσεται στο Πανεπιστήμιο Santa Barbara της Καλιφόρνιας, όπου μαζί και τα δύο αποτελούν μέρος του προγράμματος National Compiler Infrastructure project.



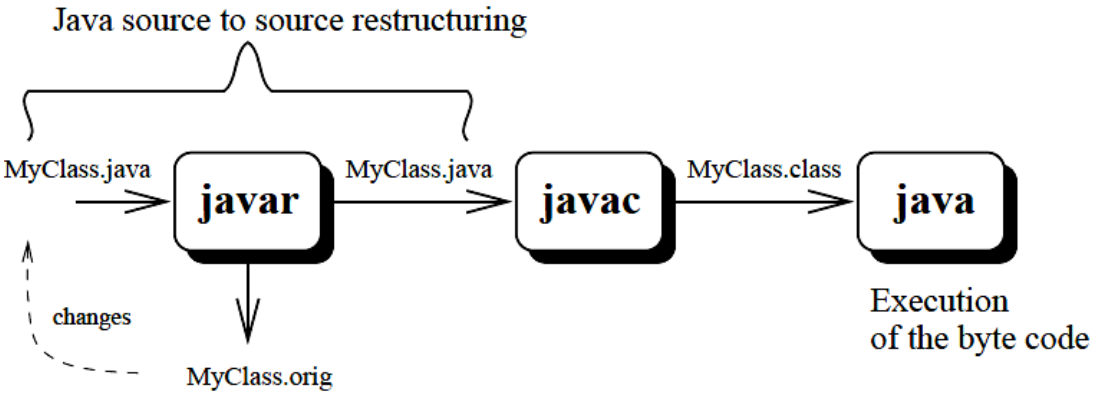
Εικόνα 3.3.2 : Μια υψηλού επιπέδου επισκόπηση του γενικού πλαισίου (overall framework) πάνω από το οποίο λειτουργεί η αναθεωρημένη έκδοση του παράλληλου μεταγλωττιστή SUIF 2.0.

3.3.2 JavaR

Ο JavaR [3, 4] είναι ένας μεταγλωττιστής αναδόμησης του πηγαίου κώδικα (source code) προγραμμάτων γραμμένα σε γλώσσα προγραμματισμού Java ενώ ο ίδιος είναι γραμμένος σε C. Μπορεί να χρησιμοποιηθεί για να εξάγει τον υπονοούμενο παραλληλισμό (implicit parallelism) από τους βρόχους επανάληψης και τις αναδρομικές μεθόδους χρησιμοποιώντας πολυνηματισμό (multithreading). Αυτό το πρωτότυπο στηρίζεται εντελώς στον προσδιορισμό του παραλληλισμού με τη βοήθεια υποδείξεων-οδηγιών/σχολίων (parallelism by means of annotations/directives). Επειδή καμία αυτόματη ανίχνευση του παραλληλισμού δεν υποστηρίζεται, ο παραλληλισμός των βρόχων πρέπει να προσδιοριστεί από τον ίδιο τον προγραμματιστή με τη βοήθεια σχολίων. Εντούτοις, ο μεταγλωττιστής παρέχει την ικανοποιητική λειτουργία για να απλοποιήσει τον παραλληλισμό των προγραμμάτων σε Java χωρίς ο προγραμματιστής να χρειάζεται να γνωρίζει κάτι

ιδιαίτερο από πολυνηματισμό (multithreading). Το εργαλείο μπορεί επίσης να παράγει παρουσιάσεις HTML των εξαγόμενων παράλληλων προγραμμάτων Java.

Στην Εικόνα 3.3.3 απεικονίζεται η προσέγγιση για την αυτόματη εξαγωγή του παραλληλισμού από ένα πρόγραμμα σε Java. Το πρόγραμμα MyClass.java με τις υποδείξεις (directives) από τον προγραμματιστή είναι αυτό που χρησιμοποιείται ως είσοδος για τον μεταγλωττιστή Javar.



Εικόνα 3.3.3 : Η διαδικασία αναδόμησης του σειριακού προγράμματος, μεταγλώττισης του παράλληλου προγράμματος και εκτέλεσης του παράλληλου ενδιάμεσου κώδικα.

Αρχικά ο μεταγλωττιστής προσδιορίζει τους βρόχους και τις αναδρομικές μεθόδους που μπορούν να παραλληλιστούν. Οι παράλληλοι βρόχοι και αναδρομικές μέθοδοι που θα τύχουν επεξεργασίας για αυτόματο παραλληλισμό προσδιορίζονται ρητά από τον προγραμματιστή χρησιμοποιώντας υποδείξεις (directives) τύπου σχολίων. Οι υποδείξεις αυτές έχουν την μορφή /*par*/ και πρέπει να τοποθετηθούν ακριβώς πριν από βρόχο επανάληψης ή την αναδρομική μέθοδο που θέλουμε να παραλληλοποιήσουμε. Στη συνέχεια ο μεταγλωττιστής μετασχηματίζει το πρόγραμμα εισόδου σε μια μορφή που χρησιμοποιεί τον πολυνηματικό μηχανισμό της Java (Java Threads) για να μετατρέψει το πρόγραμμα σε παράλληλο. Επειδή ο παραλληλισμός εκφράζεται στην Java την ίδια, το μετασχηματισμένο πρόγραμμα μπορεί να παράξει εκτελέσιμο ενδιάμεσο κώδικα (bytecode) από οποιοδήποτε μεταγλωττιστή Java (javac στην Εικόνα 3.3.3), και να εκτελεστεί στη συνέχεια από

οποιοδήποτε διερμηνέα ενδιάμεσο κώδικα (bytecode interpreter/Java Virtual Machine) (java στην Εικόνα 3.3.3). Δεδομένου ότι τα ονόματα των αρχείων παίζουν ουσιαστικό ρόλο στην Java, το μετασχηματισμένο πρόγραμμα αποθηκεύεται στο αρχείο MyClass.java αφότου έχει σωθεί ένα αντίγραφο του αρχικού προγράμματος σε ένα αρχείο με όνομα MyClass.orig.

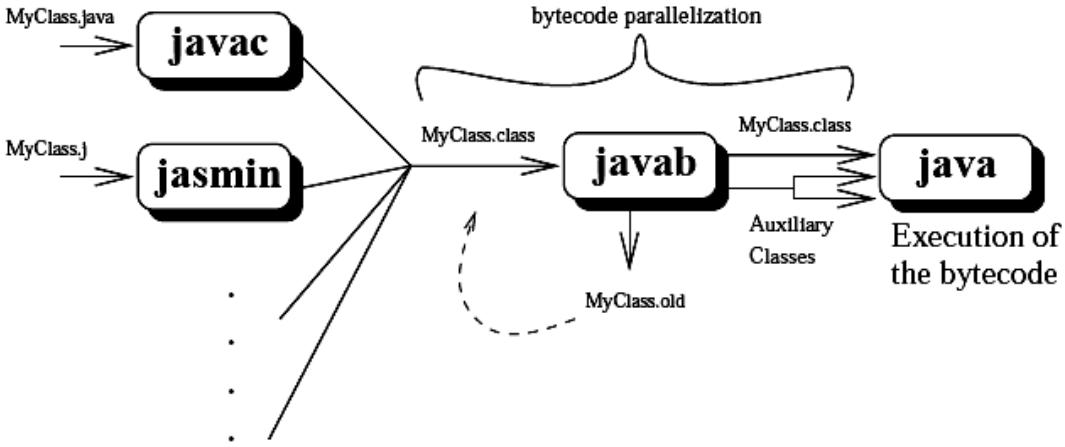
3.3.3 JavaB

Ο JavaB [1, 2] είναι ένα εργαλείο παραλληλισμού ενδιάμεσου κώδικα (bytecode) Java. Μπορεί να χρησιμοποιηθεί για να εκμεταλλευτεί τον υπονοούμενο παραλληλισμό (implicit parallelism) από τους βρόχους επανάληψης άμεσα σε επίπεδο ενδιάμεσου κώδικα (bytecode-level) χρησιμοποιώντας πολυνηματικό μηχανισμό του JVM (Java Virtual Machine). Επιπλέον, αυτό το εργαλείο παρέχει κάποια στοιχειώδη υποστήριξη για την αυτόματη ανίχνευση του υπονοούμενου παραλληλισμού βρόχων. Η αυτόματη εξαγωγή του υπονοούμενου παραλληλισμού σε επίπεδο ενδιάμεσου κώδικα μπορεί να γίνει ανεξάρτητα από το πρόγραμμα και την πλατφόρμα στην οποία ο ενδιάμεσος κώδικας τελικά θα τρέξει. Ο παραλληλοποιημένος ενδιάμεσος κώδικας παραμένει αρχιτεκτονικά ουδέτερος και μπορεί να επιτευχθεί βελτίωση (speedup) σε οποιαδήποτε πλατφόρμα που υποστηρίζει αληθινή παράλληλη εκτέλεση των νημάτων JVM.

Στην Εικόνα 3.3.4 απεικονίζεται η προσέγγιση για την αυτόματη εξαγωγή του παραλληλισμού από ένα ενδιάμεσο κώδικα JVM. Ένα αρχείο ενδιάμεσου κώδικα με όνομα MyClass.class είναι αυτό που χρησιμοποιείται ως είσοδος για το εργαλείο παραλληλισμού JavaB.

Ο μεταγλωττιστής μετασχηματίζει το πρόγραμμα εισόδου σε μια μορφή που χρησιμοποιεί τον πολυνηματικό μηχανισμό JVM για να εκμεταλλευτεί τον υπονοούμενο παραλληλισμό (implicit parallelism) από τους βρόχους επανάληψης άμεσα σε επίπεδο ενδιάμεσου κώδικα. Επειδή ο παραλληλισμός εκφράζεται σε ενδιάμεσο κώδικα JVM, το μετασχηματισμένο πρόγραμμα μπορεί ακόμα να εκτελεσθεί από οποιαδήποτε εφαρμογή του JVM και να επιτευχθεί βελτίωση

(speedup) σε οποιαδήποτε πλατφόρμα που υποστηρίζει αληθινή παράλληλη εκτέλεση των νημάτων JVM. Η αυτόματη εξαγωγή του υπονοούμενου παραλληλισμού σε επίπεδο ενδιάμεσου κώδικα μπορεί να γίνει ανεξάρτητα από το πρόγραμμα και την πλατφόρμα στην οποία ο ενδιάμεσος κώδικας τελικά θα τρέξει. Ως εκ τούτου, αυτή η προσέγγιση επιτρέπει τη βελτιστοποίηση οποιουδήποτε ενδιάμεσου κώδικα, είτε αυτός έχει παραχθεί από τον μεταγλωττιστή της Java javac, είτε αυτός έχει παραχθεί από ένα JVM assembler όπως ο Jasmin, είτε τον έχουμε κατεβάσει από το διαδίκτυο απευθείας σε αυτή τη μορφή.



Εικόνα 3.3.4 : Αυτόματος παραλληλισμός σε επίπεδο ενδιάμεσου κώδικα από το εργαλείο παραλληλισμού Javab.

Δεδομένου ότι τα ονόματα των αρχείων παίζουν ουσιαστικό ρόλο στην Java, το μετασχηματισμένο πρόγραμμα αποθηκεύεται στο αρχείο MyClass.class αφότου έχει σωθεί ένα αντίγραφο του αρχικού προγράμματος σε ένα αρχείο με όνομα MyClass.old. Μερικές βοηθητικές κλάσεις που υποστηρίζουν την παράλληλη εκτέλεση των βρόχων είναι πιθανόν επίσης να παραχθούν από το εργαλείο. Για να κρατηθεί σε περιορισμένα επίπεδα ο χρόνος, το πρωτότυπο στηρίζεται σε μια απλή αλλά λιγότερο ακριβή ανάλυση, παρά να κάνει κάποια προηγμένη ανάλυση εξάρτησης στοιχείων που κοστίζει σε χρόνο εκτέλεσης.

3.3.4 PGI C, C++ & FORTRAN Compiler

Το PGI Workstation Complete [29, 45] αποτελεί στην ουσία μια σουίτα η οποία προσφέρει μια σειρά από παράλληλους μεταγλωττιστές βελτιστοποίησης εφαρμογών για πολυπύρηνα συστήματα. Η σουίτα αυτή είναι καθαρά εμπορική (commercial) και προέρχεται από την εταιρία Portland Group με τις τιμές ανά license να κυμαίνονται από $600 μέχρι και $15.000. Οι μεταγλωττιστές που προσφέρονται καλύπτουν τις ανάγκες παραλληλοποίησης για τις γλώσσες προγραμματισμού : FORTRAN (FORTRAN 77, Fortran 90/95 και HPF), C++ και C. Το PGI Workstation Complete επίσης περιλαμβάνει OpenMP και MPI παράλληλα εργαλεία αποσφαλμάτωσης (PGDBG debugger) και μέτρησης της απόδοσης (PGPROF performance profiler). Όλοι οι μεταγλωττιστές που παρέχονται μπορούν να χρησιμοποιηθούν μόνο μέσω εντολών από κάποιο κέλυφος χωρίς να παρέχεται ούτε να υποστηρίζεται κάποιο γραφικό περιβάλλον ανάπτυξης εφαρμογών (IDE).

Σε γενικές γραμμές, η χρησιμοποίηση ενός μεταγλωττιστή PGI περιλαμβάνει τρία βήματα:

1. Δημιουργία του προγράμματος σε μια από τις γλώσσες προγραμματισμού που υποστηρίζονται (FORTRAN, C, C++). [Produce Source Code]
2. Μεταγλώττιση του προγράμματος από το προηγούμενο βήμα χρησιμοποιώντας τις κατάλληλες εντολές και υποδείξεις (Flags). [Compile Source Code]
3. Εκτέλεση, αποσφαλμάτωση ή μέτρηση απόδοσης του εκτελέσιμου αρχείου που δημιουργήθηκε στο βήμα 2. [Execute, debug, profile]

3.3.5 PROMIS Compiler

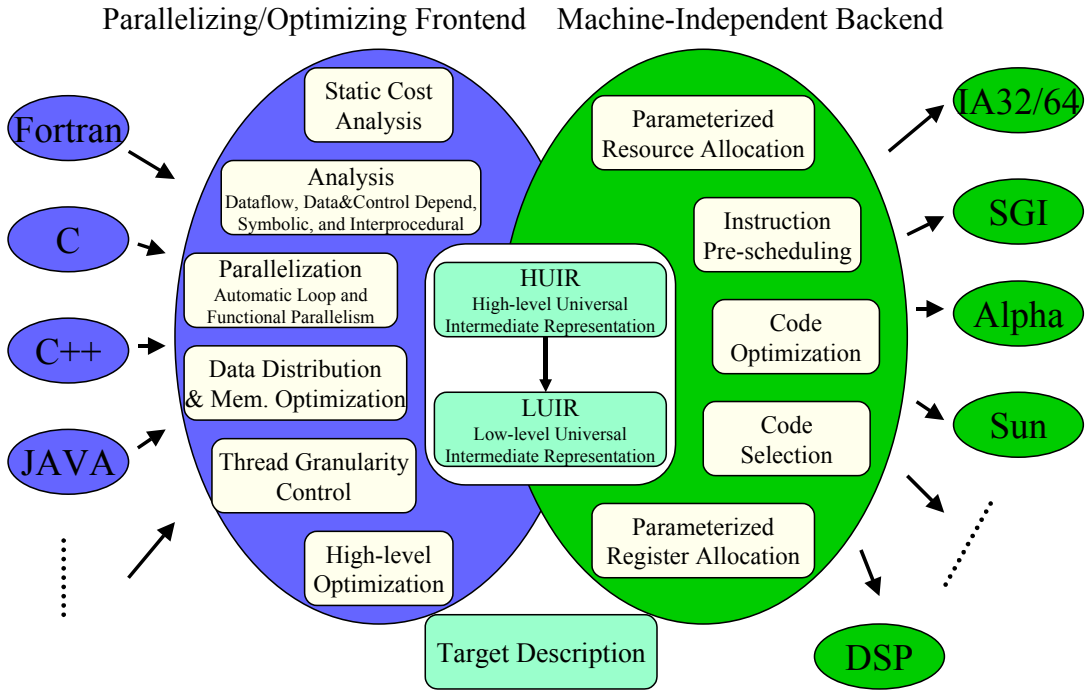
Ο PROMIS [15] είναι ένας υπερ-μεταγλωττιστής ο οποίος ενσωματώνει παραλληλοποίηση και βελτιστοποίηση τόσο σε επίπεδο κώδικα όσο και σε επίπεδο εντολών υποστηρίζοντας διάφορες γλώσσες (όπως C/C++, FORTRAN, JAVA κ.λπ.) και πολλαπλές αρχιτεκτονικές εντολών (instruction-set architectures) (όπως Intel,

AMD, ALPHA, SUN, x32, x64) χωρίς να αλλάζει καθόλου τον πυρήνα λειτουργίας του μεταγλωττιστή [Εικόνα 3.3.5]. Βελτιώνει σημαντικά την απόδοση και μπορεί πολύ εύκολα να αναδιαρθρώνεται για διάφορες διαμορφώσεις συστημάτων (embedded processors, clusters of workstations, κοινής ή κατανεμημένης μνήμης παράλληλα συστήματα). Ο σχεδιασμός του συνδυάζει παραλληλοποίηση σε επίπεδο βρόχων επαναλήψεων (loop-level parallelization) καθώς και σε επίπεδο εντολών (instruction-level parallelization – ILP), παράγοντας τα καλύτερα αποτελέσματα με τις υψηλότερες αποδόσεις και με μικρότερο κόστος μεταγλώττισης σε σχέση με τους παραδοσιακούς μεταγλωττιστές. Ο PROMIS είναι εφαρμογή ανοικτού κώδικα (opensource) και τρέχει μόνο σε Windows. Χρησιμοποιεί τις βιβλιοθήκες χαμηλού επιπέδου EDG C/C++ front ends (τις ίδιες βιβλιοθήκες χρησιμοποιούν και μεταγλωττιστές Suif, Intel's icc, PGI's pgicc και Sun's suncc) οι οποίες όμως δεν συμπεριλαμβάνονται με την εφαρμογή και η άδεια χρήσης τους (license) στοιχίζει περίπου $200.000.

Οι κύριοι στόχοι έρευνας και σχεδιασμού του πρωτοτύπου PROMIS είναι οι εξής:

- Ενσωμάτωση συμβολικής ανάλυσης καθώς και ανάλυσης δεικτών (symbolic and pointer analysis).
- Ενσωμάτωση παραλληλισμού υψηλού επιπέδου (για βρόχους και συναρτήσεις) και παραλληλισμού σε επίπεδο εντολών (ILP), όπου αυτό θα εφαρμόζεται μέσω της χρήσης μιας κοινής ενδιάμεσης αντιπροσώπευσης (intermediate representation – IR).
- Παραγωγή κώδικα που να μπορεί σχετικά εύκολα να τροποποιηθεί για μια ευρεία κατηγορία αρχιτεκτονικών (retargetable code) που υποστηρίζουν παραλληλοποίηση σε επίπεδο βρόχων επαναλήψεων (loop-level parallelization) καθώς και σε επίπεδο εντολών (ILP).
- Έρευνα για νέες προσεγγίσεις για βελτιστοποίηση κώδικα, κατανομή των καταχωριτών (register allocation) και σχηματισμό νημάτων για την επόμενη γενιά των πολυπύρηνων αρχιτεκτονικών.
- Σχεδιασμός και εφαρμογή μιας καθολικής ενδιάμεσης αντιπροσώπευσης και ενθυλάκωσή της μέσω μιας ισχυρή διασύνδεσης.

- Ανάλυση κόστους εκτέλεσης/ποιότητας απόδοσης (Trade-off) και παραγωγή κώδικα που να μπορεί άμεσα να τροποποιηθεί μεταξύ παραλληλισμού σε επίπεδο εντολών (ILP) ή σε επίπεδο νημάτων (thread-level parallelism) για μεγιστοποίηση της απόδοσης.



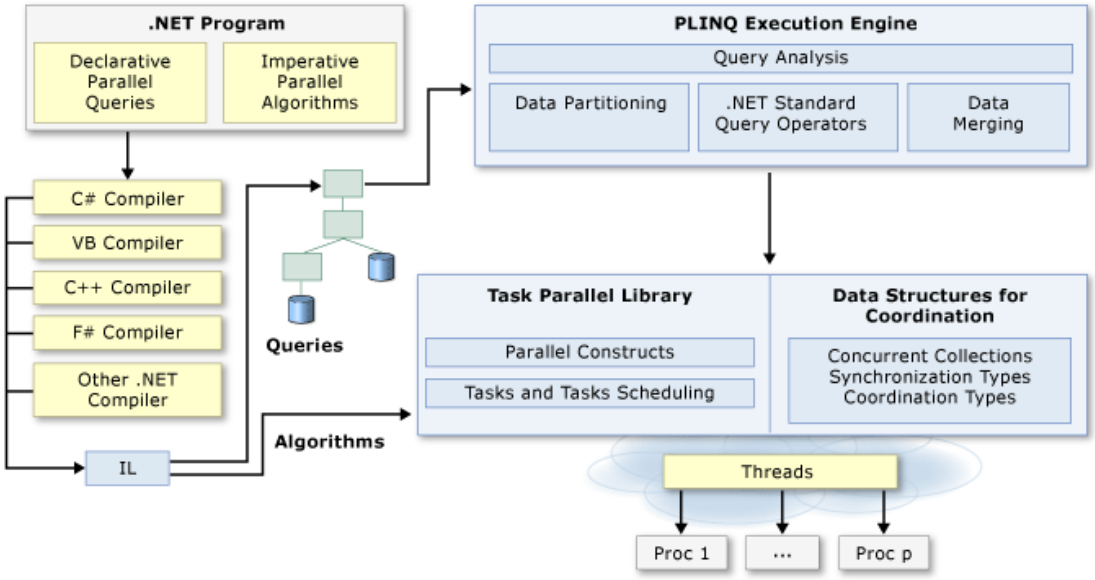
Εικόνα 3.3.5 : Η αρχιτεκτονική του πυρήνα λειτουργίας του μεταγλωττιστή PROMIS.

3.3.6 Visual Studio 2010 .Net FrameWork 4.0

Το Visual Studio 2010 και το .NET Framework 4 [26, 41] ενισχύουν την υποστήριξη για τον παράλληλο προγραμματισμό με την παροχή παράλληλων επεκτάσεων και εργαλείων. Υπάρχει πλήρης υποστήριξη για πολλαπλές γλώσσες προγραμματισμού και τους μεταγλωττιστές τους. Εκτός από την Visual Basic (VB) και C# το .NET Framework 4 καθιερώνει πλήρη υποστήριξη παραλληλισμού και για τις γλώσσες προγραμματισμού όπως ironpython, ironruby, F# και άλλους παρόμοιους μεταγλωττιστές της .NET. Σε αντίθεση με το .NET Framework 3.5, καλύπτει τόσο

τον συναρτησιακό-προγραμματισμό (functional-programming) όσο και τον αντικειμενοστρεφή-προγραμματισμό (object-oriented programming).

Η ακόλουθη απεικόνιση [Εικόνα 3.3.6] παρέχει μια υψηλού επιπέδου επισκόπηση της παράλληλης αρχιτεκτονικής προγραμματισμού στο .NET Framework 4.



Εικόνα 3.3.6 : Η παράλληλη αρχιτεκτονική προγραμματισμού στο NET Framework 4.

Το πλαίσιο των παράλληλων επεκτάσεων και εργαλείων έχει ταξινομηθεί σε τέσσερις βασικές περιοχές [41]:

1. Παράλληλη βιβλιοθήκη (Task Parallel Library – TPL) :

   Ο σκοπός της βιβλιοθήκης TPL είναι να κατασταστήσει τους υπεύθυνους για την ανάπτυξη παράλληλων εφαρμογών, προγραμματιστές, παραγωγικότερους μέσω την απλοποίησης της διαδικασίας προσθήκης παραλληλισμού και ταυτοχρονίας στις εφαρμογές. Η TPL δυναμικά αυξάνει κλιμακωτά την ταυτοχρονία έτσι ώστε να χρησιμοποιούνται αποδοτικότερα οι επεξεργαστές που είναι διαθέσιμοι στο σύστημα. Επιπλέον, χειρίζεται αποτελεσματικά τον καταμερισμό εργασίας, τον προγραμματισμό λειτουργίας και επικοινωνίας των νημάτων στο Thread Pool, την υποστήριξη

ακύρωσης εκτέλεσης των νημάτων και άλλες χαμηλού επιπέδου λεπτομέρειες. Με τη χρήση της βιβλιοθήκης TPL, μπορεί κανείς να μεγιστοποιήσει την απόδοση του κώδικά του εστιάζοντας μόνο στην εργασία που το πρόγραμμά του έχει σκοπό να επιτελεί και όχι στις λεπτομέρειες για σωστή επίτευξη παραλληλισμού. Ενσωματώνει παράλληλες εκδόσεις για τις εντολές επαναλήψεων For και ForEach στις οποίες με αποτελεσματικό καταμερισμό εργασίας για γρήγορη επικοινωνία επιτυγχάνεται αποδοτική εκτέλεση ασύγχρονων εργασιών.

2. Παράλληλη υποβολή ερωτήσεων σε βάσεις δεδομένων (Parallel Language-Integrated Query PLINQ) :

   Η παράλληλη υποβολή ερωτήσεων σε βάσεις δεδομένων (PLINQ) είναι μια παράλληλη υλοποίηση του LINQ (Language-Integrated Query) στα αντικείμενα. Η PLINQ υλοποιεί το πλήρες σύνολο των τυποποιημένων ερωτήσεων της LINQ ως μεθόδους επέκτασης έχοντας όμως κάποιες επιπλέων διαδικασίες που χρησιμοποιούνται κατά την παράλληλη εκτέλεση. Συνδυάζει έτσι την απλότητα της σύνταξης της LINQ με τη δύναμη του παράλληλου προγραμματισμού και για βάσεις δεδομένων. Ακριβώς όπως η στοχεύει στην κλιμακωτή αύξηση της ταυτοχρονίας έτσι ώστε να χρησιμοποιούνται αποδοτικότερα οι επεξεργαστές που είναι διαθέσιμοι στο σύστημα όταν εκτελούνται ερωτήσεις (queries) σε βάσεις δομένων.

3. Παράλληλες δομές δεδομένων (parallel data structures) :

   Παράλληλη υλοποίηση των κυριότερων δομών δεδομένων, όπως πίνακες και λίστες, στις οποίες αποφεύγεται η χρήση κλειδαριών (locks) αλλά αντ' αυτού χρησιμοποιούνται πιο αποδοτικές μέθοδοι οι οποίες υπόσχονται ασφαλή διαχείριση των αντικειμένων που αποτελούν τις δομές αυτές σε μια παράλληλη εκτέλεση με νήματα. Με αυτόν τον τρόπο βελτιώνεται σημαντικά η απόδοση σε περιπτώσεις σεναρίων όπου πολλαπλά νήματα προσθέτουν, αφαιρούν ή επεξεργάζονται στοιχεία σε μια συλλογή αντικειμένων που αποτελούν την κοινόχρηστη δομή. Επιπλέον ο προγραμματιστής δεν

χρειάζεται πια να ασχοληθεί ο ίδιος με το κλείδωμα των μεταβλητών για να εγγυηθεί τη σωστή εκτέλεση του κώδικά του θέλοντας ταυτόχρονα να συνδυάσει και αύξηση της απόδοσης.

4. Διαγνωστικά εργαλεία για παράλληλο κώδικα και εφαρμογές :
   Στα ήδη υπάρχοντα διαγνωστικά εργαλεία του Microsoft Visual Studio 2010 έχει προστεθεί η δυνατότητα να μπορούν να επιτελέσουν ανάλυση απόδοσης και εκτέλεση εφαρμογών στις οποίες εκτελούνται πολλαπλά νήματα. Για σκοπούς αποσφαλμάτωσης υπάρχουν απεικονίσεις που παρουσιάζουν γραφικά τα νήματα με όλες τις κλήσεις τους προς τη σωρό του προγράμματος (program stack thread calls) καθώς και την υφιστάμενη κατάσταση κάθε νήματος και την εργασία που του αναλογεί για να εκτελέσει. Ο προγραμματιστής έτσι είναι σε θέση να κατανοήσει με μεγαλύτερη ευκολία την συμπεριφορά της εφαρμογής κατά τον χρόνο εκτέλεσής της και να διορθώσει τυχόν παραλήψεις και λάθη. Υπάρχει ακόμα ο Concurrency Visualizer όπου με τις απεικονίσεις του δίνει τη δυνατότητα στον προγραμματιστή να δει πώς η πολυνηματική εφαρμογή του αλληλεπιδρά με τον εαυτό της, το υλικό, το λειτουργικό σύστημα αλλά και με επιπλέον διαδικασίες που εκτελούνται παράλληλα στο σύστημα. Οι χρονικές σχέσεις μεταξύ των νημάτων στην εφαρμογή αλλά και του συστήματος συνολικά μπορούν να παρουσιαστούν είτε με γραφικό τρόπο, είτε σε μορφή απλού κειμένου. Χρησιμοποιώντας έτσι κάποιος τον Concurrency Visualizer είναι σε θέση να αναγνωρίσει προβλήματα συμφόρησης στην επίδοση (performance bottlenecks), μη επαρκή εκμετάλλευση των επεξεργαστών (CPU underutilization), καθυστερήσεις λόγο συγχρονισμού εν αναμονής κάποιας διεργασίας εισόδου/εξόδου (I/O synchronization delays) και άλλα.

Αυτά τα χαρακτηριστικά γνωρίσματα και εργαλεία απλοποιούν την παράλληλη ανάπτυξη έτσι ώστε ο οποιοσδήποτε να μπορεί να γράψει αποδοτικό, λεπτομερή, και επεκτάσιμο παράλληλο κώδικα με έναν φυσικό ιδιωματισμό χωρίς να πρέπει να γνωρίζει πολλές λεπτομέρειες.

Τα κύρια κριτήρια, όπως η επικοινωνία και ο συγχρονισμός των δευτερευόντων λειτουργιών, θεωρούνται ως τα μεγαλύτερα εμπόδια για να μπορεί να επιτευχθεί μια αποδοτική παράλληλη εκτέλεση ενός προγράμματος. Το .NET Framework 4 όμως υπόσχεται ότι μέσα από την παράλληλη τεχνολογία βιβλιοθηκών επιτρέπει στους υπεύθυνους για την ανάπτυξη παράλληλων προγραμμάτων προγραμματιστές να καθορίσουν την ταυτοχρονία και την και την ασύγχρονη εκτέλεση χωρίς να πρέπει κατ' ανάγκη να δουλέψουν με threads, locks ή thread pools.

3.4 Ανασκόπηση επόμενου κεφαλαίου

Στο επόμενο κεφάλαιο [Κεφάλαιο 4] θα παρουσιάσουμε την όλη μεθοδολογία και προτεινόμενη λύση. Θα περιγράψουμε τον τρόπο λειτουργίας της παράλληλης βιβλιοθήκης σε γλώσσα Java που έχει υλοποιηθεί και θα αναφερθούμε στις κύριες μεθόδους από τις οποίες αποτελείται. Στην συνέχεια θα δούμε πώς έχει υλοποιηθεί ο μεταγλωττιστής αναδόμησης (parser – αναλυτής) του πηγαίου κώδικα (source code) προγραμμάτων γραμμένα σε γλώσσα προγραμματισμού Java και πώς αυτός λειτουργεί πετυχαίνοντας αυτόματη μετατροπή του σειριακού κώδικα σε παράλληλο με βάση την προηγούμενη βιβλιοθήκη. Τέλος θα παρουσιάσουμε τον τρόπο λειτουργίας του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα στο οποίο ενσωματώσαμε τον μεταγλωττιστή αναδόμησης βλέποντας ένα – ένα τα παράθυρα από τα οποία αποτελείται.

Κεφάλαιο 4

Παρουσίαση Μεθοδολογίας

4.1 Εισαγωγή

Με την ανάγκη δημιουργίας υψηλού επιπέδου απόδοσης για μια ποικιλία εφαρμογών που πρέπει να καλύπτουν τόσο τις ανάγκες των απλών καταναλωτών αλλά και των επιχειρήσεων και του επιστημονικού τομέα περάσαμε στη γενιά των παράλληλων υπολογιστών και του παράλληλου υπολογισμού. Νέες αρχιτεκτονικές για πολυπύρηνα συστήματα άρχισαν να δημιουργούνται και καινούργια προγραμματιστικά μοντέλα άρχισαν να ξεπροβάλλουν για να καλύψουν τις νέες τάσεις της τεχνολογίας, οι οποίες τώρα απευθύνοντε σε ένα ευρύ κοινό. Καινούργιοι ορίζοντες έτσι άνοιξαν στο χώρο της Πληροφορικής δημιουργώντας την ανάγκη για νέους τρόπους προσέγγισης του παράλληλου υπολογισμού. Οι προσεγγίσεις αυτές είναι ο παράλληλος προγραμματισμός και ο αυτόματος παραλληλισμός.

Όπως αναφέραμε και πιο πριν [Κεφάλαιο 1] στόχος αυτής της διπλωματικής εργασίας είναι να προταθεί μια μεθοδολογία με κύριο σκοπό να αφαιρέσουμε από τον προγραμματιστή όσο το δυνατόν περισσότερο την ανάγκη να ασχολείται με τις χαμηλού επιπέδου λεπτομέρειες (threads, thread-pools, locks, synchronization) για την δημιουργία μιας σωστής και αποδοτικής παράλληλης εφαρμογής. Για να το πετύχουμε αυτό δημιουργήσαμε μια παράλληλη βιβλιοθήκη σε γλώσσα Java καθώς και ένα ήμι-αυτόματο μεταφραστή σειριακού κώδικα Java σε παράλληλο που να κάνει χρήση της προηγούμενης βιβλιοθήκης. Με τις δυο υλοποιήσεις αυτές θα είμαστε σε θέση να καλύψουμε και τους δυο τρόπους προσέγγισης του παράλληλου υπολογισμού, παράλληλο προγραμματισμό και ο αυτόματο παραλληλισμό. Δίνουμε με αυτό τον τρόπο την δυνατότητα στον προγραμματιστή να μεγιστοποιήσει την απόδοση του κώδικά του εστιάζοντας μόνο στην εργασία που το πρόγραμμά του έχει σκοπό να επιτελεί και όχι στις λεπτομέρειες για σωστή επίτευξη παραλληλισμού. Καταστούμε έτσι τους υπεύθυνους για την ανάπτυξη παράλληλων εφαρμογών, προγραμματιστές, παραγωγικότερους μέσω την απλοποίησης της διαδικασίας προσθήκης παραλληλισμού και ταυτοχρονίας στις εφαρμογές τους.

Στη συνέχεια του υπόλοιπου κεφαλαίου θα προσπαθήσουμε να περιγράψουμε το τι ακριβώς υποστηρίζει η βιβλιοθήκη παραλληλισμού που δημιουργήσαμε καθώς και τον τρόπο με τον οποίο επιτυγχάνεται παράλληλη εκτέλεση σε ένα πολυπύρηνο σύστημα κάνοντας χρήση των παράλληλων μεθόδων που υποστηρίζει. Θα δούμε επίσης τον τρόπο υλοποίησης και λειτουργίας του αυτόματου μεταφραστή ο οποίος με βάση κάποιες απλές υποδείξεις από μέρους του προγραμματιστή αυτόματα μεταφράζει σειριακά κομμάτια κώδικα σε παράλληλα κάνοντας χρήση των μεθόδων της προηγούμενης βιβλιοθήκης. Τέλος θα παρουσιάσουμε το γραφικό εργαλείο αυτόματης μετάφρασης κώδικα που δημιουργήθηκε για να ενσωματώσει τον αυτόματο μεταφραστή σε συνδυασμό με ένα εργαλείο μέτρησης της βελτίωσης της απόδοσης (speedup BenchMark) που επιτυγχάνεται μεταξύ του αρχικού σειριακού κώδικα και του παράλληλου κώδικα που δημιουργείται όταν αυτά τρέχουν σε ένα πολυπύρηνο σύστημα.

4.2 Βιβλιοθήκη παράλληλου προγραμματισμού JACON

Η βιβλιοθήκη παράλληλου προγραμματισμού JACON (JAva-CONcorrency) δημιουργήθηκε για να βοηθήσει στη μείωση του χρόνου ανάπτυξης μιας εφαρμογής εξοικονομώντας χρόνο από τον προγραμματιστή έτσι ώστε να μην χρειάζεται ο ίδιος να μπαίνει στον κόπο για να γράψει περίπλοκα πολυνηματικά κομμάτια κώδικα για να βελτιώσει το χρόνο εκτέλεσης της εφαρμογής του. Η βιβλιοθήκη JACON μπορεί να μειώσει σημαντικά το χρόνο που ξοδεύεται για την αποσφαλμάτωση κώδικα με λάθη (bugs). Είναι γνωστό ότι οι πολυνηματικές εφαρμογές παρουσιάζουν αρκετά συχνά ατέλειες – λάθη που είναι δύσκολο να απομονωθούν και να αναπαραχθούν. Εφαρμογές με τέτοιου είδους προβλήματα όταν εκτελούνται σε συστήματα που έχουν μόνο ένα επεξεργαστή και μπορούν να διαχειριστούν μόνο ένα νήμα ταυτόχρονα, συχνά ο κώδικας μπορεί να τρέξει χωρίς να παρουσιαστεί κανένα πρόβλημα . Όταν όμως οι εφαρμογές αυτές εκτελούνται σε πραγματικά παράλληλα συστήματα με πολυπύρηνους επεξεργαστές τα λάθη και οι ανακρίβειες που παρουσιάζουν τείνουν να κάνουν την εμφάνισή τους πιο συχνά και να έχουν στο τέλος καταστροφικά αποτελέσματα. Η αποσφαλμάτωση λαθών που έχουν να κάνουν με το συγχρονισμό και την επικοινωνία νημάτων σε μια πολυνηματική εφαρμογή είναι μια πολύ χρονοβόρα και δύσκολη διαδικασία ακόμα και για προγραμματιστές που είναι έμπειροι πάνω σε αυτού του είδους θέματα. Χρησιμοποιώντας έτσι την βιβλιοθήκη JACON, ο υπεύθυνος για την ανάπτυξη πολυνηματικών εφαρμογών μπορεί να είναι βέβαιος ότι πλέον δεν είναι αναγκασμένος να καθορίσει ο ίδιος την ταυτοχρονία και την ασύγχρονη εκτέλεση της εφαρμογής του και ότι δεν χρειάζεται να ασχοληθεί με threads, thread pools, locks ή barriers [5, 6, 7, 23, 36]. Αφαιρούμε έτσι από τον προγραμματιστή την ανάγκη να καταπιαστεί με τις χαμηλού επιπέδου λεπτομέρειες της δημιουργίας και χρήσης των νημάτων καθώς και στο πώς να εφαρμόσει ασφαλή ενδοεπικοινωνία νημάτων. Αυτό απελευθερώνει τον υπεύθυνο για την ανάπτυξη παράλληλων εφαρμογών αφήνοντάς τον να εστιάσει περισσότερο στο πραγματικό πρόβλημα/σημείο το οποίο επιθυμεί να παραλληλοποιήσει/βελτιώσει, παρά στο πώς να κάνει την πολυνηματική εργασία να δουλέψει σωστά.

4.2.1 Γενική περιγραφή και πληροφορίες

Για την υλοποίηση της βιβλιοθήκης παραλληλισμού JACON χρησιμοποιήθηκε η γλώσσα προγραμματισμού Java [8, 36] και το περιβάλλον ανάπτυξης εφαρμογών (IDE) NetBeans IDE 6.9.1 [39]. Ο λόγος για τον οποίο επιλέξαμε να υλοποιήσουμε την βιβλιοθήκη σε Java είναι πρώτον για σκοπούς φορητότητας (portability) και δεύτερον για σκοπούς συμβατότητας (compatibility).

Είναι γνωστό ότι η Java είναι μια ουδέτερης αρχιτεκτονικής γλώσσα. Ο μεταγλωττιστής της Java παράγει ενδιάμεσο κώδικα, ο οποίος χρησιμοποιείται μόνο από την εικονική μηχανή της Java (Java Virtual Machine), και όχι κώδικα μηχανής, ο οποίος αναγνωρίζεται μόνο από το συγκεκριμένο σύστημα και αρχιτεκτονική πάνω στην οποία παράχθηκε. Έτσι ο ενδιάμεσος αυτός κώδικας (JVM bytecode) που δημιουργείται μπορεί να εκτελεστεί και να τρέξει σε οποιοδήποτε υπολογιστικό σύστημα οποιασδήποτε αρχιτεκτονικής υπό την προϋπόθεση ότι στο σύστημα αυτό είναι εγκατεστημένο το Java Virtual Machine. Αφού η βιβλιοθήκη αναπτύχθηκε στα πλαίσια για να καλύψει την ανάγκη προσέγγισης του παράλληλου υπολογισμού μέσω του παράλληλου προγραμματισμού θα ήταν πολύ σημαντικό να μπορεί να χρησιμοποιείται από τον προγραμματιστή σε οποιοδήποτε υπολογιστικό σύστημα ανεξαρτήτου λειτουργικού συστήματος και αρχιτεκτονικής.

Αφού η βιβλιοθήκη JACON προοριζόταν για παράλληλο προγραμματισμό εφαρμογών σε γλώσσα Java τότε ήταν επιβεβλημένο να την υλοποιήσουμε στην ίδια γλώσσα πάνω στην οποία στόχευε να λειτουργεί απαλείφοντας έτσι κάθε πιθανότητα να προκύψει κάποιο πρόβλημα ασυμβατότητας.

Η μεθοδολογία για να χρησιμοποιήσει κάποιος την βιβλιοθήκη JACON είναι αρκετά διαφορετική σε σχέση με τον τρόπο που κανονικά ένας προγραμματιστής θα διαχειριστεί τα νήματα για να γράψει μια πολυνηματική εφαρμογή. Κατά τη συγγραφή μιας πολύπλοκης εφαρμογής, ο προγραμματιστής παραδοσιακά σκέφτεται με όρους όπως νήματα (threads), κρίσιμα τμήματα (critical sections), κλειδαριές (locks/mutexes), σηματοφόρους (semaphores), ατομικές πράξεις (atomic operations)

και ούτω καθεξής. Πολλές πολυνηματικές βιβλιοθήκες (όπως εκείνες που παρέχονται από το API της Java και τις C++) απλά τυλίγουν αυτά τα κατασκευάσματα/όρους σε κατηγορίες εργαλείων και τα "πλασάρουν" στον προγραμματιστή για να τα χρησιμοποιήσει έχοντας λίγο πιο ελκυστικό τρόπο διαχείρισης. Η βιβλιοθήκη JACON από την άλλη έρχεται να αποκρύψει από τον προγραμματιστή τέτοιου είδους κατασκευάσματα κάνοντας μια διαφορετική διαχείριση των όρων αυτών. Η βιβλιοθήκη JACON επικεντρώνεται περισσότερο στον προσδιορισμό και διάσπαση του σειριακού υπολογισμού σε μικρότερες εργασίες/στόχους (tasks). Ο προγραμματιστής έτσι πρέπει να δείχνει με τον κώδικά του απλά τις εργασίες/στόχους που πρέπει να εκτελεστούν παράλληλα και η βιβλιοθήκη με τη σειρά της είναι αυτή που θα αναλάβει τον ρόλο του καταμερισμού των εργασιών αυτών στις διάφορες μονάδες εκτέλεσής (threads) καθώς και την ορθότητα της εκτέλεσης τους (locks/mutexes, barriers, synchronization). Η βιβλιοθήκη JACON έτσι, είναι εκείνη που διαχειρίζεται τις χαμηλού επιπέδου αποφάσεις για το πότε και πόσα νήματα θα δημιουργηθούν, πώς αυτά θα αλληλεπιδρούν ακίνδυνα μεταξύ τους και θα διαχειρίζονται τα κοινά μεταξύ τους δεδομένα. Για να καταλάβουμε όμως πώς όλα αυτά λειτουργούν θα περιγράψουμε στη συνέχεια τις βασικές μεθόδους και κλάσεις τις οποίες έχει στην διάθεσή του ένας προγραμματιστής που χρησιμοποιεί αυτή την βιβλιοθήκη και πώς αυτές επιτυγχάνουν τον σκοπό τους.

4.2.2 Ανάλυση λειτουργίας κύριων μεθόδων

Στην βιβλιοθήκη JACON υπάρχουν δύο βασικές κλάσεις τις οποίες έχει στην διάθεση του για να χρησιμοποιήσει κάποιος προγραμματιστής. Οι κλάσεις/αρχεία αυτά είναι το *Parallel.java* [Παράρτημα Α-2] και *ParallelAtomic.java* [Παράρτημα Α-5].

Το αρχείο *Parallel.java* περιέχει την μέθοδο *forLoop* η οποία στην ουσία αποτελεί την παράλληλη έκδοση ενός κανονικού παραδοσιακού βρόχου επανάληψης for για τη γλώσσα προγραμματισμού Java. Η μέθοδος *forLoop* μπορεί να χρησιμοποιηθεί

από τον προγραμματιστή σε δύο μορφές, όπου σε κάθε περίπτωση παίρνει διαφορετικές παραμέτρους/δεδομένα εισόδου (arguments). Οι μορφές οι οποίες υπάρχουν και ο τρόπος λειτουργίας τους είναι οι εξής :

- *public static void forLoop (final int start, final String condition,*
  final int end, final String action,
  final IForTask loopBody)
  throws CancelException

  Αποτελεί όπως είπαμε την παράλληλη έκδοση ενός κανονικού παραδοσιακού βρόχου επανάληψης for για τη γλώσσα προγραμματισμού Java. Προσομοιώνει στην ουσία ένα for statement που ξεκινά με δείκτη (index) *i = start* και εκτελεί οτιδήποτε διεργασίες υπάρχουν στο block που ακολουθεί μετά την δήλωση του for, *loopBody,* όσο ισχύει η συνθήκη *condition* (<, >, <=, >=, !=) μεταξύ του δείκτη *start* και *end* αυξάνοντας σε κάθε τέλος της εκτέλεσης με τρόπο *action* (++, --) τον δείκτη *end*. Το μέγεθος στόχου (grain size) τίθεται αυτόματα σε ένα (1), υπονοώντας ότι η εργασία που θα έχει κάθε νήμα να εκτελέσει στην συνέχεια είναι ίσο με μια εκτέλεση του *loopBody*. Ως εκ τούτου, δημιουργείται ένας στόχος (task) για κάθε αύξηση του δείκτη *i = start*, δηλαδή [*(end – start) / 1*] στόχοι, καθώς και ένα threadpool, με αριθμό νημάτων ίσο με τους διαθέσιμους πυρήνες που υπάρχουν στο υπολογιστικό σύστημα, για να διεκπεραιώσουν όλους τους στόχους. Πολλές φορές είναι δυνατόν ο προγραμματιστής να μπορεί να βελτιώσει την επίδοση προσαρμόζοντας σωστά ο ίδιος το μέγεθος στόχου (grain size). Για να μπορεί να το κάνει όμως αυτό θα πρέπει να χρησιμοποιήσει την μέθοδο που ακολουθεί πιο κάτω αφού στην ουσία η μέθοδος που περιγράφουμε τώρα απλά καλεί την επόμενη μέθοδο με *grainSize* ίσο με ένα (1). Η μέθοδος *forLoop* συλλέγει οποιοδήποτε λάθος/εξαίρεση (exception) προκύψει κατά την διάρκεια εκτέλεσης των στόχων (tasks) και αν αυτό συμβεί, τότε ακυρώνει όλες τις εργασίες που εκτελούνται εκείνη την στιγμή ρίχνοντας στο τέλος ένα *CancelException*. Με το *CancelException* ο προγραμματιστής είναι υπεύθυνος να αποφασίσει τι θα γίνει με τις υπόλοιπες εργασίες που αναμένουν προς εκτέλεση.

Παραμέτροι:

start – Ο πρώτος δείκτης από τον οποίο ξεκινά ο βρόχος επανάληψης.

condition – Η συνθήκη που πρέπει να ισχύει μεταξύ των δεικτών *start* και *end* για να συνεχίζεται η επαναληπτική εκτέλεση.

end – Ο τελευταίος δείκτης που σημαίνει το τέλος της επαναληπτικής εκτέλεσης (συμπεριλαμβανομένου ή μη αναλόγως του *condition*)

action – Η ενέργεια που εκτελείται πάνω στον δείκτη *start* μετά το πέρας της εκτέλεσης κάθε επανάληψης.

loopBody – Η δουλειά/διεργασία που πρέπει να εκτελεστεί σε κάθε επανάληψη.

- *public static void forLoop (final int start, final String condition,*
  final int end, final String action,
  final int grainSize, final IForTask loopBody)
  throws CancelException

Αποτελεί όπως είπαμε την παράλληλη έκδοση ενός κανονικού παραδοσιακού βρόχου επανάληψης for για τη γλώσσα προγραμματισμού Java. Προσομοιώνει στην ουσία ένα for statement που ξεκινά με δείκτη (index) *i = start* και εκτελεί οτιδήποτε διεργασίες υπάρχουν στο block που ακολουθεί μετά την δήλωση του for, *loopBody,* όσο ισχύει η συνθήκη *condition* (<, >, <=, >=, !=) μεταξύ του δείκτη *start* και *end* αυξάνοντας σε κάθε τέλος της εκτέλεσης με τρόπο *action* (++, --) τον δείκτη *end*. Πολλές φορές είναι δυνατόν ο προγραμματιστής να μπορεί να βελτιώσει την επίδοση προσαρμόζοντας σωστά ο ίδιος το μέγεθος στόχου (grain size). Αυτό γίνεται από μέρους του ορίζοντας ρητά το μέγεθος στόχου (grain size), υπονοώντας ότι η εργασία που θα έχει κάθε νήμα να εκτελέσει στην συνέχεια είναι ίση με *grainSize* εκτελέσεις του *loopBody*. Ως εκ τούτου, δημιουργούνται στόχοι (tasks) μεγέθους *grainSize* για κάθε αύξηση του δείκτη *i = start*, δηλαδή [*(end – start) / grainSize*] στόχοι, καθώς και ένα threadpool, με αριθμό νημάτων ίσο με τους διαθέσιμους πυρήνες που υπάρχουν στο υπολογιστικό σύστημα, για να διεκπεραιώσουν όλους τους στόχους. Η μέθοδος *forLoop*

συλλέγει οποιοδήποτε λάθος/εξαίρεση (exception) προκύψει κατά την διάρκεια εκτέλεσης των στόχων (tasks) και αν αυτό συμβεί, τότε ακυρώνει όλες τις εργασίες που εκτελούνται εκείνη την στιγμή ρίχνοντας στο τέλος ένα *CancelException*. Με το *CancelException* ο προγραμματιστής είναι υπεύθυνος να αποφασίσει τι θα γίνει με τις υπόλοιπες εργασίες που αναμένουν προς εκτέλεση.

Παραμέτροι:

start – Ο πρώτος δείκτης από τον οποίο ξεκινά ο βρόχος επανάληψης.

condition – Η συνθήκη που πρέπει να ισχύει μεταξύ των δεικτών *start* και *end* για να συνεχίζεται η επαναληπτική εκτέλεση.

end – Ο τελευταίος δείκτης που σημαίνει το τέλος της επαναληπτικής εκτέλεσης (συμπεριλαμβανομένου ή μη αναλόγως του *condition*)

action – Η ενέργεια που εκτελείται πάνω στον δείκτη *start* μετά το πέρας της εκτέλεσης κάθε επανάληψης.

grainSize – Ένας λογικός αριθμός επαναλήψεων εκτέλεσης του *loopBody* μέσα σε κάθε στόχο (task).

loopBody – Η δουλειά/διεργασία που πρέπει να εκτελεστεί σε κάθε επανάληψη.

Το αρχείο *ParallelAtomic.java* περιέχει δύο κύριες μεθόδους, την *forLoopAtomic* και την *reduce*. Η *forLoopAtomic* αποτελεί την παράλληλη έκδοση ενός κανονικού παραδοσιακού βρόχου επανάληψης for για τη γλώσσα προγραμματισμού Java με την διαφορά όμως ότι αυτή η μέθοδος υλοποιεί την ύπαρξη κάποιου μηχανισμού που να υποστηρίζει ταυτόχρονη πρόσβαση σε έναν κοινό πόρο/μεταβλητή για σκοπούς εγγραφής. Η υλοποίηση και αποτελεσματική λειτουργία αυτού του μηχανισμού επιτυγχάνεται με τη χρήση της μεθόδου *reduce,* η οποία μπορεί όμως να χρησιμοποιηθεί και ανεξάρτητη. Η μέθοδος *reduce* συνενώνει τιμές του ίδιου τύπου και υπολογίζει μια τελική τιμή (reduction) βάση μιας συνάρτησης συνένωσης.

Η μέθοδος *forLoopAtomic* μπορεί να χρησιμοποιηθεί από τον προγραμματιστή σε μια μόνο μορφή όπου παίρνει συγκεκριμένες παραμέτρους/δεδομένα εισόδου (arguments) και ο τρόπος λειτουργίας της είναι ο εξής :

- *public static double forLoopAtomic (final int start,*
  final int end,
  final String action,
  final IForAtomicTask loopBody)
  throws CancelException

  Αποτελεί όπως είπαμε την παράλληλη έκδοση ενός κανονικού βρόχου επανάληψης for για τη γλώσσα προγραμματισμού Java με την διαφορά όμως ότι αυτή η μέθοδος υλοποιεί την ύπαρξη κάποιου μηχανισμού που να υποστηρίζει ταυτόχρονη πρόσβαση σε έναν κοινό πόρο/μεταβλητή για σκοπούς εγγραφής. Προσομοιώνει στην ουσία ένα for statement που ξεκινά με δείκτη (index) *i = start* και εκτελεί οτιδήποτε διεργασίες υπάρχουν στο block που ακολουθεί μετά την δήλωση του for, *loopBody,* όσο ισχύει η συνθήκη *start* < *end* αυξάνοντας σε κάθε τέλος της εκτέλεσης τον δείκτη *end* κατά ένα. Το μέγεθος στόχου (grain size) τίθεται αυτόματα ίσο με τον λόγο του συνολικού αριθμού επαναλήψεων του βρόχου προς τον αριθμό των επεξεργαστών που διαθέτει το σύστημα στο οποίο θα τρέξει. Ο καταμερισμός της εργασίας που θα έχει κάθε νήμα να εκτελέσει γίνεται με βάση των διαθέσιμων επεξεργαστών του συστήματος κάθε φορά δυναμικά.. Ως εκ τούτου, δημιουργούνται [(end-start) / αριθμό διαθέσιμων επεξεργαστών] στόχοι προς εκτέλεση. Η μέθοδος *forLoopAtomic* υλοποιεί τον μηχανισμό ταυτόχρονης πρόσβασης σε έναν κοινό πόρο/μεταβλητή δίνοντας την δυνατότητα να μπορούν να εκτελούνται ατομικά πράξεις τύπου *action* (+=, -=, *=, /=) σε αυτήν από πολλά νήματα κατά την διάρκεια μιας παράλληλης εκτέλεσης. Αυτό επιτυγχάνεται κάνοντας χρήση της μεθόδου reduce η χρήση της οποίας θα εξηγηθεί στην συνέχεια. Μετά το πέρας της εκτέλεσης του επαναληπτικού βρόγχου η τιμή του κοινού πόρου/μεταβλητής επιστρέφεται από την μέθοδο σαν αντικείμενο τύπου *Double*. Η μέθοδος επίσης συλλέγει οποιοδήποτε λάθος/εξαίρεση (exception) προκύψει κατά την διάρκεια

εκτέλεσης των στόχων (tasks) και αν αυτό συμβεί, τότε ακυρώνει όλες τις εργασίες που εκτελούνται εκείνη την στιγμή ρίχνοντας στο τέλος ένα *CancelException*. Με το *CancelException* ο προγραμματιστής είναι υπεύθυνος να αποφασίσει τι θα γίνει με τις υπόλοιπες εργασίες που αναμένουν προς εκτέλεση.

Παραμέτροι:

start – Ο πρώτος δείκτης από τον οποίο ξεκινά ο βρόχος επανάληψης, ο οποίος αυξάνεται κατά ένα μέχρι να γίνει ίσος με τον δείκτη *end*.

end – Ο τελευταίος δείκτης που σημαίνει το τέλος της επαναληπτικής εκτέλεσης (μη συμπεριλαμβανομένου της τιμής του, ο βρόχος εκτελείται όσο ο δείκτης *start* είναι μικρότερος από το δείκτη *end*)

action – Η ενέργεια μείωσης (reduction operation) που αναλαμβάνει ο μηχανισμός ταυτόχρονης πρόσβασης να εκτελεί ατομικά πάνω στον κοινό πόρο/μεταβλητή που θέλουμε να προστατεύσουμε.

loopBody – Η δουλειά/διεργασία που πρέπει να εκτελεστεί σε κάθε επανάληψη.

Η μέθοδος *reduce* μπορεί να χρησιμοποιηθεί από τον προγραμματιστή σε δύο μορφές, όπου σε κάθε περίπτωση παίρνει διαφορετικές παραμέτρους/δεδομένα εισόδου (arguments). Οι μορφές οι οποίες υπάρχουν και ο τρόπος λειτουργίας τους είναι οι εξής :

- *public static <T> T reduce (final int start, final int end, final T initialValue,*
  final IReduceTask<T> loopBody,
  final ICombineTask<T> combineMethod)
  throws CancelException

Η μέθοδος *reduce* συνενώνει τιμές του ίδιου τύπου και υπολογίζει μια τελική τιμή (reduction) βάσει μιας συνάρτησης συνένωσης. Το *IReduceTask<T> loopBody* παίρνει έναν ακέραιο ο οποίος ανήκει μεταξύ *start* και *end* επιστρέφοντας ένα στοιχείο τύπου T, όπου το T μπορεί να αντιπροσωπεύει οποιοδήποτε τύπο ανήκει και αναγνωρίζεται στην γλώσσα προγραμματισμού

Java. Η *ICombineTask<T> combineMethod* παίρνει δύο τέτοια στοιχεία, που επιστρέφονται από το προηγούμενο, και τα συνδυάζει/συνενώνει (combine) μεταξύ τους. Η *reduce* αποφεύγει να χρησιμοποιήσει κλειδαριές (locks) διαιρώντας το πρόβλημα σε ένα δυαδικό δέντρο στόχων (binary task tree) όπου η κάθε διεργασία έχει την δική της τοπική μεταβλητή την οποία μπορεί και μπορεί να ενημερώνει χωρίς να υπάρχει φόβος να εμφανιστούν προβλήματα όπως race-conditions (πολλαπλές προσβάσεις σε έναν κοινό πόρο/μεταβλητή για σκοπούς εγγραφής χωρίς την ύπαρξη κάποιου μηχανισμού που να υποστηρίζει ταυτόχρονη πρόσβαση), deadlocks (αδιέξοδο κατά το οποίο δύο ή περισσότερες ανταγωνιστικές εργασίες (threads) περιμένουν η μια την άλλη να τελειώσει, χωρίς όμως αυτό να γίνεται από καμία) και livelocks (κατάσταση αδιεξόδου κατά την οποία πολλαπλές ανταγωνιστικές εργασίες (threads) αλλάζουν συνεχώς κατάσταση αλλά ποτέ δεν καταλήγουν σε σημείο όπου να μπορεί κάποια από αυτές να προχωρήσει). Το μέγεθος στόχου (grain size) τίθεται αυτόματα σε δύο (2), υπονοώντας ότι η εργασία που θα έχει κάθε νήμα να εκτελέσει στην συνέχεια είναι ίσο με δύο εκτελέσεις του *IReduceTask<T> loopBody* για να μπορεί και η *ICombineTask<T> combineMethod* με τη σειρά της να πάρει δύο τέτοια στοιχεία. Πολλές φορές είναι δυνατόν ο προγραμματιστής να μπορεί να βελτιώσει την επίδοση προσαρμόζοντας σωστά ο ίδιος το μέγεθος στόχου (grain size). Για να μπορεί να το κάνει όμως αυτό θα πρέπει να χρησιμοποιήσει την μέθοδο που ακολουθεί πιο κάτω αφού στην ουσία η μέθοδος που περιγράφουμε τώρα απλά καλεί την επόμενη μέθοδο με *grainSize* ίσο με δύο (2). Μετά το πέρας της εκτέλεσης του *loopBody* σε συνδυασμό με την *combineMethod* επιστρέφεται μια ενιαία τιμή τύπου T. Η μέθοδος *reduce* συλλέγει οποιοδήποτε λάθος/εξαίρεση (exception) προκύψει κατά την διάρκεια εκτέλεσης των στόχων (tasks) και αν αυτό συμβεί, τότε ακυρώνει όλες τις εργασίες που εκτελούνται εκείνη την στιγμή ρίχνοντας στο τέλος ένα *CancelException*. Με το *CancelException* ο προγραμματιστής είναι υπεύθυνος να αποφασίσει τι θα γίνει με τις υπόλοιπες εργασίες που αναμένουν προς εκτέλεση.

Παραμέτροι:

start – Ο πρώτος δείκτης από τον οποίο ξεκινά ο βρόχος επανάληψης, ο οποίος αυξάνεται κατά ένα μέχρι να γίνει ίσος με τον δείκτη *end*.

end – Ο τελευταίος δείκτης που σημαίνει το τέλος της επαναληπτικής εκτέλεσης (μη συμπεριλαμβανομένης της τιμής του, ο βρόχος εκτελείται όσο ο δείκτης *start* είναι μικρότερος από το δείκτη *end*)

initialValue – Η τιμή αρχικοποίησης της μεταβλητής που θα έχει το τελικό αποτέλεσμα.

loopBody – Η δουλειά/διεργασία που πρέπει να εκτελεστεί σε κάθε επανάληψη.

combineMethod – Η μέθοδος/συνάρτηση η οποία χρησιμοποιείται για να συνδυαστούν/συνενωθούν (combine) μεταξύ τους δύο τοπικά αποτελέσματα.

- *public static <T> T reduce (final int start, final int end,*
  final T initialValue, final int grainSize,
  final IReduceTask<T> loopBody,
  final ICombineTask<T> combineMethod)
  throws CancelException

Η μέθοδος *reduce* συνενώνει τιμές του ίδιου τύπου και υπολογίζει μια τελική τιμή (reduction) βάσει μιας συνάρτησης συνένωσης. Το *IReduceTask<T> loopBody* παίρνει έναν ακέραιο ο οποίος ανήκει μεταξύ *start* και *end* επιστρέφοντας ένα στοιχείο τύπου T, όπου το T μπορεί να αντιπροσωπεύει οποιοδήποτε τύπο ανήκει και αναγνωρίζεται στην γλώσσα προγραμματισμού Java. Η *ICombineTask<T> combineMethod* παίρνει δύο τέτοια στοιχεία, που επιστρέφονται από το προηγούμενο, και τα συνδυάζει/συνενώνει (combine) μεταξύ τους. Η *reduce* αποφεύγει να χρησιμοποιήσει κλειδαριές (locks) διαιρώντας το πρόβλημα σε ένα δυαδικό δέντρο στόχων (binary task tree) όπου η κάθε διεργασία έχει την δική της τοπική μεταβλητή την οποία μπορεί και μπορεί να ενημερώνει χωρίς να υπάρχει φόβος να εμφανιστούν προβλήματα όπως race-conditions, deadlocks και livelocks. Πολλές φορές είναι δυνατόν ο προγραμματιστής να μπορεί να βελτιώσει την επίδοση

προσαρμόζοντας σωστά ο ίδιος το μέγεθος στόχου (grain size). Αυτό γίνεται από μέρους του ορίζοντας ρητά το μέγεθος στόχου (grain size), υπονοώντας ότι η εργασία που θα έχει κάθε νήμα να εκτελέσει στην συνέχεια είναι ίση με *grainSize* εκτελέσεις του *IReduceTask<T> loopBody*. Μετά το πέρας της εκτέλεσης του *loopBody* σε συνδυασμό με την *combineMethod* επιστρέφεται μια ενιαία τιμή τύπου T. Η μέθοδος *reduce* συλλέγει οποιοδήποτε λάθος/εξαίρεση (exception) προκύψει κατά την διάρκεια εκτέλεσης των στόχων (tasks) και αν αυτό συμβεί, τότε ακυρώνει όλες τις εργασίες που εκτελούνται εκείνη την στιγμή ρίχνοντας στο τέλος ένα *CancelException*. Με το *CancelException* ο προγραμματιστής είναι υπεύθυνος να αποφασίσει τι θα γίνει με τις υπόλοιπες εργασίες που αναμένουν προς εκτέλεση.

Παραμέτροι:

start – Ο πρώτος δείκτης από τον οποίο ξεκινά ο βρόχος επανάληψης, ο οποίος αυξάνεται κατά ένα μέχρι να γίνει ίσος με τον δείκτη *end*.

end – Ο τελευταίος δείκτης που σημαίνει το τέλος της επαναληπτικής εκτέλεσης (μη συμπεριλαμβανομένης της τιμής του, ο βρόχος εκτελείται όσο ο δείκτης *start* είναι μικρότερος από το δείκτη *end*)

initialValue – Η τιμή αρχικοποίησης της μεταβλητής που θα έχει το τελικό αποτέλεσμα.

grainSize – Ένας λογικός αριθμός επαναλήψεων εκτέλεσης του *loopBody* μέσα σε κάθε στόχο (task).

loopBody – Η δουλειά/διεργασία που πρέπει να εκτελεστεί σε κάθε επανάληψη.

combineMethod – Η μέθοδος/συνάρτηση η οποία χρησιμοποιείται για να συνδυαστούν/συνενωθούν (combine) μεταξύ τους δύο τοπικά αποτελέσματα.

Τα υπόλοιπα αρχεία/κλάσεις που αποτελούν και υποστηρίζουν τις βασικές λειτουργίες της βιβλιοθήκης JACON μπορούν να χωριστούν σε πέντε (5) βασικές κατηγορίες ενώ όλα μαζί στο σύνολό τους τα αρχεία/κλάσεις είναι είκοσι τρία (23). Οι βασικές αυτές κατηγορίες είναι :

- Δημιουργία και διαχείριση εργασιών/στόχων

  Σε αυτή την κατηγορία ανήκουν στο σύνολο οχτώ αρχεία/κλάσεις που είναι υπεύθυνα για την δημιουργία και σωστή διαχείριση των εργασιών/στόχων που δημιουργούνται και πρέπει να εκτελεστούν παράλληλα. Τα αρχεία που την αποτελούν είναι τα εξής :

  Task.java – τρέχει κάθε εργασία/στόχο που δημιουργείται από την βιβλιοθήκη.

  TaskScheduler.java – είναι υπεύθυνος για τον χρονοπρογραμματισμό των εργασιών/στόχων έτσι ώστε να εκτελούνται με σωστή σειρά.

  Async.java – ξεκινά μια ασύγχρονη εργασία/στόχο που δεν επιστρέφει κάποιο αποτέλεσμα.

  Future.java – ξεκινά μια ασύγχρονη εργασία/στόχο που επιστρέφει κάποιο αποτέλεσμα.

  IForTask.java – διεπαφή (interface) που υλοποιείται όταν γίνεται χρήση της μεθόδου Parallel.forLoop.

  IForAtomicTask.java – διεπαφή (interface) που υλοποιείται όταν γίνεται χρήση της μεθόδου Parallel.forLoopAtomic.

  ICombineTask.java – διεπαφή (interface) που υλοποιείται όταν γίνεται χρήση της μεθόδου Parallel.reduce.

  IReduceTask.java – διεπαφή (interface) που υλοποιείται όταν γίνεται χρήση της μεθόδου Parallel.reduce.

- Δημιουργία και διαχείριση νημάτων :

  Σε αυτή την κατηγορία ανήκουν στο σύνολο τρία αρχεία/κλάσεις που είναι υπεύθυνα για την δημιουργία και σωστή διαχείριση των νημάτων που δημιουργούνται και πρέπει να εκτελεστούν παράλληλα. Τα αρχεία που την αποτελούν είναι τα εξής :

  JaConThread.java – χρησιμοποιείται από την κλάση JaConThreadPool για τη δημιουργία ενός νήματος της βιβλιοθήκης που θα αναλάβει την εκτέλεση κάποιας εργασίας/στόχου.

  ThreadScheduler.java – είναι υπεύθυνος για τον χρονοπρογραμματισμό και τη σωστή λειτουργία των νημάτων του JaConThreadPool.

JaConThreadPool.java – υπεύθυνο για την δημιουργία εκτέλεση και τερματισμό των εργασιών/στόχων με τη βοήθεια ενός συνόλου από νήματα.

- Επικοινωνία και ανταλλαγή μηνυμάτων :

  Σε αυτή την κατηγορία ανήκουν στο σύνολο τρία αρχεία/κλάσεις που είναι υπεύθυνα για την επικοινωνία και ανταλλαγή μηνυμάτων μεταξύ των νημάτων και των εργασιών/στόχων που καλούνται να εκτελεστούν παράλληλα. Τα αρχεία που την αποτελούν είναι τα εξής :

  MailBox.java – υπεύθυνο για την διαχείριση και προσωρινή αποθήκευση των μηνυμάτων.

  Address – αντιπροσωπεύει το γραμματοκιβώτιο διευθύνσεων μιας εργασίας/στόχου.

  Message – αντιπροσωπεύει ένα μήνυμα το οποίο μπορεί να ληφθεί στο γραμματοκιβώτιο μιας εργασίας/στόχου.

- Γενικές βοηθητικές συναρτήσεις υποστήριξης :

  Σε αυτή την κατηγορία ανήκουν στο σύνολο έξι αρχεία/κλάσεις που είναι υπεύθυνα για την επικοινωνία και ανταλλαγή μηνυμάτων μεταξύ των νημάτων και των εργασιών/στόχων που καλούνται να εκτελεστούν παράλληλα. Τα αρχεία που την αποτελούν είναι τα εξής :

  Alternative.java – χρησιμοποιείται από την κλάση Choice για την επιλογή μεταξύ διαφορετικών γεγονότων.

  Choice.java – είναι υπεύθυνο για να "ακούει" για κάποιο προκαθορισμένο γεγονός.

  Deque.java, DequeFixed.java – χρησιμοποιούνται κατά τη χρονοδρομολόγηση για να προσθέτονται και να αφαιρούνται αντίστοιχα από μια λίστα οι εργασίες/στόχοι που αναμένουν προς για εκτέλεση.

  Manager.java – διαχείριση αρχικοποιήσεων και γενικών ρυθμίσεων ολόκληρης της βιβλιοθήκης.

  Timer.java – χρησιμοποιείται για δύο λόγους, σαν ρολόι/χρονόμετρο και σαν χρονοδρομολογητής για να επιτυγχάνεται σωστός συγχρονισμός μεταξύ των

νημάτων που τρέχουν παράλληλα, αλλά και που πρόκειται να τρέξουν/τερματίσουν

- Διαχείριση και συλλογή λαθών/εξαιρέσεων (exceptions) :
  Σε αυτή την κατηγορία ανήκουν στο σύνολο τρία αρχεία/κλάσεις που είναι υπεύθυνα για την διαχείριση, συλλογή και εκτύπωση μηνυμάτων λαθών/εξαιρέσεων (exceptions) που τυχόν να προκληθούν κατά τη διάρκεια μιας παράλληλης εκτέλεσης. Τα αρχεία που την αποτελούν είναι τα εξής :
  JaConException.java – η βασική κλάση διαχείρισης και συλλογής λαθών/εξαιρέσεων (exceptions) ολόκληρης της βιβλιοθήκης.
  CancelException.java – επεκτείνει τις λειτουργίες της προηγούμενης για διαχωρισμό των λαθών/εξαιρέσεων (exceptions) που έχουν να κάνουν με ακύρωση εκτέλεσης εργασίας/στόχου.
  PoisonException.java – επεκτείνει τις λειτουργίες της προηγούμενης για διαχωρισμό των λαθών/εξαιρέσεων (exceptions) που έχουν να κάνουν με ανταλλαγή μηνυμάτων και τιμών μεταξύ των νημάτων και των εργασιών/στόχων.

4.3 Μεταγλωττιστής αναδόμησης σειριακού κώδικα Java σε παράλληλο

Όπως ήδη αναφερθήκαμε και προηγουμένως, στα πλαίσια αυτής της διπλωματικής εργασίας έχει δημιουργηθεί ένα εργαλείο αυτόματης μετάφρασης σειριακού κώδικα σε παράλληλο. Το πρόγραμμα αυτό είναι στην ουσία ένας μεταγλωττιστής αναδόμησης (parser – αναλυτής) του πηγαίου κώδικα (source code) προγραμμάτων γραμμένα σε γλώσσα προγραμματισμού Java. Μπορεί να χρησιμοποιηθεί για να εξάγει τον υπονοούμενο παραλληλισμό (implicit parallelism) από τους βρόχους επανάληψης τύπου for() χρησιμοποιώντας πολυνηματισμό (multithreading) κάνοντας εκτεταμένη χρήση των παράλληλων μεθόδων *Parallel.For* και *Parallel.AtomicFor* της βιβλιοθήκης JACON που παρουσιάστηκε στο προηγούμενο υποκεφάλαιο. Το εργαλείο αυτό στηρίζεται εντελώς στον προσδιορισμό του παραλληλισμού με τη βοήθεια υποδείξεων-οδηγιών/σχολίων (parallelism by means of annotations/directives). Επειδή καμία αυτόματη ανίχνευση του παραλληλισμού

δεν υποστηρίζεται, ο παραλληλισμός των βρόχων πρέπει να προσδιοριστεί από τον ίδιο τον προγραμματιστή με τη βοήθεια οδηγιών του τύπου *"/@prarallel*" και *"/@parallelatomic(reduction operation, shared_variable_type shared_variable_name)"* ακριβώς πριν από τον βρόχο επανάληψης που ο ίδιος θέλει να τύχει αυτόματης επεξεργασίας. Εντούτοις, ο μεταγλωττιστής παρέχει ικανοποιητική λειτουργία για να απλοποιήσει τον παραλληλισμό των προγραμμάτων σε Java χωρίς ο προγραμματιστής να χρειάζεται να γνωρίζει κάτι ιδιαίτερο από πολυνηματισμό (multithreading) ή παράλληλο προγραμματισμό.

Ο μεταγλωττιστής αναδόμησης ονομάζεται JavaParallelTranslator και έχει υλοποιηθεί σε γλώσσα προγραμματισμού Java στο περιβάλλον ανάπτυξης εφαρμογών (IDE) Eclipse SDK 3.5.2 [33] με την βοήθεια του εργαλείου δημιουργίας μεταγλωττιστών ANTLR 3.2 [28, 31] το οποίο μπορεί να εγκατασταθεί σαν πρόσθετο εργαλείο (plugin) [31] στο περιβάλλον Eclipse.

Το εργαλείο ANTLR (ANother Tool for Language Recognition) [28, 31] είναι ένα εργαλείο για γλώσσες προγραμματισμού το οποίο παρέχει ένα πλαίσιο (framework) για την δημιουργία αναγνωριστών (recognizers), διερμηνέων (interpreters), μεταγλωττιστών (compilers) και μεταφραστών (translators) μέσα από γραμματικές περιγραφές καλύπτοντας ένα ευρύ φάσμα γλωσσών προγραμματισμού.

Όπως αναφέραμε και πιο πριν, καμία αυτόματη ανίχνευση παραλληλισμού δεν υποστηρίζεται από τον ίδιο τον μεταγλωττιστή αναδόμησης, άρα ο παραλληλισμός των βρόχων πρέπει να προσδιοριστεί από τον προγραμματιστή με τη βοήθεια οδηγιών ακριβώς πριν από τον βρόχο επανάληψης που αυτός θα επιλέξει να τύχει αυτόματης επεξεργασίας. Για τον λόγο αυτό, ο προγραμματιστής είναι υπεύθυνος ο βρόγχος επανάληψης που θα επιλέξει για να τύχει αυτόματης επεξεργασίας να είναι "απολαυστικά/βολικά παράλληλος" (delightfully/pleasantly parallel). Δηλαδή ο βρόχος να είναι σε θέση να τρέξει και παράλληλα επειδή οι πολλές επιμέρους εργασίες που διεξάγονται μπορούν να εκτελεστούν με σχετική ανεξαρτησία, με μερικές ή και καθόλου εξαρτήσεις μεταξύ τους ώστε να μην χρειάζεται ειδικός συγχρονισμός των εργασιών για να αποφευχθούν προβλήματα όπως πολλαπλές

προσβάσεις σε έναν κοινό πόρο/μεταβλητή για σκοπούς εγγραφής χωρίς την ύπαρξη κάποιου μηχανισμού που να υποστηρίζει ταυτόχρονη πρόσβαση (race-conditions), αδιέξοδο κατά το οποίο δύο ή περισσότερες ανταγωνιστικές εργασίες (threads) περιμένουν η μια την άλλη να τελειώσει, χωρίς όμως αυτό να γίνεται από καμία (deadlocks) και καταστάσεις αδιεξόδου κατά την οποία πολλαπλές ανταγωνιστικές εργασίες (threads) αλλάζουν συνεχώς κατάσταση αλλά ποτέ δεν καταλήγουν σε σημείο όπου να μπορεί κάποια από αυτές να προχωρήσει (livelocks).

Οι υποδείξεις για παραλληλισμό που μπορούν να αναγνωριστούν από τον μεταγλωττιστή αναδόμησης JavaParallelTranslator είναι δύο τύπων :

- *"/@prarallel"* : υπονοείται ότι ο βρόχος επανάληψης που ακολουθεί είναι "απολαυστικά/βολικά παράλληλος" (delightfully/pleasantly parallel). Δηλαδή ο βρόχος να είναι σε θέση να εκτελεστεί παράλληλα επειδή οι πολλές επιμέρους εργασίες που διεξάγονται μπορούν να εκτελεστούν με σχετική ανεξαρτησία, με μερικές ή και καθόλου εξαρτήσεις μεταξύ τους. Ο μεταγλωττιστής αναδόμησης θα αντικαταστήσει την βασική δήλωση του βρόχου επανάληψης *'for' '(' simpleforControl ')'* με την παράλληλη έκδοση της μεθόδου που προσφέρεται από την βιβλιοθήκη παράλληλου προγραμματισμού JACON (Prarallel.forLoop), χωρίς όμως να χρειάζεται να κάνει οποιαδήποτε άλλη αλλαγή στην δομή του statement block που ακολουθεί.

- *"/@parallelatomic(reduction_operation, shared_variable_type shared_variable_name)"* : υπονοείται ότι ο βρόχος επανάληψης που ακολουθεί μπορεί να εκτελεστεί παράλληλα υπό την προϋπόθεση ότι ο μεταγλωττιστής αναδόμησης θα λάβει υπόψη του τον κοινό πόρο/μεταβλητή *shared_variable_name* τύπου *shared_variable_type*. Έτσι ο JavaParallelTranslator είναι υπεύθυνος να προνοήσει για την ύπαρξη μηχανισμού που να επιτρέπει την ταυτόχρονη πρόσβαση στον κοινό πόρο/μεταβλητή, ώστε να μπορούν να εκτελεστούν πάνω της παράλληλα πράξεις εγγραφής του τύπου *shared_variable_name reduction_operation shared_variable_name*, όπου το *reduction_operation* μπορεί να

περιλαμβάνει πράξεις όπως +=, -=, *= ή /= (π.χ. counter += counter). Άρα ο μεταγλωττιστής αναδόμησης εκτός του ότι θα αντικαταστήσει την βασική δήλωση του βρόχου επανάληψης *'for' '(' simpleforControl ')'* με την παράλληλη έκδοση της μεθόδου που προσφέρεται από την βιβλιοθήκη παράλληλου προγραμματισμού JACON (PrarallelAtomic.forLoopAtomic), θα πρέπει επίσης να κάνει και μερικές αλλαγές στην δομή του statement block που ακολουθεί έτσι ώστε να μπορεί να λειτουργήσει σωστά ο μηχανισμός ταυτόχρονης πρόσβασης σε κοινό πόρο/μεταβλητή κατά την διάρκεια μιας παράλληλης εκτέλεσης.

Ο JavaParallelTranslator αποτελείται από πέντε βασικά αρχεία. Τα αρχεία αυτά είναι : *JavaParallelTranslator.java, JavaParallelTranslator.g, JavaParallelTranslator.tokens, JavaParallelTranslatorLexer.java* και *JavaParallelTranslatorParser.java*. Στην συνέχεια θα εξηγήσουμε αναλυτικά τη χρήση του κάθε αρχείου από αυτά καθώς και την διαδικασία με την οποία ο μεταγλωττιστής αναδόμησης επιτυγχάνει την αυτόματη μετάφραση του σειριακού κώδικα Java σε παράλληλο.

Το αρχείο JavaParallelTranslator.g περιέχει την γραμματική περιγραφή για τη γλώσσα προγραμματισμού Java καθώς και τις περιγραφές για την αναγνώριση των υποδείξεων-οδηγιών/σχολίων (annotations/directives) που χρειάζονται για να γίνει η σωστή αναγνώριση και ανάλυση των βρόχων επαναλήψεων που θα τύχουν επεξεργασίας έτσι ώστε να παραλληλιστούν. Για την δημιουργία του χρησιμοποιήσαμε την βασική γραμματική για αναγνώριση της γλώσσας προγραμματισμού Java την οποία και τροποποιήσαμε προσθέτοντας περιγραφές για αναγνώριση των παράλληλων οδηγιών καθώς και ορισμού της παράλληλης βιβλιοθήκης (import library) στην αρχή του κώδικα. Πιο συγκεκριμένα οι τροποποιήσεις που κάναμε είναι οι εξής :

- Στην περιγραφή για την αναγνώριση από τον μεταγλωττιστή ενός *statement* (όπου μεταξύ άλλων το statement μπορεί να περιλαμβάνει δηλώσεις για *if, for, while, do .. while, switch* κ.λπ.) προσθέσαμε την δυνατότητα αναγνώρισης και για *parallelblock*.

- Το *parallelblock* περιλαμβάνει αναγνωρίσεις για δύο ειδών δηλώσεις, *parallelAtomicFor* και *parallelFor*, των οποίων η προτεραιότητα αναγνώρισής τους είναι μεγαλύτερη από εκείνην του απλού κανονικού for.
- Στο *parallelAtomicFor* αναγνωρίζεται η εξής δήλωση :
  '/@parallelatomic' '(' reduseAction ',' varType varName ')'
  'for' '(' simpleforControl ')' statement
  ενώ αντίστοιχα στο *parallelFor* αναγνωρίζεται η δήλωση :
  '/@parallel''
  'for' '(' simpleforControl ')' statement
  Με την πρώτη γραμμή (‘/@parallel…’) επιτυγχάνεται αναγνώριση της οδηγίας που πρέπει να υπάρχει πριν από το statement που θέλουμε να παραλληλιστεί περιγράφοντας το είδος του παραλληλισμού και παρέχοντας τυχόν επιπρόσθετες πληροφορίες (reduseAction, varType, varName). Με την δεύτερη γραμμή ('for' '(' simpleforControl ')' statement) επιτυγχάνεται αναγνώριση δήλωσης ενός κανονικού for statement block που όμως οι μεταβλητές ελέγχου του (simpleforControl) τυγχάνουν επεξεργασίας για να μπορούν να χρησιμοποιηθούν κατάλληλα στην μεταφρασμένη τελική δήλωση μέσω των παράλληλων μεθόδων *Parallel.For* και *Parallel.AtomicFor* της βιβλιοθήκης JACON οι οποίες θα αντικαταστήσουν στην ουσία το κομμάτι '(' simpleforControl ')' της δήλωσης του for που θα παραλληλιστεί.
- Εάν αναγνωριστεί κάποιο *parallelblock* τότε είμαστε σε θέση να κάνουμε import την βιβλιοθήκη JACON στην αρχή του αρχείου που πήραμε σαν δεδομένο εισόδου. Αυτό γίνεται προσθέτοντας σχετικό κώδικα κατά την αναγνώριση του κανόνα classOrInterfaceDeclaration ο οποίος εκτελείται κατά την εκκίνηση της συντακτικής αναγνώρισης του κώδικα.

Τα αρχεία JavaParallelTranslator.tokens, JavaParallelTranslatorLexer.java και JavaParallelTranslatorParser.java δημιουργούνται αυτόματα από το εργαλείο ANTLR

Το αρχείο JavaParallelTranslator.tokens περιέχει όλες τις γραμματικές ενδείξεις (tokens) που μπορούν να αναγνωριστούν μέσα από την γραμματική που περιέχεται στο προηγούμενο αρχείο JavaParallelTranslator.g. Οι γραμματικές αυτές ενδείξεις αντιστοιχούνται με ένα μοναδικό αναγνωριστικό (identifier) τύπου integer, κάτι παρόμοιο όπως γίνεται και με τους χαρακτήρες στον πίνακα ASCII, το οποίο αναγνωριστικό και χρησιμοποιείται από τον μεταγλωττιστή κατά την φάση της γραμματικής αναγνώρισης.

Το αρχείο JavaParallelTranslatorLexer.java περιέχει τον αναδρομικό συντακτικό αναλυτή καθόδου (recursive-descent lexer) ο οποίος σπάζει τα δεδομένα σε μικρότερα στοιχεία, σύμφωνα με ένα σύνολο κανόνων που περιγράφουν τη δομή τους. Άρα στην ουσία το αρχείο αυτό περιέχει μια μέθοδο για κάθε κανόνα της γραμματικής που υπάρχει στο αρχείο JavaParallelTranslator.g έτσι ώστε να μπορεί να αναγνωρίσει την γραμματική δομή σε ένα σύνολο από χαρακτήρες αντικαθιστώντας το κάθε σύνολο που βρίσκει με ένα token, δημιουργώντας μια ακολουθία από αυτά.

Το αρχείο JavaParallelTranslatorParser.java περιέχει τον αναδρομικό parser καθόδου (recursive-descent parser) ο οποίος σπάζει τα δεδομένα σε μικρότερα στοιχεία, σύμφωνα με ένα σύνολο κανόνων που περιγράφουν τη δομή τους. Άρα στην ουσία το αρχείο αυτό περιέχει μια μέθοδο για κάθε κανόνα της γραμματικής που υπάρχει στο αρχείο JavaParallelTranslator.g έτσι ώστε να μπορεί να αναγνωρίσει την γραμματική δομή σε ένα σύνολο από tokens τα οποία έχουν δημιουργηθεί προηγουμένως από τον JavaParallelTranslatorLexer.

Η διαφορά μεταξύ ενός parser και ενός lexer είναι ότι, ο lexer αναγνωρίζει την γραμματική δομή σε ένα σύνολο από χαρακτήρες και δημιουργεί μια ακολουθία από tokens ενώ ο parser αναγνωρίζει την γραμματική δομή σε ένα σύνολο από tokens (τα οποία έχουν δημιουργηθεί προηγουμένως από τον lexer).

Το αρχείο JavaParallelTranslator.java είναι αυτό που θα χρησιμοποιήσουμε στην ουσία για να δοκιμάσουμε την γραμματική με κάποια δεδομένα εισόδου (το αρχείο

με τον σειριακό κώδικα σε γλώσσα Java). Περιέχει κώδικα για την δημιουργία αντικειμένων όπως : CharStream, JavaParallelTranslatorLexer, TokenRewriteStream, JavaParallelTranslatorParser. Το αντικείμενο τύπου CharStream περιέχει το αρχείο εισόδου με τον πηγαίο κώδικα και στην ουσία είναι μια ακολουθία χαρακτήρων. Το αντικείμενο τύπου JavaParallelTranslatorLexer είναι ο lexer ο οποίος αναλύει την ακολουθία χαρακτήρων του CharStream. Το αντικείμενο τύπου TokenRewriteStream παίρνει από τον lexer τα tokens που εξάγονται και δημιουργεί μια ακολουθία από αυτά. Το αντικείμενο τύπου JavaParallelTranslatorParser αναλύει την ακολουθία από tokens του TokenRewriteStream καλώντας ένα από τους κανόνες που υπάρχουν στην γραμματική. Το αποτέλεσμα που προκύπτει μετά την γραμματική αναγνώριση και τις μετατροπές που προκύπτουν από τους lexer και parser είναι αυτό που μένει στο TokenRewriteStream.

Ο μεταγλωττιστής αναδόμησης JavaParallelTranslator στην ουσία κάνει γραμματική αναγνώριση του κώδικα Java που παίρνει σαν είσοδο και αποφασίζει κατά πόσο το αρχείο είναι συντακτικά ορθό. Αν στον κώδικα αναγνωριστεί κάποιο parallelblock τότε αυτό αυτόματα παραλληλίζεται κάνοντας χρήση των παράλληλων μεθόδων της βιβλιοθήκης JACON. Οι υποδείξεις για παραλληλισμό (*"/@prarallel"* ή *"/@parallelatomic(reduction operation, shared variable type shared variable name)"*) γίνονται σχόλια ενώ η πρώτη γραμμή στη δήλωση του for (*"for(simpleforControl)"*) αντικαθίσταται από την παράλληλη μέθοδο της βιβλιοθήκης JACON αφήνοντας εντελώς ανέπαφο το ουσιαστικό block του for το οποίο και περιγράφει την εργασία που γίνεται όσο ισχύει η προϋπόθεση (condition) του for. Παραδείγματα parallelblocks και αποτελεσμάτων που προκύπτουν μετά την αυτόματη μετατροπή τους σε παράλληλα από τον JavaParallelTranslator παρουσιάζονται πιο κάτω [Εικόνα 4.1, Εικόνα 4.2] .

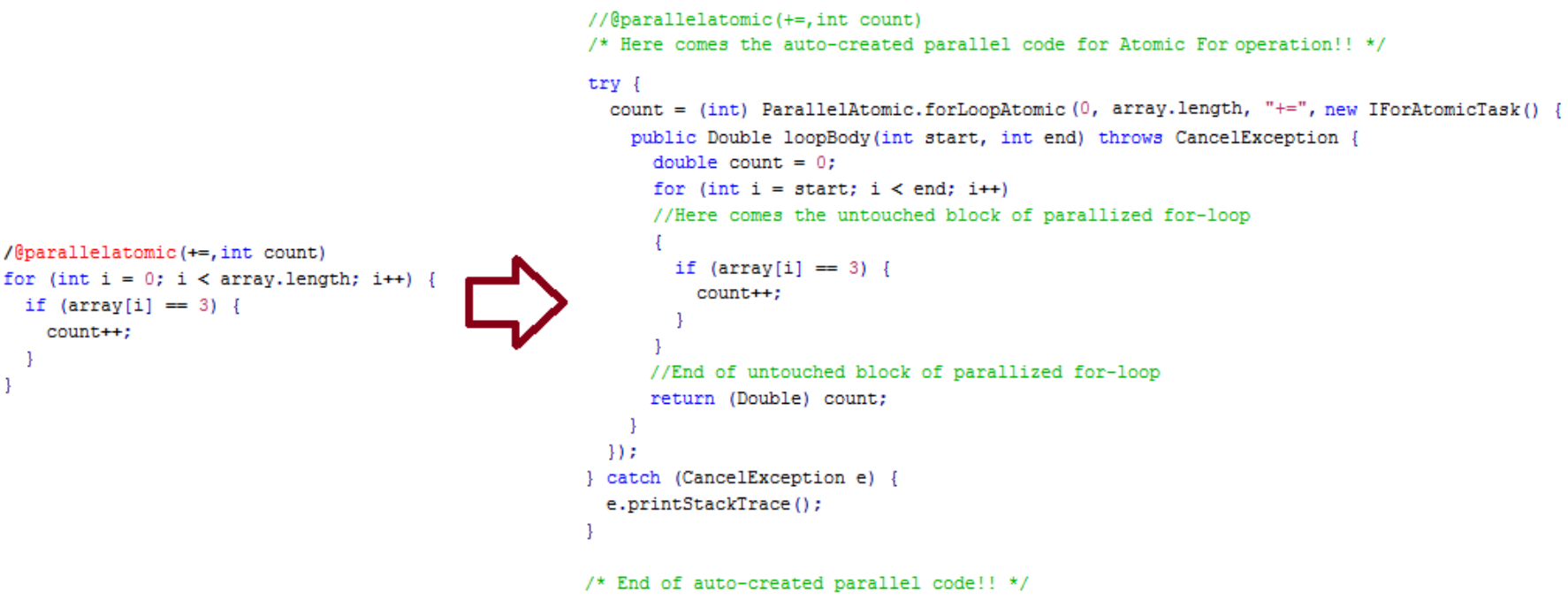
```java
/@parallel
for (int i = 0; i < NDIM; i++) {
  for (int j = 0; j < NDIM; j++) {
    long sum = 0;
    for (int k = 0; k < NDIM; k++) {
      sum += a[i][k] * b[k][j];
    }
    c[i][j] = sum;
  }
}
```

```java
//@parallel
/* Here comes the auto-created parallel code for For statement block!! */

try {
  Parallel.forLoop(0, "<", NDIM, "++", new IForTask() {
    public void loopBody(int i) throws CancelException
    //Here comes the untouched block of parallized for-loop
    {
      for (int j = 0; j < NDIM; j++) {
        long sum = 0;
        for (int k = 0; k < NDIM; k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] = sum;
      }
    }
    //End of untouched block of parallized for-loop
  });
} catch (CancelException e) {
  e.printStackTrace();
}

/* End of auto-created parallel code!! */
```

Εικόνα 4.1 : Παράδειγμα parallelblock με υπόδειξη *"/@prarallel"* και το αποτέλεσμα που προκύπτει μετά την αυτόματη μετάφραση από τον JavaParallelTranslator.



Εικόνα 4.2 : Παράδειγμα parallelblock με υπόδειξη *"/@parallelatomic(reduction operation, shared_variable_type shared_variable_name)"* και το αποτέλεσμα που προκύπτει μετά την αυτόματη μετάφραση από τον JavaParallelTranslator.

4.4 Γραφικό εργαλείο αυτόματης μετάφρασης κώδικα

Στο κομμάτι αυτό θα γίνει μια παρουσίαση της λειτουργίας του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα που έχει υλοποιηθεί και αυτό στα πλαίσια της διπλωματικής εργασίας.

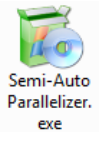
Το εργαλείο αυτό ονομάζεται Semi-AutoParallelizer και έχει υλοποιηθεί σε γλώσσα προγραμματισμού Java στο περιβάλλον ανάπτυξης εφαρμογών (IDE) NetBeans IDE 6.9.1 [39]. Επίσης έχει χρησιμοποιηθεί και το Smart Install Maker 5.02 [43] για την δημιουργία ενός οδηγού εγκατάστασης (installation wizard/installer) το οποίο καθοδηγεί τον χρήστη για την σωστή εγκατάσταση της εφαρμογής πάνω στο σύστημα του.

4.4.1 Εγκατάσταση Εφαρμογής

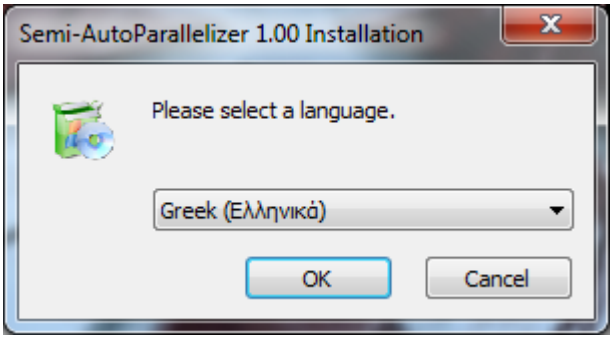
Οι απαιτήσεις συστήματος που έχει το εργαλείο/εφαρμογή έτσι ώστε να μπορεί να τρέξει σωστά είναι οι εξής :

- Λειτουργικό σύστημα Windows XP (x32/x64), Windows Vista (x32/x64), Windows Seven (x32/x64).
- Java Development Kit (JDK) από την Sun Microsystems, οποιαδήποτε έκδοση μεγαλύτερη της 5.0.
- Ανάλυση οθόνης υπολογιστή ίση ή μεγαλύτερη από 1152 x 768.

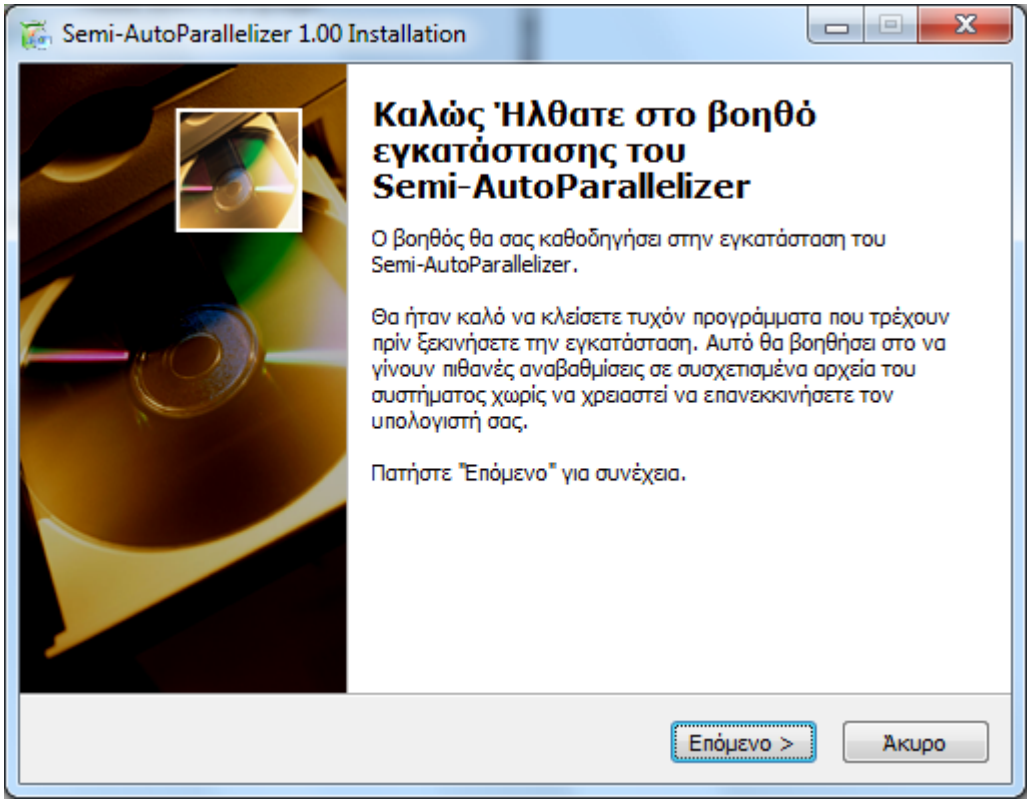
Αρχικά θα πρέπει να εγκαταστήσουμε την εφαρμογή για να μπορεί να τρέξει σωστά πάνω στο υπολογιστικό μας σύστημα. Για να το κάνουμε αυτό, πολύ απλά εκτελούμε τον οδηγό εγκατάστασης που έχει δημιουργηθεί ειδικά για αυτή την εφαρμογή και ακλουθούμε τις απλές οδηγίες και βήματα που μας παρουσιάζονται.

Εκτελώντας το τον οδηγό εγκατάστασης Semi-AutoParallelizer.exe 

ανοίγει ένα παράθυρο το οποίο μας ζητά να επιλέξουμε αρχικά την γλώσσα για τον οδηγό εγκατάστασης [Εικόνα 4.3], επιλέγουμε Ελληνικά και πατούμε το *OK*.



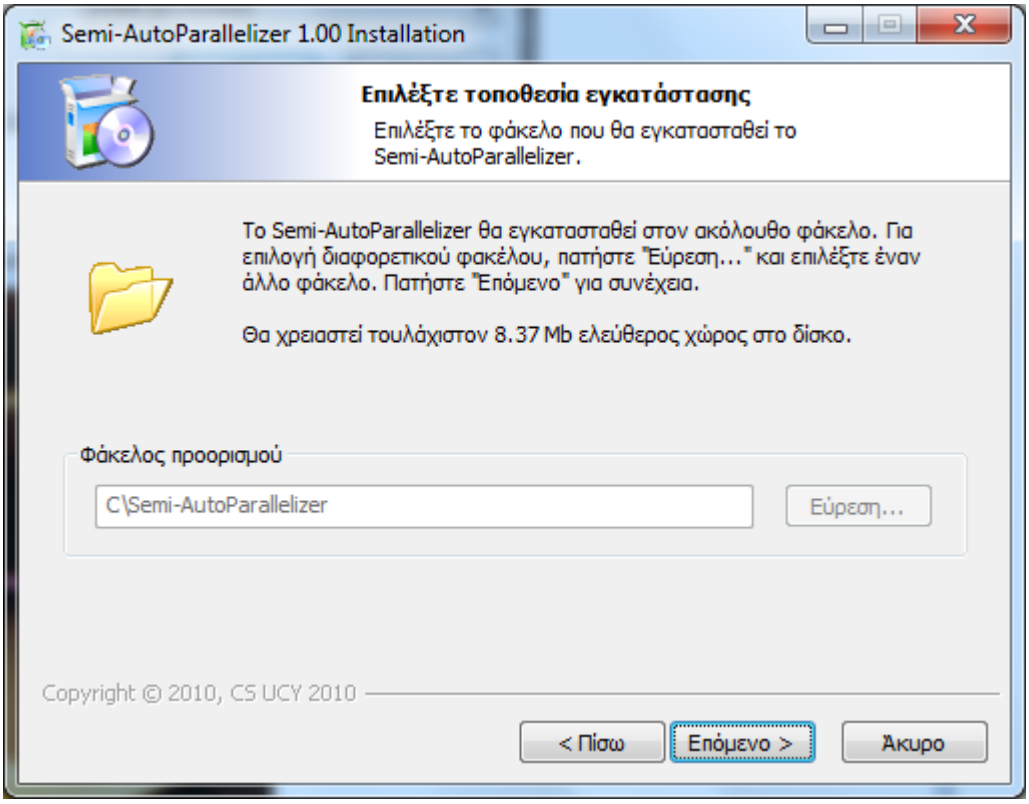
Εικόνα 4.3 : Παράθυρο οδηγού εγκατάστασης του Semi-AutoParallelizer για επιλογή της γλώσσας εγκατάστασης.



Εικόνα 4.4 : Παράθυρο οδηγού εγκατάστασης του Semi-AutoParallelizer με σύντομο χαιρετισμό και βασικές υποδείξεις.

Αφού επιλέξουμε την γλώσσα και πατήσουμε το κουμπί *OK* στο επόμενο παράθυρο [Εικόνα 4.4] ακολουθεί σύντομος χαιρετισμός καθώς και κάποιες βασικές οδηγίες

για να κυλήσει ομαλά η εγκατάσταση. Υπάρχουν δύο επιλογές στο κάτω μέρος του παραθύρου, μια για ακύρωση της διαδικασίας και μια για να προχωρήσουμε. Επιλέγοντας το κουμπί *Επόμενο* > η διαδικασία εγκατάστασης προχωρά στο επόμενο παράθυρο διαφορετικά μας ζητά επιβεβαίωση για την ακύρωση της εγκατάστασης.

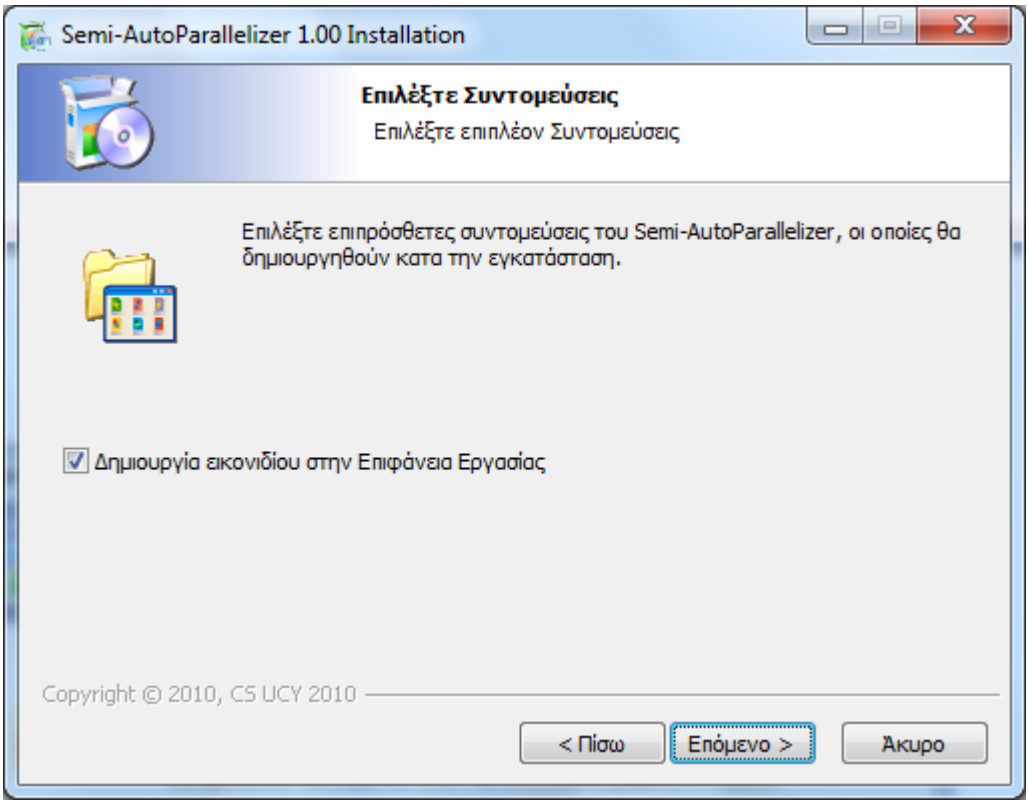


Εικόνα 4.5 : Παράθυρο οδηγού εγκατάστασης του Semi-AutoParallelizer για επιλογή φακέλου εγκατάστασης.

Στο επόμενο παράθυρο καλούμαστε να επιλέξουμε την τοποθεσία εγκατάστασης [Εικόνα 4.5]. Για να λειτουργεί όμως σωστά το γραφικό εργαλείο αυτόματης μετάφρασης κώδικα θα πρέπει να εγκατασταθεί οπωσδήποτε στον φάκελο *C:\Semi-AutoParallelizer,* γι' αυτόν τον λόγο η επιλογή για υπόδειξη διαφορετικού φακέλου είναι απενεργοποιημένη. Επίσης γίνεται αναφορά για τον ελεύθερο χώρο στον δίσκο που απαιτείται για να γίνει η εγκατάσταση και είναι περίπου 8.5 MB. Για να

προχωρήσουμε στο επόμενο παράθυρο της διαδικασίας πρέπει να πατήσουμε το κουμπί *Επόμενο >*.

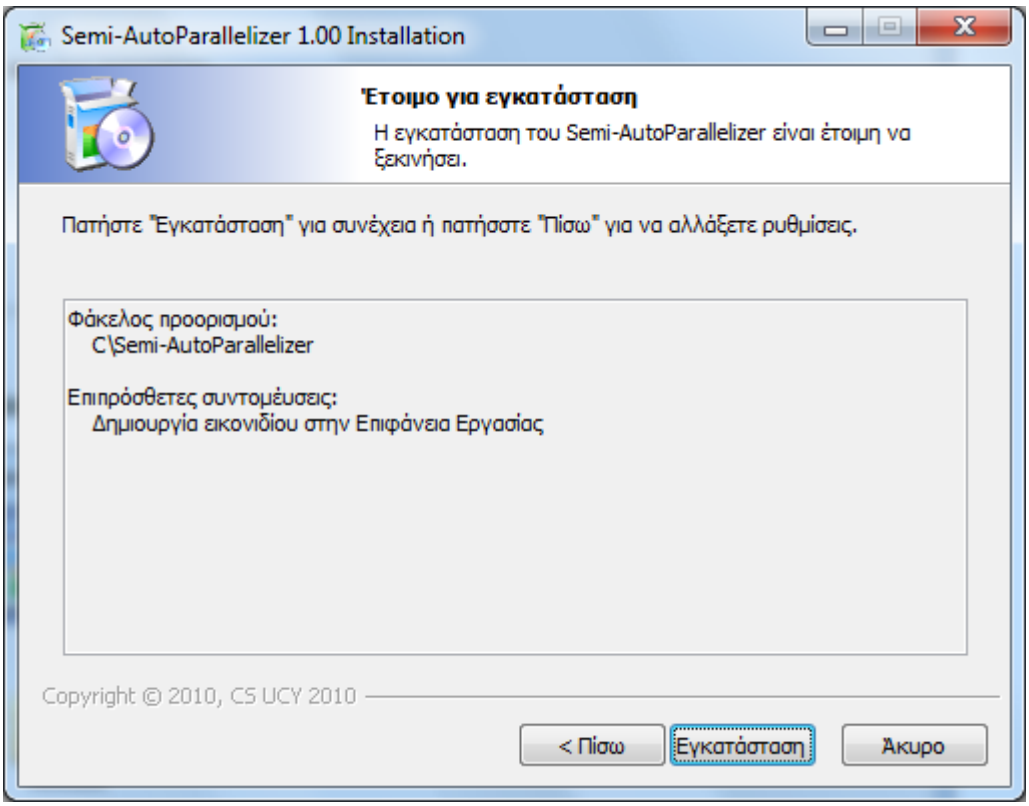
Αμέσως μετά την επιλογή φακέλου εγκατάστασης ακολουθεί ένα παράθυρο στο οποίο μας ζητείται να αποφασίσουμε κατά πόσο επιθυμούμε να δημιουργηθεί επιπλέον συντόμευση (shortcut) στην επιφάνεια εργασίας (Desktop) [Εικόνα 4.6]. Αν το επιθυμούμε αφήνουμε την σχετική επιλογή και πατούμε στο κουμπί *Επόμενο >*, διαφορετικά αφαιρούμε την επιλογή και προχωρούμε και πάλι στο επόμενο παράθυρο.



Εικόνα 4.6 : Παράθυρο οδηγού εγκατάστασης του Semi-AutoParallelizer για επιλογή επιπρόσθετων συντομεύσεων.

Στο επόμενο παράθυρο είμαστε ήδη έτοιμοι να ξεκινήσουμε την εγκατάσταση αφού έχουμε επιλέξει όλες τις επιμέρους λεπτομέρειες που μας ζητήθηκαν. Στο παράθυρο αυτό [Εικόνα 4.7] παρουσιάζονται περιληπτικά όλες οι προηγούμενες επιλογές μας

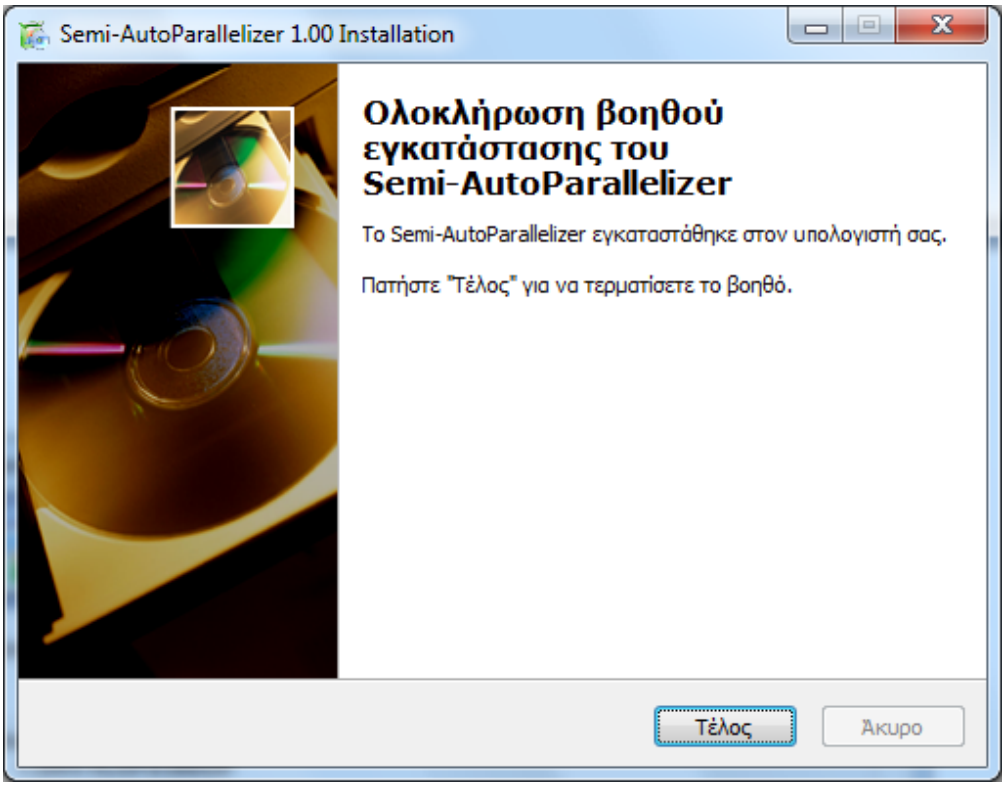
σε συνοπτική μορφή και καλούμαστε να πατήσουμε το κουμπί *Εγκατάσταση* για να συνεχίσουμε. Μπορούμε πατώντας το κουμπί < *Πίσω* να πάμε σε προηγούμενα παράθυρα και να αλλάξουμε τις ρυθμίσεις μας.



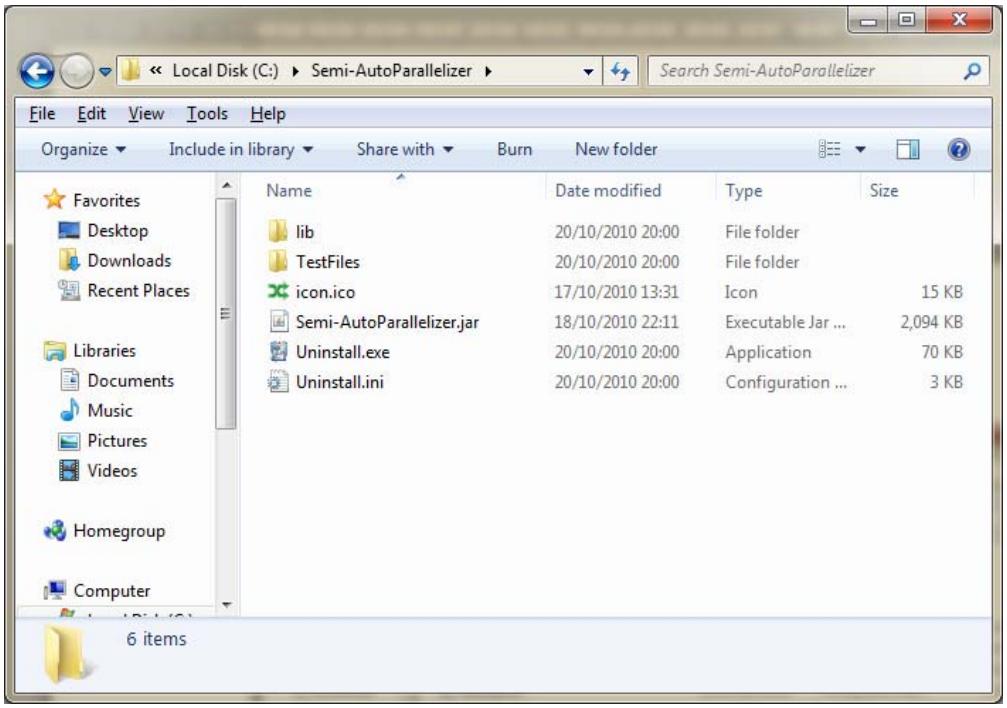
Εικόνα 4.7 : Παράθυρο οδηγού εγκατάστασης του Semi-AutoParallelizer έτοιμο για να ξεκινήσει την εγκατάσταση.

Αφού πατήσουμε το κουμπί *Εγκατάσταση* και αυτή ολοκληρωθεί μας παρουσιάζεται τελικά ένα παράθυρο που μας ενημερώνει ότι η εγκατάσταση έχει ολοκληρωθεί και μας ζητείται να πατήσουμε το κουμπί Τέλος για να κλείσει και το παράθυρο αυτό [Εικόνα 4.8].

Μετά το τέλος της εγκατάστασης έχει ήδη δημιουργηθεί ο φάκελος με όλα τα αρχεία που χρειάζονται [Εικόνα 4.9] για να μπορεί να λειτουργήσει η εφαρμογή του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα και δεν μένει παρά να το δοκιμάσουμε.



Εικόνα 4.8 : Παράθυρο οδηγού εγκατάστασης του Semi-AutoParallelizer που μας ενημερώνει για την επιτυχή ολοκλήρωση της εγκατάστασής του.



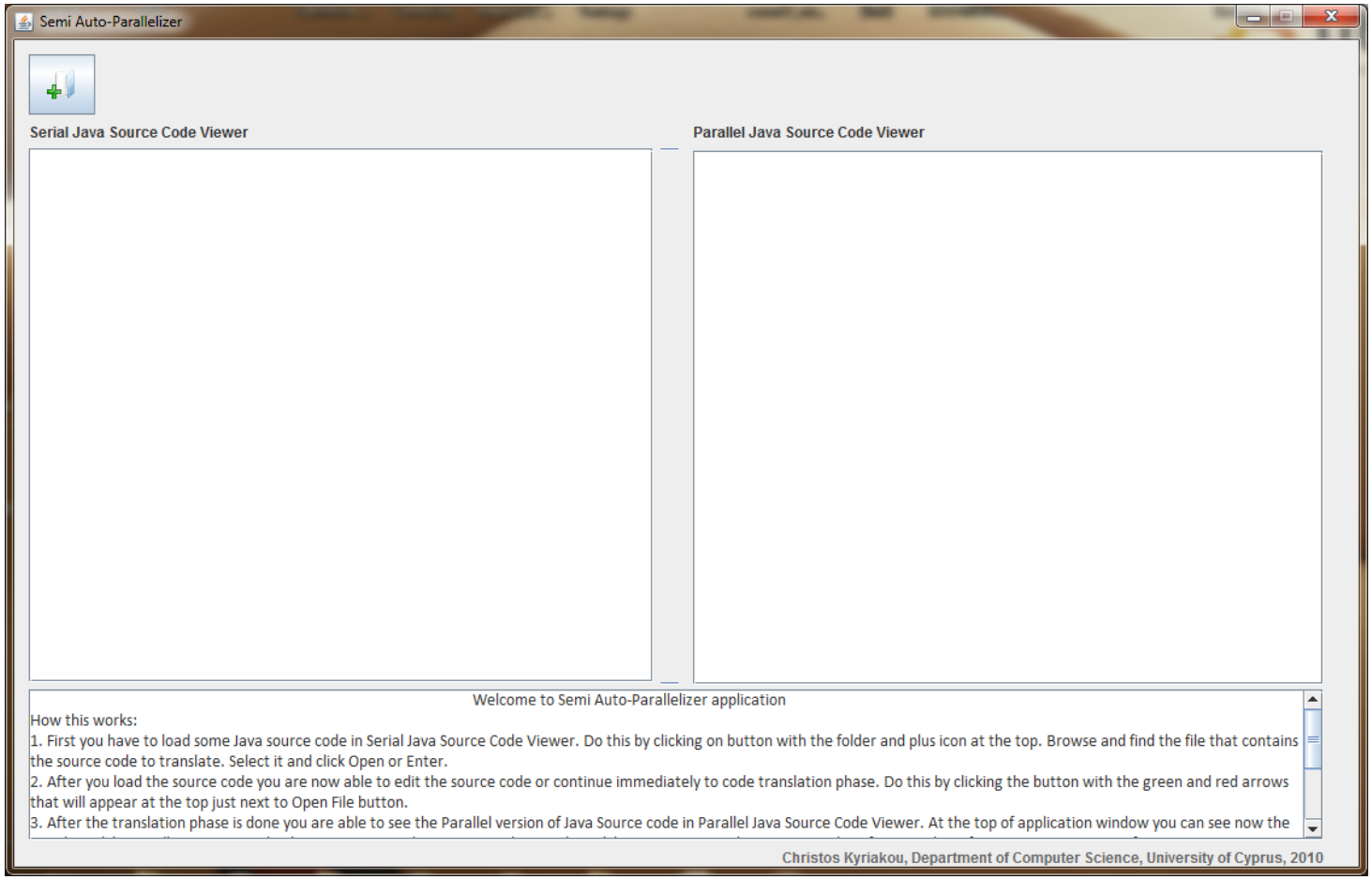
Εικόνα 4.9 : Ο φάκελος με όλα τα αρχεία που χρειάζονται για να μπορεί να λειτουργήσει ο Semi-AutoParallelizer.

4.4.2 Παρουσίαση Εργαλείου

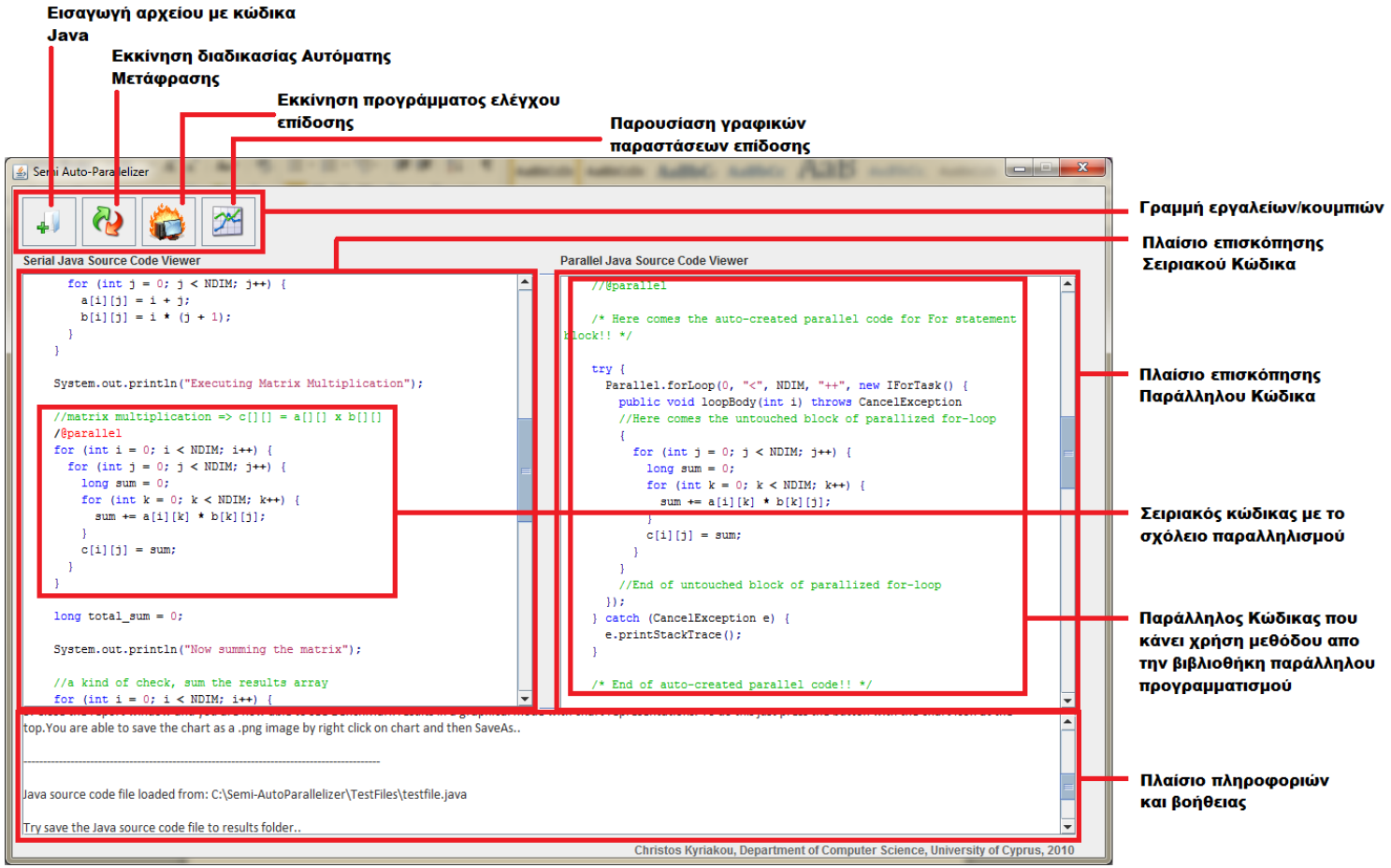
Για να τρέξουμε την εφαρμογή του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα απλά πατούμε διπλό κλικ στο εικονίδιο συντόμευσης Semi-AutoParallelizer

που βρίσκεται στην επιφάνεια εργασίας μας και δημιουργήθηκε αφού τρέξαμε προηγουμένως τον οδηγό εγκατάστασης του εργαλείου.

Εκκινώντας την εφαρμογή του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα έχουμε τώρα μπροστά μας το αρχικό παράθυρο της εφαρμογής [Εικόνα 4.10] που χωρίζεται σε τρία βασικά μέρη. Το πρώτο μέρος είναι αυτό στο πάνω μέρος του παραθύρου και το οποίο θα περιέχει τα κουμπιά με τα οποία ο χρήστης θα μπορεί να αλληλεπιδρά με την εφαρμογή. Στο αρχικό παράθυρο εμφανίζεται μόνο ένα κουμπί, αλλά στην συνέχεια στο σύνολό τους θα γίνουν τέσσερα. Το δεύτερο κυρίως μέρος είναι τα δύο πλαίσια επισκόπησης κώδικα που βρίσκονται στο μέσο του παραθύρου. Αριστερά είναι το πλαίσιο επισκόπησης του σειριακού κώδικα ενώ ακριβώς δίπλα στα δεξιά είναι το πλαίσιο επισκόπησης του παράλληλου κώδικα που δημιουργείται μετά την αυτόματη μετάφραση του σειριακού. Έτσι ο χρήστης είναι σε θέση πολύ εύκολα να συγκρίνει τις δύο εκδόσεις κώδικα που θα έχει μιας και το εργαλείο χρωματίζει τον κώδικα σύμφωνα με την σύνταξη της γλώσσας Java (Java syntax highlighting) όπως επίσης διορθώνει την μορφοποίηση του κειμένου έτσι ώστε ο κώδικας να είναι σωστά στοιχισμένος και άρα πιο ευανάγνωστος. Το τρίτο και τελευταίο μέρος του παραθύρου της εφαρμογής είναι αυτό που βρίσκεται ακριβώς κάτω από τα δύο πλαίσια επισκόπησης κώδικα στο κάτω μέρος του παραθύρου. Σε αυτό το πλαίσιο ο χρήστης μπορεί να δει αρχικά κάποιες βασικές οδηγίες και βήματα τα οποία μπορεί να ακολουθήσει για χρησιμοποιήσει το εργαλείο. Κατά την διάρκεια εκτέλεσης των διαφόρων λειτουργιών από την εφαρμογή θα εμφανίζονται εκεί κάποιες χρήσιμες πληροφορίες που έχουν να κάνουν με αρχεία και αποτελέσματα που αποθηκεύονται αυτόματα από το εργαλείο για να μπορεί και αργότερα ο χρήστης να τα χρησιμοποιήσει.



Εικόνα 4.10 : Το αρχικό παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα που εμφανίζεται κατά την εκκίνηση της εφαρμογής.

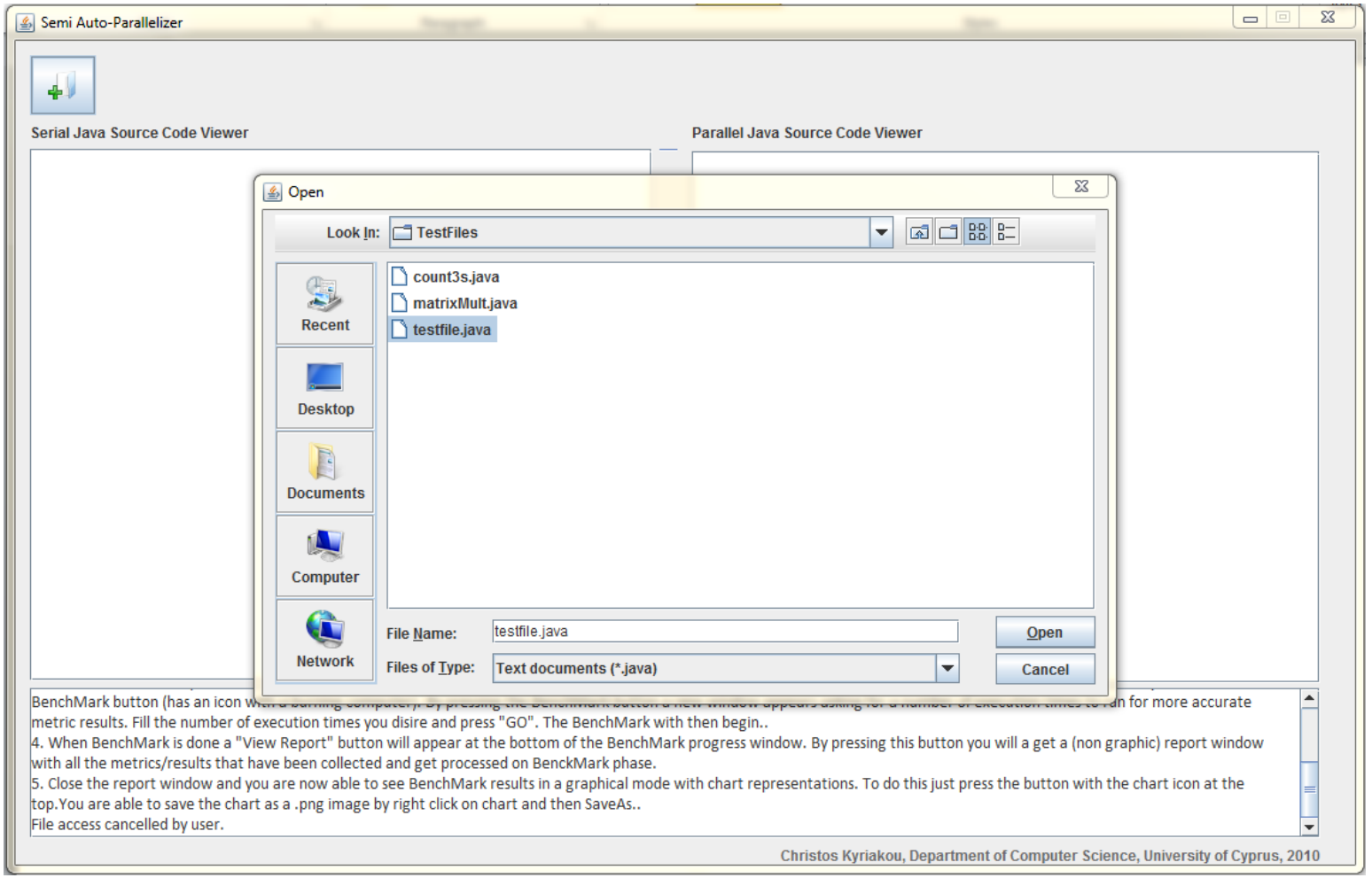


Εικόνα 4.11 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με επεξηγήσεις για τις λειτουργίες που μπορεί να προσφέρει.

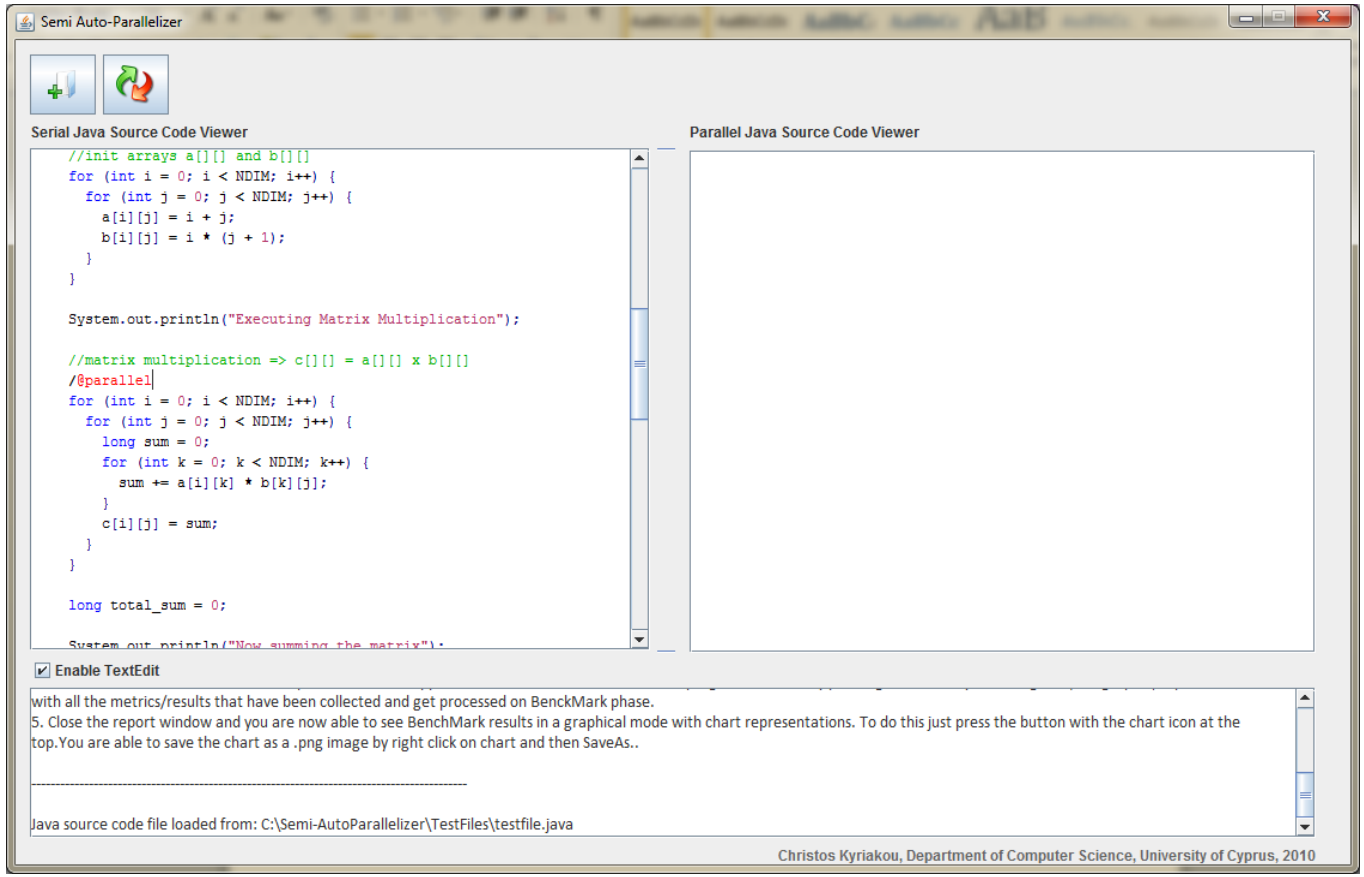
Έχοντας ανοικτό το αρχικό παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα, εκτός από το να διαβάσουμε τις σύντομες οδηγίες χρήσης στο κάτω μέρος του παραθύρου, το μόνο που μπορούμε να κάνουμε είναι να επιλέξουμε το κουμπί εισαγωγής αρχείου κώδικα Java  το οποίο θα μας δώσει την δυνατότητα να αναζητήσουμε και να επιλέξουμε όποιο αρχείο κώδικα εμείς θέλουμε με προέκταση .java [Εικόνα 4.12]. Στο παράθυρο (Open) που μας εμφανίζεται πλοηγούμαστε στον επιθυμητό κατάλογο με το αρχείο που θέλουμε να εισάγουμε στην εφαρμογή, το επιλέγουμε και πατούμε *Enter* ή το κουμπί *Open* στο κάτω αριστερό μέρος του παραθύρου.

Αφού επιλέξουμε το αρχείο με κώδικα Java και το φορτώσουμε στην εφαρμογή είμαστε τώρα σε θέση να δούμε τον πηγαίο κώδικα, που περιείχε το αρχείο, μορφοποιημένο και χρωματισμένο σε ευανάγνωστη μορφή στο πλαίσιο επισκόπησης σειριακού κώδικα. Μπορούμε επίσης να επεξεργαστούμε/τροποποιήσουμε τον κώδικα ενεργοποιώντας την επιλογή *Enable TextEdit* που βρίσκεται ακριβώς κάτω από το πλαίσιο επισκόπησης του σειριακού κώδικα που μόλις φορτώσαμε [Εικόνα 4.13].

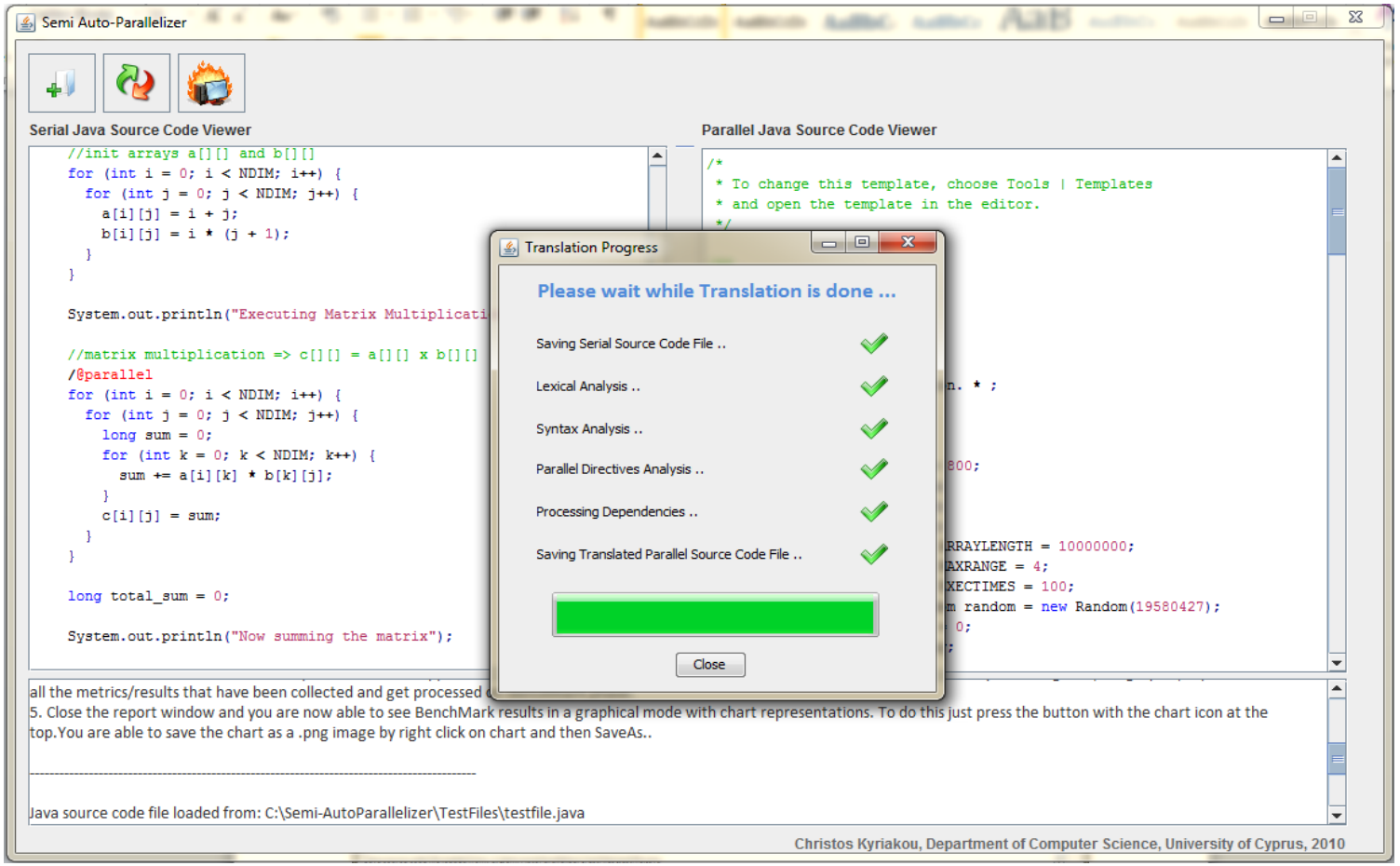
Αφού τελειώσουμε με τυχόν τροποποιήσεις στον σειριακό πηγαίο κώδικα Java μπορούμε πλέον να προχωρήσουμε στην διαδικασία αυτόματης μετατροπής του σειριακού κώδικα σε παράλληλο. Για να το κάνουμε αυτό πρέπει να επιλέξουμε το δεύτερο κουμπί που εμφανίστηκε ακριβώς δίπλα από το κουμπί εισαγωγής κώδικα στο πάνω μέρος του παραθύρου. Πατώντας το κουμπί αυτόματης μετάφρασης  ξεκινά η διαδικασία μετατροπής του σειριακού κώδικα Java σε παράλληλο, ενώ εμείς μπορούμε να δουμε τα βήματα που εκτελούνται και ολοκληρόνονται από τον μεταγλωττιστή αναδόμησης JavaParallelTranslator που ενσωματώσαμε στο εργαλείο (Translation Progress). Αν ο κώδικας που υπήρχε φορτωμένος στο πλαίσιο επισκόπησης του σειριακού κώδικα δεν περιείχε συντακτικά/γραμματικά λάθη τότε όλα τα βήματα της διαδικασίας αυτόματης μετάφρασης ολοκληρώνονται με επιτυχία [Εικόνα 4.14].



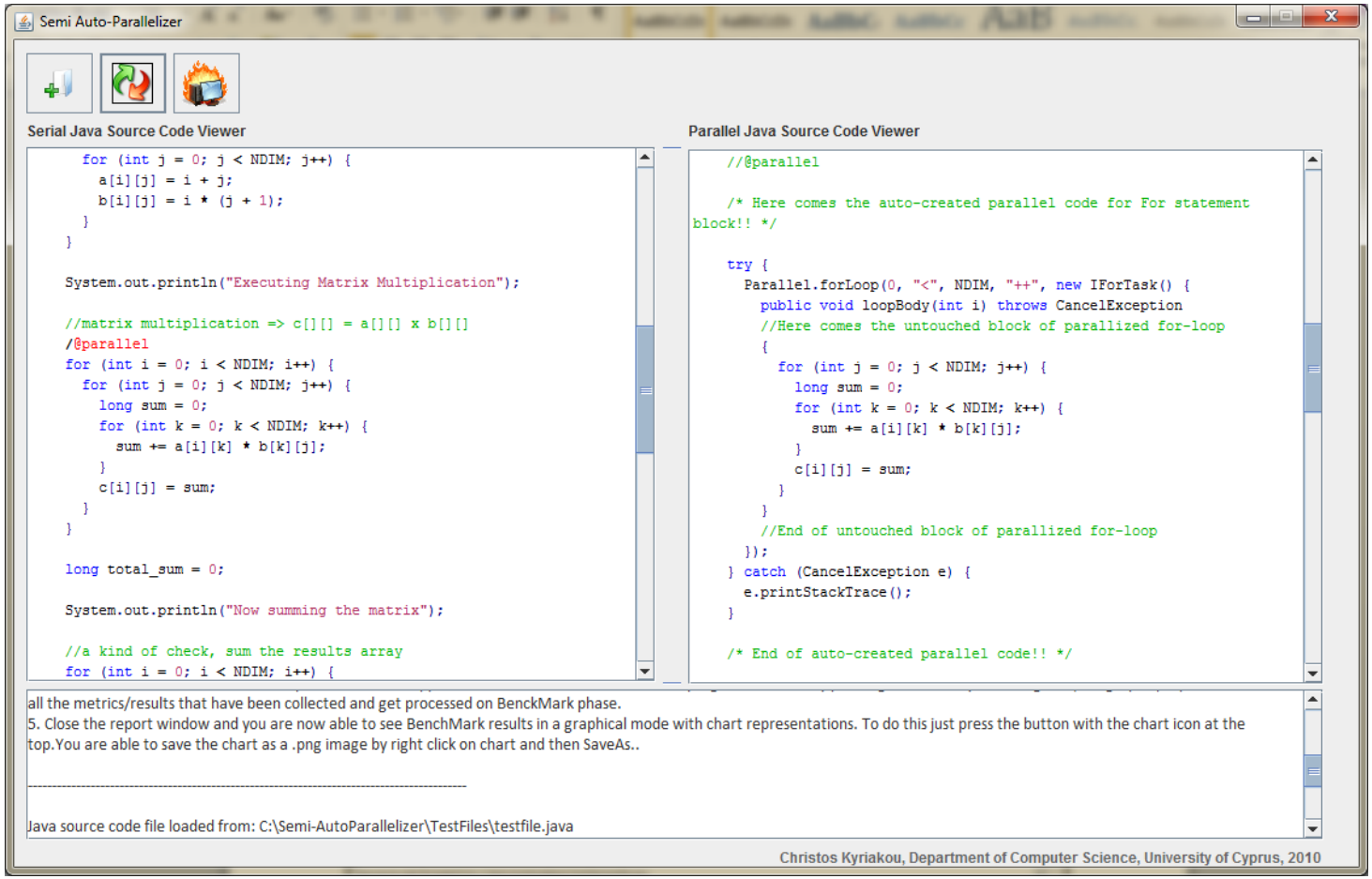
Εικόνα 4.12 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα για εισαγωγή αρχείου κώδικα Java.



Εικόνα 4.13 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με φορτωμένο τον σειριακό κώδικα στο πλαίσιο επισκόπησης σειριακού κώδικα και ενεργοποιημένη την επιλογή Enable TextEdit για δυνατότητα επεξεργασίας/τροποποίησης του.



Εικόνα 4.14 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με όλα τα βήματα της διαδικασίας αυτόματης μετάφρασης ολοκληρωμένα με επιτυχία.



Εικόνα 4.15 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με τις δύο εκδόσεις κώδικα, σειριακή και παράλληλη, που εμφανίζονται στα δύο αντίστοιχα πλαίσια επισκόπησης κώδικα.

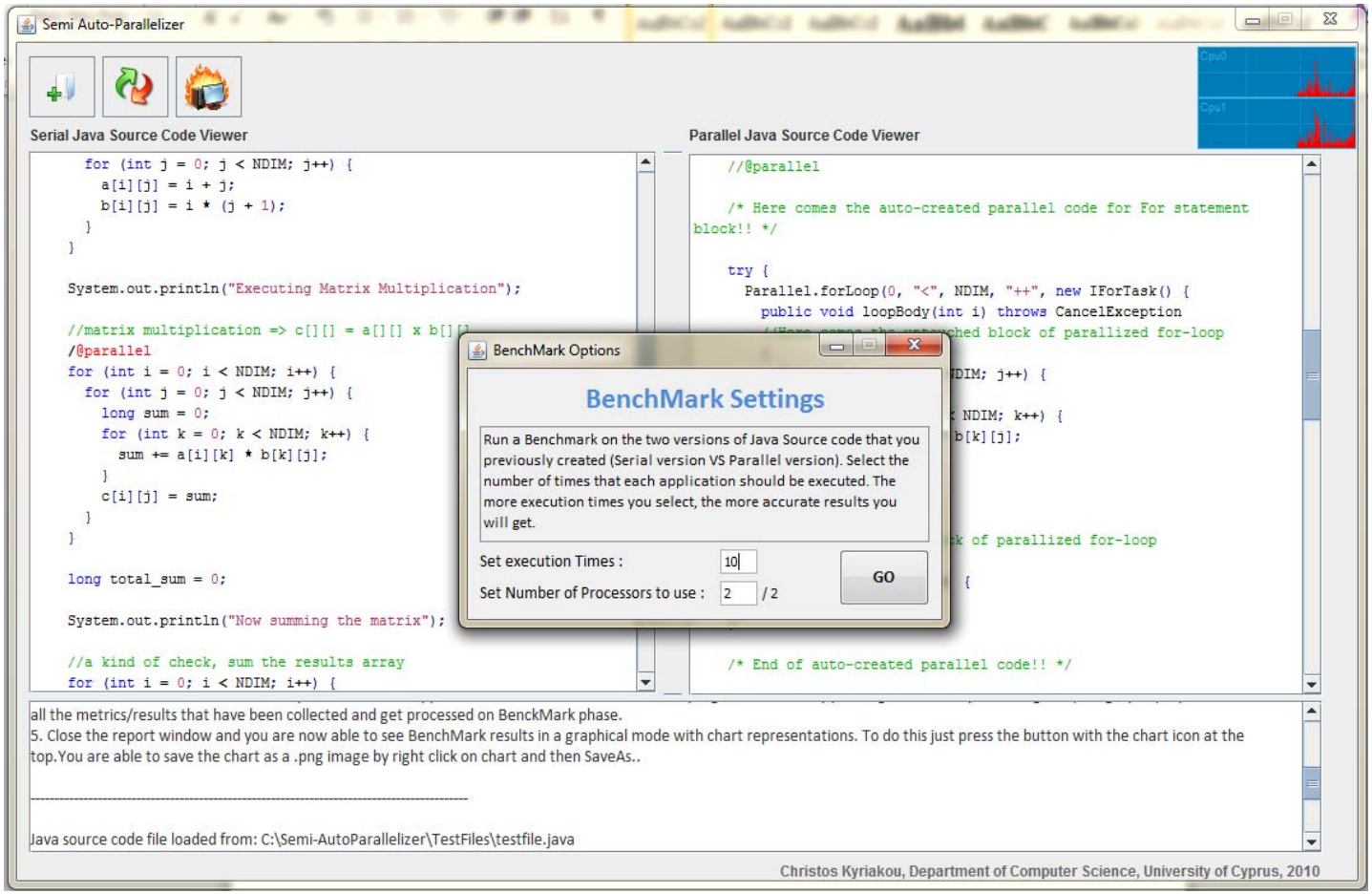
Η σειριακή έκδοση του κώδικα αποθηκεύεται στον φάκελο *Results* που δημιουργείται στον κατάλογο εγκατάστασης την εφαρμογής (*C:\Semi-AutoParallelizer*) με όνομα αρχείου ίδιο με εκείνο που αρχικά φορτώσαμε με την επιπλέον όμως προέκταση *.orig* . Η παράλληλη έκδοση του κώδικα που μόλις δημιουργήθηκε αποθηκεύεται και αυτή στον φάκελο *Results* με το αρχικό όμως όνομα του σειριακού αρχείου που φορτώσαμε. Αυτό γίνεται γιατί τα ονόματα αρχείων στην Java παίζουν σημαντικό ρόλο και επίσης δεν μπορούμε να έχουμε ταυτόχρονα στον ίδιο φάκελο δύο αρχεία με το ίδιο ακριβώς όνομα και προέκταση. Έτσι τώρα ο χρήστης έχει και τις δύο εκδόσεις κώδικα και μπορεί πολύ εύκολα να τρέξει την παράλληλη έκδοση που μόλις δημιουργήθηκε, και σίγουρα τον ενδιαφέρει περισσότερο, από μόνος του μέσω της γραμμής εντολών (Command Prompt) χωρίς να χρειαστεί να κάνει οποιαδήποτε αλλαγή στο αρχείο. Η παράλληλη έκδοση του κώδικα φορτώνεται στο πλαίσιο επισκόπησης παράλληλου κώδικα επίσης μορφοποιημένη και χρωματισμένη σε ευανάγνωστη μορφή. Αφού κλείσουμε το παράθυρο που εμφανίζει την εξέλιξη προόδου της διαδικασίας μετάφρασης με τα επιμέρους βήματα, πατώντας το κουμπί *Close* στο κάτω μέρος του παραθύρου, είμαστε σε θέση να συγκρίνουμε τις δύο εκδόσεις κώδικα, σειριακή και παράλληλη, που εμφανίζονται στα δύο αντίστοιχα πλαίσια επισκόπησης κώδικα [Εικόνα 4.15].

Έχοντας τώρα και τις δύο εκδόσεις κώδικα, σειριακή και παράλληλη, έχουμε την δυνατότητα να τρέξουμε το πρόγραμμα ελέγχου επίδοσης (BenchMark). Με αυτό το εργαλείο μπορούμε να τρέξουμε βασικά τις δυο εκδόσεις κώδικα πάνω στο σύστημά μας και να πάρουμε διάφορες μετρικές που έχουν να κάνουν με χρόνους εκτέλεσης και ποσοστά βελτίωσης της επίδοσης της παράλληλης έκδοσης που δημιουργήθηκε έναντι της σειριακής. Για να ξεκινήσουμε το BenchMark απλά πατούμε στο τρίτο κουμπί που εμφανίστηκε ακριβώς δίπλα από το κουμπί αυτόματης μετάφρασης στο πάνω μέρος του παραθύρου. Πατώντας το κουμπί ελέγχου της επίδοσης  ανοίγει ένα παράθυρο (BenchMark Options) για καθορισμό κάποιων βασικών ρυθμίσεων της διαδικασίας ελέγχου της επίδοσης [Εικόνα 4.16]. Στο παράθυρο αυτό μας ζητείται να καθορίσουμε τον αριθμό των επαναλήψεων που θέλουμε να

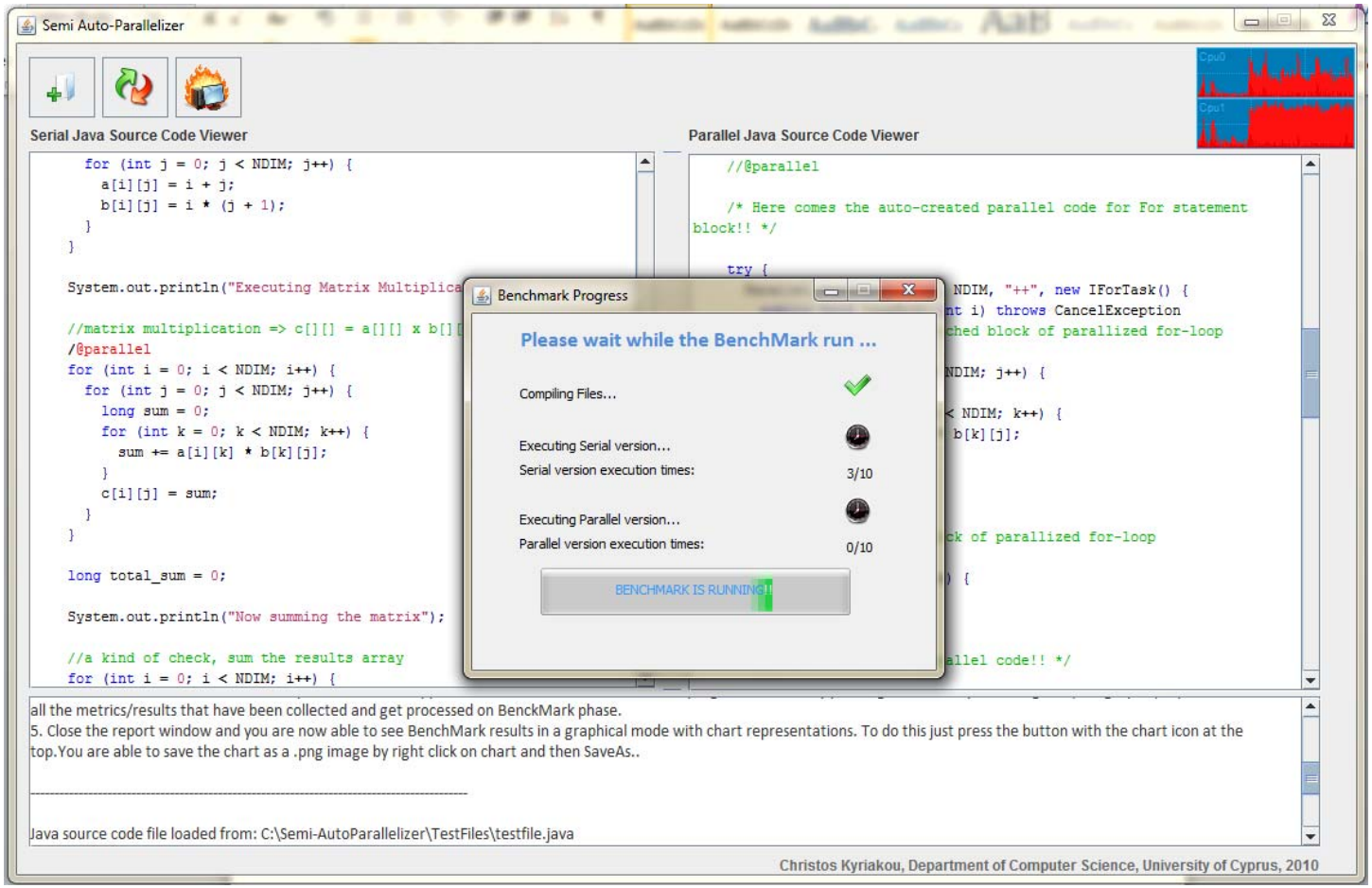
εκτελεστούν οι δύο εκδόσεις καθώς και τον αριθμό των επεξεργαστών (cups'/cores) που θέλουμε να συμμετάσχουν στην εκτέλεση.

Ο αριθμός των επαναλήψεων εκτέλεσης είναι σημαντικός να καθοριστεί για σκοπούς μεγαλύτερης ακρίβειας στους τελικούς χρόνους εκτέλεσης των δυο εκδόσεων του προγράμματος που έχουμε. Με μια μόνο εκτέλεση οι χρόνοι που θα πάρουμε δεν θα είναι αντιπροσωπευτικοί γιατί μπορεί οποιαδήποτε άλλη διαδικασία που τρέχει στο παρασκήνιο να επηρεάσει την εκτέλεση και κατά συνέπεια και τις μετρήσεις μας. Με περισσότερες όμως από μια εκτελέσεις για κάθε έκδοση κώδικα αυτό το πρόβλημα περιορίζεται γιατί αθροίζονται όλοι οι επιμέρους χρόνοι εκτέλεσης και εμείς παίρνουμε τον μέσο όρο χρόνου εκτέλεσης, έτσι τα αποτελέσματα είναι πιο ακριβή. Όσο μεγαλύτερος είναι ο αριθμός των επαναλήψεων εκτέλεσης τόσο πιο ακριβή θα είναι τα αποτελέσματά μας αλλά και τόσο μεγαλύτερος θα είναι ο συνολικός χρόνος εκτέλεσης του BenchMark. Άρα από τον χρήστη εξαρτάται πόσο θα το θέσει, η προτεινόμενη αρχική τιμή είναι δέκα επαναλήψεις.

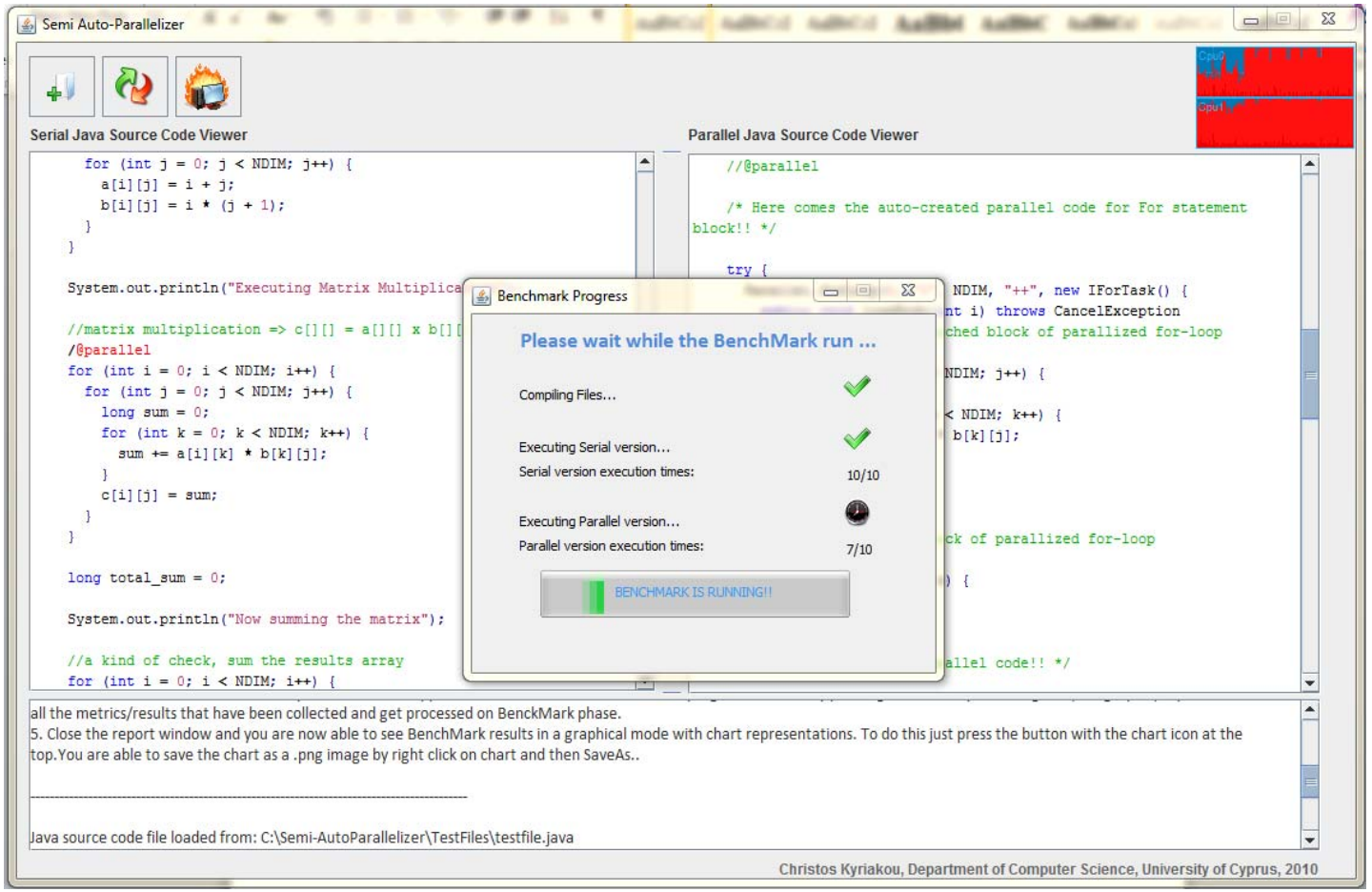
Όσον αφορά τον καθορισμό των επεξεργαστών (cups'/cores) που θέλουμε να συμμετέχουν κατά τη διάρκεια της εκτέλεσης, αυτή η επιλογή δίνεται στο χρήστη για να μπορεί να κάνει συγκρίσεις της βελτίωσης της επίδοσης που έχει από το εργαλείο μας πάνω σε ένα σύστημα με πολλαπλούς επεξεργαστές. Μπορεί έτσι να έχει την δυνατότητα να προβλέψει σε ένα βαθμό την πιθανή βελτίωση στην επίδοση για ένα παρόμοιο σύστημα το οποίο θα διαφέρει ουσιαστικά στον αριθμό των επεξεργαστών που προσφέρει. Επίσης μπορούμε με αυτό το χαρακτηριστικό να συγκρίνουμε την επίδοση μεταξύ της αρχικής σειριακής έκδοσης, που ουσιαστικά εκτελείται χρησιμοποιώντας έναν επεξεργαστή, έναντι τις επίδοσης της παράλληλης έκδοσης την οποία θα εκτελέσουμε σε έναν επεξεργαστή χρησιμοποιώντας πολυνηματισμό. Έτσι θα έχουμε την δυνατότητα να μελετήσουμε τις επιπλέον καθυστερήσεις (overheads) ή τυχόν βελτιώσεις που προσφέρει μια πολυνηματική εκτέλεση σε ένα μη παράλληλο σύστημα με ένα επεξεργαστή.



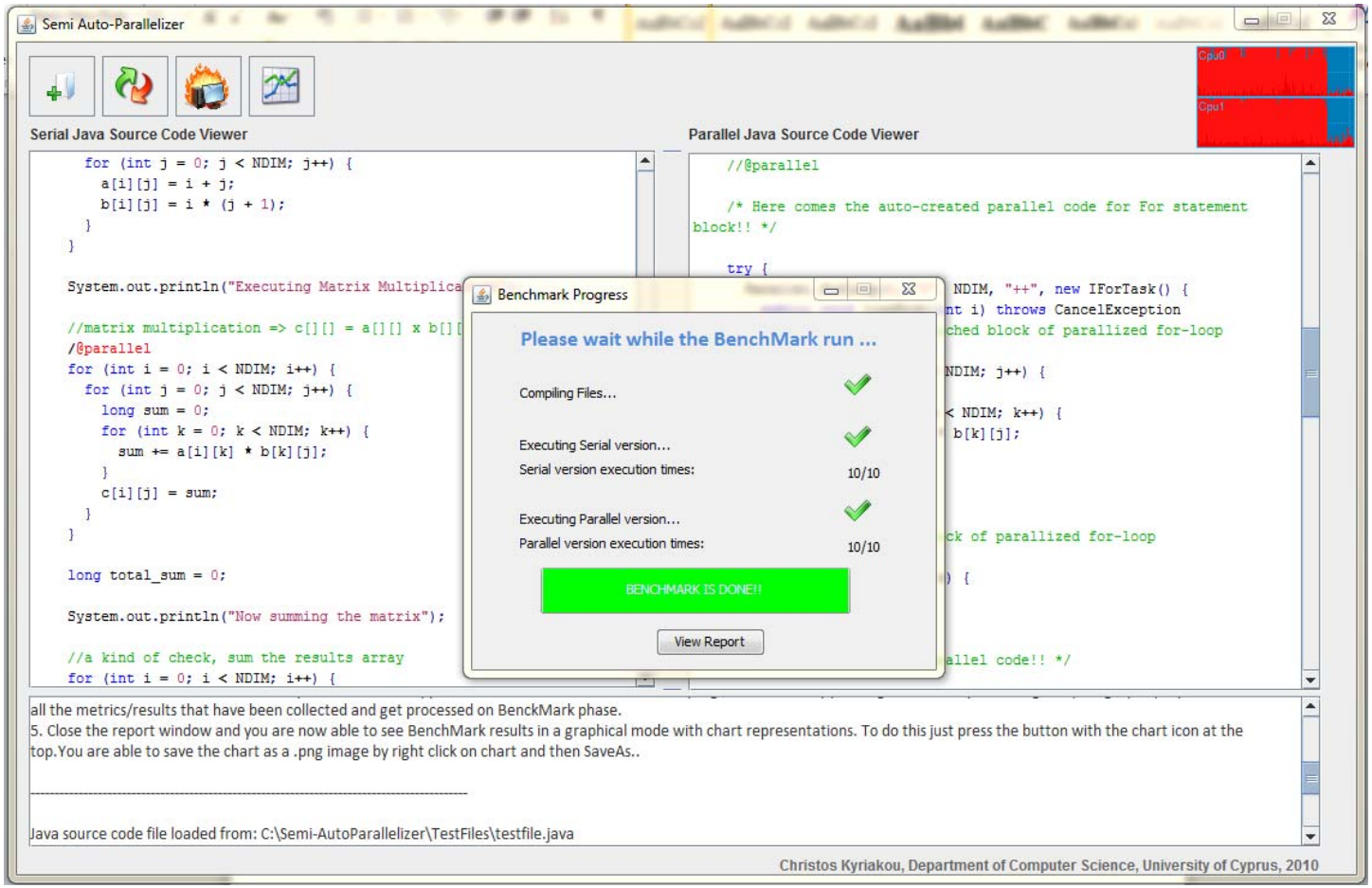
Εικόνα 4.16 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο για τον καθορισμό των βασικών ρυθμίσεων της διαδικασίας ελέγχου επίδοσης.



Εικόνα 4.17 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο που δείχνει την πρόοδο/εξέλιξη της σειριακής εκτέλεσης για το BenchMark.



Εικόνα 4.18 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο που δείχνει την πρόοδο/εξέλιξη της παράλληλης εκτέλεσης για το BenchMark.

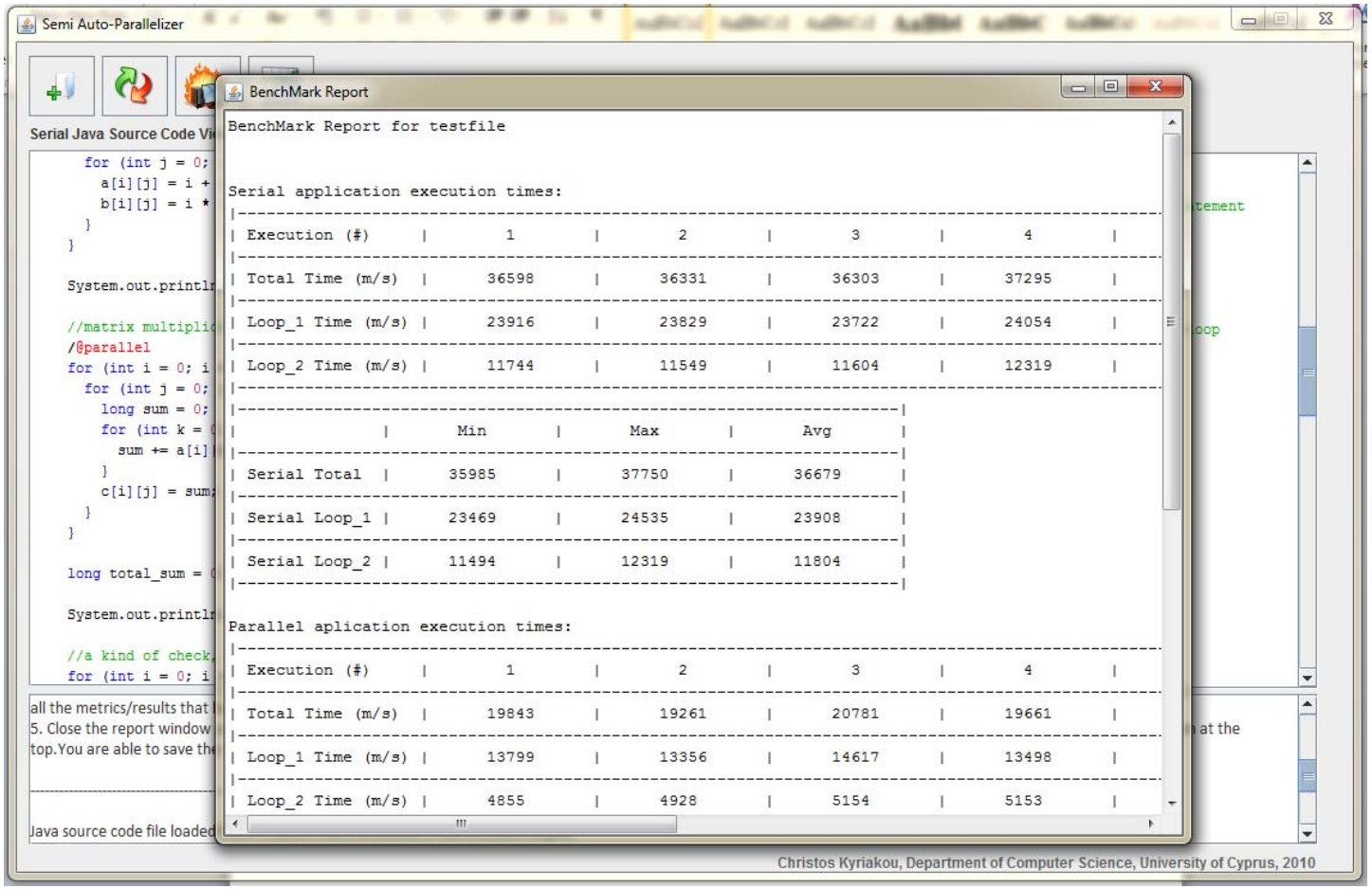


Εικόνα 4.19 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο που δείχνει ολοκληρωμένα με επιτυχία όλα τα βήματα της διαδικασίας ελέγχου της επίδοσης (BenchMark).

Πατώντας το κουμπί ελέγχου της επίδοσης εκτός από το παράθυρο για καθορισμό των βασικών ρυθμίσεων του BenchMark, στην πάνω δεξιά γωνία του κεντρικού παραθύρου του εργαλείου, εμφανίζεται γραφική αντιπροσώπευση του ποσοστού χρήσης (utilization) των διαθέσιμων επεξεργαστών που μας παρέχει το υπολογιστικό σύστημα στο οποίο εργαζόμαστε. Αυτό είναι χρήσιμο για το χρήστη αφού ξεκινήσει το πρόγραμμα ελέγχου επίδοσης και αρχίσει η εκτέλεση των προγραμμάτων. Με τη βοήθεια της γραφικής αναπαράστασης του ποσοστού χρήσης (utilization) των διαθέσιμων επεξεργαστών θα μπορεί πολύ εύκολα να παρατηρήσει ότι κατά την διάρκεια εκτέλεσης της σειριακής έκδοσης χρησιμοποιείται μόνο ένας επεξεργαστής, ενώ κατά την διάρκεια εκτέλεσης της παράλληλης έκδοσης χρησιμοποιούνται τόσοι επεξεργαστές όσοι αποφασίστηκαν στο προηγούμενο βήμα του καθορισμού των βασικών ρυθμίσεων [Εικόνα 4.17 και Εικόνα 4.18].

Αφού ο χρήστης καθορίσει τις βασικές ρυθμίσεις του BenchMark, αριθμό των επαναλήψεων εκτέλεσης και αριθμό επεξεργαστών προς χρήση, είναι έτοιμος να ξεκινήσει την διαδικασία ελέγχου της επίδοσης πατώντας το κουμπί *GO*. Όταν το κάνει αυτό, το παράθυρο των βασικών ρυθμίσεων κλείνει και ανοίγει ένα παράθυρο (BenchMark Progress) που δείχνει την πρόοδο/εξέλιξη της εκτέλεσης για το BenchMark. Αρχικά μεταγλωττίζονται τα αρχεία με τον κώδικα που δημιουργήσαμε, τόσο της σειριακής όσο και της παράλληλης έκδοσης, ενώ στην συνέχεια ξεκινά πρώτα η εκτέλεση της σειριακής έκδοσης [Εικόνα 4.17] και αφού αυτή ολοκληρωθεί ξεκινά και η εκτέλεση της παράλληλης έκδοσης [Εικόνα 4.18].

Μετά το πέρας της εκτέλεσης και της παράλληλης έκδοσης, δεδομένου ότι όλα τα βήματα εκτελέστηκαν κανονικά χωρίς να προκύψει οποιοδήποτε πρόβλημα, στο κάτω μέρος του παραθύρου που εμφανίζει την εξέλιξη προόδου του BenchMark εμφανίζεται ένα κουμπί με την ονομασία *View Report* [Εικόνα 4.19]. Πατώντας το κουμπί *View Report*, το παράθυρο προόδου του BenchMark καθώς και η γραφική αντιπροσώπευση του ποσοστού χρήσης (utilization) των διαθέσιμων επεξεργαστών κλείνουν και εμφανίζεται ένα παράθυρο (BenchMark Report) το οποίο παρουσιάζει τα αποτελέσματα που προέκυψαν από την εκτέλεση της διαδικασίας ελέγχου της επίδοσης [Εικόνα 4.20].



Εικόνα 4.19 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο το οποίο παρουσιάζει τα αποτελέσματα που προέκυψαν από την εκτέλεση της διαδικασίας ελέγχου της επίδοσης.

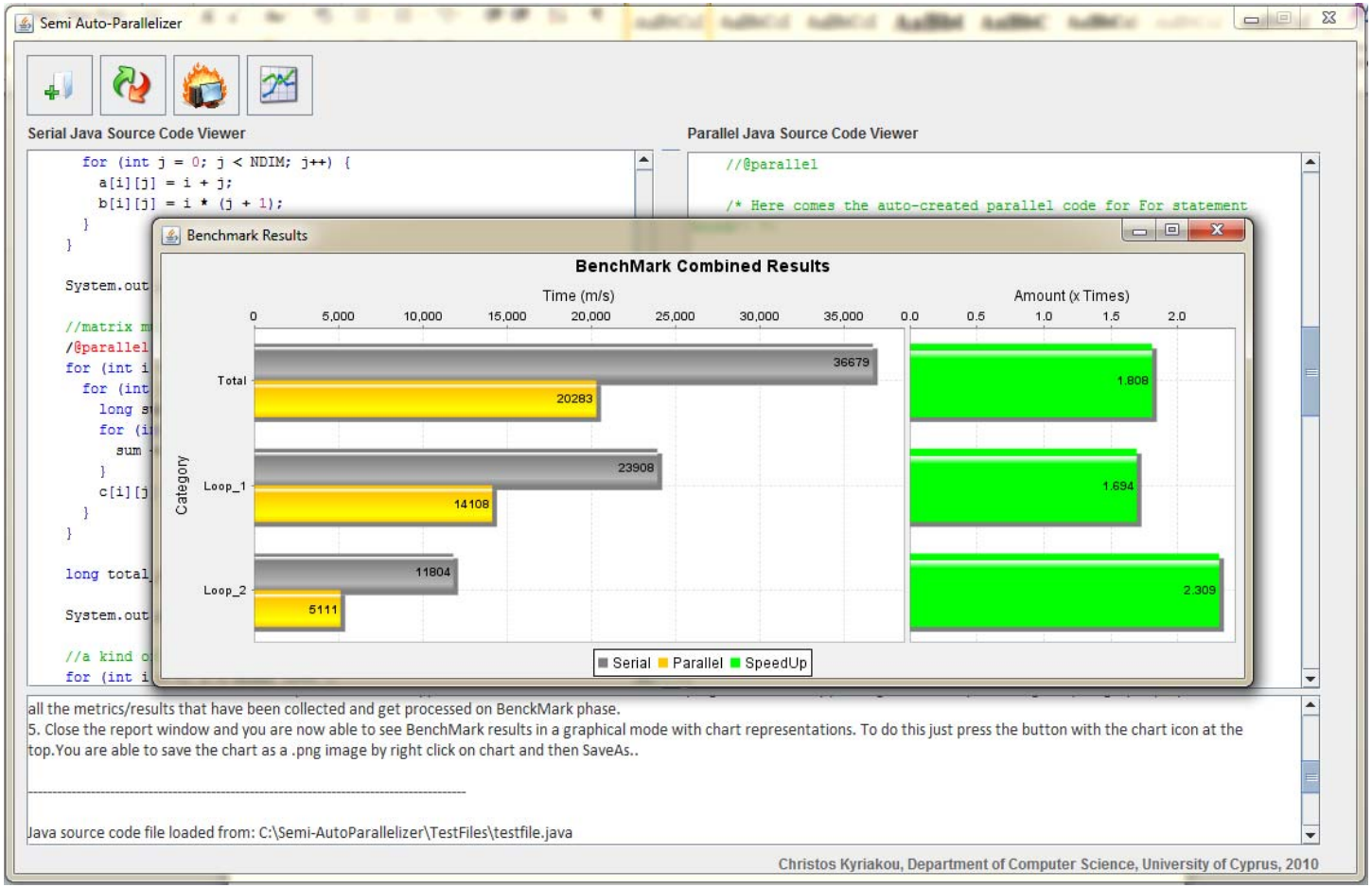
Στο παράθυρο με τα αποτελέσματα που προέκυψαν από την εκτέλεση της διαδικασίας ελέγχου της επίδοσης παρουσιάζονται αναλυτικά όλοι οι χρόνοι εκτέλεσης που μετρήθηκαν για κάθε έκδοση προγράμματος που δημιουργήθηκε προηγουμένως, καθώς και τα ποσοστά βελτίωσης (speedup) που πέτυχαν. Αναλυτικότερα, αρχικά παρουσιάζονται οι συνολικοί χρόνοι εκτέλεσης ολόκληρου του προγράμματος της σειριακής έκδοσης για κάθε εκτέλεση καθώς και απομονωμένοι οι χρόνοι εκτέλεσης των blocks των βρόγχων επανάληψης που στην παράλληλη έκδοση έχουν τύχει παραλληλοποίησης. Ακολουθεί ένας πίνακας που συνοψίζει όλους τους προηγούμενους χρόνους για κάθε κατηγορία (συνολικό χρόνο προγράμματος και επιμέρους χρόνους μόνο για βρόγχους επανάληψης) σε τρείς ομάδες, ελάχιστο χρόνο εκτέλεσης (Min), μέγιστο χρόνο εκτέλεσης (Max) και μέσο όρο χρόνου εκτέλεσης (Avg). Στην συνέχεια κατά παρόμοιο τρόπο παρουσιάζονται και οι χρόνοι εκτέλεσης για την παράλληλη έκδοση του προγράμματος. Τέλος υπάρχει ένας πίνακας με τις τιμές/ποσοστά βελτίωσης της απόδοσης (speedup) της παράλληλης έκδοσης σε σχέση με την σειριακή, τόσο σε επίπεδο συνολικού προγράμματος όσο και σε επίπεδο των επιμέρους βρόγχων επανάληψης που έτυχαν τροποποίησης στην παράλληλη έκδοση. Οι τιμές για το speedup προκύπτουν από τον λόγο του παράλληλου χρόνου εκτέλεσης προς τον σειριακό για κάθε κατηγορία ξεχωριστά (συνολικό πρόγραμμα και επιμέρους βρόχοι επανάληψης).

Ο λόγος που αποφασίσαμε να λαμβάνονται μετρικές τόσο για επίπεδο συνολικού χρόνου εκτέλεσης προγράμματος όσο και για επίπεδο χρόνου εκτέλεσης των επιμέρους βρόγχων επανάληψης που έτυχαν τροποποίησης στην παράλληλη έκδοση, είναι για να μπορεί πιο εύκολα και αποτελεσματικά ο χρήστης να δει ακριβώς πόση βελτίωση στην επίδοση επωφελείται κάνοντας χρήση των εργαλείων μας. Με το να λαμβάνονται μετρικές για τους χρόνους εκτέλεσης ξεχωριστά των βρόγχων επανάληψης που στοχεύουμε να βελτιώσουμε σε σχέση με το συνολικό πρόγραμμα μπορούμε πιο εύκολα να αντιληφθούμε τον βαθμό βελτίωσης που επιτυγχάνουμε. Όπως αναφέραμε και στο Κεφάλαιο 2, μέσα από τον νόμο του Amdahl καταλήγουμε στο συμπέρασμα ότι το speedup (βελτίωση στην επίδοση) που μπορούμε να πάρουμε από μια εφαρμογή που χρησιμοποιεί πολλαπλούς επεξεργαστές σε έναν παράλληλο υπολογισμό, περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει. Άρα ο

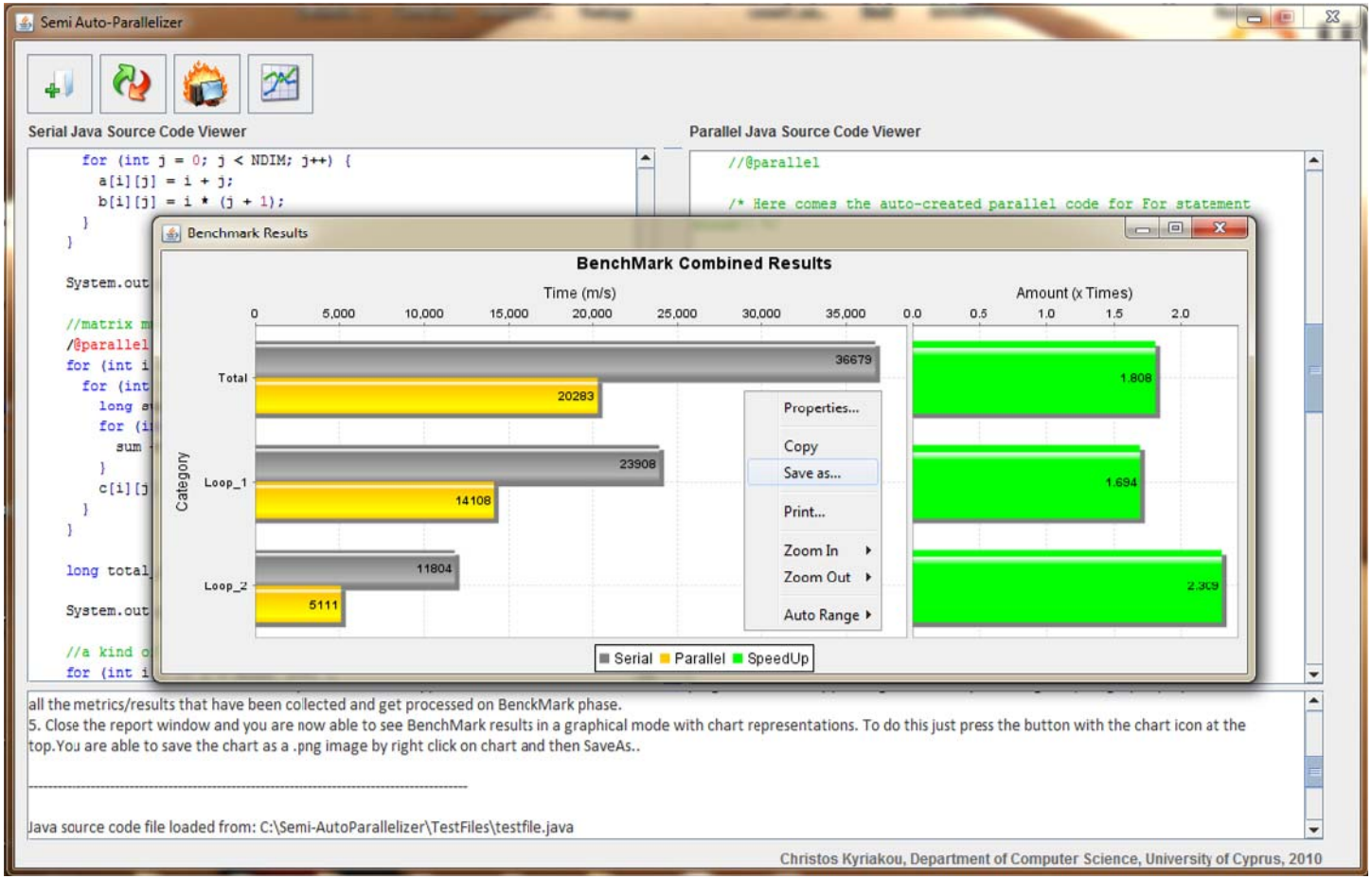
μόνος τρόπος για να δούμε τις πραγματικές βελτιώσεις στην επίδοση μιας παράλληλης εφαρμογής είναι να μελετήσουμε απομονωμένα τα κομμάτια τα οποία έτυχαν επεξεργασίας για να εκτελούν παράλληλο υπολογισμό.

Οτιδήποτε εμφανίζεται στο παράθυρο των αποτελεσμάτων που προέκυψαν από την εκτέλεση της διαδικασίας ελέγχου της επίδοσης αυτόματα αποθηκεύονται σε ένα αρχείο με όνομα ίδιο με το βασικό όνομα της κλάσης που αποτελεί το πρόγραμμα σε Java το οποίο χειριζόμαστε. Το αρχείο με τα αποτελέσματα αυτά αποθηκεύεται στον φάκελο *Results* που έχει δημιουργηθεί στον κατάλογο εγκατάστασης την εφαρμογής (*C:\Semi-AutoParallelizer*) με όνομα όπως αναφέραμε πιο πάνω και προέκταση *.txt*. Κάθε φορά που τρέχουμε την διαδικασία ελέγχου της επίδοσης για το ίδιο αρχικό σειριακό πρόγραμμα (με κοινές ή διαφορετικές τροποποιήσεις κάθε φορά) τα αποτελέσματα προσαρτούνται (append) στο αρχείο αποτελεσμάτων, αν αυτό υπάρχει, σημειώνοντας την ημερομηνία και ώρα τερματισμού της εκτέλεσης της διαδικασίας BenchMark κάθε φορά για να μπορεί ο χρήστης να έχει συγκριτικά αποτελέσματα για την εφαρμογή του στα οποία να μπορεί να ανατρέξει σε μελλοντικό στάδιο.

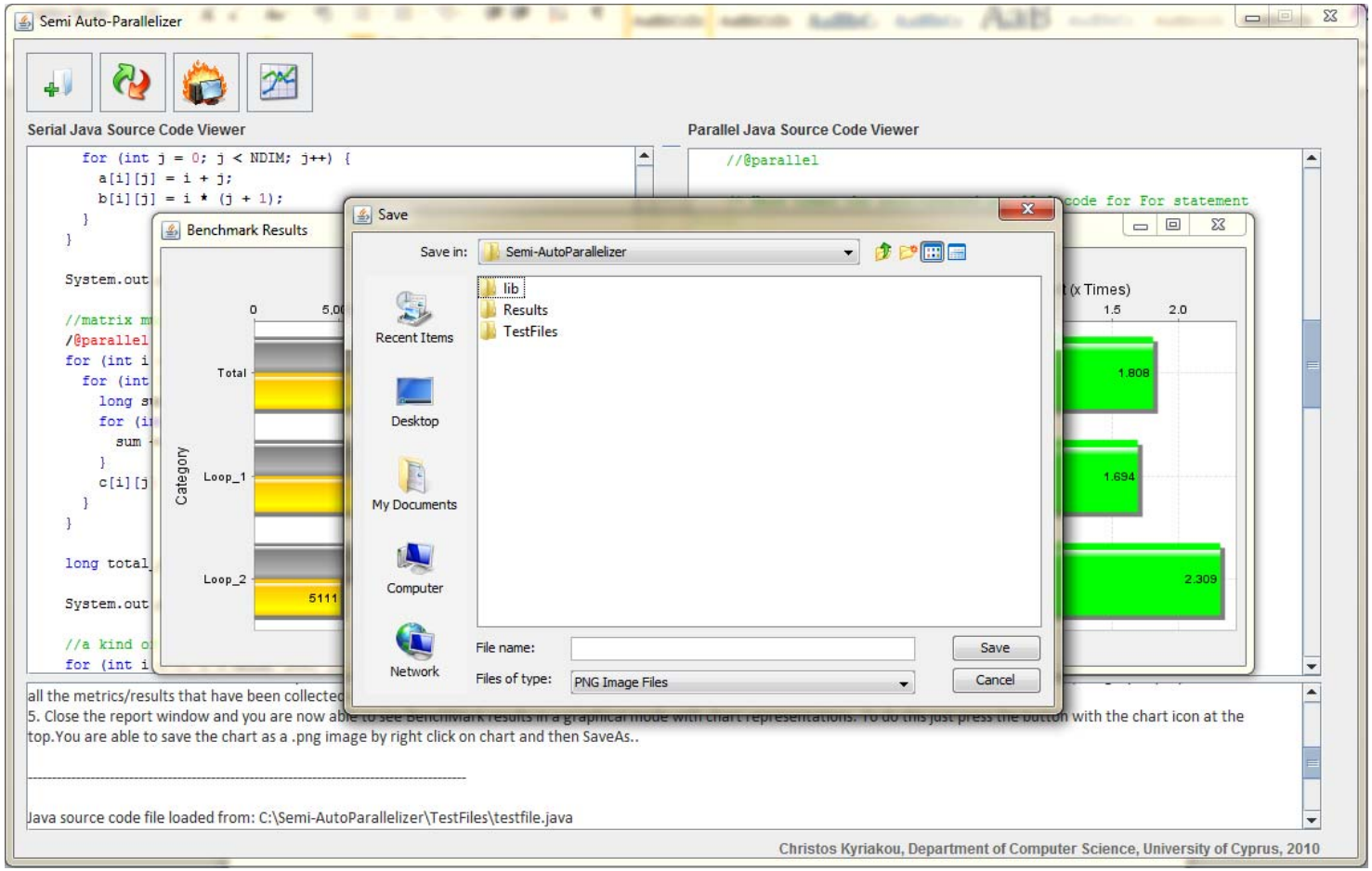
Κλείνοντας το παράθυρο αποτελεσμάτων της διαδικασίας ελέγχου της επίδοσης (BenchMark) μπορούμε να προχωρήσουμε και να δούμε κάποιες γραφικές παραστάσεις που μπορούν να δημιουργηθούν με βάση τα προηγούμενα αποτελέσματα. Για να το κάνουμε αυτό, απλά επιλέγουμε το κουμπί που εμφανίζεται δίπλα από το κουμπί εκκίνησης της διαδικασία ελέγχου της επίδοσης. Πατώντας το κουμπί παρουσίασης των γραφικών παραστάσεων  ανοίγει ένα παράθυρο το ποίο εμφανίζει γραφικές αναπαραστάσεις των αποτελεσμάτων που μόλις πήραμε εκτελώντας το BenchMark. Στο παράθυρο δημιουργείται μια γραφική παράσταση που εμφανίζει τον μέσο όρο χρόνου εκτέλεσης τόσο του συνολικού προγράμματος όσο και των επιμέρους βρόγχων επανάληψης που έτυχαν τροποποίησης της σειριακής και παράλληλης εκτέλεσής τους. Επίσης στο ίδιο παράθυρο εμφανίζεται και γραφική παράσταση με τις τιμές βελτίωσης/επιδείνωσης της απόδοσης (speedup) της παράλληλης εκτέλεσης σε σχέση με τη σειριακή [Εικόνα 4.20].



Εικόνα 4.20 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο που περιέχει τις γραφικές παραστάσεις των αποτελεσμάτων που προέκυψαν από την εκτέλεση της διαδικασίας ελέγχου της επίδοσης.



Εικόνα 4.21 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο που περιέχει τις γραφικές παραστάσεις και το μενού επιλογών που έχει στην διάθεσή του ο χρήστης.



Εικόνα 4.22 : Παράθυρο του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα με το παράθυρο πλοήγησης και αποθήκευσης των γραφικών παραστάσεων σε μορφή εικόνας.

Ο χρήστης έχει την δυνατότητα να αποθηκεύσει το παράθυρο αυτό με τις δύο γραφικές παραστάσεις σε μορφή εικόνας. Για να το κάνει αυτό απλά πατά δεξί κλικ μέσα στο παράθυρο και του εμφανίζεται ένα μενού επιλογών [Εικόνα 4.21]. Από το μενού επιλογών επιλέγοντας το *Save as...* ανοίγει ένα νέο παράθυρο από το οποίο καλείται να πλοηγηθεί και να επιλέξει κατάλογο στον οποίον θέλει να αποθηκεύσει τις γραφικές παραστάσεις [Εικόνα 4.22]. Αφού επιλέξει τον επιθυμητό κατάλογο θα πρέπει στην συνέχεια να δώσει ένα όνομα για το αρχείο εικόνας που θα αποθηκευθεί και τέλος να πατήσει το κουμπί *Save*. Ο χρήστης έχει την δυνατότητα να αποθηκεύσει τις γραφικές παραστάσεις μόνο σε μορφή εικόνας τύπου *PNG*.

4.5 Ανασκόπηση επόμενου κεφαλαίου

Στο επόμενο κεφάλαιο [Κεφάλαιο 5] θα προσπαθήσουμε να επαληθεύσουμε και να πιστοποιήσουμε τις εισηγήσεις που προτάθηκαν για τη βελτιστοποίηση προγραμμάτων σε Java. Θα αναλύσουμε μερικές πειραματικές μετρήσεις/αποτελέσματα της επίδοσης διαφόρων προγραμμάτων που μπορούν να παραλληλιστούν με τα εργαλεία που έχουμε δημιουργήσει μέσα από γραφικές παραστάσεις με χρόνους εκτέλεσης για διαφορετικό αριθμό πυρήνων πάνω στο ίδιο παράλληλο σύστημα.

Κεφάλαιο 5

Πειραματικές μετρήσεις

5.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα προσπαθήσουμε μέσα από πειραματικές μετρήσεις και συγκριτικά αποτελέσματα επιδόσεων να επαληθεύσουμε και να πιστοποιήσουμε τις εισηγήσεις που προτάθηκαν για τη βελτιστοποίηση προγραμμάτων σε Java όταν αυτά τρέχουν σε ένα πολυπύρηνο υπολογιστικό σύστημα με αρχιτεκτονική Κοινής Μνήμης (Shared Memory). Κάνοντας χρήση του προγράμματος ελέγχου της επίδοσης (BenchMark) μέσα από το γραφικό εργαλείο αυτόματης μετάφρασης κώδικα που έχει υλοποιηθεί στα πλαίσια της διπλωματικής εργασίας, το οποίο και παρουσιάσαμε στο προηγούμενο κεφάλαιο, θα εκτελέσουμε πειραματικές μετρήσεις για ένα σύνολο προγραμμάτων.

Τα προγράμματα [Παράρτημα Β] τα οποία θα χρησιμοποιήσουμε για τις πειραματικές μετρήσεις είναι όλα γραμμένα σε γλώσσα προγραμματισμού Java και στο σύνολό τους είναι εφτά (7). Στην συνέχεια παρατίθενται και τα εφτά αυτά προγράμματα με μια μικρή περιγραφή για το τι ακριβώς υλοποιεί το καθένα :

1. *Black_White_Game.java*

   Υλοποίηση ενός παιχνιδιού με μία σκακιέρα από μαύρα και άσπρα πιόνια που προσομοιώνει τον πιο κάτω αλγόριθμο και κανόνες παιχνιδιού :

   - Σε ένα πίνακα 1024x1024 υπάρχουν περίπου 25% μαύρα (B), 25% άσπρα (W) και 50% κενά (E) τετράγωνα.
   - Σε κάθε γύρο του παιχνιδιού όλα τα άσπρα (W) και μαύρα (B) τετράγωνα μπορούν να μετακινηθούν μια θέση.
   - Πρώτα μετακινούνται όλα τα άσπρα (W) και στην συνέχεια όλα τα μαύρα (B).
   - Τα άσπρα (W) μπορούν να κινηθούν ΜΟΝΟ ΔΕΞΙΑ και δεδομένου ότι το δεξί τετραγωνάκι είναι κενό (E).
   - Τα μαύρα (B) μπορούν να κινηθούν ΜΟΝΟ ΚΑΤΩ και δεδομένου ότι το από κάτω τετραγωνάκι είναι κενό (E).
   - Μετά τις μετακινήσεις, ολόκληρος ο πίνακας ελέγχεται αν υπάρχει κάποιος υποπίνακας 9x9 στον οποίον όλα τα τετραγωνάκια να είναι κενά (E). Σε περίπτωση που βρεθεί τέτοιος πίνακας τότε το παιχνίδι τερματίζει.

   Αν δεν βρεθεί ο κενός 9x9 υποπίνακας τότε το παιχνίδι επαναλαμβάνεται. Το παιχνίδι τελειώνει μετά από 100 γύρους αν δεν βρεθεί κενός υποπίνακας.

2. *ComputePi.java*

   Εφαρμογή υπολογισμού της μαθηματικής σταθεράς **π** που είναι ένας πραγματικός αριθμός και μπορεί να οριστεί ως ο λόγος του μήκους της περιφέρειας ενός κύκλου προς τη διάμετρό του στην Ευκλείδεια γεωμετρία. Ο αλγόριθμος υπολογιστικής προσέγγισης που χρησιμοποιήσαμε είναι η πιο κάτω σειρά :

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots.$$

Εικόνα 5.1 : Ο αλγόριθμος υπολογιστικής προσέγγισης της μαθηματικής σταθεράς π .

3. *Count3s.java*

   Πρόγραμμα που καταμετρά τις εμφανίσεις του αριθμού 3 κάνοντας προσπέλαση σε ένα μονοδιάστατο πίνακα που περιέχει ακέραιους αριθμούς από το 0 μέχρι το 4 κατανεμημένους σε τυχαία σειρά μέσα στον πίνακα.

4. *DCT2Algorithm.java*

   Η διακριτή μετατροπή συνημίτονου (discrete cosine transform - DCT) βοηθά στον διαχωρισμό μιας εικόνας σε διάφορα μέρη (ή φασματικές υποζώνες) διαφορετικής σημασίας (όσον αφορά την ποιότητα της εικόνας). Ο αλγόριθμος DCT είναι κάτι παρόμοιο με τον αλγόριθμο διακριτού μετασχηματισμού Fourier (discrete Fourier transform) που χρησιμοποιείται για να μετασχηματίζει ένα σήμα ή μια εικόνα από τη χωρική περιοχή (spatial domain) στο πεδίο συχνότητάς (frequency domain) της. Ο αλγόριθμος υπολογιστικής προσέγγισης που χρησιμοποιήσαμε φαίνεται πιο κάτω :

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{matrix}$$

where and

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}}, & p = 0 \\ \sqrt{\frac{2}{M}}, & 1 \le p \le M\text{-}1 \end{cases} \qquad \alpha_q = \begin{cases} \frac{1}{\sqrt{N}}, & q = 0 \\ \sqrt{\frac{2}{N}}, & 1 \le q \le \text{N-}1 \end{cases}$$

Εικόνα 5.2 : Ο αλγόριθμος υπολογιστικής προσέγγισης της διακριτής μετατροπής συνημίτονου (DCT).

5. *Mandelbrot.java*

   Το σύνολο Mandelbrot, είναι ένα μαθηματικό σύνολο σημείων στο μιγαδικό επίπεδο, το όριο του οποίου αποτελεί ένα fractal (ένα κατακερματισμένο γεωμετρικό σχήμα που μπορεί να χωριστεί σε τμήματα, καθένα από τα οποία (τουλάχιστον κατά προσέγγιση) αποτελεί αντίγραφο του συνόλου του σε μειωμένο όμως μέγεθος). Το σύνολο Mandelbrot αποτελεί το σύνολο όλων

των μιγαδικών αριθμών *z* όπου η ακολουθία τους που ορίζεται από την επανάληψη παραμένει οριοθετημένη (bounded). Αυτό σημαίνει ότι υπάρχει ένας αριθμός B έτσι ώστε η απόλυτη τιμή όλων των τιμών *z(n)* που επαναλαμβάνονται δεν ξεπερνούν ποτέ την τιμή του B. Η οριοθετημένη ακολουθία μπορεί να έχει ή όχι ένα όριο. Για παράδειγμα, αν *z = 0* τότε *z (n) = 0* για κάθε *n*, έτσι ώστε το όριο της ακολουθίας που φαίνεται πιο κάτω (1) να είναι μηδενική. Από την άλλη πλευρά, αν *z = i* (i είναι η φανταστική μονάδα), τότε η ακολουθία ταλαντεύεται μεταξύ *i* και *i-1*, οπότε εξακολουθεί να οριοθετείται αλλά δεν συγκλίνει σε ένα όριο.

$$z(0) = z,\ z_{n+1} = z_n^2 + c,\ n=0,1,2,\ \ldots \qquad (1)$$

Περιορίζοντας την μεταβλητή n στην φόρμουλα (1) στο πεδίο των πραγματικών αριθμών μεταξύ -2 έως 0 παίρνουμε μια πιο κατανοητή, μεγαλύτερης όμως πολυπλοκότητας, εικόνα σε σχέση με την απλή, κενή, εικόνα που θα παίρναμε αν οι τιμές του n ορίζονταν στο πεδίο από 0 έως άπειρο.

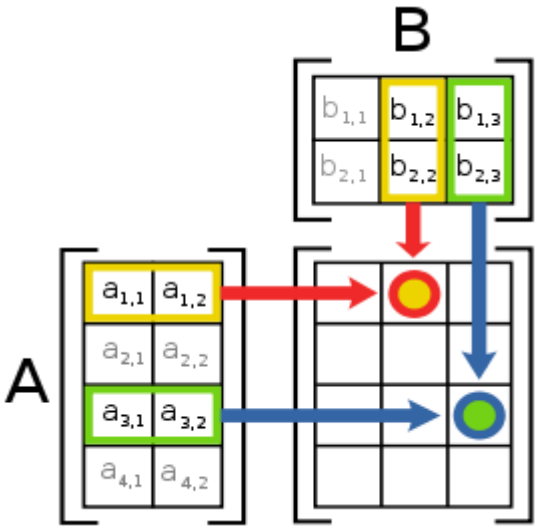
6. *MatrixMult.java*

   Εφαρμογή πολλαπλασιασμού δύο δισδιάστατων πινάκων. Το αποτέλεσμα που προκύπτει από τον πολλαπλασιασμό των δύο αυτών πινάκων αποθηκεύεται σε έναν τρίτο πίνακα. Ο πολλαπλασιασμός γίνεται με βάση τον τύπο και την φόρμουλα στην Εικόνα 5.3 .

7. *GameOfLife.java*

   Το Game of Life ή αλλιώς Conway's Game είναι ένα κυψελοειδές αυτόματο παιχνίδι που επινοήθηκε από το βρετανικό μαθηματικό John Horton Conway το 1970. Το Game of Life είναι ένα παιχνίδι που δεν χρειάζεται συμμετοχή παικτών, που σημαίνει ότι η εξέλιξή του καθορίζεται από την αρχική κατάσταση και δεν απαιτεί καμία περαιτέρω εισαγωγή στοιχείων. Τον κόσμο του παιχνιδιού αποτελεί ένα άπειρο δισδιάστατο ορθογώνιο πλέγμα των τετραγωνικών κελιών, κάθε ένα από τα οποία μπορεί να βρίσκεται σε μία από

$$(AB)_{i,j} = \sum_{k=1}^{p} A_{ik} B_{kj},$$



Εικόνα 5.3 : Ο αλγόριθμος και η φόρμουλα υπολογιστικής προσέγγισης πολλαπλασιασμού δύο δισδιάστατων πινάκων.

Οι κανόνες πάνω στους οποίους στηρίζεται η λειτουργία του παιχνιδιού και ισχύουν για κάθε βήμα είναι οι εξής :

- Οποιοδήποτε ζωντανό κελί με λιγότερους από δύο ζωντανούς γείτονες πεθαίνει (under-population).
- Οποιοδήποτε ζωντανό κελί με περισσότερους από τρεις ζωντανούς γείτονες πεθαίνει, (overcrowding).
- Οποιοδήποτε ζωντανό κελί με δύο ή τρεις ζωντανούς γείτονες ζει μέχρι την επόμενη γενεά.
- Οποιοδήποτε νεκρό κελί με ακριβώς τρεις ζωντανούς γείτονες γίνεται ζωντανό (reproduction).

Η πρώτη "γενιά" δημιουργείται εφαρμόζοντας τους πιο πάνω κανόνες ταυτόχρονα σε κάθε κελί. Οι κανόνες συνεχίζουν να εφαρμόζονται

επανειλημμένα για να δημιουργήσουν τις περαιτέρω γενεές οι οποίες στην ουσία είναι άπειρες.

5.2 Πειραματικές μετρήσεις και συγκριτικά αποτελέσματα επιδόσεων

Όλες οι πειραματικές μετρήσεις έγιναν στο ίδιο υπολογιστικό σύστημα για να μπορούν τα αποτελέσματα που παράχθηκαν να είναι συγκρίσιμα μεταξύ τους. Το υπολογιστικό σύστημα που χρησιμοποιήσαμε για τις πειραματικές μας μετρήσεις ήταν ένας διακομιστής/εξυπηρετητής τύπου BladeServer ο οποίος ενσωματώνει τέσσερις (4) επεξεργαστές τύπου Intel(R) Xeon(R) CPU E7450 όπου ο καθένας έχει έξι (6) πυρήνες (cores). Ο κάθε πυρήνας έχει δυνατότητα διαχείρισης ενός μόνο νήματος (thread) και τρέχει σε συχνότητες των 2.4 GHz. Άρα στο σύνολό του το υπολογιστικό σύστημα που χρησιμοποιήσαμε έχει υπολογιστική ισχύ της τάξης των 57.6 GHz αφού στην ουσία αποτελείται από 24 πυρήνες σύνολο (4 επεξεργαστές x 6 πυρήνες), όπου ο καθένας τους τρέχει στα 2.4 GHz. Το σύστημα διέθετε επίσης 32 GB φυσική μνήμη (physical memory/RAM) και το λειτουργικό σύστημα το οποίο έτρεχε ήταν Microsoft Windows Server 2003 Enterprise Edition Service Pack 2 (Build 3790).

Το σενάριο που ακολουθήθηκε σε όλες τις εκτελέσεις των πειραμάτων για κάθε ένα από τα 7 προγράμματα ελέγχου ήταν το εξής:

1. Μέσω του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα (Semi-Auto Parallelizer) φορτώνεται η σειριακή έκδοση πηγαίου κώδικα του προγράμματος ελέγχου για το οποίο θα εκτελέσουμε πειραματικές μετρήσεις.
2. Επιλέγονται οι βρόγχοι επανάληψης τύπου for() που μπορούν και θέλουμε να τύχουν επεξεργασίας/βελτιστοποίησης (για να παραλληλιστούν) και προσθέτουμε το ανάλογο και σωστό σχόλιο παραλληλισμού που απαιτείται ακριβώς πριν από τον βρόγχο.
3. Πατούμε το κουμπί αυτόματης μετάφρασης κώδικα και το εργαλείο δημιουργεί αυτόματα την παράλληλη έκδοση του προγράμματος που θα χρησιμοποιηθεί στην συνέχεια.

4. Εκκινούμε το πρόγραμμα ελέγχου της επίδοσης (BenchMark) μέσα από το γραφικό εργαλείο αυτόματης μετάφρασης κώδικα και θέτουμε τον αριθμό εκτελέσεων (Execution Times) για κάθε έκδοση προγράμματος (σειριακή και παράλληλη) ίσο με 10. Επίσης θέτουμε τον αριθμό των πυρήνων (Number of Cores) που θέλουμε να συμμετέχουν κατά την διάρκεια εκτέλεσης ίσο με 24.
5. Πατώντας το κουμπί *GO* ξεκινούμε τη διαδικασία εκτέλεσης των δύο εκδόσεων προγραμμάτων για το εργαλείο BenchMark και περιμένουμε να τελειώσει η διαδικασία εκτέλεσης.
6. Αφού τελειώσει η διαδικασία εκτέλεσης και για τις δύο εκδόσεις προγραμμάτων και δούμε ότι παράχθηκε η έκθεση αναφοράς των αποτελεσμάτων (BenchMark Report) κλείνουμε το παράθυρο με τα αποτελέσματα.
7. Επαναλαμβάνουμε τα βήματα 4 έως 6 για 5 επιπλέον φορές. Σε κάθε επανάληψη εκτέλεσης των βημάτων αλλάζουμε μόνο τον αριθμό των πυρήνων (Number of Cores) που θέλουμε να συμμετέχουν κατά την διάρκεια εκτέλεσης ως εξής : 16, 8, 4, 2 και 1, αντίστοιχα για κάθε επανάληψη εκτέλεσης των βημάτων.

Ο λόγος που αποφασίσαμε να χρησιμοποιήσουμε το πρόγραμμα ελέγχου της επίδοσης (BenchMark) μέσα από το γραφικό εργαλείο αυτόματης μετάφρασης κώδικα για τα πειράματα και τις μετρικές μας είναι γιατί από το συγκεκριμένο εργαλείο λαμβάνονται μετρικές τόσο για επίπεδο συνολικού χρόνου εκτέλεσης προγράμματος, όσο και για επίπεδο χρόνου εκτέλεσης των επιμέρους βρόγχων επανάληψης που έτυχαν τροποποίησης για την παράλληλη έκδοση. Όπως αναφέραμε και προηγουμένως [Κεφάλαιο 4], τον διαχωρισμό αυτόν του χρόνου εκτέλεσης τον θεωρούμε σημαντικό για να μπορούμε πιο εύκολα και αποτελεσματικά να δούμε ακριβώς πόση βελτίωση στην απόδοση επωφελούμαστε κάνοντας χρήση των παράλληλων μεθόδων που υπάρχουν στην βιβλιοθήκη JACON σε συνδυασμό με την ευκολία που μας προσφέρουν τα εργαλεία αυτόματου παραλληλισμού που δημιουργήσαμε. Με το να λαμβάνονται μετρικές για τους χρόνους εκτέλεσης ξεχωριστά των βρόγχων επανάληψης που στοχεύουμε να βελτιώσουμε σε σχέση με το συνολικό πρόγραμμα μπορούμε πιο εύκολα να αντιληφθούμε τον βαθμό

βελτίωσης που επιτυγχάνουμε. Όπως προαναφέραμε και στο Κεφάλαιο 2, μέσα από τον νόμο του Amdahl καταλήγουμε στο συμπέρασμα ότι η βελτίωση στην επίδοση (speedup) που μπορούμε να πάρουμε από μια εφαρμογή που χρησιμοποιεί πολλαπλούς επεξεργαστές σε έναν παράλληλο υπολογισμό, περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει. Άρα ο μόνος τρόπος για να δούμε τις πραγματικές βελτιώσεις στην επίδοση μιας παράλληλης εφαρμογής είναι να μελετήσουμε απομονωμένα τα κομμάτια τα οποία έτυχαν επεξεργασίας για να εκτελούν παράλληλο υπολογισμό.

Πιο κάτω παρατίθενται οι πίνακες με τα συνολικά συγκριτικά αποτελέσματα για τους χρόνους εκτέλεσης και για τα 7 διαφορετικά προγράμματα ελέγχου που χρησιμοποιήσαμε. Στους πίνακες αυτούς μπορούμε να δούμε τους χρόνους εκτέλεσης για το κάθε πρόγραμμα ελέγχου ξεχωριστά, τόσο για επίπεδο συνολικού χρόνου εκτέλεσης προγράμματος, όσο και για επίπεδο χρόνου εκτέλεσης των επιμέρους βρόγχων επανάληψης που έτυχαν τροποποίησης για την παράλληλη έκδοση. Όλοι οι χρόνοι που παρουσιάζονται αποτελούν το μέσο όρο των χρόνων εκτέλεσης και των 10 επαναλήψεων εκτέλεσης για τις οποίες έτρεξαν τα προγράμματα ελέγχου κάθε φορά με διαφορετικό όμως αριθμό πυρήνων που μπορούσαν να συμμετάσχουν στην εκτέλεση.

Στους πίνακες παρουσιάζονται επίσης και οι βαθμοί βελτίωσης της επίδοσης (speedup) που προκύπτουν από τους χρόνους που έχουν ληφθεί. Ο βαθμός βελτίωσης της επίδοσης (speedup) υπολογίζεται ως ο λόγος του σειριακού χρόνου εκτέλεσης προς τον παράλληλο χρόνο εκτέλεσης.

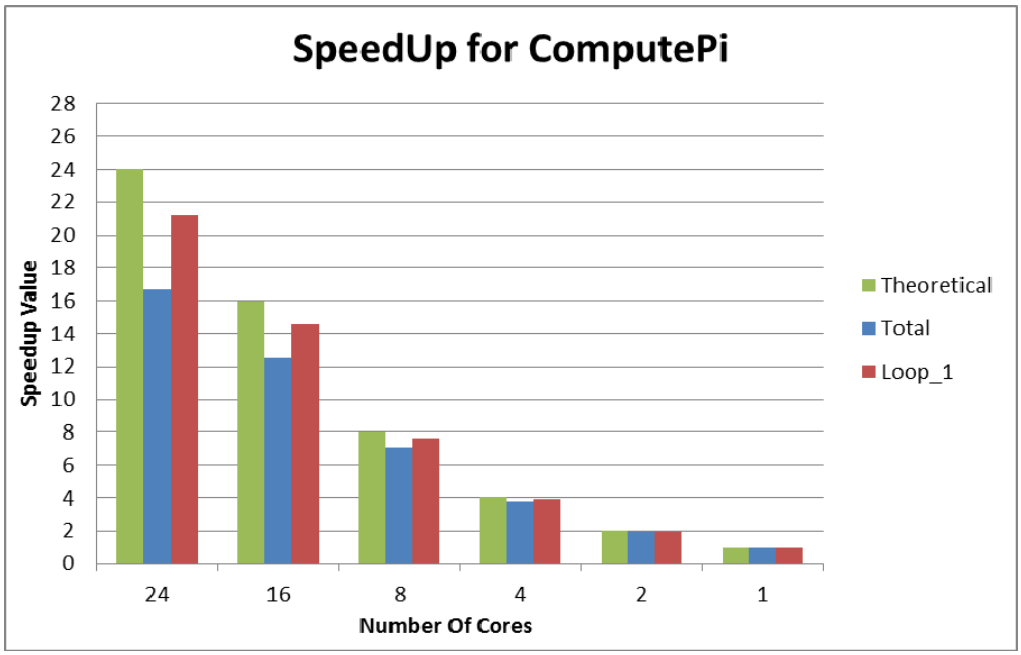
Τέλος, στους πίνακες παρουσιάζονται και τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για τις παράλληλες εκδόσεις προγραμμάτων που δημιουργούνται από τα εργαλεία αυτόματου παραλληλισμού μας. Τα ποσοστά αποδοτικότητας (efficiency) έχουν να κάνουν με το βαθμό βελτιστοποίησης της παράλληλης έκδοσης, ένατι της θεωρητικής μέγιστης βελτιστοποίησης που θα μπορούσαμε να επιτύχουμε σε ένα παράλληλο υπολογιστικό σύστημα με πολλαπλούς πυρήνες. Η θεωρητική μέγιστη βελτιστοποίηση (theoretical speedup) που μπορεί να επιτευχθεί

για μια πλήρη παραλληλοποιημένη (100%) πολυνηματική εφαρμογή που εκτελείται σε ένα παράλληλο υπολογιστικό σύστημα με πολλαπλούς πυρήνες είναι ίση με τον αριθμό των πυρήνων που χρησιμοποιούνται κάθε φορά στην επιμέρους εκτέλεση (π.χ. για ένα παράλληλο υπολογιστικό σύστημα με 24 πυρήνες η θεωρητική μέγιστη βελτιστοποίηση (theoretical speedup) που μπορεί να επιτευχθεί για μια παράλληλη έκδοση εφαρμογής είναι ίση με 24.0, δηλ. 24 φορές πιο γρήγορη εκτέλεση).

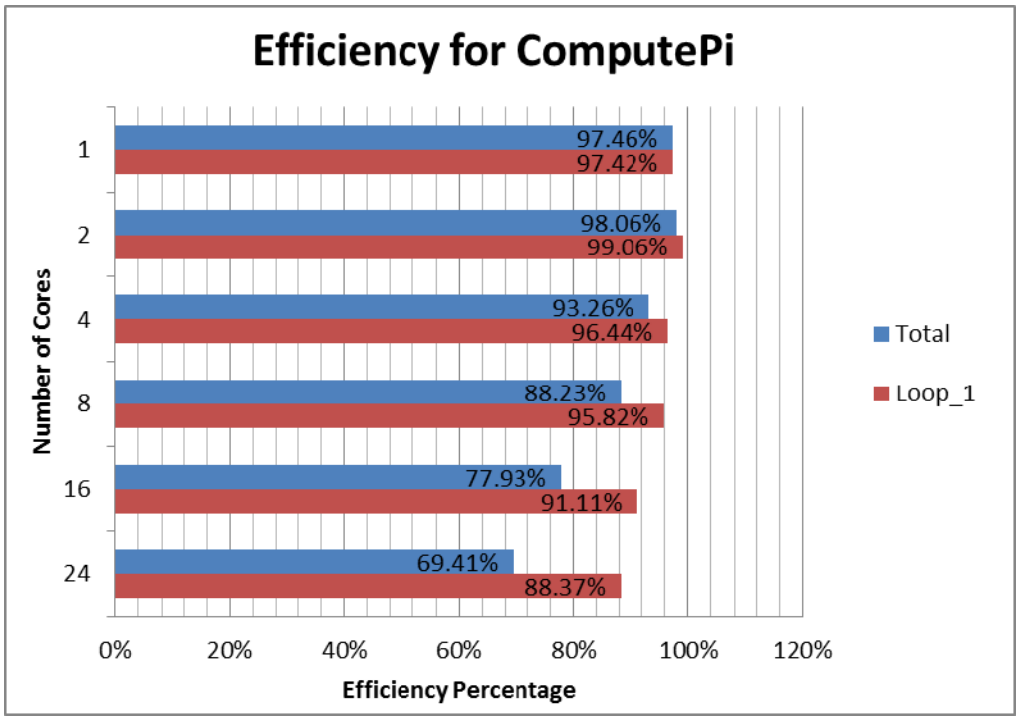
Εκτός από τους πίνακες με τα συνολικά συγκριτικά αποτελέσματα για τους χρόνους εκτέλεσης και για τα 7 διαφορετικά προγράμματα ελέγχου που χρησιμοποιήσαμε, παρατίθενται επίσης και οι γραφικές αναπαραστάσεις της βελτίωσης στην απόδοση (speedup) και του ποσοστού αποδοτικότητας (efficiency) που προκύπτουν για τις παράλληλες εκδόσεις προγραμμάτων που δημιουργούνται από τα εργαλεία αυτόματου παραλληλισμού μας. Μέσω των γραφικών αναπαραστάσεων είναι πιο εύκολο για μας να αναλύσουμε τα συγκριτικά αποτελέσματα που παράχθηκαν από τα πειράματα τα οποία τρέξαμε. Στην γραφική αναπαράσταση της βελτίωσης στην απόδοση (speedup) εκτός από το βαθμό βελτίωσης της επίδοσης σε επίπεδο συνολικού χρόνου εκτέλεσης προγράμματος και σε επίπεδο χρόνου εκτέλεσης των επιμέρους βρόγχων επανάληψης που έτυχαν τροποποίησης για την παράλληλη έκδοση, παρουσιάζεται επίσης και η θεωρητική μέγιστη βελτιστοποίηση (theoretical speedup) που θα μπορούσε να επιτευχθεί από κάθε παράλληλη έκδοση εφαρμογής σε σχέση με τον συνολικό αριθμό πυρήνων που συμμετείχαν κάθε φορά στην εκτέλεση.

ComputePi.java								
Cores #	**Serial Times**		**Parallel Times**		**SpeedUp Value**		**Efficiency Percentage**	
	Total	**Loop_1**	**Total**	**Loop_1**	**Total**	**Loop_1**	**Total**	**Loop_1**
24	19406	19152	1165	903	16.658	21.209	69.41%	88.37%
16	19413	19170	1557	1315	12.468	14.578	77.93%	91.11%
8	19453	19209	2756	2506	7.058	7.665	88.23%	95.82%
4	19338	19099	5184	4951	3.730	3.858	93.26%	96.44%
2	19501	19252	9943	9717	1.961	1.981	98.06%	99.06%
1	19351	19110	19856	19617	0.975	0.974	97.46%	97.42%

Πίνακας 5.1 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου ComputePi.java .



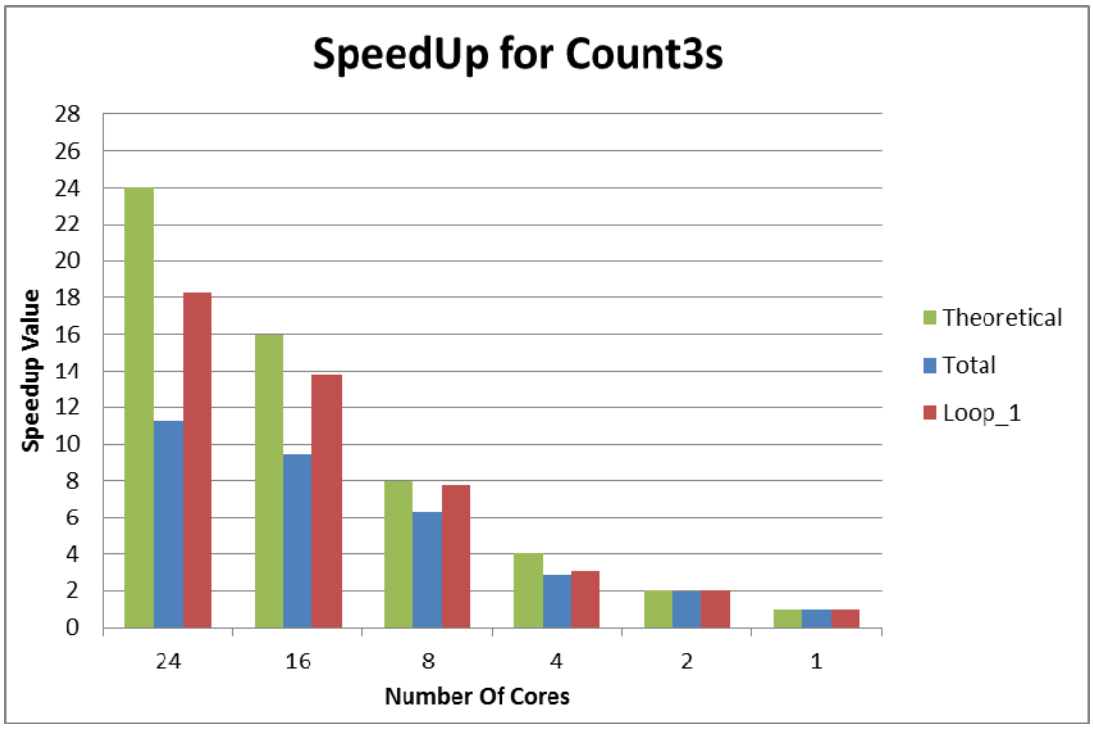
Εικόνα 5.3 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου ComputePi .



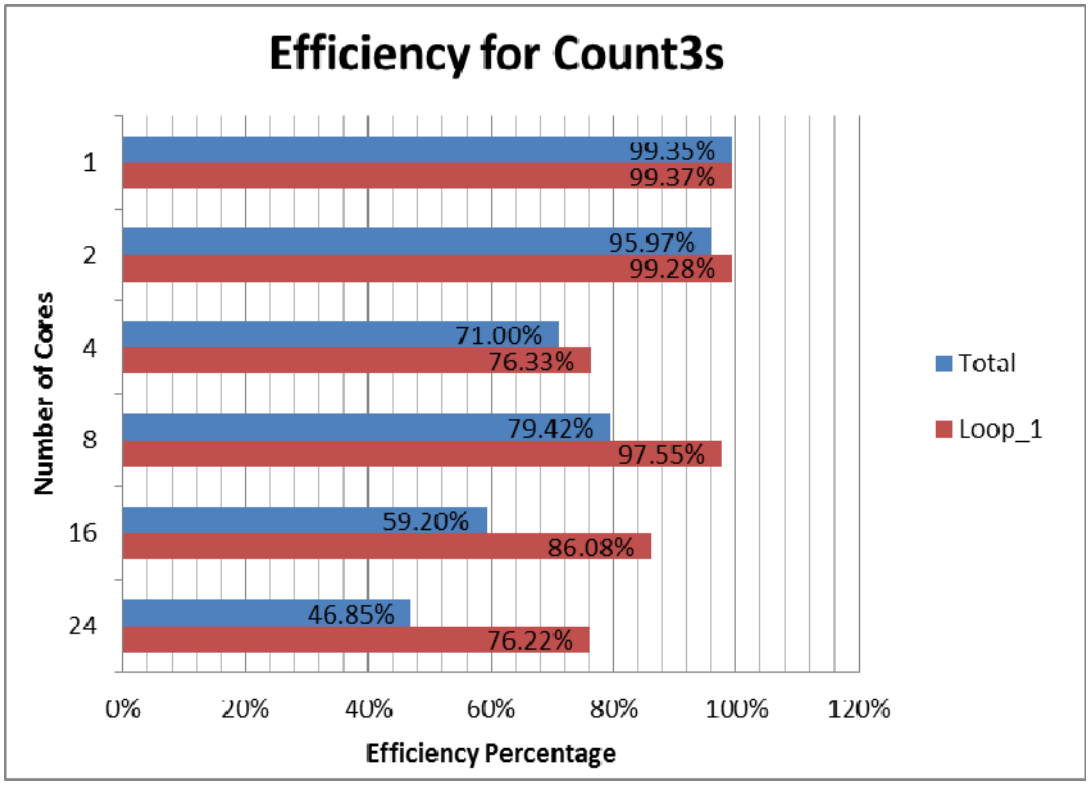
Εικόνα 5.4 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου ComputePi .

Count3s.java								
Cores #	Serial Times		Parallel Times		SpeedUp Value		Efficiency Percentage	
	Total	Loop_1	Total	Loop_1	Total	Loop_1	Total	Loop_1
24	17373	16792	1545	918	11.245	18.292	46.85%	76.22%
16	17401	16803	1837	1220	9.473	13.773	59.20%	86.08%
8	17378	16787	2735	2151	6.354	7.804	79.42%	97.55%
4	17374	16787	6118	5498	2.840	3.053	71.00%	76.33%
2	17378	16793	9054	8457	1.919	1.986	95.97%	99.28%
1	17371	16784	17485	16890	0.993	0.994	99.35%	99.37%

Πίνακας 5.2 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου Count3s.java .



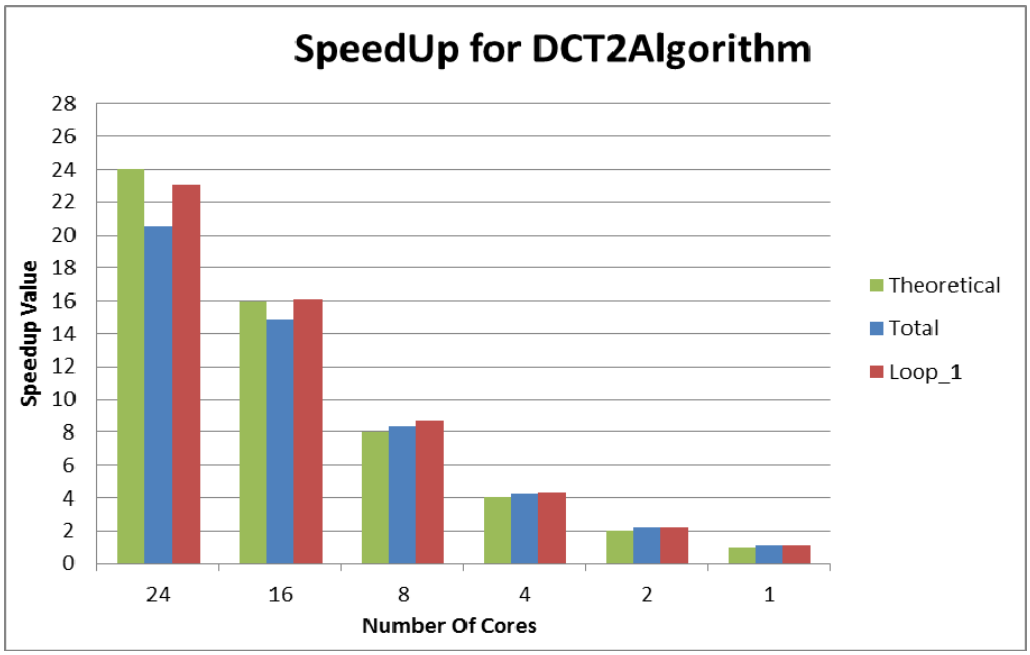
Εικόνα 5.5 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου Count3s .



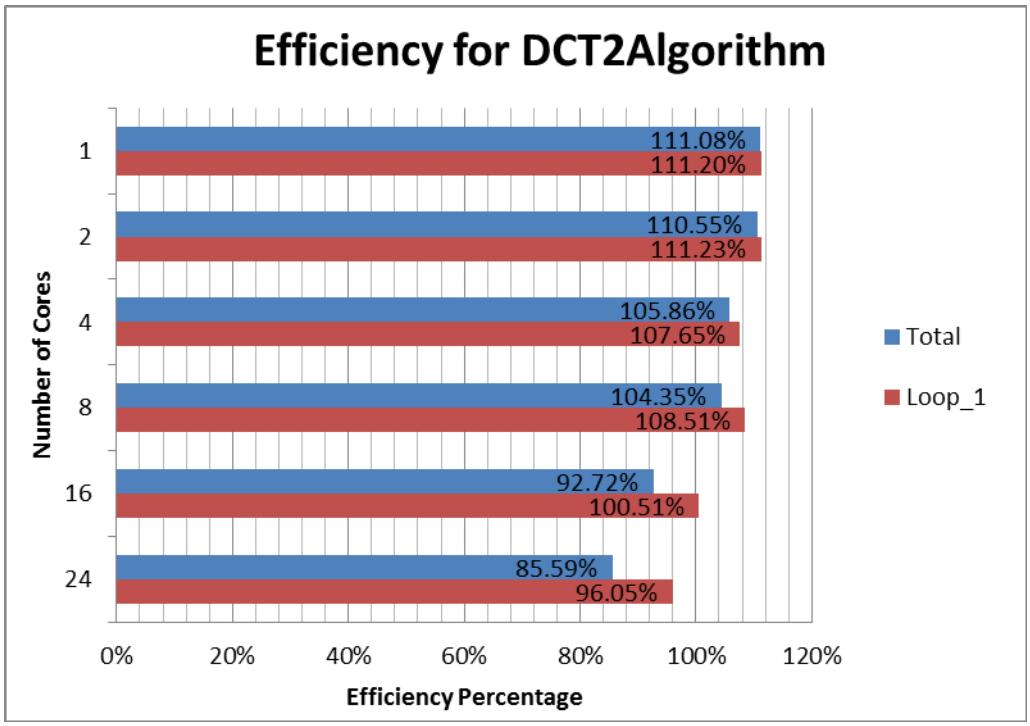
Εικόνα 5.6 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου Count3s .

DCT2Algorithm.java								
Cores #	**Serial Times**		**Parallel Times**		**SpeedUp Value**		**Efficiency Percentage**	
	Total	**Loop_1**	**Total**	**Loop_1**	**Total**	**Loop_1**	**Total**	**Loop_1**
24	44573	44331	2170	1923	20.541	23.053	85.59%	96.05%
16	44609	44368	3007	2759	14.835	16.081	92.72%	100.51%
8	45787	45531	5485	5245	8.348	8.681	104.35%	108.51%
4	45171	44921	10668	10432	4.234	4.306	105.86%	107.65%
2	45796	45554	20712	20477	2.211	2.225	110.55%	111.23%
1	17371	16784	17485	16890	0.993	0.994	99.35%	99.37%

Πίνακας 5.3 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου DCT2Algorithm.java .



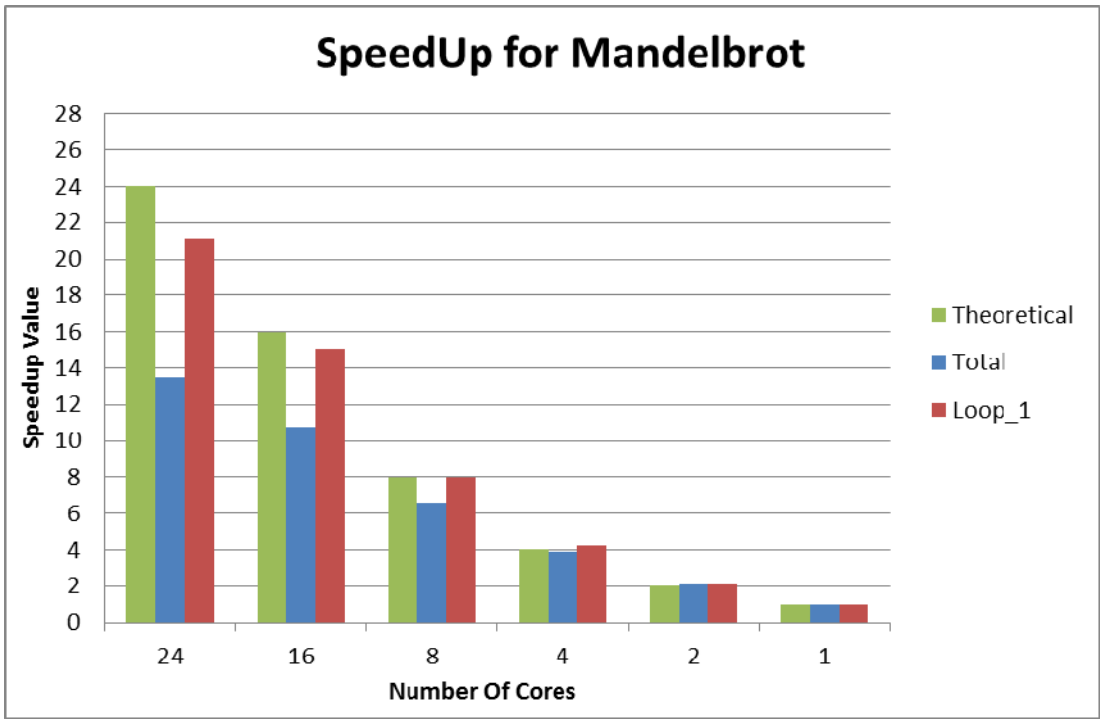
Εικόνα 5.7 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου DCT2Algorithm .



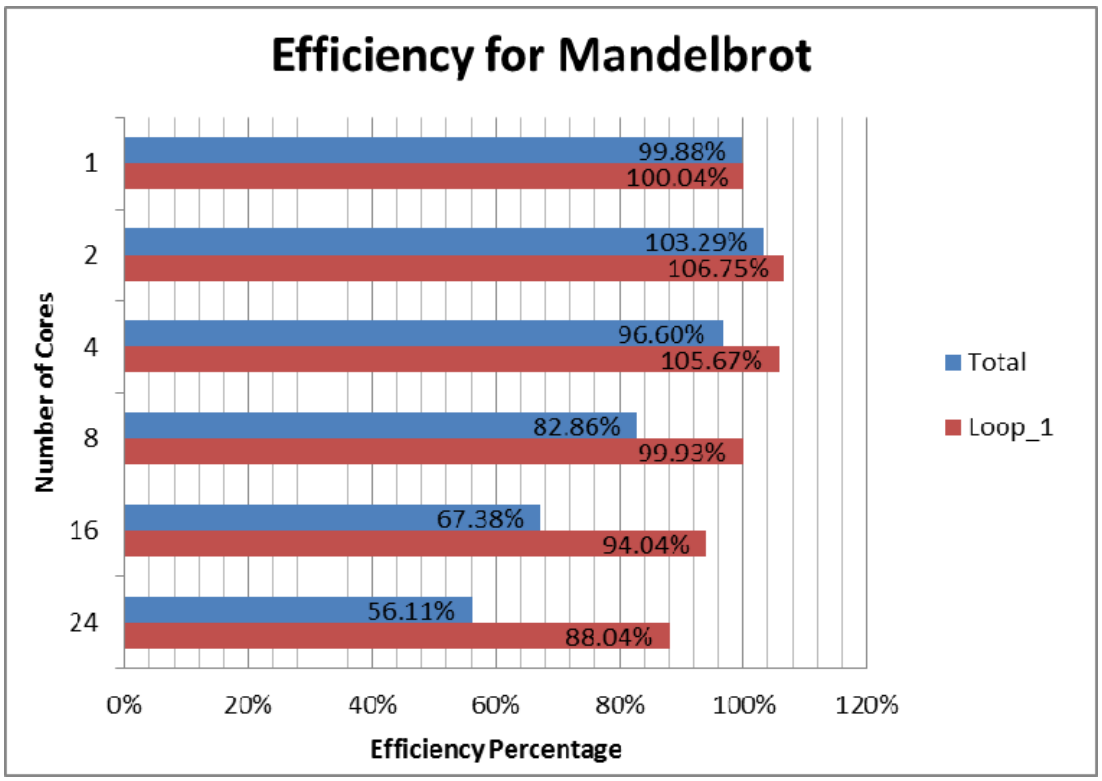
Εικόνα 5.8 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου DCT2Algorithm .

Mandelbrot.java								
Cores #	Serial Times		Parallel Times		SpeedUp Value		Efficiency Percentage	
	Total	Loop_1	Total	Loop_1	Total	Loop_1	Total	Loop_1
24	11528	11220	856	531	13.467	21.130	56.11%	88.04%
16	11568	11240	1073	747	10.781	15.047	67.38%	94.04%
8	11534	11224	1740	1404	6.629	7.994	82.86%	99.93%
4	11534	11218	2985	2654	3.864	4.227	96.60%	105.67%
2	11548	11247	5590	5268	2.066	2.135	103.29%	106.75%
1	11564	11239	11578	11235	0.999	1.000	99.88%	100.04%

Πίνακας 5.4 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου Mandelbrot.java.



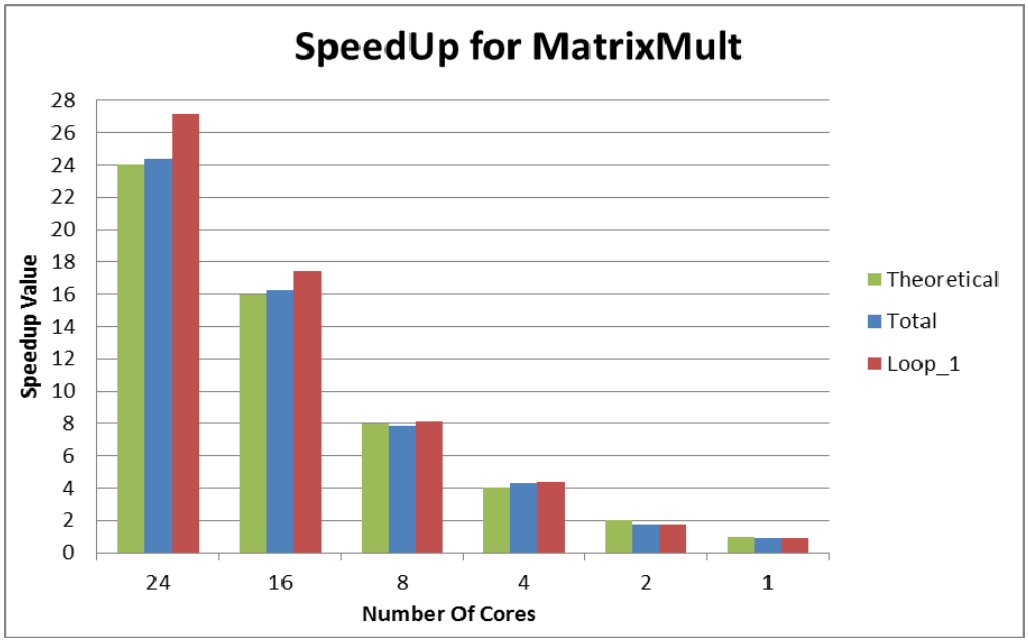
Εικόνα 5.9 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου Mandelbrot .



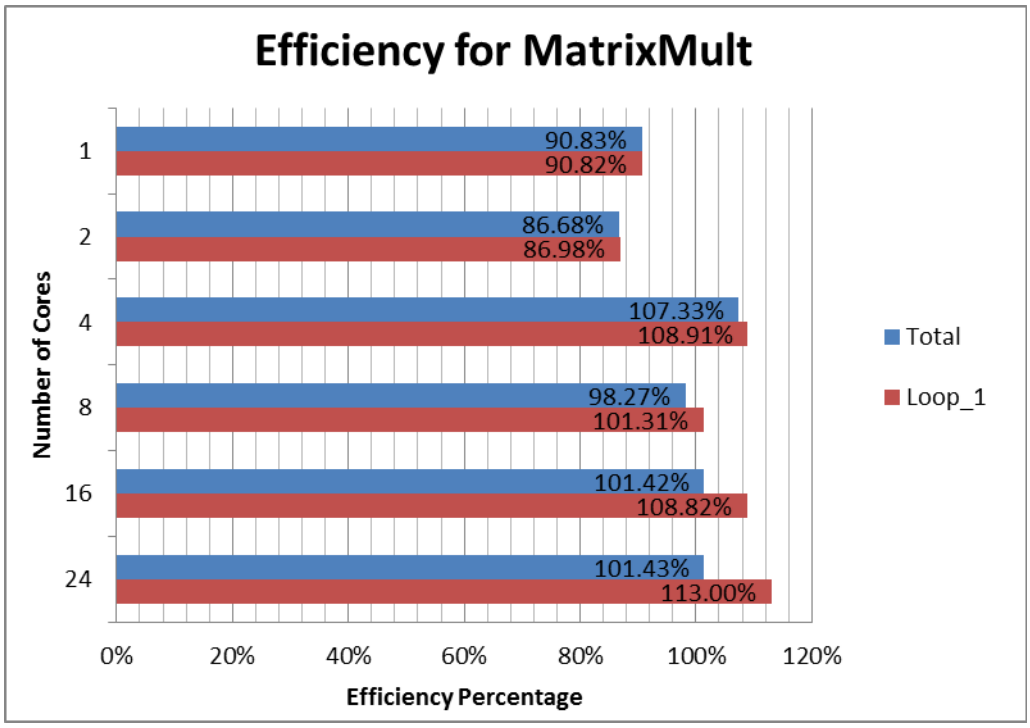
Εικόνα 5.10 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου Mandelbrot .

MatrixMult.java								
Cores #	Serial Times		Parallel Times		SpeedUp Value		Efficiency Percentage	
	Total	Loop_1	Total	Loop_1	Total	Loop_1	Total	Loop_1
24	86366	86001	3548	3171	24.342	27.121	101.43%	113.00%
16	86327	85961	5320	4937	16.227	17.412	101.42%	108.82%
8	86313	85946	10979	10604	7.862	8.105	98.27%	101.31%
4	86341	85966	20112	19734	4.293	4.356	107.33%	108.91%
2	86311	85941	49785	49402	1.734	1.740	86.68%	86.98%
1	86374	86012	95093	94703	0.908	0.908	90.83%	90.82%

Πίνακας 5.5 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου MatrixMult.java.



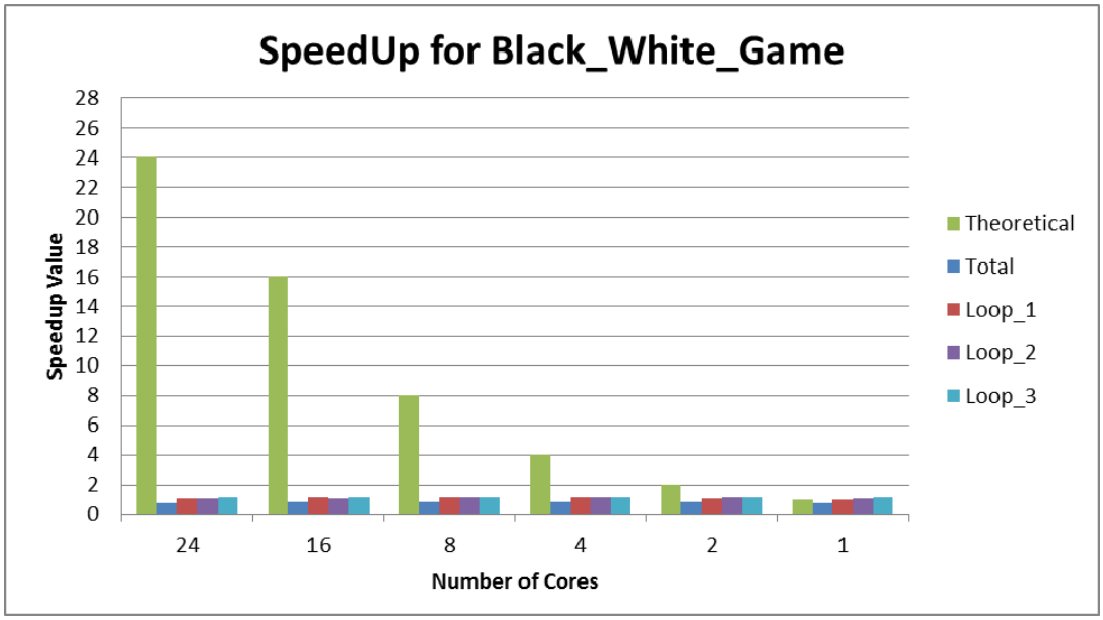
Εικόνα 5.11 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου MatrixMult.java .



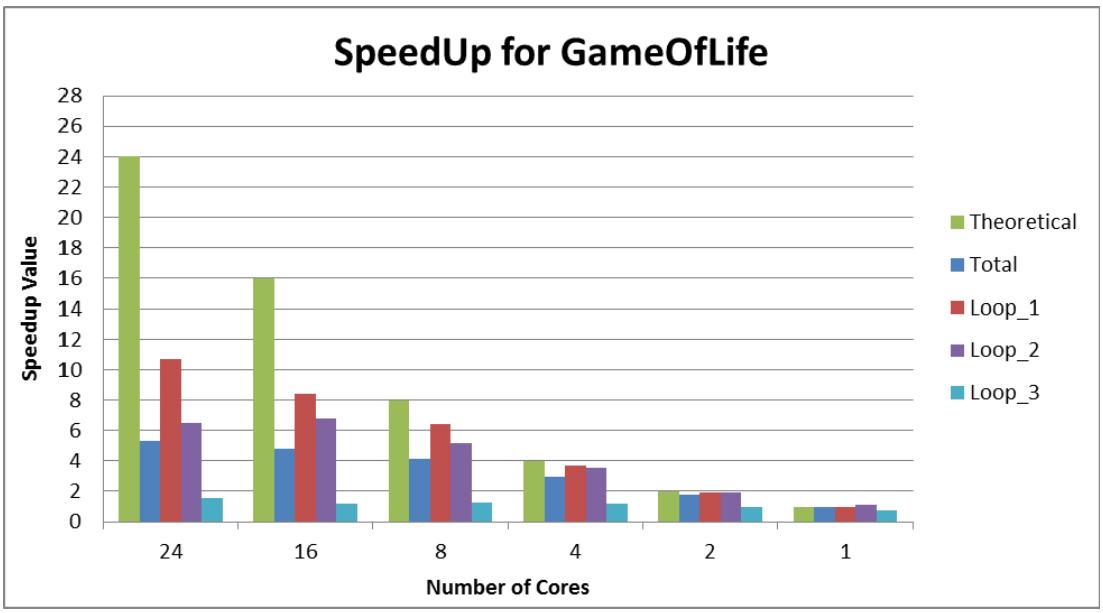
Εικόνα 5.12 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου MatrixMult.java .

Black_White_Game.java															
Cores #	**Serial Times**				**Parallel Times**				**SpeedUp Value**				**Efficiency Percentage**			
	Total	**Loops**			**Total**	**Loops**			**Total**	**Loops**			**Total**	**Loops**		
		1	**2**	**3**		**1**	**2**	**3**		**1**	**2**	**3**		**1**	**2**	**3**
24	19895	17	17	18	24145	16	16	15	0.824	1.063	1.063	1.200	3.43%	4.43%	4.43%	5.00%
16	19893	17	17	18	24078	15	16	15	0.826	1.133	1.063	1.200	5.16%	7.08%	6.64%	7.50%
8	19891	17	17	18	24023	15	15	15	0.828	1.133	1.133	1.200	10.35%	14.17%	14.17%	15.00%
4	19885	17	17	18	23928	15	15	15	0.831	1.133	1.133	1.200	20.78%	28.33%	28.33%	30.00%
2	19899	17	17	18	23996	16	15	15	0.829	1.063	1.133	1.200	41.46%	53.13%	56.67%	60.00%
1	19890	17	17	18	24218	17	16	15	0.821	1.000	1.063	1.200	82.13%	100.00%	106.25%	120.00%

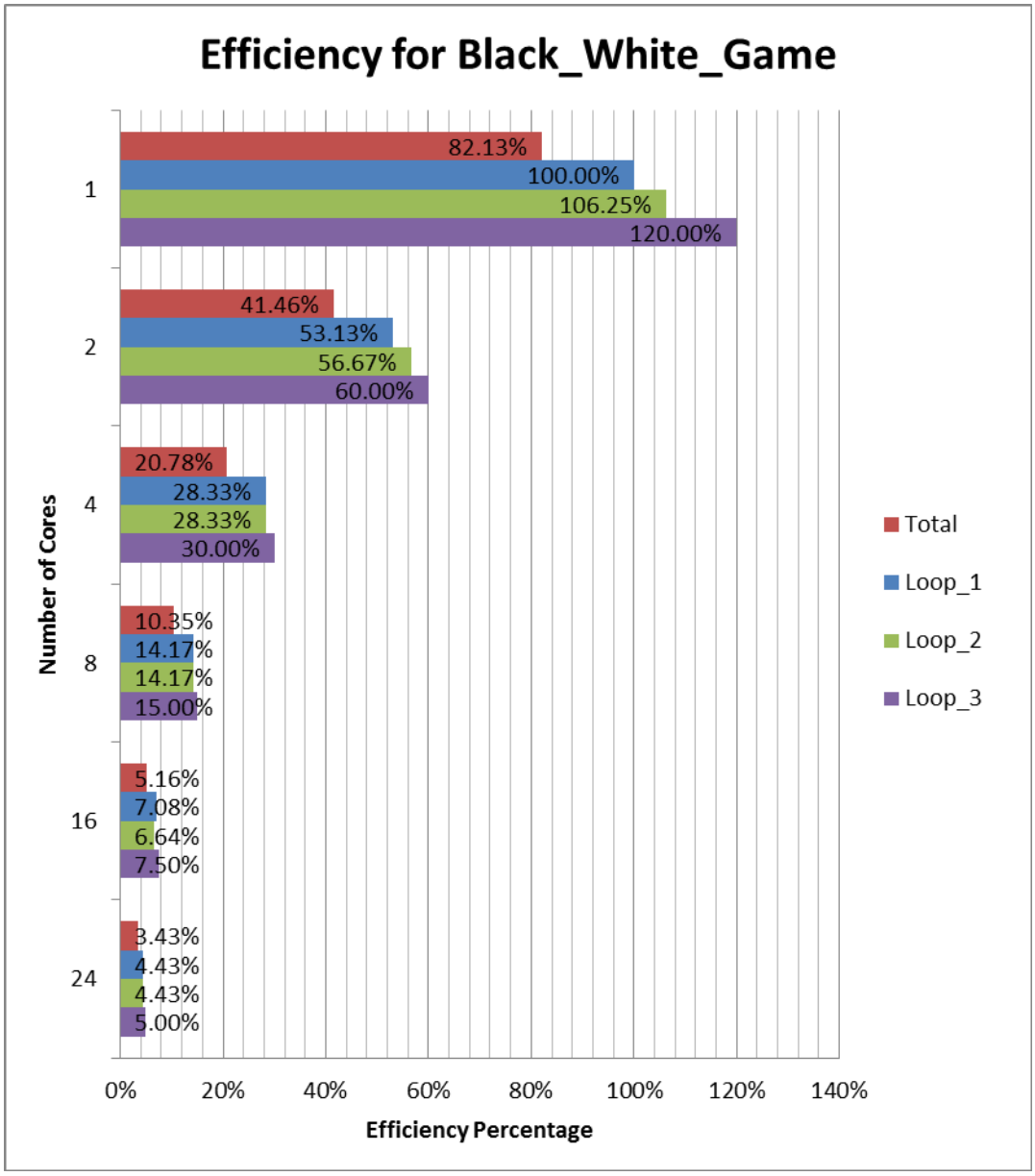
Πίνακας 5.6 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου Black_White_Game.java .



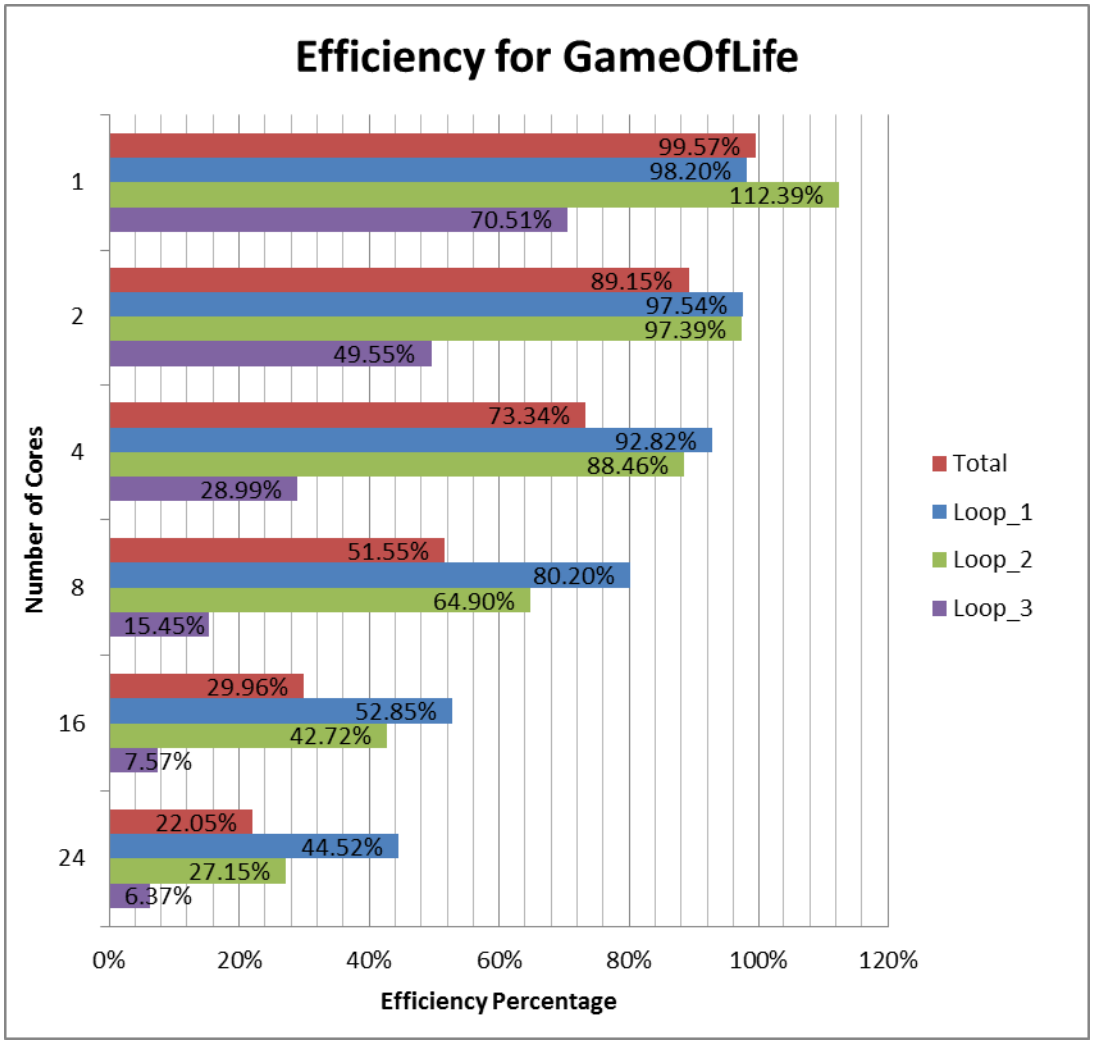
Εικόνα 5.13 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου Black_White_Game.java .



Εικόνα 5.15 : Γραφική παράσταση που παρουσιάζει την βελτίωση στην επίδοση (Speedup) για το πρόγραμμα ελέγχου GameOfLife.java .



Εικόνα 5.14 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου Black_White_Game.java .



Εικόνα 5.16 : Γραφική παράσταση που παρουσιάζει τα ποσοστά αποδοτικότητας (efficiency) που προκύπτουν για την παράλληλη έκδοση του προγράμματος ελέγχου GameOfLife.java .

GameOfLife.java																
Cores #	**Serial Times**				**Parallel Times**				**SpeedUp Value**				**Efficiency Percentage**			
	Total	**Loops**			**Total**	**Loops**			**Total**	**Loops**			**Total**	**Loops**		
		1	**2**	**3**		**1**	**2**	**3**		**1**	**2**	**3**		**1**	**2**	**3**
24	42765	3056	782	110	8082	286	120	72	5.291	10.685	6.517	1.528	22.05%	44.52%	27.15%	6.37%
16	42855	3061	786	109	8940	362	115	90	4.794	8.456	6.835	1.211	29.96%	52.85%	42.72%	7.57%
8	42784	3054	784	110	10375	476	151	89	4.124	6.416	5.192	1.236	51.55%	80.20%	64.90%	15.45%
4	42754	3052	782	109	14574	822	221	94	2.934	3.713	3.538	1.160	73.34%	92.82%	88.46%	28.99%
2	42840	3059	785	110	24028	1568	403	111	1.783	1.951	1.948	0.991	89.15%	97.54%	97.39%	49.55%
1	42718	3051	780	110	42901	3107	694	156	0.996	0.982	1.124	0.705	99.57%	98.20%	112.39%	70.51%

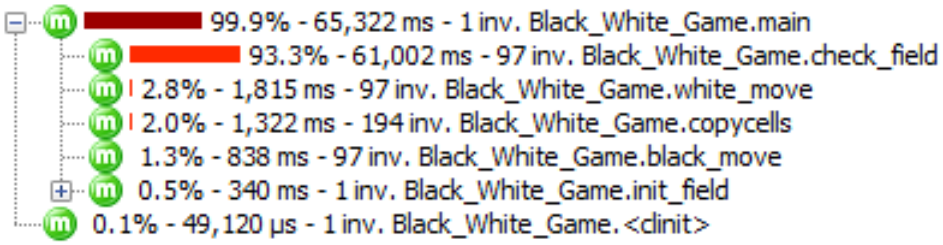
Πίνακας 5.7 : Πίνακας συνολικών αποτελεσμάτων των πειραματικών μετρήσεων για το πρόγραμμα ελέγχου GameOfLife.java .

Βλέποντας σε συνολικό επίπεδο τα συγκριτικά αποτελέσματα των πειραμάτων μας και για τα 7 προγράμματα ελέγχου παρατηρούμε ότι δύο (MatrixMult [Παράρτημα Β-31] και DCT2Algorithm [Παράρτημα Β-20]) από τα εφτά προγράμματα ελέγχου παρουσιάζουν απόλυτη βελτίωση στην επίδοση, της τάξεως της θεωρητικής μέγιστης βελτίωσης που θα μπορούσαμε να έχουμε, με το πρόγραμμα ελέγχου MatrixMult να παρουσιάζει τα καλύτερα αποτελέσματα βελτίωσης της επίδοσης. Τρία (ComputePi [Παράρτημα Β-16], Count3s [Παράρτημα Β-18] και Mandelbrot [Παράρτημα Β-28]) από τα εφτά προγράμματα ελέγχου παρουσιάζουν πολύ καλή βελτίωση στην επίδοση πλησιάζοντας αρκετά την θεωρητική μέγιστη βελτίωση που θα μπορούσαν να πετύχουν. Τέλος, σε δύο (Black_White_Game [Παράρτημα Β-10] και GameOfLife [Παράρτημα Β-22]) από τα εφτά προγράμματα ελέγχου οι βελτιώσεις στην επίδοση φαίνεται να είναι σε αρκετά χαμηλά επίπεδα με το πρόγραμμα ελέγχου Black_White_Game να παρουσιάζει τα χειρότερα αποτελέσματα βελτίωσης της απόδοσης.

Με βάση τις πιο πάνω πολύ γενικές παρατηρήσεις επιλέξαμε να επικεντρωθούμε σε λεπτομερή επεξήγηση των αποτελεσμάτων για δύο από τα συνολικά εφτά προγράμματα ελέγχου που χρησιμοποιήθηκαν για τις πειραματικές μας μετρήσεις. Τα δύο αυτά προγράμματα ελέγχου με τα οποία θα ασχοληθούμε στην συνέχεια είναι αυτά με την καλύτερη και χειρότερη βελτίωση στην επίδοση αντίστοιχα (MatrixMult και Black_White_Game). Αναλύοντας τόσο την δομή τους όσο και τον τρόπο με τον οποίο οι παράλληλες εκδόσεις τους εκτελέστηκαν ουσιαστικά στο παράλληλο σύστημα που χρησιμοποιήσαμε θα είμαστε σε θέση να καταλάβουμε ακριβώς τι σημαίνουν για μας τα αποτελέσματα τα οποία προκύπτουν. Θα εξηγήσουμε δηλαδή ότι τα αποτελέσματα της βελτίωσης στην απόδοση που προκύπτουν μέσα από τους χρόνους εκτέλεσής τους ήταν ουσιαστικά τα αναμενόμενα και στις δύο περιπτώσεις. Με βάση την ανάλυση και τις επεξηγήσεις που θα δώσουμε θα είμαστε έτσι σε θέση να καταλάβουμε και να κατανοήσουμε πώς ακριβώς προέκυψαν και τα υπόλοιπα αποτελέσματα για τα άλλα 5 προγράμματα ελέγχου που χρησιμοποιήθηκαν για τους πειραματικούς ελέγχους.

Θα ξεκινήσουμε αναλύοντας τα αποτελέσματα για το πρόγραμμα ελέγχου Black_White_Game για το οποίο όπως μπορούμε να παρατηρήσουμε από τα δεδομένα που παραθέσαμε πιο πριν [Πίνακας 5.6, Εικόνα 5.13 και Εικόνα 5.14] τα αποτελέσματα που προκύπτουν για την βελτίωση της επίδοσης είναι απογοητευτικά σε κάθε εκτέλεση της εφαρμογής αυτής στο παράλληλο υπολογιστικό σύστημα που χρησιμοποιήσαμε. Τα όσα θα αναλύσουμε καθώς και τα συμπεράσματα στα οποία θα καταλήξουμε θα είναι κοινά και για το πρόγραμμα ελέγχου GameOfLife, αφού η δομή του προγράμματος και του τρόπου εκτέλεσής τους καθώς και τα αποτελέσματα που προκύπτουν από τις πειραματικές μετρήσεις είναι παρόμοια.

Όπως αναφέραμε και στην αρχή αυτού του κεφαλαίου (5.1 Εισαγωγή) το πρόγραμμα ελέγχου Black_White_Game αποτελεί την υλοποίηση ενός παιχνιδιού με μία σκακιέρα από μαύρα και άσπρα πιόνια το οποίο ακολουθεί κάποιους βασικούς κανόνες για την κίνηση των πιονιών και σε κάθε γύρο ελέγχεται αν υπάρχει στην σκακιέρα κενός χώρος μεγέθους 9x9. Το πρόγραμμα ουσιαστικά αποτελείται από 8 μεθόδους συνολικά, μια εκ των οποίων είναι η print_board(), για την εκτύπωση της σκακιέρας, η οποία μέθοδος δεν καλείται ποτέ. Τρέχοντας τον JProfiler 6.1.1 [37] για το πρόγραμμά μας μπορούμε να δούμε ποιες μέθοδοι απαιτούν τον περισσότερο χρόνο εκτέλεσης [Εικόνα 5.17].



Εικόνα 5.17 : Τα αποτελέσματα που προκύπτουν τρέχοντας τον JProfiler για την σειριακή έκδοση του προγράμματος ελέγχου Black_White_Game.

Από τα αποτελέσματα που προκύπτουν μπορούμε να διακρίνουμε ότι η μέθοδος check_field() είναι αυτή που καταναλώνει το 93.3% του συνολικού ποσοστού χρόνου εκτέλεσης για ολόκληρο το πρόγραμμα. Οι υπόλοιπες μέθοδοι

καταναλώνουν ποσοστό χρόνου εκτέλεσης μικρότερο του 7.0% και οι πέντε μαζί συνολικά. Οι μέθοδοι που εμείς ήταν δυνατόν και θελήσαμε να παραλληλίσουμε με το εργαλείο αυτόματου παραλληλισμού ήταν οι white_move(), black_move() και copycells() όπου και οι τρείς όμως μαζί έχουν συνολικό ποσοστό χρόνου εκτέλεσης ίσο με 6.1%. Θέλαμε να παραλληλίσουμε και την μέθοδο check_field() αλλά αυτό δεν ήταν δυνατόν γιατί υπάρχουν εξαρτήσεις που αφορούν τους ίδιους τους δείκτες για τους βρόγχους for() και επίσης η δομή ελέγχου των βρόγχων for() είναι αρκετά περίπλοκη και δεν υποστηρίζεται η ανάλυσή της από τον αυτόματο μεταφραστή JavaParallelTranslator [Εικόνα 5.18].

```
//Checking field for an empty subtable
public static void check_field() {

  int[] total = new int[3];
  int space = FIELD_SIZE - WIN_FIELD;

  for (int i = 0; i <= space; i++) {
    for (int j = 0; j <= space; j++) {
      for (int m = 0; m < 3; m++) {
        total[m] = 0;
      }

      for (int k = i; k < i + WIN_FIELD; k++) {
        for (int l = j; l < j + WIN_FIELD; l++) {
          total[(int)(field[1][k][l])]++;
        }
      }

      if (total[EMPTY] >= WIN_FIELD * WIN_FIELD) {
        System.out.printf("Empty table at: i = %d, j = %d\n", i, j);
        moving = 0;
        i = FIELD_SIZE - WIN_FIELD;
        break;
      }
    }
  }
}
```

Εικόνα 5.18 : Ο κώδικας της μεθόδου check_field() όπου με κόκκινη σημείωση φαίνονται οι εξαρτήσεις που αφορούν τους ίδιους τους δείκτες για τους βρόγχους for() ενώ με πράσινη σημείωση φαίνεται η δομή ελέγχου των βρόγχων for() της οποίας δεν υποστηρίζεται η ανάλυσή της από τον αυτόματο μεταφραστή JavaParallelTranslator.

Η παράλληλη έκδοση του προγράμματος ελέγχου Black_White_Game [Παράρτημα Β-13] που δημιουργήθηκε από τον αυτόματο μεταφραστή JavaParallelTranslator ήταν κατά 6.1% παράλληλο και κατά συνέπεια, 93.3% σειριακό. Ανακαλώντας ξανά τα συμπεράσματα που προκύπτουν μέσα από τον νόμο του Amdahl, που αναφέραμε στο Κεφάλαιο 2, η μέγιστη αναμενόμενη βελτίωση μιας εφαρμογής περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει [Εικόνα 2.3]. Υπολογίζοντας τον τύπο που ορίζει την βελτίωση της επίδοσης (speedup) [Εικόνα 2.2] με τα δεδομένα που έχουμε για το πρόγραμμα ελέγχου Black_White_Game, δηλαδή με S (ποσοστό του προγράμματος που είναι σειριακό) ίσο με 93.9% (0.939) και 1-S (ποσοστό του προγράμματος που είναι παράλληλο) ίσο με 6.1% (0.061) καταλήγουμε στο αποτέλεσμα ότι μέγιστη αναμενόμενη βελτίωση (speedup) δεν μπορεί να ξεπερνά το 0.8, πράγμα που επαληθεύεται και μέσα από τις πειραματικές μας μετρήσεις για το πρόγραμμα ελέγχου Black_White_Game [Πίνακας 5.6]. Άρα στην ουσία μιλούμε για αρνητική βελτίωση (speed-down) αφού η τιμή του speedup είναι μικρότερη από 1.

Τα πιο πάνω ισχύουν αποκλειστικά και μόνο για τα αποτελέσματα του συνολικού χρόνου εκτέλεσης (total execution time), όσον αφορά τους χρόνους εκτέλεσης για τους επιμέρους βρόγχους for() που έτυχαν επεξεργασίας από τον παράλληλο μεταφραστή (Loop_# Times) τα πράγματα είναι διαφορετικά.

Όπως προαναφέραμε, ήταν δυνατόν να παραλληλίσουμε τρείς από τις οκτώ μεθόδους που είχαμε συνολικά όπου κάθε μία από αυτές αποτελείτο και από ένα βρόγχο επανάληψης for() [Εικόνα 5.19]. Οι παράλληλες μέθοδοι που δημιουργούνται από τον αυτόματο μεταφραστή JavaParallelTranslator φαίνονται στην Εικόνα 5.20. Στην περιγραφή λειτουργίας των κύριων μεθόδων της βιβλιοθήκης JACON, που αναφέραμε στην αρχή του κεφαλαίου αυτού (4.2.2 Ανάλυση λειτουργίας κύριων μεθόδων), η παράλληλη μέθοδος *Prarallel.forLoop* δημιουργεί ένα στόχο (task) για κάθε αύξηση του δείκτη *i = start*, δηλαδή δημιουργούνται [*(end – start)*] στόχοι, καθώς και ένα threadpool, με αριθμό νημάτων ίσο με τους διαθέσιμους πυρήνες που υπάρχουν στο υπολογιστικό σύστημα, για να διεκπεραιώσουν όλους τους στόχους. Κάθε πυρήνας στο παράλληλο

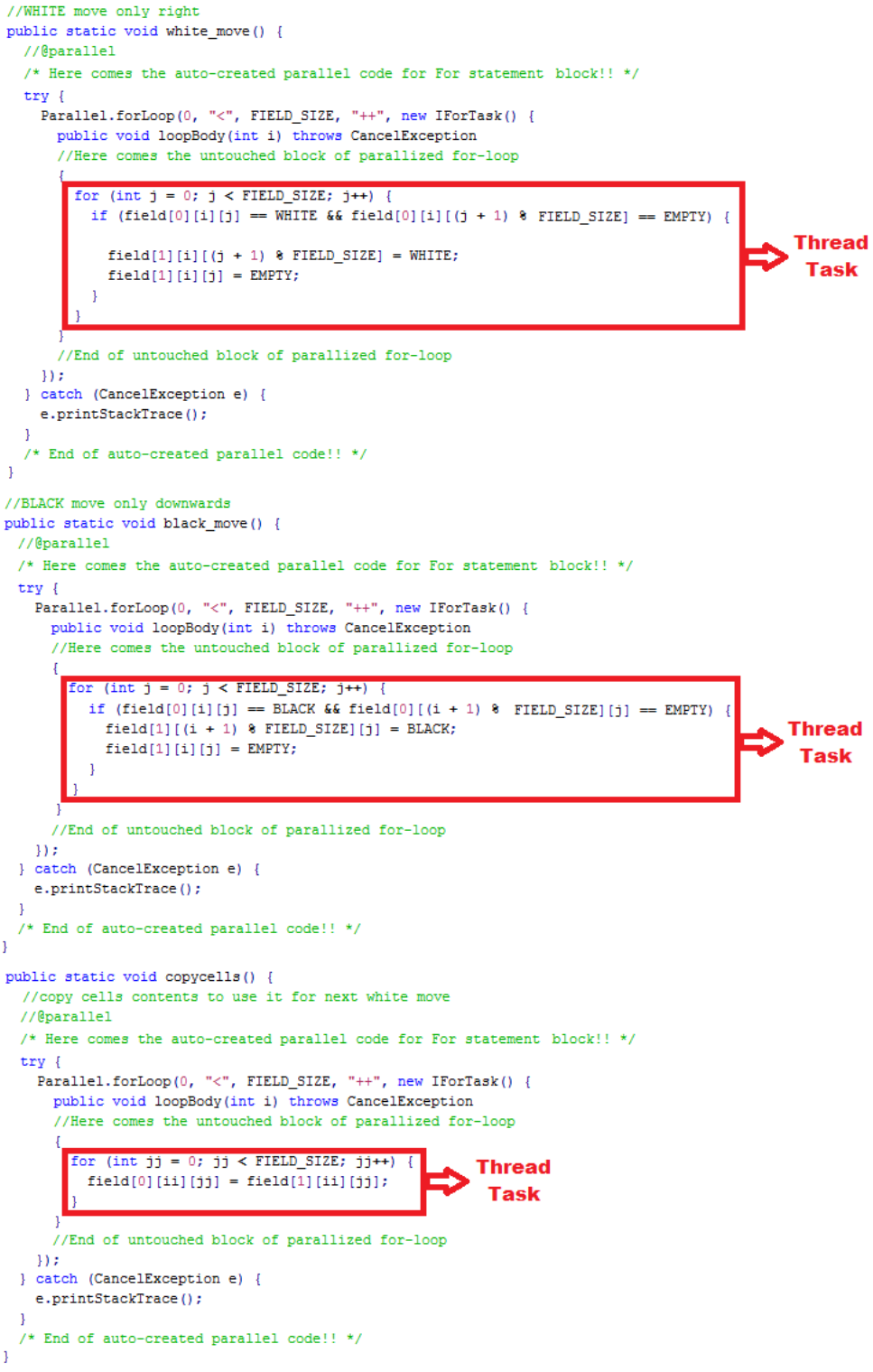
σύστημα που χρησιμοποιήσαμε για τις πειραματικές μας μελέτες, εκτελεί πολλαπλούς στόχους κάθε φορά κατά τη διάρκεια εκτέλεσης της παράλληλης μεθόδου *Prarallel.forLoop* που υπάρχει σε κάθε μια από τις μεθόδους που παραλληλίστηκαν.

```
//WHITE move only right
public static void white_move() {
  /@parallel
  for (int i = 0; i < FIELD_SIZE; i++) {
    for (int j = 0; j < FIELD_SIZE; j++) {
      if (field[0][i][j] == WHITE && field[0][i][(j + 1) % FIELD_SIZE] == EMPTY) {
        field[1][i][(j + 1) % FIELD_SIZE] = WHITE;
        field[1][i][j] = EMPTY;
      }
    }
  }
}

//BLACK move only downwards
public static void black_move() {
  /@parallel
  for (int i = 0; i < FIELD_SIZE; i++) {
    for (int j = 0; j < FIELD_SIZE; j++) {
      if (field[0][i][j] == BLACK && field[0][(i + 1) % FIELD_SIZE][j] == EMPTY) {
        field[1][(i + 1) % FIELD_SIZE][j] = BLACK;
        field[1][i][j] = EMPTY;
      }
    }
  }
}

public static void copycells() {
  //copy cells contents to use it for next white move
  /@parallel
  for (int ii = 0; ii < FIELD_SIZE; ii++) {
    for (int jj = 0; jj < FIELD_SIZE; jj++) {
      field[0][ii][jj] = field[1][ii][jj];
    }
  }
}
```

Εικόνα 5.19 : Οι τρείς μέθοδοι που μπορούσαμε να παραλληλίσουμε για το πρόγραμμα ελέγχου Black_White_Game, όπου κάθε μια από αυτές αποτελείται και από ένα βρόγχο επανάληψης for().



Εικόνα 5.20 : Οι τρείς παράλληλες μέθοδοι του προγράμματος ελέγχου Black_White_Game που δημιουργήθηκαν από τον αυτόματο μεταφραστή JavaParallelTranslator. Σε κόκκινο πλαίσιο φαίνονται οι επιμέρους στόχοι που ουσιαστικά εκτελούνται από τα νήματα.

Στην προηγούμενη εικόνα [Εικόνα 5.20], βλέπουμε σε κόκκινο πλαίσιο τους στόχους (tasks) που θα εκτελεστούν από τα νήματα που θα δημιουργηθούν κατά την παράλληλη εκτέλεση. Μπορούμε να παρατηρήσουμε ότι το μέγεθος εργασίας που έχει να κάνει κάθε νήμα είναι πολύ μικρή (προσπέλαση ενός μονοδιάστατου πίνακα μεγέθους 1024 θέσεων και απλά ανάθεση μιας τιμής). Λόγω του ότι ο χρόνος εκτέλεσης των στόχων είναι ελάχιστος, τα νήματα γίνονται πολύ ανταγωνιστικά κατά τη διάρκεια εκτέλεσής τους και έχουμε συμφόρηση κατά το συγχρονισμό της σειράς εκτέλεσής τους από τον χρονοπρογραμματιστή (TaskScheduler). Η συμφόρηση αυτή δημιουργεί επιπλέον αύξηση στο ποσοστό χρόνου εκτέλεσης που απαιτούν οι τρείς μέθοδοι που παραλληλίστηκαν και εκτελούνται μέσω της παράλληλης βιβλιοθήκης JACON σε σχέση με το ποσοστό χρόνου που είχαν στην σειριακή έκδοση. Το συνολικό ποσοστό χρόνου εκτέλεσης για την σειριακή έκδοση και των τριών μεθόδων ήταν της τάξης του 6% [Εικόνα 5.17] ενώ στην παράλληλη έκδοση το ποσοστό αυτό αυξάνεται σε 37% [Εικόνα 5.21]. Λόγω των επιπλέον αυτών καθυστερήσεων όσο περισσότεροι πυρήνες συμμετέχουν στην παράλληλη εκτέλεση τόσο λιγότερη είναι η βελτίωση στην επίδοση όσον αφορά τους επιμέρους βρόγχους που παραλληλίσαμε [Εικόνα 5.13 και Εικόνα 5.14].

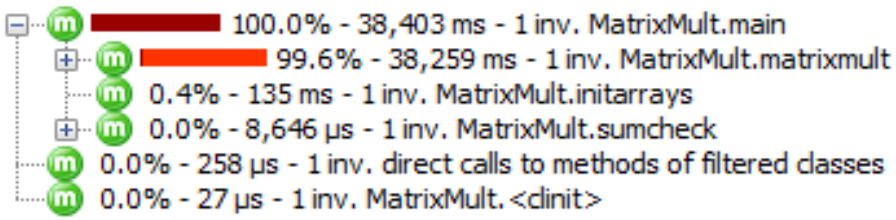
Εικόνα 5.21 : Τα αποτελέσματα που προκύπτουν τρέχοντας τον JProfiler για την παράλληλη έκδοση του προγράμματος ελέγχου Black_White_Game.

Με βάση την ανάλυση και τις υποδείξεις που προηγήθηκαν μπορούμε να πούμε ότι τα αρνητικά αποτελέσματα που προέκυψαν για τα προγράμματα ελέγχου Black_White_Game και GameOfLife (που όπως αναφέραμε έχουν παρόμοια δομή

προγράμματος και τρόπο εκτέλεσης) είναι απολύτως δικαιολογημένα και αναμενόμενα.

Το δεύτερο πρόγραμμα ελέγχου με το οποίο θα ασχοληθούμε και θα αναλύσουμε τα αποτελέσματά του είναι το MatrixMult. Στα δεδομένα που παραθέσαμε πιο πριν [Πίνακας 5.5, Εικόνα 5.11 και Εικόνα 5.12] τα αποτελέσματα που προκύπτουν για τη βελτίωση της επίδοσης είναι άκρως ικανοποιητικά σε κάθε εκτέλεση της εφαρμογής αυτής στο παράλληλο υπολογιστικό σύστημα που χρησιμοποιήσαμε. Τα όσα θα αναλύσουμε καθώς και τα συμπεράσματα στα οποία θα καταλήξουμε θα είναι κοινά και για το πρόγραμμα ελέγχου DCT2Algorithm, αφού η δομή του προγράμματος και του τρόπου εκτέλεσής του καθώς και τα αποτελέσματα που προκύπτουν από τις πειραματικές μετρήσεις είναι παρόμοια.

Όπως αναφέραμε και στην αρχή αυτού του κεφαλαίου (5.1 Εισαγωγή) το πρόγραμμα ελέγχου MatrixMult αποτελεί την υλοποίηση μιας εφαρμογής πολλαπλασιασμού δύο δισδιάστατων πινάκων. Το αποτέλεσμα που προκύπτει από τον πολλαπλασιασμό των δύο πινάκων αποθηκεύεται σε ένα τρίτο πίνακα. Ο πολλαπλασιασμός γίνεται με βάση τον τύπο και την φόρμουλα στην Εικόνα 5.3 . Το πρόγραμμα ουσιαστικά αποτελείται από 4 μεθόδους συνολικά, μια εκ των οποίων είναι η main(). Οι υπόλοιπες μέθοδοι είναι οι matrixmult(), για τον πολλαπλασιασμό των πινάκων, initarrays(), για την αρχικοποίηση των πινάκων, και sumcheck(), για υποτυπώδη έλεγχο του αποτελέσματος. Τρέχοντας τον JProfiler 6.1.1 [37] για το πρόγραμμά μας μπορούμε να δούμε ποιες μέθοδοι απαιτούν τον περισσότερο χρόνο εκτέλεσης [Εικόνα 5.22].



Εικόνα 5.22 : Τα αποτελέσματα που προκύπτουν τρέχοντας τον JProfiler για την παράλληλη έκδοση του προγράμματος ελέγχου MatrixMult.

Από τα αποτελέσματα που προκύπτουν μπορούμε να δούμε ότι η μέθοδος matrixmult() είναι αυτή που καταναλώνει το 99.6% του συνολικού ποσοστού χρόνου εκτέλεσης για ολόκληρο το πρόγραμμα. Οι υπόλοιπες μέθοδοι καταναλώνουν ποσοστό χρόνου εκτέλεσης μικρότερο του 0.5% και οι δύο μαζί συνολικά. Η μέθοδος που θα παραλληλίσουμε είναι η matrixmult() η οποία υλοποιεί τον πολλαπλασιασμό πινάκων και κατέχει και το μεγαλύτερο ποσοστό χρόνου εκτέλεσης.

Η παράλληλη έκδοση του προγράμματος ελέγχου MatrixMult [Παράρτημα Β-32] που δημιουργήθηκε από τον αυτόματο μεταφραστή JavaParallelTranslator ήταν κατά 99.6% παράλληλο και κατά συνέπεια, 0.4% σειριακό. Ανακαλώντας ξανά τα συμπεράσματα που προκύπτουν μέσα από τον νόμο του Amdahl [5, 7, 10, 19], που αναφέραμε στο Κεφάλαιο 2, η μέγιστη αναμενόμενη βελτίωση μιας εφαρμογής περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει [Εικόνα 2.3]. Υπολογίζοντας τον τύπο που ορίζει την βελτίωση της επίδοσης (speedup) [Εικόνα 2.2] με τα δεδομένα που έχουμε για το πρόγραμμα ελέγχου MatrixMult, δηλαδή με S (ποσοστό του προγράμματος που είναι σειριακό) ίσο με 0.4% (0.004) και 1-S (ποσοστό του προγράμματος που είναι παράλληλο) ίσο με 99.6% (0.996) καταλήγουμε στο αποτέλεσμα ότι μέγιστη αναμενόμενη βελτίωση (speedup) είναι 22 φορές.

Βλέποντας όμως τον πίνακα αποτελεσμάτων για το πρόγραμμα ελέγχου MatrixMult παρατηρούμε ότι η τιμή για την αναμενόμενη βελτίωση στην επίδοση (speedup) με συμμετοχή 24 πυρήνων ξεπερνά τη θεωρητική μέγιστη τιμή που είναι 22 φορές. Για συνολικούς χρόνους εκτέλεσης (total) η τιμή βελτίωσης της επίδοσης είναι 24.3 φορές ενώ για τον χρόνο εκτέλεσης του βρόγχου επανάληψης που παραλληλίσαμε η βελτίωση στην επίδοση φτάνει τις 27.1 φορές. Ανάλογες τιμές έχουμε και στα αποτελέσματα για τις εκτελέσεις με συμμετοχή 16 και $4^{ων}$ πυρήνων. Αυτό το φαινόμενο ονομάζεται έξοχη γραμμική επιτάχυνση/βελτίωση στην επίδοση (super linear speedup). Κάποιες φορές δηλαδή, η βελτίωση στην επίδοση που μπορεί να επιτευχθεί κατά τη διάρκεια εκτέλεσης μιας πολυνηματικής εφαρμογής πάνω σε ένα πολυπύρηνο σύστημα είναι δυνατόν να ξεπερνά τη θεωρητική μέγιστη αναμενόμενη

τιμή βελτίωσης της επίδοσης [5, 10, 13, 19]. Η κυριότερη αιτία για την οποία μπορεί να παρουσιαστεί αυτό το φαινόμενο είναι λόγω επίδρασης της κρυφής μνήμης (cache effect) από τις διαφορετικές ιεραρχίες μνήμης που υπάρχουν στα σύγχρονα υπολογιστικά συστήματα και επεξεργαστές. Σε ένα παράλληλο υπολογισμό δεν αλλάζει μόνο ο αριθμός των πυρήνων που συμμετέχουν στην παράλληλη εκτέλεση αλλά και το μέγεθος της συσσωρευμένης κρυφής μνήμης των επεξεργαστών. Λόγω του ότι σε μία παράλληλη εκτέλεση το πρόβλημα σπάζει σε μικρότερα κομμάτια και κατανέμεται στους διαθέσιμους πυρήνες με τη βοήθεια των νημάτων, αυτά τα κομμάτια μεταφρασμένα σε δεδομένα, μπορούν να χωρέσουν στην κρυφή μνήμη που διαθέτει ο κάθε πυρήνας. Αυτός ο διαχωρισμός των δεδομένων μπορεί να μειώσει δραματικά τον χρόνο εκτέλεσης ενός υπολογισμού αφού στο σύνολό τους τα δεδομένα βρίσκονται "κοντά" στον επεξεργαστή (L1 cache memory) και δεν χρειάζεται να γίνει προσπέλαση των επόμενων επιπέδων μνήμης (L2 cache memory, L3 cache memory, RAM) για αναζήτηση δεδομένων που χρειάζονται στην εκτέλεση. Έτσι λόγω αυτού του φαινομένου η επίδοση στην εκτέλεση μιας πολυνηματικής εφαρμογής σε ένα πολυπύρηνο υπολογιστικό σύστημα βελτιώνεται περεταίρω και έχει σαν αποτέλεσμα να παρουσιάζεται επιπλέον επιτάχυνση (speedup) η οποία στο τελικό της σύνολο να είναι μεγαλύτερη της θεωρητικής μέγιστης.

```
public static void matrixmult() {

  //matrix multiplication => c[][] = a[][] x b[][]
  /@parallel
  for (int i = 0; i < NDIM; i++) {
    for (int j = 0; j < NDIM; j++) {
      long sum = 0;
      for (int k = 0; k < NDIM; k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] = sum;
    }
  }
}
```

Εικόνα 5.23 : Η μέθοδος matrixmult() που θα παραλληλίσουμε για το πρόγραμμα ελέγχου MatrixMult με τον τριπλά φωλιασμένο βρόχο for().

Όπως προαναφέραμε, ήταν δυνατόν να παραλληλίσουμε την κύρια μέθοδο υπολογισμού του πολλαπλασιασμού πινάκων (matrixmult()) που κατέχει και το μεγαλύτερο ποσοστό χρόνου εκτέλεσης στην εφαρμογή (99.6%). Η μέθοδος αυτή αποτελείται από ένα τριπλά φωλιασμένο βρόχο επανάληψης τύπου for() [Εικόνα 5.23]. Η παράλληλη μέθοδος που δημιουργείται από τον αυτόματο μεταφραστή JavaParallelTranslator φαίνεται στην Εικόνα 5.23. Όπως αναφέραμε στην περιγραφή λειτουργίας των κύριων μεθόδων της βιβλιοθήκης JACON, στην αρχή του κεφαλαίου αυτού (4.2.2 Ανάλυση λειτουργίας κύριων μεθόδων), η παράλληλη μέθοδος *Prarallel.forLoop* δημιουργεί ένα στόχο (task) για κάθε αύξηση του δείκτη *i = start*, δηλαδή δημιουργούνται [*(end – start)*] στόχοι, καθώς και ένα threadpool, με αριθμό νημάτων ίσο με τους διαθέσιμους πυρήνες που υπάρχουν στο υπολογιστικό σύστημα και συμμετέχουν για να διεκπεραιώσουν όλους τους στόχους. Κάθε πυρήνας στο παράλληλο σύστημα που χρησιμοποιήσαμε για τις πειραματικές μας μελέτες, εκτελεί πολλαπλούς στόχους κάθε φορά κατά τη διάρκεια εκτέλεσης της παράλληλης μεθόδου *Prarallel.forLoop* που υπάρχει στη μέθοδο που παραλληλίστηκε.

Στην Εικόνα 5.24 βλέπουμε σε κόκκινο πλαίσιο τον στόχο (task) που θα εκτελεστεί από τα νήματα που θα δημιουργηθούν κατά την παράλληλη εκτέλεση. Μπορούμε να παρατηρήσουμε ότι μέγεθος εργασίας που έχει να κάνει κάθε νήμα είναι αρκετά μεγάλο (προσπέλαση ενός δισδιάστατου πίνακα μεγέθους 1536 x 1536 θέσεων, πολλαπλασιασμός ολόκληρης γραμμής με στήλη και ανάθεση της τελικής τιμής στον πίνακα αποτελεσμάτων). Λόγω του ότι ο χρόνος εκτέλεσης των στόχων είναι ικανοποιητικός, τα νήματα δεν γίνονται πολύ ανταγωνιστικά κατά τη διάρκεια εκτέλεσης τους και έχουμε ικανοποιητικό συγχρονισμό της σειράς εκτέλεσης τους από τον χρονοπρογραμματιστή (TaskScheduler) καθώς και εξισορρόπηση φόρτου εργασίας (load balancing) μεταξύ των νημάτων. Ελάχιστο πρόβλημα δημιουργείται κατά την εκτέλεση της παράλληλης έκδοσης σε ένα μόνο πυρήνα όπου εκεί βλέπουμε να παρουσιάζεται ελάχιστη αρνητική βελτίωση στην επίδοση (0.908) λόγω καθυστερήσεων από την επιπλέον δημιουργία και προσάρτηση εργασίας στο νήμα που έχει η παράλληλη έκδοση ένατι της σειριακής.

```
public static void matrixmult() {

  //matrix multiplication => c[][] = a[][] x b[][]
  //@parallel
  /* Here comes the auto-created parallel code for For statement block!! */
  try {
    Parallel.forLoop(0, "<", NDIM, "++", new IForTask() {
      public void loopBody(int i) throws CancelException
      //Here comes the untouched block of parallized for-loop
      {
        for (int j = 0; j < NDIM; j++) {
          long sum = 0;
          for (int k = 0; k < NDIM; k++) {
            sum += a[i][k] * b[k][j];
          }
          c[i][j] = sum;
        }
      }
      //End of untouched block of parallized for-loop
    });
  } catch (CancelException e) {
    e.printStackTrace();
  }
  /* End of auto-created parallel code!! */
}
```

Εικόνα 5.24 : Η παράλληλη μέθοδος του προγράμματος ελέγχου MatrixMult που δημιουργήθηκε από τον αυτόματο μεταφραστή JavaParallelTranslator. Σε κόκκινο πλαίσιο φαίνεται ο στόχος που ουσιαστικά εκτελείται από τα νήματα.

Σε γενικές γραμμές μπορούμε να πούμε ότι τα αποτελέσματα που πήραμε από τις πειραματικές μετρήσεις που εκτελέσαμε επαληθεύουν και να πιστοποιούν τις εισηγήσεις που προτάθηκαν για τη βελτιστοποίηση προγραμμάτων σε Java [Κεφάλαιο 1, 1.2 Σκοπός]. Κάνοντας χρήση της παράλληλης βιβλιοθήκης προγραμματισμού JACON σε συνδυασμό με το γραφικό εργαλείο ήμι-αυτόματης μετάφρασης σειριακού κώδικα Java σε παράλληλο, Semi-Auto Parallelizer, που να κάνει χρήση της προηγούμενης βιβλιοθήκης ο προγραμματιστής είναι σε θέση να μεγιστοποιήσει την απόδοση του κώδικά του εστιάζοντας μόνο στην εργασία που το πρόγραμμά του έχει σκοπό να επιτελεί και όχι στις λεπτομέρειες για σωστή επίτευξη παραλληλισμού. Καταστούμε έτσι τους υπεύθυνους για την ανάπτυξη παράλληλων

εφαρμογών, προγραμματιστές, παραγωγικότερους μέσω την απλοποίησης της διαδικασίας προσθήκης παραλληλισμού και ταυτοχρονίας στις εφαρμογές τους.

Κεφάλαιο 6

Συμπεράσματα

6.1 Εισαγωγή

Οδηγούμενοι από την ανάγκη δημιουργίας υψηλού επιπέδου απόδοσης για μια ποικιλία εφαρμογών που πρέπει να καλύπτουν τόσο τις ανάγκες των απλών καταναλωτών αλλά και των επιχειρήσεων και του επιστημονικού τομέα περάσαμε στη γενιά των παράλληλων υπολογιστών και του παράλληλου υπολογισμού. Ο παραλληλισμός είναι μια γνωστή έννοια που προκύπτει συχνά στη καθημερινή μας ζωή. Το σημαντικό όμως είναι ότι ο παραλληλισμός μπορεί να συμβάλει από πολλές απόψεις στη σταθερή βελτίωση της απόδοσης των υπολογιστών. Με βάση αυτό, νέες αρχιτεκτονικές και καινούργια προγραμματιστικά μοντέλα άρχισαν να δημιουργούνται ανοίγοντας έτσι καινούργιους ορίζοντες στο χώρο της Πληροφορικής.

Η εξέλιξη της αρχιτεκτονικής μικροεπεξεργαστών εξαρτάται από τις μεταβαλλόμενες πτυχές της τεχνολογίας. Η αύξηση της πυκνότητας και της ταχύτητας των transistors ανά επεξεργαστή, η μνήμη και η συμπεριφορά προγράμματος γίνονται όλο και περισσότερο σημαντικά στην βιομηχανία αρχιτεκτονικής. Ενώ η τεχνολογία επιτρέπει περισσότερο σύνθετες σχεδιάσεις επεξεργαστών, υπάρχουν όμως φυσικά και προγραμματιστικά όρια στη χρησιμότητα αυτής της πολυπλοκότητας. Τα φυσικά όρια

περιλαμβάνουν τα όρια συσκευών/επεξεργαστών καθώς επίσης και τα πρακτικά όρια που αφορούν τη δύναμη και το κόστος. Τα όρια συμπεριφοράς προγράμματος προκύπτουν από απρόβλεπτα γεγονότα που εμφανίζονται κατά τη διάρκεια μιας εκτέλεσης. Οι αρχιτεκτονικές και οι εφαρμογές που επεκτείνουν αυτά τα όρια είναι ζωτικής σημασίας στη συνεχή εξέλιξη των επεξεργαστών.

Παρουσιάστηκε η ανάγκη για νέους τρόπους προσέγγισης του παράλληλου υπολογισμού. Οι προσεγγίσεις αυτές είναι ο παράλληλος προγραμματισμός και ο αυτόματος παραλληλισμός. Κάποιος που θέλει να εμπλακεί με τον παράλληλο υπολογισμό εξετάζοντας τις προσεγγίσεις αυτές με βάση κάποια κριτήρια είναι σε θέση να αποφασίσει ποιά ταιριάζει κάθε φορά στις ανάγκες του.

Τα παράλληλα προγραμματιστικά μοντέλα υλοποιούνται με διάφορους τρόπους : βιβλιοθήκες οι οποίες δημιουργούνται ως επί το πλείστον από σειριακές γλώσσες, ως γλωσσικές επεκτάσεις ή σαν πλήρη νέα πρότυπα εκτέλεσης. Είναι επίσης κατά προσέγγιση ταξινομημένα σε δύο είδη συστημάτων – αρχιτεκτονικών : αρχιτεκτονική Κοινής Μνήμης (Shared Memory) και αρχιτεκτονική Κατανεμημένης Μνήμης (Distributed Memory).

Οι περισσότεροι από τους σημερινούς αυτόματους μεταφραστές για παράλληλα συστήματα κάνουν μετάφραση πηγαίου κώδικα σε πηγαίο κώδικα (source to source) χρησιμοποιώντας υποδείξεις – οδηγίες openMP, MPI ή κάτι παρόμοιας σύνταξης, ενώ μερικοί άλλοι μεταφράζουν άμεσα σε κώδικα μηχανής. Αυτό μπορεί να γίνει είτε εντελώς αυτόματα, είτε με τη βοήθεια του προγραμματιστή χρησιμοποιώντας ένα γραφικό εργαλείο περιβάλλοντος (Graphical User Interface – GUI). Η πιο συνηθισμένη αυτόματα παραλληλοποιήσιμη γλώσσα προγραμματισμού είναι η FORTRAN και αυτό οφείλεται κυρίως στην έλλειψη δεικτών. Διάφορες τεχνικές και προσεγγίσεις παρουσιάζονται στα διαφορετικά προγράμματα όπου το κάθε ένα από αυτά συμβάλλει στην έρευνα της αυτόματης παραλληλοποίησης είτε για αρχιτεκτονικές Κοινής Μνήμης (Shared Memory), είτε για αρχιτεκτονικές Κατανεμημένης Μνήμης (Distributed Memory).

Όπως αναφέραμε και πιο πριν [Κεφάλαιο 1] κύριος στόχος αυτής της διπλωματικής εργασίας ήταν να προταθεί μια μεθοδολογία με κύριο σκοπό να αφαιρέσουμε από τον προγραμματιστή όσο το δυνατόν περισσότερο την ανάγκη να ασχολείται με τις χαμηλού επιπέδου λεπτομέρειες (threads, thread-pools, locks, synchronization) για την δημιουργία μιας σωστής και αποδοτικής παράλληλης εφαρμογής. Για να το πετύχουμε αυτό δημιουργήσαμε μια παράλληλη βιβλιοθήκη σε γλώσσα Java καθώς και ένα ημι-αυτόματο μεταφραστή σειριακού κώδικα Java σε παράλληλο που για να πετύχει τον σκοπό του κάνει χρήση της προηγούμενης βιβλιοθήκης. Οι προτάσεις μας έχουν να κάνουν αποκλειστικά με την έρευνα της αυτόματης παραλληλοποίησης για αρχιτεκτονική Κοινής Μνήμης (Shared Memory). Με τις δυο υλοποιήσεις που προτείναμε είμαστε σε θέση να καλύψουμε και τους δυο τρόπους προσέγγισης του παράλληλου υπολογισμού, παράλληλο προγραμματισμό και αυτόματο παραλληλισμό για την αρχιτεκτονική Κοινής Μνήμης (Shared Memory). Δίνουμε με αυτό τον τρόπο τη δυνατότητα στον προγραμματιστή να μεγιστοποιήσει την απόδοση του κώδικά του εστιάζοντας μόνο στην εργασία που το πρόγραμμά του έχει σκοπό να επιτελεί και όχι στις λεπτομέρειες για σωστή επίτευξη παραλληλισμού. Καταστούμε έτσι τους υπεύθυνους για την ανάπτυξη παράλληλων εφαρμογών, προγραμματιστές, παραγωγικότερους μέσω την απλοποίησης της διαδικασίας προσθήκης παραλληλισμού και ταυτοχρονίας στις εφαρμογές τους.

6.2 Συμπεράσματα

Η μελέτη αυτή είχε ως στόχο να εισηγηθεί μια μεθοδολογία που να μπορεί να προσφέρει βοήθεια στη βελτιστοποίηση απόδοσης και επίδοσης ενός προγράμματος κάνοντας αποδοτική χρήση των πολυπύρηνων συστημάτων που κατακλύζουν την σημερινή εποχή αφαιρώντας από τον προγραμματιστή όσο το δυνατόν περισσότερο την ανάγκη να ασχολείται με τις χαμηλού επιπέδου λεπτομέρειες. Για τις απαιτήσεις αυτής της διπλωματικής εργασίας, μεγαλύτερο μέρος αφιερώθηκε στην δημιουργία της παράλληλης βιβλιοθήκης σε γλώσσα Java καθώς και ενός γραφικού εργαλείου ήμι-αυτόματου μεταφραστή σειριακού κώδικα Java σε παράλληλο. Ο τρόπος με τον οποίο διεξάχθηκε η υλοποίησή του είναι αρκετά αφαιρετικός και επαναχρησιμοποιήσιμος, με αποτέλεσμα στον μέλλον να μπορεί να βοηθήσει στην περαιτέρω επέκτασή τους.

Όσον αφορά τα αποτελέσματα των πειραματικών μετρήσεων της επίδοσης διαφόρων μικρών προγραμμάτων που έρχονται να επαληθεύσουν τις εισηγήσεις που προτάθηκαν για βελτιστοποίηση προγραμμάτων Java, μπορούμε να συμπεράνουμε τα εξής:

- ✓ Είναι σημαντικό να γνωρίζουμε τη δομή και τον τρόπο λειτουργίας της εφαρμογής/προγράμματος που θέλουμε να βελτιώσουμε. Έχοντας αυτό υπ' όψη, είμαστε σε θέση να αποφασίσουμε ποιά κομμάτια της εφαρμογής μας θέλουμε και είναι σημαντικό να βελτιώσουμε.

- ✓ Η μεθοδολογία που προτάθηκε και τα εργαλεία που δημιουργήθηκαν για να την υποστηρίξουν καταστούν τον προγραμματιστή απόλυτα υπεύθυνο για τη σωστή επιλογή των βρόγχων επανάληψης που θα τύχουν αυτόματης επεξεργασίας. Ο προγραμματιστής είναι υπεύθυνος ο βρόγχος επανάληψης που θα επιλέξει για να τύχει αυτόματης επεξεργασίας να είναι "απολαυστικά/βολικά παράλληλος" (delightfully/pleasantly parallel).

- ✓ Η μέγιστη αναμενόμενη βελτίωση στην επίδοση μιας εφαρμογής περιορίζεται από το ποσοστό του σειριακού κομματιού που έχει έναντι του παράλληλου. Αν το ποσοστό του παράλληλου μέρους μιας εφαρμογής είναι μικρότερο του 50% τότε η μέγιστη αναμενόμενη βελτίωση δεν μπορεί να ξεπερνά τις 2 φορές για οποιονδήποτε αριθμό πυρήνων συμμετέχουν στην παράλληλη εκτέλεση.

- ✓ Δεν φτάνει μόνο να παραλληλοποιήσουμε/βελτιώσουμε κάποιο κομμάτι του κώδικα που να μπορεί να εκτελεστεί παράλληλα. Πρέπει αυτό το κομμάτι να έχει ουσιαστικό ρόλο στη συνολική εκτέλεση της εφαρμογής που θέλουμε να βελτιώσουμε. Επιλέγοντας μη ουσιαστικά κομμάτια κώδικα που απαιτούν ελάχιστο χρόνο εκτέλεσης έχουμε μεγάλες πιθανότητες να καταλήξουμε σε κάποιο αρνητικό αποτέλεσμα επιτυγχάνοντας τελικά αρνητική βελτίωση.

- ✓ Τα τεχνικά χαρακτηριστικά του παράλληλου υπολογιστικού συστήματος πάνω στο οποίο θα εκτελεστεί η παράλληλη έκδοση της εφαρμογής παίζουν σημαντικότατο ρόλο για τα τελικά αποτελέσματα που έχουν να κάνουν με τα ποσοστά βελτίωσης της επίδοσης. Εκτός από τον αριθμό των επεξεργαστών από

τους οποίους αποτελείται το παράλληλο σύστημα θα πρέπει να λαμβάνεται υπόψη και η επίδραση που θα έχει η κρυφή μνήμη (cache effect) [6, 7, 13, 19] μέσα από τις διαφορετικές ιεραρχίες μνήμης που υπάρχουν στα σύγχρονα υπολογιστικά συστήματα και επεξεργαστές. Το μέγεθος της συσσωρευμένης κρυφής μνήμης των επεξεργαστών σε ένα πολυπύρηνο υπολογιστικό σύστημα μπορεί να βελτιώσει περεταίρω την επίδοση μιας παράλληλης πολυνηματικής εφαρμογής και να έχει σαν αποτέλεσμα επιπλέον επιτάχυνση (speedup) η οποία στο τελικό της σύνολο να είναι μερικές φορές ακόμα και μεγαλύτερη της θεωρητικής μέγιστης (super linear speedup).

- ✓ Τα εργαλεία που αναπτύχθηκαν στα πλαίσια αυτής της διπλωματικής εργασίας για να υποστηρίξουν τις εισηγήσεις που προτάθηκαν για βελτιστοποίηση της επίδοσης και απόδοσης εφαρμογών που εκτελούνται σε συστήματα με περισσότερους από έναν επεξεργαστές λύνουν τα χέρια των προγραμματιστών και κάνουν πιο εύκολη την δουλειά τους. Καταστούμε έτσι τους υπεύθυνους για την ανάπτυξη παράλληλων εφαρμογών, προγραμματιστές, παραγωγικότερους μέσω την απλοποίησης της διαδικασίας προσθήκης παραλληλισμού και ταυτοχρονίας στις εφαρμογές τους. Ο προγραμματιστής έχει τη δυνατότητα να μεγιστοποιήσει την απόδοση του κώδικά του εστιάζοντας μόνο στην εργασία που το πρόγραμμά του έχει σκοπό να επιτελεί και όχι στις λεπτομέρειες για σωστή επίτευξη παραλληλισμού.

6.3 Μελλοντική Εργασία

Σε αυτό το κομμάτι θα προσπαθήσουμε να προτείνουμε κάποιες εισηγήσεις και προτάσεις οι οποίες μπορούν να θεωρηθούν σαν επιπλέον μελλοντική εργασία και μελέτη. Αφορούν βελτιστοποιήσεις και προσθήκες οι οποίες θα μπορούσαν να επεκτείνουν τις υφιστάμενες μεθοδολογίες και εργαλεία τα οποία προτάθηκαν και υλοποιήθηκαν προηγουμένως σαν μέρος αυτής της διπλωματικής εργασίας και μελέτης.

- ✓ Η βιβλιοθήκη παράλληλου προγραμματισμού JACON υλοποιήθηκε με αρκετά αφαιρετικό και επαναχρησιμοποιήσιμο τρόπο. Αυτό έχει σαν αποτέλεσμα στον μέλλον να μπορεί να βοηθήσει στην περαιτέρω επέκταση της. Θα μπορούσε να

επεκταθεί με ενσωμάτωση περεταίρω παράλληλων μεθόδων για υποστήριξη και των υπολοίπων βρόγχων επανάληψης (foreach, while, do..while) που υπάρχουν και υποστηρίζονται από την γλώσσα προγραμματισμού Java. Επίσης θα μπορούσαν να προστεθούν παράλληλες μέθοδοι για αντικατάσταση συνηθισμένων πράξεων που κάνουν χρήση ευρέως διαδεδομένων αλγορίθμων, όπως για παράδειγμα ταξινόμηση στοιχείων με βάση τον αλγόριθμο Quick Sort (parallel.quicksort).

- ✓ Η παρούσα παράλληλη μέθοδος Parallel.atomic που υπάρχει στην βιβλιοθήκη παράλληλου προγραμματισμού JACON υποστηρίζει την ύπαρξη μηχανισμού για ταυτόχρονη πρόσβαση σε έναν κοινό πόρο/μεταβλητή για σκοπούς ενημέρωσής του. Θα μπορούσε να επεκταθεί υποστηρίζοντας κάποιο μηχανισμό ταυτόχρονης πρόσβασης σε περισσότερους από ένα κοινούς πόρους/μεταβλητές για σκοπούς ενημέρωσής τους.

- ✓ Η βιβλιοθήκη παράλληλου προγραμματισμού JACON έχει υλοποιηθεί υποστηρίζοντας μόνο μεθόδους για παράλληλη εκτέλεση και σε αρχιτεκτονικές Κοινής Μνήμης (Shared Memory). Θα μπορούσε να γίνει επέκτασή της έτσι ώστε να υποστηρίζει και αρχιτεκτονικές Κατανεμημένης Μνήμης (Distributed Memory). Επίσης θα μπορούσε να γίνει περεταίρω εξέλιξή της για να μπορεί να υποστηρίζει εκτέλεση παράλληλων εφαρμογών και στην αρχιτεκτονική υλικού και λογισμικού CUDA (Compute Unified Device Architecture) που σχεδιάστηκε για την υποστήριξη της παράλληλης επεξεργασίας σε κάρτες γραφικών (GPU – Graphics Processing Unit) της nVidia.

- ✓ Τα εργαλεία και ο τρόπος υλοποίησης του μεταγλωττιστή αναδόμησης JavaParallelTranslator θα μπορούσαν να χρησιμοποιηθούν για την ανάπτυξη παρόμοιων μεταγλωττιστών αναδόμησης και για άλλες γλώσσες προγραμματισμού (όπως C, C++, C# κ.α.) και να ενσωματωθούν και αυτά στο γραφικό εργαλείο ημιαυτόματης μετάφρασης πηγαίου κώδικα Semi-AutoParallelizer.

- ✓ Ο ήδη υπάρχων μεταγλωττιστής αναδόμησης JavaParallelTranslator θα μπορούσε να επεκταθεί με περισσότερες τεχνικές ανάλυσης έτσι ώστε να

λαμβάνει υπόψη και τα τεχνικά χαρακτηριστικά του παράλληλου υπολογιστικού συστήματος για το οποίο προορίζεται να εκτελεστεί η παράλληλη έκδοση της εφαρμογής που θα παραχθεί. Γνωρίζοντας τα τεχνικά χαρακτηριστικά του παράλληλου υπολογιστικού συστήματος θα μπορούσε να προσθέσει επιπλέον βελτιστοποιήσεις στη δομή του κώδικα και στις παράλληλες μεθόδους που χρησιμοποιούνται (π.χ. ορίζοντας ρητά την παράμετρο μεγέθους στόχου (grain size)) κάνοντας την παράλληλη έκδοση της εφαρμογής ακόμα πιο αποδοτική για συγκεκριμένο παράλληλο υπολογιστικό σύστημα.

- ✓ Ο μεταγλωττιστής αναδόμησης JavaParallelTranslator θα μπορούσε να τροποποιηθεί προφέροντας την υποστήριξη αυτόματης ανίχνευσης παραλληλισμού από τον ίδιο χωρίς να χρειάζεται ο παραλληλισμός των βρόχων να πρέπει να προσδιοριστεί από τον προγραμματιστή με τη βοήθεια οδηγιών ακριβώς πριν από τον βρόχο επανάληψης που αυτός θα επιλέξει να τύχει αυτόματης επεξεργασίας. Έτσι ο μεταγλωττιστής αναδόμησης θα μπορούσε να είναι υπεύθυνος έτσι ώστε ο βρόγχος επανάληψης που θα επιλέξει για να τύχει αυτόματης επεξεργασίας να είναι "απολαυστικά/βολικά παράλληλος" (delightfully/pleasantly parallel). Αυτό θα ήταν δυνατόν να επιτευχθεί ενσωματώνοντας κάποιου είδους ανάλυσης εξαρτήσεων (dependence analysis) στον ίδιο τον μεταγλωττιστή αναδόμησης. Η επίτευξη τέτοιου στόχου σίγουρα είναι αρκετά υψηλή αφού απαιτείται χρήση εξειδικευμένης ανάλυσης εξαρτήσεων για βρόγχους επανάληψης και προσπέλασης πινάκων πράγμα που κάνει την υλοποίηση αυτού του στόχου πάρα πολύ δύσκολη.

- ✓ Τροποποίηση του γραφικού εργαλείου ημιαυτόματης μετάφρασης πηγαίου κώδικα, Semi-AutoParallelizer, έτσι ώστε να μπορεί να εγκατασταθεί και να λειτουργεί και σε άλλα λειτουργικά συστήματα εκτός των Windows (π.χ. Linux, MacOS κ.α.) καθώς και να μπορεί να υποστηρίζει επιπλέον αρχιτεκτονικές εκτός της Κοινής Μνήμης (Shared Memory) (π.χ. Κατανεμημένης Μνήμης (Distributed Memory), αρχιτεκτονική υλικού και λογισμικού CUDA (Compute Unified Device Architecture) κ.α.).

- ✓ Το γραφικό εργαλείο ημιαυτόματης μετάφρασης πηγαίου κώδικα, Semi-AutoParallelizer, θα μπορούσε να εμπλουτιστεί με επιπλέον λειτουργίες ενσωματώνοντας εργαλεία τα οποία θα μπορούσε να χρησιμοποιήσει ο χρήστης/προγραμματιστής έτσι ώστε να διευκολυνθεί στο να αποφασίσει ποια κομμάτια του κώδικά του χρειάζονται βελτιστοποίηση καθώς και ποια από αυτά μπορούν να τύχουν παραλληλισμού χωρίς να δημιουργήσουν αλλοιώσεις αποτελεσμάτων ή άλλα προβλήματα κατά τη διάρκεια μιας παράλληλης εκτέλεσης. Τέτοιου είδους εργαλεία θα μπορούσαν να είναι : εργαλείο δυναμικής ανάλυσης της συμπεριφοράς και του τρόπου εκτέλεσης μιας εφαρμογής κατά τη διάρκεια εκτέλεσής της και παρουσίαση στατιστικών στοιχείων που έχουν να κάνουν με ποσοστά συχνότητας και διάρκειας εκτέλεσης κλήσεων συναρτήσεων (profiling tool), εργαλεία ανάλυσης κώδικα εντολών και δεδομένων προγράμματος (control and data flow analysis), εργαλείο ανάλυσης εξαρτήσεων σε επίπεδο βρόχων επανάληψης και παρουσίαση στατιστικών στοιχείων που έχουν να κάνουν με προσβάσεις σε πίνακες και τροποποίησης στοιχείων τους καθώς και με αποφάσεις σε εντολές διακλάδωσης εντός του βρόγχου επανάληψης (loop dependence analysis).

Βιβλιογραφία

[1] Aart J.C. Bik and Dennis B. Gannon, "javab - a prototype bytecode parallelization tool". Computer Science Dept., Indiana University, Lindley Hall 215, Bloomington, Indiana 47405-4101, USA, June 1998.

[2] Aart J.C. Bik and Dennis B. Gannon, "javab Manual (version 1.0BETA)". Computer Science Dept., Indiana University, Lindley Hall 215, Bloomington, Indiana 47405-4101, USA, June 1998.

[3] Aart J.C. Bik and Dennis B. Gannon, "javar - a prototype Java restructuring compiler". Computer Science Dept., Indiana University, Lindley Hall 215, Bloomington, Indiana 47405-4101, USA, June 1998.

[4] Aart J.C. Bik and Dennis B. Gannon, "javar Manual (version 1.3BETA)". Computer Science Dept., Indiana University, Lindley Hall 215, Bloomington, Indiana 47405-4101, USA, June 1998.

[5] Aho V. Alfred, Lam S. Monica, Sethi Ravi, Ullman D. Jeffrey, Compilers: Principles, Techniques, and Tools, 2nd Edition, Addison-Wesley, 2007.

[6] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Introduction to Parallel Computing, Second Edition, 2003.

[7] Calvin Lin, Larry Snyder, Principles of Parallel Programming, Addison-Wesley, 2009.

[8] Eckel Bruce, Thinking in Java, 4th Edition, Prentice-Hall, 2006.

[9] Fermi, NVIDIA’s Next Generation CUDA Compute Architecture, 2009.

[10] Gustafson, J. L., “Reevaluating Amdahl’s Law,“ Communications of the ACM, Volume 31, 1987, pp. 532–533.

[11] Hall M. W., Anderson J. M., Amarasinghe S.P., Murphy B. R., Liao S.W., Lam M., Maximizing multiprocessor performance with the SUIF compiler, IEEE Computer, Volume 29, Number 12, pages 84-89, December 1996.

[12] Heather Hanson, Stephen W., Keckler Doug Burger, Coordinated Power, Energy, and Temperature Management for High-Performance Microprocessors, Computer Architecture and Technology Laboratory, 1996.

[13] Helmbold, D. P. and McDowell, C. E., “Modeling Speedup(n) greater than n,” 1989, International Conference on Parallel Processing Proceedings, Volume III, 1989, pp. 219–225.

[14] Henry Kasim, Verdi March, Rita Zhang, Simon See, Survey on Parallel Programming Model, J. Cao et al. (Eds.): NPC 2008, LNCS 5245, pp. 266–275, 2008, IFIP International Federation for Information Processing, 2008.

[15] Hideki Saito , Nicholas Stavrakos , Steven Carroll , Constantine Polychronopoulos , Alex Nicolau, The Design of the PROMIS Compiler (1999), International Journal of Parallel Programming - Special issue on international symposium on high performance computing 1997, part I archive, Volume 28 Issue 2, April 2000.

[16] Holger M. Kienle, "A SUIF Java Compiler". Technical Report TRCS98-18, August, 1998.

[17] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, EECS Department,

University of California, Berkeley, No. UCB/EECS-2006-183, December 18, 2006.

[18] M. J. Flynn, Basic issues in microprocessor architecture, Journal of Systems Architecture: the EUROMICRO Journal archive, Volume 45, Issue 12-13, Pages: 939 - 948, June 1999.

[19] Mark D. Hill and Michael R. Marty, Amdahl's Law in the Multicore Era, Volume 41 , Issue 7, 0018-9162, IEEE Computer Society Press Los Alamitos, CA, USA, July 2008.

[20] Marc Loy, Robert Eckstein, Java Swing, 2nd Edition, O'Reilly Media, 2002.

[21] Mary W. Hall , Saman P. Amarasinghe , Brian R. Murphy , Shih-Wei Liao , Monica S. Lam, Interprocedural parallelization analysis in SUIF, ACM Transactions on Programming Languages and Systems (TOPLAS), v.27 n.4, p.662-731, July 2005.

[22] Peter Marksteiner, High-performance computing — an overview, Vienna University Computer Center Universitätsstraße 7, A-1010, Vienna, Austria, February 11, 1999.

[23] Rajiv Gupta, Santosh Pande, Kleanthis Psarris, Vivek Sarkar, Compilation techniques for parallel systems, Parallel Computing, Vol. 25, No. 13--14, 1999, pp. 1741-1783.

[24] Robert P. Wilson , Robert S. French , Christopher S. Wilson , Saman P. Amarasinghe , Jennifer M. Anderson , Steve W. K. Tjiang , Shih-Wei Liao , Chau-Wen Tseng , Mary W. Hall , Monica S. Lam , John L. Hennessy, SUIF: an infrastructure for research on parallelizing and optimizing compilers, ACM SIGPLAN Notices, v.29 n.12, p.31-37, Dec. 1994.

[25] S. Furber, The Future of Computer Technology and its Implications for the Computer Industry, Oxford University Press Oxford, UK, Volume 51, Issue 6, Pages 735-740, November 2008.

[26] Stephen Toub, Patterns of Parallel Programming, Understanding and Applying Parallel Patterns with the .net framework 4 and Visual C#, Parallel Computing Platform, Microsoft Corporation, February 2010.

[27] Subhasis Banerjee, G. Surendra, S.K. Nandy, On the effectiveness of phase based regression models to trade power and performance using dynamic processor adaptation, Journal of Systems Architecture, Vol. 54, pages 797–815, 2008.

[28] Terence Parr, The Definitive ANTLR Reference Building Domain-Specific Languages, 2009.

[29] The Portland Group, PGI Fortran & C Accelerator Programming Model current features, ver. 1.2, March 2010.

[30] Tom R. Halfhill, PARALLEL PROCESSINGWITH CUDA, Microprocessor Report, 2008.

[31] Antlr.org home, http://www.antlr.org/, Accessed: June 2010.

[32] CUDA, http://www.nvidia.com/object/cuda_home.html, Accessed: February 2010.

[33] Eclipse.org home, http://www.eclipse.org, Accessed: May 2010.

[34] Fortress, http://projectfortress.sun.com/Projects/Community, Accessed: February 2010.

[35] Java™ Tutorials, http://download.oracle.com/javase/tutorial/, Accessed: June 2010.

[36] Java 6.0 API Documentation, http://download.oracle.com/javase/6/docs/api/, Accessed: June 2010.

[37] Jprofiler home, http://www.ej-technologies.com/products/jprofiler/overview.html, Accessed: October 2010.

[38] MPI, http://www.mcs.anl.gov/research/projects/mpi/, Accessed: February 2010.

[39] Netbeans.org home, http://www.netbeans.org/, Accessed: June 2010.

[40] OpenMP, http://openmp.org/wp/, Accessed: February 2010.

[41] Parallel Programming in the .NET Framework, http://msdn.microsoft.com/en-us/library/dd460693%28VS.100%29.aspx, Accessed: February 2010.

[42] Pthreads, https://computing.llnl.gov/tutorials/pthreads/, Accessed: February 2010.

[43] Smart Install Maker home, http://www.sminstall.com/, Accessed: August 2010.

[44] SUIF Compiler Infrastructure, http:/suif.stanford.edu/, Accessed: February 2010.

[45] The Portland Group, PGI Documentation http://www.pgroup.com/resources/docs.htm, Accessed: February 2010.

[46] UPC, http://upc.lbl.gov/docs/, Accessed: February 2010.

Παράρτημα Α

Κυριότεροι κώδικες υλοποίησης της βιβλιοθήκης παράλληλου προγραμματισμού, JACON

IForAtomicTask.java

```
/*
 * JaCon (Java Concurrency) - Software library for parallel programming
 * Copyright (C) 2010 Christos Kyriacou UCY
 */
package com.cs.ucy.ac.cy.jacon;

/**
 * This interface should be implemented when using the
<code>Parallel.forLoopAtomic</code>
 * method.
 * @see Parallel
 */
public interface IForAtomicTask
{
        /**
         * This method will be executed one time per iteration in your loop.
         * @param start The start iteration value.
          * @param end The end iteration value.
          * @return An Object value
         * @throws CancelException
          */
        public Object loopBody(int start, int end) throws CancelException;
}
```

IForTask.java

```
/*
 * JaCon (Java Concurrency) - Software library for parallel programming
 * Copyright (C) 2010 Christos Kyriacou UCY
 */
package com.cs.ucy.ac.cy.jacon;

/**
 * This interface should be implemented when using the <code>Parallel.forLoop</code>
 * method.
 * @see Parallel
 */
public interface IForTask
{
        /**
         * This method will be executed one time per iteration in your loop.
         * @param i The current iteration value.
         * @throws CancelException
          */
        public void loopBody(int i) throws CancelException;
}
```

IReduceTask.java

```
/*
 * JaCon (Java Concurrency) - Software library for parallel programming
 * Copyright (C) 2010 Christos Kyriacou UCY
 */
package com.cs.ucy.ac.cy.jacon;

/**
 * This interface should be implemented when using the <code>Parallel.reduce</code>
method.
 * @param <T> At type that must be associative
 * @see ICombineTask
 * @see Parallel
 */
public interface IReduceTask<T>
{
	/**
	 *
	 * @param i The index, which is incremented by one after each iteration
	 * @return The reduced value
	 * @throws CancelException
	 */
	public T reduce(int i) throws CancelException;
}
```

Parallel.java

```
/*
 * JaCon (Java Concurrency) - Software library for parallel programming
 * Copyright (C) 2010 Christos Kyriacou UCY
 */
package com.cs.ucy.ac.cy.jacon;

/**
 * <code>Parallel</code> is a static class providing a basic parallel method
 * such as <code>forLoop</code>.<br/>
 * The method is implemented with JaCon's built-in scheduler that allows multiple tasks
to run
 * efficiently on the available processors/cores. It's important to understand that the
built-in scheduler
 * is not preemptive, meaning that it's not meant for executing tasks that run forever.
If a task temporarily
 * blocks, the scheduler detects it and ensures that the processor is kept busy
executing another task.
 * For more information about the scheduler, see the {@link Task} class.
 * @see Task
 */
public class Parallel
{
	private Parallel() { }
	/**
	 * A parallel version of a standard <code>for</code> loop.
	 * <code>forLoop</code> is similar to a standard for loop starting at index i =
<code>start</code> and running as long as condition <code>condition</code> between
<code>start</code> and <code>end</code> is true, and changing index i as
<code>action</code> describes.
	 * <code>loopBody</code> is called once in each iteration.<br/>
	 * Grain size is implicitly set to one, implying that the unit of work is one
call to <code>loopBody</code>. Hence one task is created for
	 * each index i. It's often possible to increase performance by adjusting the
grain size. To do so use
	 * <code>forLoop(int start, int end, int grainSize, IForTask
loopBody)</code>.<br/>
```

```
        * <code>forLoop</code> catches any uncaught exception raised inside the tasks.
Once the exception is caught <code>forLoop</code> cancels all
        * running tasks that it has started and throws a <code>CancelException</code>,
containing the original exception. JaCon cancels all
        * tasks but it doesn't just brutally stop them - the user defined code must
detect and handle the cancellation.
        * @param start First index.
         * @param condition Condition to be checked so the the for loop runs as long as
condition <code>condition</code> between <code>start</code> and <code>end</code> is
true.
        * @param end Last index, that indicates the end of loop execution (included or
not as referred by <code>condition</code>).
         * @param action Increase/Decrease action that will performed on
<code>start</code> after each <code>loopbody</code> execution.
        * @param loopBody The work to be done in each iteration.
        * @throws CancelException is thrown if one or more of the tasks cancel
themselves, either by throwing an uncaught
        * exception, or by explicitly invoking <code>cancel</code> on the current
<code>Task</code>.
        */
       public static void forLoop(final int start, final String condition, final int
end, final String action, final IForTask loopBody) throws CancelException
       {
              forLoop(start, condition, end, action, 1, loopBody);
       }
       /**
        * A parallel version of a standard for loop.
        * </summary>
        * <remarks>
        * <code>forLoop</code> is similar to a standard for loop starting at index i =
<code>start</code> and running as long as condition <code>condition</code> between
<code>start</code> and <code>end</code> is true, and changing index i as
<code>action</code> describes.
        * <code>loopBody</code> is called once in each iteration and index i is
incremented by one after each iteration. <br/>
        * <cde>grainSize</code> specifies a reasonable number of iterations in each
task. If the number of iterations is larger than <code>grainSize</code>, data
        * is split in two and handled separately.
        * Adjusting the grain size can lead to better performance, but the optimal grain
        * size depends on the problem at hand. Increasing the grain size will decrease
the amount of tasks,
        * and thus decrease the overhead of setting up tasks. But a small amount of
tasks might introduce some load balancing problems,
        * if no tasks are available for free processors. Decreasing the grain size will
increase the amount of tasks and thereby ensure
        * better load balancing, but on the other hand it will also increase the
overhead of setting up tasks. Use
        * <code>Parallel.forLoop(int start, int end, IForTask loopBody)</code> at first
and once your code is working, experiment with
        * the grain size.<br/>
        * <code>forLoop</code> catches any uncaught exception raised inside the tasks.
Once the exception is caught <code>forLoop</code> cancels all
        * running tasks it has started and throws a <code>CancelException</code>,
containing the original exception. JaCon cancels  all tasks but it doesn't
        * just brutally stop them - the user defined code
        * must detect and handle the cancellation.
        * @param start First index.
         * @param condition Condition to be checked so the the for loop runs as long as
condition <code>condition</code> between <code>start</code> and <code>end</code> is
true.
        * @param end Last index, that indicates the end of loop execution (included or
not as referred by <code>condition</code>).
         * @param action Increase/Decrease action that will performed on
<code>start</code> after each <code>loopbody</code> execution.
        * @param grainSize A reasonable number of work iterations in each task
        * @param loopBody The work to be done in each iteration.
        * @throws CancelException is thrown if one or more of the tasks cancel
themselves, either by throwing an uncaught
        * exception, or by explicitly invoking <code>cancel</code> on the current
<code>Task</code>.
        * @throws IllegalArgumentException is thrown if the grain size is less than 1.
        */
       public static void forLoop(final int start, final String condition, final int
end, final String action, final int grainSize, final IForTask loopBody) throws
CancelException
```

```
	{
		if(grainSize < 1)
			throw new IllegalArgumentException("Grain size cannot be less than
1.");
		 if (!(condition.equals("<") || condition.equals("<=") ||
condition.equals(">") || condition.equals(">=") || condition.equals("!=")))
		    throw new IllegalArgumentException("Condition '" + condition + "'
cannot be proccess in parallel.");
		 if (!(action.equals("++") || action.equals("--")))
		     throw new IllegalArgumentException("Condition '" + condition + "'
cannot be proccess in parallel.");
		new Async()
		{
			public void run() throws CancelException
			{
				traverseFor(start, condition, end, action, grainSize,
loopBody);
			}
		}.waitFor();
	}
	 // ************** Auxiliary methods *****************
	private static void traverseFor(final int start, final String condition, final
int end, final String action, final int grainSize, final IForTask loopBody) throws
CancelException
	{
		 int num;
		 if (action.equals("--"))
		     num = start - end;
		 else
		     num = end - start;
		if (num > grainSize)
		{
			  final int middle;
			  if (action.equals("--"))
			      middle = start - (num / 2) ;
			  else
			      middle = (num / 2) + start;
			 Async task = new Async()
			 {
				public void run() throws CancelException
				{
					traverseFor(start, condition, middle, action,
grainSize, loopBody);
				}
			 }.start();
			 traverseFor(middle, condition, end, action, grainSize, loopBody);
			 task.waitFor();
		}
		else
		{
		     if (condition.equals("<") && action.equals("++")){
		         //System.out.println("Will be executed : for(int i = " + start +
"; i < " + end + "; i++)");
			 for (int i = start; i < end; i++)
				loopBody.loopBody(i);
		     }
		     if (condition.equals("<=") && action.equals("++")){
		         //System.out.println("Will be executed : for(int i = " + start +
"; i <= " + end + "; i++)");
			 for (int i = start; i <= end; i++)
				loopBody.loopBody(i);
		     }
		     if (condition.equals(">") && action.equals("--")){
		         //System.out.println("Will be executed : for(int i = " + start +
"; i > " + end + "; i--)");
			 for (int i = start; i > end; i--)
				loopBody.loopBody(i);
		     }
		     if (condition.equals(">=") && action.equals("--")){
		         //System.out.println("Will be executed : for(int i = " + start +
"; i >= " + end + "; i--)");
			 for (int i = start; i >= end; i--)
				loopBody.loopBody(i);
		     }
		     if (condition.equals("!=") && action.equals("++")){
```

```
                        //System.out.println("Will be executed : for(int i = " + start +
"; i != " + end + "; i++)");
                        for (int i = start; i != end; i++)
                              loopBody.loopBody(i);
                    }
                    if (condition.equals("!=") && action.equals("--")){
                        //System.out.println("Will be executed : for(int i = " + start +
"; i != " + end + "; i--)");
                       for (int i = start; i != end; i--)
                              loopBody.loopBody(i);
                    }
                }
        }
}
```

ParallelAtomic.java

```
/*
 * JaCon (Java Concurrency) - Software library for parallel programming
 * Copyright (C) 2010 Christos Kyriacou UCY
 */
package com.cs.ucy.ac.cy.jacon;

/**
 * <code>ParallelAtomic</code> is a static class providing some basic parallel methods
 * such as <code>forLoopAtomic</code> and <code>reduce</code>.<br/>
 * All methods are implemented with JaCon's built-in scheduler that allows multiple
tasks to run
 * efficiently on the available processors/cores. It's important to understand that the
built-in scheduler
 * is not preemptive, meaning that it's not meant for executing tasks that run forever.
If a task temporarily
 * blocks, the scheduler detects it and ensures that the processor is kept busy
executing another task.
 * For more information about the scheduler, see the {@link Task} class.
 * @see Task
 */
public class ParallelAtomic {
    private ParallelAtomic() {}
    /**
     * Calculates and combines values to a single value.
     * <code>reduce</code> combines multiple associative values into a single value. The
<code>IReduceTask</code> <code>loopBody</code>
     * takes an integer between <code>start</code> and <code>end</code> and returns an
element of type  <code>T</code>. The <code>ICombineTask</code> takes two such
     * elements and combine them. <br/>
     * <code>reduce</code> avoids using locks by dividing the problem into a binary task
tree where each task
     * updates it's own local value.<br/>
     * Grain size is implicit set to two. It's often possible to increase performance by
adjusting the grain size.
     * To do so use <code>reduce(int start, int end, T initialValue, int grainSize,
IReduceTask loopBody, ICombineTask combineMethod)</code>.
     * Parallel.<br/>
     * <code>reduce</code> catches any uncaught exception raised inside the tasks. Once
the exception is caught <code>reduce</code> cancels all
     * running tasks and throws a <code>CancelException</code>, containing the original
exception. JaCon cancels all
     * tasks but it doesn't just brutally stop them - the user defined code must detect
and handle the cancellation.
     * @param <T> A type that must be associative
     * @param start First index, which is incremented by one until it equals
<code>end</code>.
     * @param end Last index, which is not included in the loop. The loop runs as long
as <code>start</code> is less than <code>end</code>.
     * @param initialValue The initial value for the result.
     * @param loopBody A task that is executed once for each iteration.
     * @param combineMethod A task used to combine the local results
     * @return A single value of type <code>T</code>
```

```
     * @throws CancelException is thrown if one or more of the tasks cancel themselves,
either by throwing an uncaught
     * exception, or by explicitly invoking <code>cancel</code> on the current
<code>Task</code>.
     * @example <a href="doc-files/ReduceExample.java.html"><code>reduce</code>
example</a>
     */
    public static <T> T reduce(final int start, final int end, final T initialValue,
final IReduceTask<T> loopBody, final ICombineTask<T> combineMethod) throws
CancelException {
        return reduce(start, end, initialValue, 1, loopBody, combineMethod);
    }

    /**
     * Calculates and combines values to a single value.
     * <code>reduce</code> combines multiple associative values into a single value. The
<code>IReduceTask</code> <code>loopBody</code>
     * takes an integer between <code>start</code> and <code>end</code> and returns an
element of type  <code>T</code>. The <code>ICombineTask</code> takes two such
     * elements and combine them. <br/>
     * <code>reduce</code> avoids using locks by dividing the problem into a binary task
tree where each task
     * updates it's own local value.<br/>
     * <code>grainSize</code> specifies a reasonable number of iterations in each task.
If the number of iterations is more than <code>grainSize</code>, data
     * is split in two and handled separately.
     * Adjusting the grain size can lead to better performance, but the optimal grain
size depends on the problem at hand.
     * Increasing the grain size will decrease the
     * amount of tasks, and thus decrease the overhead of setting up tasks. But a small
amount of tasks could introduce some load
     * balancing problems, if no tasks are available for free processors. <br/>
     * <code>reduce</code> catches any uncaught exception raised inside the tasks. Once
the exception is caught <code>reduce</code> cancels all
     * running tasks and throws a <code>CancelException</code>, containing the original
exception. JaCon cancels all
     * tasks but it doesn't just brutally stop them - the user defined code must detect
and handle the cancellation.
     * @param <T> A type that must be associative
     * @param start First index, which is incremented by one until it equals
<code>end</code>.
     * @param end Last index, which is not included in the loop. The loop runs as long
as <code>start</code> is less than <code>end</code>.
     * @param initialValue The initial value for the result.
     * @param grainSize A reasonable number of iterations in each task
     * @param loopBody A task that is executed once for each iteration.
     * @param combineMethod A task used to combine the local results
     * @return A single value of type <code>T</code>
     * @throws CancelException is thrown if one or more of the tasks cancel themselves,
either by throwing an uncaught
     * exception, or by explicitly invoking <code>cancel</code> on the current
<code>Task</code>.
     * @throws IllegalArgumentException is thrown if the grain size is less than 1.
     * @example <a href="doc-files/ReduceExample.java.html"><code>reduce</code>
example</a>
     */
    public static <T> T reduce(final int start, final int end, final T initialValue,
final int grainSize, final IReduceTask<T> loopBody, final ICombineTask<T> combineMethod)
throws CancelException {
        if (grainSize < 1) {
            throw new IllegalArgumentException("Grain size cannot be less than 1.");
        }
        if (end - start < 1) {
            return initialValue;
        }
        Future<T> f = new Future<T>() {
            public T run() throws CancelException {
                return combineMethod.combine(initialValue, traverseReduce(start, end,
grainSize, loopBody, combineMethod));
            }
        }.start();
        return f.result();
    }

    /**
```

```
     * A parallel version of a standard <code>for</code> loop that support atomic
handling on a shared variable.
     * For this method to work and return the correct vallue of the shared variable,
this shared variable must be manually declared to be returned at the end of
<code>forLoopAtomic</code> as {@link IForAtomicTask} class commands
     * <code>forLoopAtomic</code> is similar to a standard for loop starting at index i
= <code>start</code> and running as long as running as long as i is less than
<code>end</code>.
     * <code>loopBody</code> is called once in each iteration.<br/>
     * <code>forLoop</code> catches any uncaught exception raised inside the tasks. Once
the exception is caught <code>forLoop</code> cancels all
     * running tasks that it has started and throws a <code>CancelException</code>,
containing the original exception. JaCon cancels all
     * tasks but it doesn't just brutally stop them - the user defined code must detect
and handle the cancellation.
     * @param start First index, which is incremented by one until it equals end.
     * @param end Last index, which is not included in the loop. The for loop runs as
long as <code>start</code> is less than <code>end</code>.
     * @param action The reduction action that will performed on shared variable that is
returned at the end of operation
     * @param loopBody The work to be done in each iteration.
     * @throws CancelException is thrown if one or more of the tasks cancel themselves,
either by throwing an uncaught
     * exception, or by explicitly invoking <code>cancel</code> on the current
<code>Task</code>.
     * @example <a href="doc-files/ForExample.java.html"><code>forLoop</code>
example</a>
     */
    public static double forLoopAtomic(final int start, final int end, final String
action, final IForAtomicTask loopBody) throws CancelException {
        int iterations = end - start;
        return parallelEstimate(iterations, action, loopBody);
    }

    // ************** Auxiliary methods *****************

    private static <T> T traverseReduce(final int start, final int end, final int
grainSize, final IReduceTask<T> loopBody, final ICombineTask<T> combineMethod) throws
CancelException {
        int num = end - start;
        T result;
        if (num > grainSize) {
            final int middle = (num / 2) + start;
            Future<T> task = new Future<T>() {
                public T run() throws CancelException {
                    return traverseReduce(start, middle, grainSize, loopBody,
combineMethod);
                }
            }.start();
            result = traverseReduce(middle, end, grainSize, loopBody, combineMethod);
            return combineMethod.combine(result, task.result());
        }
        if (num == 1) {
            return loopBody.reduce(start);
        }
        result = combineMethod.combine(loopBody.reduce(start), loopBody.reduce(start +
1));
        for (int i = start + 2; i < end; i++) {
            result = combineMethod.combine(result, loopBody.reduce(i));
        }
        return result;
    }

    public static double parallelEstimate(final int iterations, final String action,
final IForAtomicTask loopBody) throws CancelException {
        return new ParallelCalculator(iterations, action, loopBody).result();
    }

    /**
     * <code>ParallelCalculator</code> is a static class providing Auxiliary methods for
ParallelAtomic
     */
    public static class ParallelCalculator extends Future<Double> {

        private int iterations;
```

```
        private int missiter;
        private int finalend;
        private int procCount;
        private String action;
        private IForAtomicTask loopBody;

        /************************************************
         * To simplify the code we just do an integer division
         * to split the iteration number. This means that
         * that we can miss up to procCount-1 iterations.
         ************************************************/
        public ParallelCalculator(final int iterations, final String action, final
IForAtomicTask loopBody) {
            this.procCount = Manager.getProcessorCount();
            this.iterations = iterations / procCount;
            this.missiter = iterations % procCount;
            this.finalend = iterations;
            this.action = action;
            this.loopBody = loopBody;
        }

        @Override
        public Double run() throws CancelException {
            // We use the reduce method to distribute work to tasks
            // and combine the results.
            // The grainSize parameter is 1 => one task for each iteration.
            double initialVal;
            if (action.equals("+=") || action.equals("-="))
                initialVal=0.0;
            else
                initialVal=1.0;
            Double sum = reduce(0, procCount, initialVal, 1, new Helper(), new
Reducer(action));
            return sum;
        }

        /************************************************
         * This class is used in the ParallelAtomic.reduce method.
         * Helper performes the parallel execution
         ************************************************/
        private class Helper implements IReduceTask<Double> {

            volatile int start = -(iterations);
            volatile int end = 0;
            public Double reduce(int i) throws CancelException {
                start += iterations;
                end += iterations;
                if (missiter != 0 && end == (finalend - 1)) {
                    end += procCount - 1;
                }
                // we call the sequential method.
                return (Double) estimate(start, end, loopBody);
            }
        }

        public static Object estimate(final int start, final int end, final
IForAtomicTask loopBody) throws CancelException {
            return loopBody.loopBody(start, end);
        }

        /************************************************
         * This class is used in the ParallelAtomic.reduce method.
         * When the parallel execution is done, all returned values
         * are gathered and Reducer execute the reduction operation
         ************************************************/
        private class Reducer implements ICombineTask<Double> {

            private String action;
            public Reducer(final String action) {
                this.action = action;
            }
            public Double combine(Double op1, Double op2) throws CancelException {
                if (action.equals("+=")) {
                    return adder(op1, op2);
                } else if (action.equals("-=")) {
```

```
                    return substractor(op1, op2);
                } else if (action.equals("*=")) {
                    return multiplier(op1, op2);
                } else if (action.equals("/=")) {
                    return divider(op1, op2);
                } else {
                    throw new IllegalArgumentException("Illigal reduction action: " +
action);
                }
            }

            public Double adder(Double op1, Double op2) {
                return op1 + op2;
            }

            public Double substractor(Double op1, Double op2) {
                return op1 - op2;
            }

            public Double multiplier(Double op1, Double op2) {
                return op1 * op2;
            }

            public Double divider(Double op1, Double op2) {
                if (op2 != 0) {
                    return op1 / (double)op2;
                } else {
                    throw new IllegalArgumentException("Devision by zero: " + op1 + "/"
+ op2);
                }
            }
        }
    }
}
```

Παράρτημα Β

Αρχεία ελέγχου της βιβλιοθήκης παράλληλου προγραμματισμού JACON και του γραφικού εργαλείου αυτόματης μετάφρασης κώδικα Semi-AutoParallelizer

Black_White_Game.java

```
/*
 * Black_White_Game.java
 * Ylopoihsh enos paixnidioy me mayra kai aspra pionia poy prosomoiwnei ton pio katw
algorithmo kai kanones paixnidioy :
 * Se ena pinaka 1024x1024 yparxoyn peripoy 25% mayra (B), 25% aspra (W) kai 50% kena
(E) tetragwna.
 * Se kathe gyro toy paixnidioy ola ta aspra (W) kai mayra (B) tetragwna mporoyn na
metakinhthoyn mia thesh.
 * Prwta metakinoyntai ola ta aspra (W) kai sthn synexeia ola ta mayra (B).
 * Ta aspra (W) mporoyn na kinhthoyn MONO DEKsIA kai dedomenoy oti to deksi tetragwnaki
einai keno (E).
 * Ta mayra (B) mporoyn na kinhthoyn MONO KATW kai dedomenoy oti to apo katw tetragwnaki
einai keno (E).
 * Meta tis metakinhseis, oloklhros o pinakas elegxetai an yparxei kapoios ypopinakas
9x9 ston opoion ola ta tetragwnakia na einai kena (E).
 * Se periptwsh poy brethei tetoios pinakas tote to paixnidi termatizei.An den brethei o
kenos 9x9 ypopinakas tote to paixnidi epanalambanetai.
 * To paixnidi teleiwnei meta apo 100 gyroys an den brethei kenos ypopinakas.
 */

//=~15%-20% parallel


import java.util.*;

public class Black_White_Game {

    public static final int EMPTY = 0;
    public static final int BLACK = 1;
    public static final int WHITE = 2;
    public static final int WIN_FIELD = 9;
    public static final int FIELD_SIZE = 1024;
    public static final int MAX_TURNS = 100;
    public static char[][][] field = new char[2][FIELD_SIZE][FIELD_SIZE];
    public static int moving = 1;

    public static void init_field() {
        int rand_num = 0;
        RandomNumbers.seed(281);

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < FIELD_SIZE; j++) {
                for (int k = 0; k < FIELD_SIZE; k++) {
                    field[i][j][k] = EMPTY;
                }
            }
        }
        //Set blocks
        for (int i = 0; i < FIELD_SIZE; i++) {
            for (int j = 0; j < FIELD_SIZE; j++) {
                rand_num = RandomNumbers.nextNumber() % 100;
                if (rand_num < 25) {
                    field[0][i][j] = field[1][i][j] = WHITE;
                } else if (rand_num < 50) {
```

```
                    field[1][i][j] = field[1][i][j] = BLACK;
                }
            }
        }
    }

    public static void print_board() {
        int i;
        int j = 0;

        System.out.print("Printing final field:\n");
        for (i = 0; i < FIELD_SIZE; i++) {
            for (j = 0; j < FIELD_SIZE; j++) {
                switch (field[1][i][j]) {
                    case EMPTY: {
                        System.out.print("X");
                        break;
                    }
                    case BLACK: {
                        System.out.print("B");
                        break;
                    }
                    case WHITE: {
                        System.out.print("W");
                        break;
                    }
                }
                System.out.print(" ");
            }
            System.out.print("\n");
        }
    }

    //WHITE move only right
    public static void white_move() {
        ///@parallel
        for (int i = 0; i < FIELD_SIZE; i++) {
            for (int j = 0; j < FIELD_SIZE; j++) {
                if (field[0][i][j] == WHITE && field[0][i][(j + 1) % FIELD_SIZE] ==
EMPTY) {
                    field[1][i][(j + 1) % FIELD_SIZE] = WHITE;
                    field[1][i][j] = EMPTY;
                }
            }
        }
    }

    //BLACK move only downwards
    public static void black_move() {
        ///@parallel
        for (int i = 0; i < FIELD_SIZE; i++) {
            for (int j = 0; j < FIELD_SIZE; j++) {
                if (field[0][i][j] == BLACK && field[0][(i + 1) % FIELD_SIZE][j] ==
EMPTY) {
                    field[1][(i + 1) % FIELD_SIZE][j] = BLACK;
                    field[1][i][j] = EMPTY;
                }
            }
        }
    }

    //Checking field for an empty subtable
    public static void check_field() {

        int[] total = new int[3];
        int space=FIELD_SIZE - WIN_FIELD;

        for (int i = 0; i <= space; i++) {
            for (int j = 0; j <= space; j++) {
                for (int m = 0; m < 3; m++) {
                    total[m] = 0;
                }

                for (int k = i; k < i + WIN_FIELD; k++) {
                    for (int l = j; l < j + WIN_FIELD; l++) {
```

```
                        total[(int) (field[1][k][l])]++;
                    }
                }

                if (total[EMPTY] >= WIN_FIELD * WIN_FIELD) {
                    //gettimeofday(&t_fin,NULL);
                    System.out.printf("Empty table coordinates: i = %d, j = %d\n", i,
j);
                    moving = 0;
                    i = FIELD_SIZE - WIN_FIELD;
                    break;
                }
            }
        }
    }

    public static void copycells(){
        //copy cells contents to use it for next white move
            ///@parallel
            for (int ii = 0; ii < FIELD_SIZE; ii++) {
                for (int jj = 0; jj < FIELD_SIZE; jj++) {
                    field[0][ii][jj] = field[1][ii][jj];
                }
            }
    }

    public static void main(String[] args) {
        int turns = 0;
        turns = 0;
        moving = 1;

        init_field();

        while ((turns < MAX_TURNS) && moving != 0) {
            copycells();
            white_move();

            copycells();
            black_move();

            check_field();

            turns++;
        }

        //print_board();
        System.out.printf("\nGame over after %d turns\n\n", turns);
    }
}

final class RandomNumbers {

    private static Random r;

    static int nextNumber() {
        if (r == null) {
            seed();
        }

        return r.nextInt();
    }

    static int nextNumber(int ceiling) {
        if (r == null) {
            seed();
        }

        return r.nextInt(ceiling);
    }

    static void seed() {
        r = new Random();
    }

    static void seed(int seed) {
```

```
        r = new Random(seed);
    }

    public static void main(String[] args) {

    }
}
```

Black_White_Game_mult.java

```
/*
 * Black_White_Game.java
 * Ylopoihsh enos paixnidioy me mayra kai aspra pionia poy prosomoiwnei ton pio katw
algorithmo kai kanones paixnidioy :
 * Se ena pinaka 1024x1024 yparxoyn peripoy 25% mayra (B), 25% aspra (W) kai 50% kena
(E) tetragwna.
 * Se kathe gyro toy paixnidioy ola ta aspra (W) kai mayra (B) tetragwna mporoyn na
metakinhthoyn mia thesh.
 * Prwta metakinoyntai ola ta aspra (W) kai sthn synexeia ola ta mayra (B).
 * Ta aspra (W) mporoyn na kinhthoyn MONO DEKsIA kai dedomenoy oti to deksi tetragwnaki
einai keno (E).
 * Ta mayra (B) mporoyn na kinhthoyn MONO KATW kai dedomenoy oti to apo katw tetragwnaki
einai keno (E).
 * Meta tis metakinhseis, oloklhros o pinakas elegxetai an yparxei kapoios ypopinakas
9x9 ston opoion ola ta tetragwnakia na einai kena (E).
 * Se periptwsh poy brethei tetoios pinakas tote to paixnidi termatizei.An den brethei o
kenos 9x9 ypopinakas tote to paixnidi epanalambanetai.
 * To paixnidi teleiwnei meta apo 100 gyroys an den brethei kenos ypopinakas.
 */

//=~15%-20% parallel

import java.util. * ;
import com.cs.ucy.ac.cy.jacon. * ;

public class Black_White_Game_mult {

  public static final int EMPTY = 0;
  public static final int BLACK = 1;
  public static final int WHITE = 2;
  public static final int WIN_FIELD = 9;
  public static final int FIELD_SIZE = 1024;
  public static final int MAX_TURNS = 100;
  public static char[][][] field = new char[2][FIELD_SIZE][FIELD_SIZE];
  public static int moving = 1;

  public static void init_field() {
    int rand_num = 0;
    RandomNumbers.seed(281);

    for (int i = 0; i < 2; i++) {
      for (int j = 0; j < FIELD_SIZE; j++) {
        for (int k = 0; k < FIELD_SIZE; k++) {
          field[i][j][k] = EMPTY;
        }
      }
    }
    //Set blocks
    for (int i = 0; i < FIELD_SIZE; i++) {
      for (int j = 0; j < FIELD_SIZE; j++) {
        rand_num = RandomNumbers.nextNumber() % 100;
        if (rand_num < 25) {
          field[0][i][j] = field[1][i][j] = WHITE;
        } else if (rand_num < 50) {
          field[1][i][j] = field[1][i][j] = BLACK;
        }
      }
    }
```

```
}

public static void print_board() {
  int i;
  int j = 0;

  System.out.print("Printing final field:\n");
  for (i = 0; i < FIELD_SIZE; i++) {
    for (j = 0; j < FIELD_SIZE; j++) {
      switch (field[1][i][j]) {
      case EMPTY:
        {
          System.out.print("X");
          break;
        }
      case BLACK:
        {
          System.out.print("B");
          break;
        }
      case WHITE:
        {
          System.out.print("W");
          break;
        }
      }
      System.out.print(" ");
    }
    System.out.print("\n");
  }
}

//WHITE move only right
public static void white_move() {

  //@parallel

  /* Here comes the auto-created parallel code for For statement block!! */

  try {
    Parallel.forLoop(0, "<", FIELD_SIZE, "++", new IForTask() {
      public void loopBody(int i) throws CancelException
      //Here comes the untouched block of parallized for-loop
      {
        for (int j = 0; j < FIELD_SIZE; j++) {
          if (field[0][i][j] == WHITE && field[0][i][(j + 1) % FIELD_SIZE] == EMPTY) {
            field[1][i][(j + 1) % FIELD_SIZE] = WHITE;
            field[1][i][j] = EMPTY;
          }
        }
      }
      //End of untouched block of parallized for-loop
    });
  } catch (CancelException e) {
    e.printStackTrace();
  }

  /* End of auto-created parallel code!! */

}

//BLACK move only downwards
public static void black_move() {

  //@parallel

  /* Here comes the auto-created parallel code for For statement block!! */

  try {
    Parallel.forLoop(0, "<", FIELD_SIZE, "++", new IForTask() {
      public void loopBody(int i) throws CancelException
      //Here comes the untouched block of parallized for-loop
      {
        for (int j = 0; j < FIELD_SIZE; j++) {
          if (field[0][i][j] == BLACK && field[0][(i + 1) % FIELD_SIZE][j] == EMPTY) {
```

```
            field[1][(i + 1) % FIELD_SIZE][j] = BLACK;
            field[1][i][j] = EMPTY;
          }
        }
      }
      //End of untouched block of parallized for-loop
    });
  } catch (CancelException e) {
    e.printStackTrace();
  }

  /* End of auto-created parallel code!! */

}

//Checking field for an empty subtable
public static void check_field() {

  int[] total = new int[3];
  int space = FIELD_SIZE - WIN_FIELD;

  for (int i = 0; i <= space; i++) {
    for (int j = 0; j <= space; j++) {
      for (int m = 0; m < 3; m++) {
        total[m] = 0;
      }

      for (int k = i; k < i + WIN_FIELD; k++) {
        for (int l = j; l < j + WIN_FIELD; l++) {
          total[(int)(field[1][k][l])]++;
        }
      }

      if (total[EMPTY] >= WIN_FIELD * WIN_FIELD) {
        //gettimeofday(&t_fin,NULL);
        System.out.printf("Empty table coordinates: i = %d, j = %d\n", i, j);
        moving = 0;
        i = FIELD_SIZE - WIN_FIELD;
        break;
      }
    }
  }
}

public static void copycells() {
  //copy cells contents to use it for next white move

  //@parallel

  /* Here comes the auto-created parallel code for For statement block!! */

  try {
    Parallel.forLoop(0, "<", FIELD_SIZE, "++", new IForTask() {
      public void loopBody(int ii) throws CancelException
      //Here comes the untouched block of parallized for-loop
      {
        for (int jj = 0; jj < FIELD_SIZE; jj++) {
          field[0][ii][jj] = field[1][ii][jj];
        }
      }
      //End of untouched block of parallized for-loop
    });
  } catch (CancelException e) {
    e.printStackTrace();
  }

  /* End of auto-created parallel code!! */

}

public static void main(String[] args) {
  int turns = 0;
  turns = 0;
  moving = 1;
```

```
    init_field();

    while ((turns < MAX_TURNS) && moving != 0) {
      copycells();
      white_move();

      copycells();
      black_move();

      check_field();

      turns++;
    }

    //print_board();
    System.out.printf("\nGame over after %d turns\n\n", turns);
  }
}



final class RandomNumbers {

  private static Random r;

  static int nextNumber() {
    if (r == null) {
      seed();
    }

    return r.nextInt();
  }

  static int nextNumber(int ceiling) {
    if (r == null) {
      seed();
    }

    return r.nextInt(ceiling);
  }

  static void seed() {
    r = new Random();
  }

  static void seed(int seed) {
    r = new Random(seed);
  }

  public static void main(String[] args) {

  }
}
```

ComputePi.java

```
/*
 * ComputePi.java
 * Efarmogi ypologismoy tis mathimatikis statheras p pou einai enas pragmatikos arithmos
kai mporei na oristei ws o logos tou mikous tis perifereias enos kyklou pros ti diametro
tou stin Efkleideia gewmetria.
 */

package all;


public class ComputePi {
```

```
    static int N = 300000000;

    public static void computepi() {
        double pi = 0;

        //compute pi using pi=4(-1^k/(2k+1)) , 0<k<infinite
        ///@parallelatimic(+=,double pi)
        for (int i = 0; i < N; i++) {
            pi += ((4 * Math.pow(-1, i)) / (double)((2 * i) + 1));
        }

        System.out.printf("%.8f ..\n",pi);
    }

    public static void main(String[] args) {
        computepi();
    }
}
```

ComputePi_mult.java

```
/*
 * ComputePi.java
 * Efarmogi ypologismoy tis mathimatikis statheras p pou einai enas pragmatikos arithmos
kai mporei na oristei ws o logos tou mikous tis perifereias enos kyklou pros ti diametro
tou stin Efkleideia gewmetria.
 */

package all;

import com.cs.ucy.ac.cy.jacon.*;

public class ComputePi_mult {

    static int N = 300000000;

    public static void computepi_mult() {
        double pi = 0;

        //compute pi using pi=4(-1^k/(2k+1)) , 0<k<infinite

        ///@parallelatimic(+=,double pi)

        /* Here comes the auto-created parallel code for Atomic For operation!! */

        try {
            pi = (double) ParallelAtomic.forLoopAtomic(0, N, "+=", new IForAtomicTask()
{

                public Double loopBody(int start, int end) throws CancelException {
                    double pi = 0;
                    for (int i = start; i < end; i++)
                    //Here comes the untouched block of parallized for-loop
                    {
                        pi += ((4 * Math.pow(-1, i)) / (double)((2 * i) + 1));
                    }
                    //End of untouched block of parallized for-loop
                    return pi;
                }
            });
        } catch (CancelException e) {
            e.printStackTrace();
        }
        System.out.printf("%.8f ..\n",pi);
    }

    public static void main(String[] args) {
        computepi_mult();
```

```
    }
}
```

Count3s.java

```
/*
 * Count3s.java
 * Programma poy katametra tis emfaniseis toy arithmoy 3 kanontas prospelash se ena
monodiastato pinaka poy periexei akeraioys arithmoys apo to 0 mexri to 4 katanemhmenoys
se tyxaia seira mesa ston pinaka.
 */

package all;

import java.util.*;

public class Count3s {

    private static final int ARRAYLENGTH = 10000000;
    private static final int MAXRANGE = 4;
    private static final int EXECTIMES = 100;
    private static final Random random = new Random(19580427);
    private static int count = 0;
    private static int[] array;

    public static void count3s() {
        array = new int[ARRAYLENGTH];

        //initialize the elements in the array
        for (int i = 0; i < array.length; i++) {
            array[i] = random.nextInt(MAXRANGE);
        }

        System.out.println("Executing Count 3s");

        //count the number of threes in array[]
        //@parallelatomic(+=,int count)
        for (int i = 0; i < array.length; i++) {
            if (array[i] == 3) {
                //some stupid intensive work
                for (int j = 0; j < EXECTIMES; j++) {
                    array[i] = array[i] * array[i];
                    array[i] = (int) Math.sqrt(array[i]);
                }
                count++;
            }
        }

        //Number of 3s = 2498713
        System.out.println(count + " threes counted");
    }

    public static void main(String[] args) {
        count3s();
    }
}
```

Count3s_mult.java

```
/*
 * Count3s.java
```

```
 * Programma poy katametra tis emfaniseis toy arithmoy 3 kanontas prospelash se ena
monodiastato pinaka poy periexei akeraioys arithmoys apo to 0 mexri to 4 katanemhmenoys
se tyxaia seira mesa ston pinaka.
 */

package all;

import com.cs.ucy.ac.cy.jacon.*;
import java.util.Random;

public class Count3s_mult {
  private static final int ARRAYLENGTH = 10000000;
  private static final int MAXRANGE = 4;
  private static final int EXECTIMES = 100;
  private static final Random random = new Random(19580427);
  private static int count = 0;
  private static int[] array;

  public static void count3s() {

    array = new int[ARRAYLENGTH];

    //initialize the elements in the array
    for (int i = 0; i < array.length; i++) {
      array[i] = random.nextInt(MAXRANGE);
    }

    System.out.println("Executing Count 3s");

    //count the number of threes in array[]

    //@parallelatomic(+=,int count)

    /* Here comes the auto-created parallel code for Atomic For operation!! */

    try {
      count = (int) ParallelAtomic.forLoopAtomic(0, array.length, "+=", new
IForAtomicTask() {
        public Double loopBody(int start, int end) throws CancelException {
          double count = 0;
          for (int i = start; i < end; i++)
          //Here comes the untouched block of parallized for-loop
          {
            if (array[i] == 3) {
              //some stupid intensive work
              for (int j = 0; j < EXECTIMES; j++) {
                array[i] = array[i] * array[i];
                array[i] = (int) Math.sqrt(array[i]);
              }
              count++;
            }
          }
          //End of untouched block of parallized for-loop
          return (Double) count;
        }
      });
    } catch (CancelException e) {
      e.printStackTrace();
    }

    /* End of auto-created parallel code!! */


    //Number of 3s = 2498713
    System.out.println(count + " threes counted");
  }

  public static void main(String[] args) {
    count3s();
  }
}
```

DCT2Algorithm.java

```
/*
 * DCT2Algorithm.java
 * H diakrith metatroph synhmitonoy (discrete cosine transform - DCT) bohtha ston
diaxwrismo mias eikonas se diafora merh (h fasmatikes ypozwnes) diaforetikhs shmasias
(oson afora thn poiothta ths eikonas).
 * O algorithmos DCT einai kati paromoio me ton algorithmo diakritoy metasxhmatismoy
Fourier (discrete Fourier transform) poy xrhsimopoieitai gia na metasxhmatizei ena shma
h mia eikona apo th xwrikh perioxh (spatial domain) sto pedio syxnothtas (frequency
domain) ths.
 * http://www.mathworks.com/help/toolbox/images/ref/dct2.html
 */

package all;

import java.text.DecimalFormat;
import java.util.Random;

public class DCT2Algorithm {

    static double[][] A;
    static double[][] B;
    static DecimalFormat twoDForm = new DecimalFormat("#.##");

    public static void dctprocess() {

        final int M = 120;
        final int N = 120;
        A = new double[M][N];
        B = new double[M][N];
        Random random = new Random(19580427);

        //init array A
        for (int p = 0; p < M; p++) {
            for (int q = 0; q < N; q++) {
                A[p][q] = random.nextInt(255);
            }
        }

//        System.out.println("Array A: \n");
//        ptrArray(A);

        //execute dct algorithm
        ///@parallel
        for (int p = 0; p < M; p++) {
            for (int q = 0; q < N; q++) {
                double value = 0;
                double ap = 0;
                double aq = 0;
                if (p == 0) {
                    ap = 1 / (Math.sqrt(M));
                } else {
                    double temp = 2 / (double) M;
                    ap = Math.sqrt(temp);
                }
                if (q == 0) {
                    aq = 1 / (Math.sqrt(N));
                } else {
                    double temp = 2 / (double) N;
                    aq = Math.sqrt(temp);
                }
                for (int m = 0; m < M; m++) {
                    for (int n = 0; n < N; n++) {
                        value += A[m][n] * Math.cos((Math.PI * (2 * m + 1) * p) /
(double) (2 * M)) * Math.cos((Math.PI * (2 * n + 1) * q) / (double) (2 * N));
                    }
                }
                //System.out.println("ap=" + ap + ", aq=" + aq + ", value=" + value);
                B[p][q] = (ap * aq * value);
            }
        }
```

```
//        System.out.println("\n\nArray B: \n");
//        ptrArray(B);
    }

    public static void ptrArray(double a[][]) {
        for (int p = 0; p < a.length; p++) {
            for (int q = 0; q < a[p].length; q++) {
                System.out.print(Double.valueOf(twoDForm.format(a[p][q])) + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        dctprocess();
    }
}
```

DCT2Algorithm_mult.java

```
/*
 * DCT2Algorithm.java
 * H diakrith metatroph synhmitonoy (discrete cosine transform - DCT) bohtha ston
diaxwrismo mias eikonas se diafora merh (h fasmatikes ypozwnes) diaforetikhs shmasias
(oson afora thn poiothta ths eikonas).
 * O algorithmos DCT einai kati paromoio me ton algorithmo diakritoy metasxhmatismoy
Fourier (discrete Fourier transform) poy xrhsimopoieitai gia na metasxhmatizei ena shma
h mia eikona apo th xwrikh perioxh (spatial domain) sto pedio syxnothtas (frequency
domain) ths.
 * http://www.mathworks.com/help/toolbox/images/ref/dct2.html
 */

package all;

import java.text.DecimalFormat;
import java.util.Random;


import com.cs.ucy.ac.cy.jacon.*;

public class DCT2Algorithm_mult {

    static double[][] A;
    static double[][] B;
    static DecimalFormat twoDForm = new DecimalFormat("#.##");

    public static void dctprocess() {

        final int M = 120;
        final int N = 120;
        A = new double[M][N];
        B = new double[M][N];
        Random random = new Random(19580427);

        //init array A
        for (int p = 0; p < M; p++) {
            for (int q = 0; q < N; q++) {
                A[p][q] = random.nextInt(255);
            }
        }

//        System.out.println("Array A: \n");
//        ptrArray(A);

        //execute dct algorithm

        //@parallel
```

```
        /* Here comes the auto-created parallel code for For statement block!! */

        try {
            Parallel.forLoop(0, "<", M, "++", new IForTask() {

                public void loopBody(int p) throws CancelException //Here comes the
untouched block of parallized for-loop
                {
                    for (int q = 0; q < N; q++) {
                        double value = 0;
                        double ap = 0;
                        double aq = 0;
                        if (p == 0) {
                            ap = 1 / (Math.sqrt(M));
                        } else {
                            double temp = 2 / (double) M;
                            ap = Math.sqrt(temp);
                        }
                        if (q == 0) {
                            aq = 1 / (Math.sqrt(N));
                        } else {
                            double temp = 2 / (double) N;
                            aq = Math.sqrt(temp);
                        }
                        for (int m = 0; m < M; m++) {
                            for (int n = 0; n < N; n++) {
                                value += A[m][n] * Math.cos((Math.PI * (2 * m + 1) * p)
/ (double) (2 * M)) * Math.cos((Math.PI * (2 * n + 1) * q) / (double) (2 * N));
                            }
                        }
                        //System.out.println("ap=" + ap + ", aq=" + aq + ", value=" +
value);
                        B[p][q] = (ap * aq * value);
                    }
                }
                //End of untouched block of parallized for-loop
            });
        } catch (CancelException e) {
            e.printStackTrace();
        }

        /* End of auto-created parallel code!! */

//        System.out.println("\n\nArray B: \n");
//        ptrArray(B);
    }

    public static void ptrArray(double a[][]) {
        for (int p = 0; p < a.length; p++) {
            for (int q = 0; q < a[p].length; q++) {
                System.out.print(Double.valueOf(twoDForm.format(a[p][q])) + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        dctprocess();
    }
}
```

GameOfLife.java

```
/*
 * GameOfLife.java
 * To Game of Life h alliws Conway's Game einai ena kypseloeides aytomato paixnidi poy
epinohthhke apo to bretaniko mathhmatiko John Horton Conway to 1970. To Game of Life
einai ena paixnidi poy den xreiazetai symmetoxh paiktwn, poy shmainei oti h ekseliksh
```

```
toy kathorizetai apo thn arxikh katastash kai den apaitei kamia peraiterw eisagwgh
stoixeiwn.
 * Ton kosmo toy paixnidioy apotelei ena apeiro disdiastato orthogwnio plegma twn
tetragwnikwn keliwn, kathe ena apo ta opoia mporei na brisketai se mia apo tis pithanes
katastaseis, zwntanos h nekros.
 * Kathe keli allhlepidra me oktw geitones toy, ta opoia einai ta kelia poy einai amesa
orizontia, katheta, h diagwnia dipla toy.
 * Oi kanones panw stoys opoioys sthrizetai h leitoyrgia toy paixnidioy kai isxyoyn gia
kathe bhma einai oi ekshs :
 * 1.  Opoiodhpote zwntano keli me ligoteroys apo dyo zwntanoys geitones pethainei
(under-population).
 * 2.  Opoiodhpote zwntano keli me perissoteroys apo treis zwntanoys geitones pethainei,
(overcrowding).
 * 3.  Opoiodhpote zwntano keli me dyo h treis zwntanoys geitones zei mexri thn epomenh
genea.
 * 4.  Opoiodhpote nekro keli me akribws treis zwntanoys geitones ginetai zwntano
(reproduction).
 * H prwth genia poy dhmioyrgeitai me efarmozontas toys pio panw kanones taytoxrona se
kathe keli.
 * Oi kanones synexizoyn na efarmozontai epaneilhmmena gia na dhmioyrghsoyn tis
peraiterw genees oi opoies sthn oysia einai apeires.
 */

package all;

import java.util.Random;

public class GameOfLife {

    static Random random = new Random(19580427);
    static int loops = 1000;
    static int cells = 1024;
    static int generation=0, population=0, newborncell=0;
    static int w, h;
    static boolean life[][] = new boolean[cells][cells];
    static boolean next[][] = new boolean[cells][cells];
    static boolean remain[][] = new boolean[cells][cells];
    static boolean startflag = false;

    public static void main(String[] args) {
        randomstart();
        runGame();
        printBoard();
    }

    static public void runGame() {
        while (loops != generation) {
            population = 0;
            ///@parallelatomic(+=, int population)
            for (int i = 0; i < cells; i++) {
                for (int j = 0; j < cells; j++) {
                    int a = 0;
                    if (life[(i + cells - 1) % cells][j]) {
                        a++; //left
                    }
                    if (life[(i + 1) % cells][j]) {
                        a++; //right
                    }
                    if (life[i][(j + cells - 1) % cells]) {
                        a++; //up
                    }
                    if (life[i][(j + 1) % cells]) {
                        a++; //down
                    }
                    if (life[(i + cells - 1) % cells][(j + cells - 1) % cells]) {
                        a++; //upper left
                    }
                    if (life[(i + 1) % cells][(j + cells - 1) % cells]) {
                        a++; //upper right
                    }
                    if (life[(i + cells - 1) % cells][(j + 1) % cells]) {
                        a++; //downer left
                    }
                    if (life[(i + 1) % cells][(j + 1) % cells]) {
                        a++; //downer right
```

```
                }
                if (life[i][j]) {
                    if (a == 2 || a == 3) {
                        next[i][j] = true;
                        population++;
                    } else {
                        next[i][j] = false;
                    }
                } else {
                    if (a == 3) {
                        next[i][j] = true;
                        population++;
                    } else {
                        next[i][j] = false;
                    }
                }
            }
        }
        judgeremain();
        nextgeneration();
        generation++;
    }
}

static public void judgeremain() {
    newborncell = 0;
    ///@parallelatomic(+=, int newborncell)
    for (int i = 0; i < cells; i++) {
        for (int j = 0; j < cells; j++) {
            if (life[i][j] == next[i][j]) {
                remain[i][j] = true;
            } else {
                remain[i][j] = false;
                if (next[i][j]) {
                    newborncell++;
                }
            }
        }
    }
}

static public void nextgeneration() {
    ///@parallel
    for (int i = 0; i < cells; i++) {
        for (int j = 0; j < cells; j++) {
            life[i][j] = next[i][j];
        }
    }
}

static public void clearcell() {
    ///@parallel
    for (int i = 0; i < cells; i++) {
        for (int j = 0; j < cells; j++) {
            life[i][j] = false;
        }
    }
    generation = 0;
    population = 0;
    newborncell = 0;
}

static public void randomstart() {
    clearcell();
    for (int i = 0; i < cells; i++) {
        for (int j = 0; j < cells; j++) {
            if (random.nextInt() % 2 == 0) {
                life[i][j] = true;
            } else {
                life[i][j] = false;
            }
        }
    }
}
```

```
    static public void printBoard() {
        for (int i = 0; i < cells; i++) {
            for (int j = 0; j < cells; j++) {
                if (remain[i][j]) {
                    System.out.print(" ");
                } else {
                    System.out.print("X");
                }

            }
            System.out.println();
        }
    }
}
```

GameOfLife_mult.java

```
/*
 * GameOfLife.java
 * To Game of Life h alliws Conway's Game einai ena kypseloeides aytomato paixnidi poy
epinohthhke apo to bretaniko mathhmatiko John Horton Conway to 1970. To Game of Life
einai ena paixnidi poy den xreiazetai symmetoxh paiktwn, poy shmainei oti h ekseliksh
toy kathorizetai apo thn arxikh katastash kai den apaitei kamia peraiterw eisagwgh
stoixeiwn.
 * Ton kosmo toy paixnidioy apotelei ena apeiro disdiastato orthogwnio plegma twn
tetragwnikwn keliwn, kathe ena apo ta opoia mporei na brisketai se mia apo tis pithanes
katastaseis, zwntanos h nekros.
 * Kathe keli allhlepidra me oktw geitones toy, ta opoia einai ta kelia poy einai amesa
orizontia, katheta, h diagwnia dipla toy.
 * Oi kanones panw stoys opoioys sthrizetai h leitoyrgia toy paixnidioy kai isxyoyn gia
kathe bhma einai oi ekshs :
 * 1.  Opoiodhpote zwntano keli me ligoteroys apo dyo zwntanoys geitones pethainei
(under-population).
 * 2.  Opoiodhpote zwntano keli me perissoteroys apo treis zwntanoys geitones pethainei,
(overcrowding).
 * 3.  Opoiodhpote zwntano keli me dyo h treis zwntanoys geitones zei mexri thn epomenh
genea.
 * 4.  Opoiodhpote nekro keli me akribws treis zwntanoys geitones ginetai zwntano
(reproduction).
 * H prwth genia poy dhmioyrgeitai me efarmozontas toys pio panw kanones taytoxrona se
kathe keli.
 * Oi kanones synexizoyn na efarmozontai epaneilhmmena gia na dhmioyrghsoyn tis
peraiterw genees oi opoies sthn oysia einai apeires.
 */

package all;

import java.util.Random;

import com.cs.ucy.ac.cy.jacon. * ;

public class GameOfLife_mult {

  static Random random = new Random(19580427);
  static int loops = 1000;
  static int cells = 1024;
  static int generation = 0, population = 0, newborncell = 0;
  static int w, h;
  static boolean life[][] = new boolean[cells][cells];
  static boolean next[][] = new boolean[cells][cells];
  static boolean remain[][] = new boolean[cells][cells];
  static boolean startflag = false;

  public static void main(String[] args) {
    randomstart();
    runGame();
    printBoard();
  }
```

```
  static public void runGame() {
    while (loops != generation) {
      population = 0;

      //@parallelatomic(+=,int population)

      /* Here comes the auto-created parallel code for Atomic For operation!! */

      try {
        population = (int) ParallelAtomic.forLoopAtomic(0, cells, "+=", new
IForAtomicTask() {
          public Double loopBody(int start, int end) throws CancelException {
            double population = 0;
            for (int i = start; i < end; i++)
            //Here comes the untouched block of parallized for-loop
            {
              for (int j = 0; j < cells; j++) {
                int a = 0;
                if (life[(i + cells - 1) % cells][j]) {
                  a++; //left
                }
                if (life[(i + 1) % cells][j]) {
                  a++; //right
                }
                if (life[i][(j + cells - 1) % cells]) {
                  a++; //up
                }
                if (life[i][(j + 1) % cells]) {
                  a++; //down
                }
                if (life[(i + cells - 1) % cells][(j + cells - 1) % cells]) {
                  a++; //upper left
                }
                if (life[(i + 1) % cells][(j + cells - 1) % cells]) {
                  a++; //upper right
                }
                if (life[(i + cells - 1) % cells][(j + 1) % cells]) {
                  a++; //downer left
                }
                if (life[(i + 1) % cells][(j + 1) % cells]) {
                  a++; //downer right
                }
                if (life[i][j]) {
                  if (a == 2 || a == 3) {
                    next[i][j] = true;
                    population++;
                  } else {
                    next[i][j] = false;
                  }
                } else {
                  if (a == 3) {
                    next[i][j] = true;
                    population++;
                  } else {
                    next[i][j] = false;
                  }
                }
              }
            }
            //End of untouched block of parallized for-loop
            return (Double) population;
          }
        });
      } catch (CancelException e) {
        e.printStackTrace();
      }

      /* End of auto-created parallel code!! */

      judgeremain();
      nextgeneration();
      generation++;
    }
  }
```

```
  static public void judgeremain() {
    newborncell = 0;

    //@parallelatomic(+=,int newborncell)

    /* Here comes the auto-created parallel code for Atomic For operation!! */

    try {
      newborncell = (int) ParallelAtomic.forLoopAtomic(0, cells, "+=", new
IForAtomicTask() {
        public Double loopBody(int start, int end) throws CancelException {
          double newborncell = 0;
          for (int i = start; i < end; i++)
          //Here comes the untouched block of parallized for-loop
          {
            for (int j = 0; j < cells; j++) {
              if (life[i][j] == next[i][j]) {
                remain[i][j] = true;
              } else {
                remain[i][j] = false;
                if (next[i][j]) {
                  newborncell++;
                }
              }
            }
          }
          //End of untouched block of parallized for-loop
          return (Double) newborncell;
        }
      });
    } catch (CancelException e) {
      e.printStackTrace();
    }

    /* End of auto-created parallel code!! */

  }

  static public void nextgeneration() {

    //@parallel

    /* Here comes the auto-created parallel code for For statement block!! */

    try {
      Parallel.forLoop(0, "<", cells, "++", new IForTask() {
        public void loopBody(int i) throws CancelException
        //Here comes the untouched block of parallized for-loop
        {
          for (int j = 0; j < cells; j++) {
            life[i][j] = next[i][j];
          }
        }
        //End of untouched block of parallized for-loop
      });
    } catch (CancelException e) {
      e.printStackTrace();
    }

    /* End of auto-created parallel code!! */

  }

  static public void clearcell() {

    //@parallel

    /* Here comes the auto-created parallel code for For statement block!! */

    try {
      Parallel.forLoop(0, "<", cells, "++", new IForTask() {
        public void loopBody(int i) throws CancelException
        //Here comes the untouched block of parallized for-loop
        {
```

```
          for (int j = 0; j < cells; j++) {
            life[i][j] = false;
          }
        }
        //End of untouched block of parallized for-loop
      });
    } catch (CancelException e) {
      e.printStackTrace();
    }

    /* End of auto-created parallel code!! */

    generation = 0;
    population = 0;
    newborncell = 0;
  }

  static public void randomstart() {
    clearcell();
    for (int i = 0; i < cells; i++) {
      for (int j = 0; j < cells; j++) {
        if (random.nextInt() % 2 == 0) {
          life[i][j] = true;
        } else {
          life[i][j] = false;
        }
      }
    }
  }

  static public void printBoard() {
    for (int i = 0; i < cells; i++) {
      for (int j = 0; j < cells; j++) {
        if (remain[i][j]) {
          System.out.print(" ");
        } else {
          System.out.print("X");
        }

      }
      System.out.println();
    }
  }
}
```

Mandelbrot.java

```
/*
 * Mandelbrot.java
 * To synolo Mandelbrot, einai ena mathhmatiko synolo shmeiwn sto migadiko epipedo, to
orio toy opoioy apotelei ena fractal (ena katakermatismeno gewmetriko sxhma poy mporei
na xwristei se tmhmata, kathena apo ta opoia (toylaxiston kata proseggish) apotelei
antigrafo toy synoloy toy se meiwmeno omws megethos).
 * To synolo Mandelbrot apotelei to synolo olwn twn migadikwn arithmwn z opoy h
akoloythia toys poy orizetai apo thn epanalhpsh paramenei oriothethmenh (bounded). Ayto
shmainei oti yparxei enas arithmos B etsi wste h apolyth timh olwn twn timwn z(n) poy
epanalambanontai den ksepernoyn pote thn timh toy B.
 * H oriothethmenh akoloythia mporei na exei h oxi ena orio. Gia paradeigma, an z = 0
tote z (n) = 0 gia kathe n, etsi wste to orio ths akoloythias poy fainetai pio katw (1)
na einai mhdenikh.
 * Apo thn allh pleyra, an z = i (i einai h fantastikh monada), tote h akoloythia
talanteyetai metaksy i kai i-1, opote eksakoloythei na oriotheteitai alla den sygklinei
se ena orio.
 *  z(0) = z, zn+1 = zn2 + c, n=0,1,2, ...   (1)
 * Periorizontas thn metablhth n sthn formoyla (1) sto pedio twn pragmatikwn arithmwn
metaksy -2 ews 0 pairnoyme mia pio katanohth, megalyterhs omws periplokothtas,  eikona
se sxesh me thn aplh, kenh, eikona poy tha pairname an oi times toy n orizontan sto
pedio apo 0 ews apeiro.
```

```
 */

package all;

import java.awt.Color;

public class Mandelbrot {

    static final int width = 700;
    static final int height = 700;
    static Color[][] colors = new Color[width][height];
    static final double xMin = -1.9;
    static final double xMax = 0.7;
    static final double yMin = -1.3;
    static final double yMax = 1.3;
    static final int iterations = 15000;

    // Draws the Mandelbrot set using a single thread.
    // The time spent calculating all points is returned.
    // The image-drawing time is left out, because this can only
    // be done in a single thread.
    public static void MandelbrotDraw() {

        final double mandWidth = xMax - xMin;
        final double mandHeight = yMax - yMin;

        ///@parallel
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                double xCoord = xMin + ((mandWidth / width) * x);
                double yCoord = yMin + ((mandHeight / height) * y);

                colors[x][y] =
MandelbrotMethods.mapToColor(MandelbrotMethods.solvePoint(xCoord, yCoord, iterations));
            }
        }

    }

    public static void main(String[] args) {
        MandelbrotDraw();
    }
}

class MandelbrotMethods {
    // Solves a single point in the Mandelbrot set
    // and returns the number of iterations spent.
    // The iterations parameter specifies the maximum
    // number of iterations.

    public static int solvePoint(double x, double y, int iterations) {
        double r = 0.0, s = 0.0;
        double next_r, next_s;
        int iteration = 0;

        while ((r * r + s * s) <= 4.0) {
            next_r = r * r - s * s + x;
            next_s = 2 * r * s + y;
            r = next_r;
            s = next_s;
            if (++iteration == iterations) {
                break;
            }
        }

        return iteration;
    }

    // Maps an iteration number to a color.
    // The colors can be tweaked by fiddling
    // with this method.
    public static Color mapToColor(int val) {
        int r = (val * 12) % 256;
        int g = (val * 16) % 256;
        int b = (val * 5) % 256;
```

```
        return new Color(r, g, b);
    }
}
```

Mandelbrot_mult.java

```
/*
 * Mandelbrot.java
 * To synolo Mandelbrot, einai ena mathhmatiko synolo shmeiwn sto migadiko epipedo, to
orio toy opoioy apotelei ena fractal (ena katakermatismeno gewmetriko sxhma poy mporei
na xwristei se tmhmata, kathena apo ta opoia (toylaxiston kata proseggish) apotelei
antigrafo toy synoloy toy se meiwmeno omws megethos).
 * To synolo Mandelbrot apotelei to synolo olwn twn migadikwn arithmwn z opoy h
akoloythia toys poy orizetai apo thn epanalhpsh paramenei oriothethmenh (bounded). Ayto
shmainei oti yparxei enas arithmos B etsi wste h apolyth timh olwn twn timwn z(n) poy
epanalambanontai den ksepernoyn pote thn timh toy B.
 * H oriothethmenh akoloythia mporei na exei h oxi ena orio. Gia paradeigma, an z = 0
tote z (n) = 0 gia kathe n, etsi wste to orio ths akoloythias poy fainetai pio katw (1)
na einai mhdenikh.
 * Apo thn allh pleyra, an z = i (i einai h fantastikh monada), tote h akoloythia
talanteyetai metaksy i kai i-1, opote eksakoloythei na oriotheteitai alla den sygklinei
se ena orio.
 *  z(0) = z, zn+1 = zn2 + c, n=0,1,2, ...   (1)
 * Periorizontas thn metablhth n sthn formoyla (1) sto pedio twn pragmatikwn arithmwn
metaksy -2 ews 0 pairnoyme mia pio katanohth, megalyterhs omws periplokothtas,  eikona
se sxesh me thn aplh, kenh, eikona poy tha pairname an oi times toy n orizontan sto
pedio apo 0 ews apeiro.
 */

package all;

import java.awt.Color;

import com.cs.ucy.ac.cy.jacon.*;

public class Mandelbrot_mult {

  static final int width = 700;
  static final int height = 700;
  static Color[][] colors = new Color[width][height];
  static final double xMin = -1.9;
  static final double xMax = 0.7;
  static final double yMin = -1.3;
  static final double yMax = 1.3;
  static final int iterations = 15000;

  // Draws the Mandelbrot set using a single thread.
  // The time spent calculating all points is returned.
  // The image-drawing time is left out, because this can only
  // be done in a single thread.

  public static void MandelbrotDraw() {

    final double mandWidth = xMax - xMin;
    final double mandHeight = yMax - yMin;


    //@parallel

    /* Here comes the auto-created parallel code for For statement block!! */

    try {
      Parallel.forLoop(0, "<", height, "++", new IForTask() {
        public void loopBody(int y) throws CancelException
        //Here comes the untouched block of parallized for-loop
        {
          for (int x = 0; x < width; x++) {
```

```
            double xCoord = xMin + ((mandWidth / width) * x);
            double yCoord = yMin + ((mandHeight / height) * y);

            colors[x][y] =
MandelbrotMethods.mapToColor(MandelbrotMethods.solvePoint(xCoord, yCoord, iterations));
          }
        }
        //End of untouched block of parallized for-loop
      });
    } catch (CancelException e) {
      e.printStackTrace();
    }

    /* End of auto-created parallel code!! */


  }

  public static void main(String[] args) {
    MandelbrotDraw();
  }
}



class MandelbrotMethods {
  // Solves a single point in the Mandelbrot set
  // and returns the number of iterations spent.
  // The iterations parameter specifies the maximum
  // number of iterations.
  public static int solvePoint(double x, double y, int iterations) {
    double r = 0.0, s = 0.0;
    double next_r, next_s;
    int iteration = 0;

    while ((r * r + s * s) <= 4.0) {
      next_r = r * r - s * s + x;
      next_s = 2 * r * s + y;
      r = next_r;
      s = next_s;
      if (++iteration == iterations) {
        break;
      }
    }

    return iteration;
  }

  // Maps an iteration number to a color.
  // The colors can be tweaked by fiddling
  // with this method.
  public static Color mapToColor(int val) {
    int r = (val * 12) % 256;
    int g = (val * 16) % 256;
    int b = (val * 5) % 256;

    return new Color(r, g, b);
  }
}
```

MatrixMult.java

```
/*
 * MatrixMult.java
 * Efarmogh pollaplasiasmoy dyo disdiastatwn pinakwn.
 * To apotelesma poy prokyptei apo ton pollaplasiasmo twn dyo aytwn pinakwn
apothhkeyetai se ena trito pinaka.
 */

public class MatrixMult {
```

```
    private static int NDIM = 1000;
    private static long[][] a;
    private static long[][] b;
    private static long[][] c;

    public static void initarrays(){
        a = new long[NDIM][NDIM];
        b = new long[NDIM][NDIM];
        c = new long[NDIM][NDIM];

        //init arrays a[][] and b[][]
        for (int i = 0; i < NDIM; i++) {
            for (int j = 0; j < NDIM; j++) {
                a[i][j] = i + j;
                b[i][j] = i * (j + 1);
            }
        }
    }

    public static void sumcheck(){
        long total_sum = 0;

        System.out.println("Now summing the matrix");

        //a kind of check, sum the results array
        for (int i = 0; i < NDIM; i++) {
            for (int j = 0; j < NDIM; j++) {
                total_sum += c[i][j];
            }
        }

        //Total sum = 76390280640000000
        System.out.println("Total sum: " + total_sum);
    }

    public static void matrixmult() {

        System.out.println("Executing Matrix Multiplication");

        //matrix multiplication => c[][] = a[][] x b[][]
        //@parallel
        for (int i = 0; i < NDIM; i++) {
            for (int j = 0; j < NDIM; j++) {
                long sum = 0;
                for (int k = 0; k < NDIM; k++) {
                    sum += a[i][k] * b[k][j];
                }
                c[i][j] = sum;
            }
        }
    }

    public static void main(String[] args) {
        initarrays();
        matrixmult();
        sumcheck();
    }
}
```

MatrixMult_mult.java

```
/*
 * MatrixMult.java
 * Efarmogh pollaplasiasmoy dyo disdiastatwn pinakwn.
 * To apotelesma poy prokyptei apo ton pollaplasiasmo twn dyo aytwn pinakwn
apothhkeyetai se ena trito pinaka.
 */
```

```
import com.cs.ucy.ac.cy.jacon. * ;

public class MatrixMult_mult {

  private static int NDIM = 1500;
  private static long[][] a;
  private static long[][] b;
  private static long[][] c;

  public static void initarrays() {
    a = new long[NDIM][NDIM];
    b = new long[NDIM][NDIM];
    c = new long[NDIM][NDIM];

    //init arrays a[][] and b[][]
    for (int i = 0; i < NDIM; i++) {
      for (int j = 0; j < NDIM; j++) {
        a[i][j] = i + j;
        b[i][j] = i * (j + 1);
      }
    }
  }

  public static void sumcheck() {
    long total_sum = 0;

    System.out.println("Now summing the matrix");

    //a kind of check, sum the results array
    for (int i = 0; i < NDIM; i++) {
      for (int j = 0; j < NDIM; j++) {
        total_sum += c[i][j];
      }
    }

    //Total sum = 76390280640000000
    System.out.println("Total sum: " + total_sum);
  }

  public static void matrixmult() {

    System.out.println("Executing Matrix Multiplication");

    //matrix multiplication => c[][] = a[][] x b[][]

    //@parallel

    /* Here comes the auto-created parallel code for For statement block!! */

    try {
      Parallel.forLoop(0, "<", NDIM, "++", new IForTask() {
        public void loopBody(int i) throws CancelException
        //Here comes the untouched block of parallized for-loop
        {
          for (int j = 0; j < NDIM; j++) {
            long sum = 0;
            for (int k = 0; k < NDIM; k++) {
              sum += a[i][k] * b[k][j];
            }
            c[i][j] = sum;
          }
        }
        //End of untouched block of parallized for-loop
      });
    } catch (CancelException e) {
      e.printStackTrace();
    }

    /* End of auto-created parallel code!! */

  }

  public static void main(String[] args) {
    initarrays();
```

```
    matrixmult();
    sumcheck();
  }
}
```