ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ: Ηλεκτρονικής και Υπολογιστών
ΕΡΓΑΣΤΗΡΙΟ: Συστημάτων Υπολογιστών

Διπλωματική Εργασία

του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών της Πολυτεχνικής Σχολής του Πανεπιστημίου Πατρών

Κομηνού Χαράλαμπου-Γαβριήλ του Βρεττού

Αριθμός Μητρώου: 6544

Θέμα

«Υπολογιστικές εφαρμογές σε περιβάλλον παράλληλης επεξεργασίας»

Επιβλέπων
Χούσος Ευθύμιος, Καθηγητής

Αριθμός Διπλωματικής Εργασίας:

Πάτρα, Οκτώβριος 2013

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η Διπλωματική Εργασία με θέμα

«Υπολογιστικές εφαρμογές σε περιβάλλον παράλληλης επεξεργασίας»

Του φοιτητή του Τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών

Κομηνού Χαράλαμπου του Βρεττού

Αριθμός Μητρώου:6544

Παρουσιάστηκε δημόσια και εξετάστηκε στο Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών στις
……../……../……

Ο Επιβλέπων Καθηγητής

Χούσος Ευθύμιος
Καθηγητής

Ο Διευθυντής του Τομέα

Χούσος Ευθύμιος
Καθηγητής

Αριθμός Διπλωματικής Εργασίας:

Θέμα: «Υπολογιστικές εφαρμογές σε περιβάλλον παράλληλης επεξεργασίας»

Φοιτητής:
Κομηνός Χαράλαμπος Γαβριήλ

Επιβλέπων Καθηγητής:
Χούσος Ευθύμιος

Περίληψη

Η παρούσα διπλωματική εργασία πραγματοποιήθηκε κατά το διάστημα 2012-2013 υπό την επίβλεψη του καθηγητή Χούσου Ευθύμιου στο Εργαστήριο Συστημάτων Υπολογιστών (CSL) του Πανεπιστημίου Πατρών. Στόχος της εργασίας είναι η προσπάθεια επίλυσης ενός προβλήματος χρονοπρογραμματισμού εξετάσεων (ETP), με χρήση πληροφορημένου γενετικού αλγορίθμου. Στην εργασία αυτή θα παρουσιαστούν, τα βασικά μοντέλα λειτουργίας των γενετικών αλγορίθμων, του ETP καθώς και παρουσίαση βασικών εννοιών των παράλληλων συστημάτων.
Τέλος παρουσιάζεται ο σειριακός κώδικας που υλοποιήθηκε σε ANSI-C και στην συνέχεια γίνεται σύγκριση με τον παράλληλο κώδικα που υλοποιήθηκε με MPI-C και παρουσιάζονται τα αποτελέσματα της σύγκρισης μεταξύ των δύο.

Abstract

The Aim of this thesis which was completed during the 2012/2013 academic year under the supervision of professor Housos Eythimios at the Computer Systems Laboratory (CSL) at the University of Patras is to solve the Examination Timetabling Problem (ETP) with the aid of an informed genetic algorithm. I will present the basic model under which the genetic algorithms operate and some information about the ETP and general parallel systems. To conclude we will present our serial ANSI-C code and compare it with the parallel MPI-C code that we build and compare the two results.

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω αρχικά όλους αυτούς που με βοήθησαν τόσα χρόνια στην προσπάθεια μου να πάρω πτυχίο. Πρώτα θα ήθελα να ευχαριστήσω τον κύριο Χούσο για την απόφαση του να δώσει διπλωματική πριν τέσσερα χρόνια σε φοιτητή που χρωστούσε τόσα μαθήματα και ελπίζω να είναι ικανοποιημένος από την ποιότητα της διπλωματικής αυτής. Στη συνέχεια θα ήθελα να ευχαριστήσω τον διδακτορικό φοιτητή Βασίλη Κολώνια για την βοήθεια του και την καθοδήγηση του, ειδικά προς το τέλος της εργασίας. Επιπλέον εκφράζω την ευχαρίστηση μου στα παιδιά που με φιλοξένησαν στις εξεταστικές όταν είχα ξενοικιάσει και δουλέψαμε μαζί στις εξεταστικές (Νάσος, Αλέξια). Τέλος ευχαριστώ την οικογένεια μου ήταν πάντα εκεί όταν τους χρειάστηκα σε αυτό τον μακρύ και δύσκολο αγώνα .

Η Διπλωματική αυτή αφιερώνεται στην μνήμη του εξαίρετου φίλου, ιατρού, επιστήμονα και πάνω από όλα ανθρώπου, κ Αργύρη Μιχαλόπουλου ο οποίος έφυγε πολύ άδικα και πολύ νωρίς (6/4/13) λίγο πριν την απόκτηση του πτυχίου μου στην απόκτηση του οποίου βοήθησε ουκ ολίγες φορές με συμβουλές και οδηγίες. Είμαι βέβαιος ότι είναι περήφανος όπου και αν είναι.

ΚΕΦΑΛΑΙΟ 1 : ΠΕΡΙΛΗΨΗ

Στα πλαίσια της διπλωματικής αυτής θα ασχοληθούμε με την επίλυση ενός προβλήματος χρονοπρογραμματισμού. Ο χρονοπρογραμματισμός των εξετάσεων είναι μια πολύ σημαντική λειτουργία που λαμβάνει χώρα σε όλα τα ιδρύματα του κόσμου και είναι η διαδικασία κατά την οποία προσπαθεί το ίδρυμα να δημιουργήσει ένα πρόγραμμα για την εξεταστική, το οποίο εξ ορισμού να είναι υλοποιήσιμο δηλαδή κανένας φοιτητής να μην πρέπει να παραβρεθεί σε δύο αμφιθέατρα την ίδια στιγμή. Αν και σαν ιδέα ακούγεται απλό, στην πραγματικότητα είναι ένα πολύ δύσκολο πρόβλημα για το οποίο δεν υπάρχει ακριβής αλγόριθμος επίλυσης διότι ο χώρος των λύσεων, είναι πρακτικά άπειρος.

Αφού λοιπόν αποκλείσαμε την πιθανότητα ακριβής επίλυσης πρέπει να στραφούμε σε κάτι άλλο. Από το σύνολο των τρόπων επίλυσης που υπάρχουν στη βιβλιογραφία θα στραφούμε σε μια κατηγορία υπερευρετικών με την ονομασία γενετικοί αλγόριθμοι και βασίζονται στη διαδικασία της φυσικής εξέλιξης του Δαρβίνου. Τα υπερευρετικά είναι ένας τρόπος επίλυσης και σκέψης, κατά τον οποίο δεχόμαστε ότι το τελικό αποτέλεσμα δεν θα είναι βέλτιστο αλλά θα είναι κοντά σε αυτό. Για αυτή όμως την χαλάρωση των περιορισμών κερδίζουμε πάρα πολύ σε χρόνο εκτέλεσης και καταλήγουμε σε αποτελέσματα τα οποία είναι πολύ καλά και τα αποκτάμε γρήγορα. Οι γενετικοί αλγόριθμοι ειδικεύονται στην επίλυση προβλημάτων στα οποία ο χώρος είναι πολύ μεγάλος διότι ψάχνουν ταυτόχρονα σε πάρα πολλά σημεία. Ο τρόπος λειτουργίας τους είναι ο εξής. Δημιουργείται ένας αρχικός πληθυσμός. Αυτός ο πληθυσμός, ο οποίος αποτελείται από ένα σύνολο από χρωμοσώματα αντιπροσωπεύει μια πιθανή λύση στο πρόβλημα. Ο τρόπος δημιουργίας αυτού του πληθυσμού μπορεί να είναι είτε τυχαίος είτε όχι και στόχος του αλγορίθμου είναι να βελτιώσει αυτό τον πληθυσμό σαν σύνολο. Αυτή η βελτίωση λαμβάνει χώρα με την χρήση τριών στοχαστικών τελεστών που είναι η επιλογή, η αναπαραγωγή και η μετάλλαξη.

Ο γενετικός λοιπόν ξεκινάει με έναν πληθυσμό και εκτελεί με την σειρά τα εξής. Πρώτος τελεστής είναι η επιλογή και στόχος της είναι, από τον αρχικό πληθυσμό, να επιλέξει και να δημιουργήσει έναν νέο πληθυσμό ίσου μεγέθους ο οποίος προέρχεται στην ουσία από τους καλύτερους της προηγούμενης γενιάς. Από τις πολλές μεθόδους επιλογής, στα πλαίσια της εργασίας επιλέξαμε να παίξουμε το λεγόμενο τουρνουά δηλαδή επιλέγεται τυχαία ένα υποσύνολο λύσεων οι οποίες παίζουν μεταξύ τους τουρνουά και η καλύτερη κερδίζει και περνάει στην επόμενη γενιά. Ο επόμενος τελεστής είναι η αναπαραγωγή και στόχος της είναι να μιμηθεί την φυσική αναπαραγωγή. Η ιδέα πίσω από αυτό τον τελεστή είναι ότι, όπως και στη φύση, τα παιδιά δύο γονέων αναμένεται να έχουν καλύτερα χαρακτηριστικά και από τους δύο γονείς. Αν συνδυάσουμε αυτό το γεγονός και με τον επόμενο τελεστή της μετάλλαξης, δηλαδή με την τυχαία μετατροπή ενός χαρακτηριστικού του πληθυσμού, οδηγούμαστε σε έναν νέο πληθυσμό, που θα αντικαταστήσει τον παλαιό, και αναμένεται σαν σύνολο να έχει καλύτερα χαρακτηριστικά, δηλαδή να πλησιάσει σε μια λύση για το πρόβλημα. Ο αλγόριθμος επαναλαμβάνει τα ανωτέρω βήματα έναν αριθμό φορών, που τις ονομάζουμε γενιές, και τερματίζει έχοντας ιδανικά βρει λύσεις στο πρόβλημα.

Με αυτό λοιπόν το υπόβαθρο δημιουργήθηκε ένα κώδικας σε ANSI-C ο οποίο λύνει τα προβλήματα χρονοπρογραμματισμού του Carter. Το Carter είναι ένα σύνολο προβλημάτων της βιβλιογραφίας όπου οι ερευνητές των προβλημάτων χρονοπρογραμματισμού δοκιμάζουν την ποιότητα των λύσεων της έρευνάς τους. Δοκιμάζοντας και εμείς τον τρόπο μας καταλήξαμε να λύσουμε οκτώ από τα δεκατρία προβλήματα που μας δόθηκαν μέσα σε εβδομήντα γενιές. Αν και ικανοποιητικό αποτέλεσμα στη συνέχεια δόθηκε άλλη μία ευκαιρία στον αλγόριθμο που δημιουργήσαμε, αυτή την φόρα όμως του δόθηκαν και κάποιες ήδη γνωστές λύσεις από την βιβλιογραφία με σκοπό να δούμε αν ο αλγόριθμος θα βελτιώσει τα αποτελέσματα του όπως και έγινε.

Κοιτώντας τους χρόνους εκτέλεσης του αλγορίθμου, δημιουργείται η ανάγκη περαιτέρω μείωσης τους διότι η αύξηση του μεγέθους του προβλήματος κατά έναν παράγοντα δύο, τετραπλασιάζει τον χρόνο εκτέλεσης του αλγορίθμου. Θα στραφούμε λοιπόν στην παράλληλη επεξεργασία για προσπαθήσουμε να μειώσουμε τον χρόνο εκτέλεσης. Η παράλληλη επεξεργασία, που γνωρίζει άνθηση, βασίζεται στο σκεπτικό του διαχωρισμού ενός μεγάλου προβλήματος σε μικρότερα και επίλυση των μικρότερων. Όπως είναι γνωστό η καρδιά όλων των συστημάτων υπολογιστών είναι ο επεξεργαστής. Σε αυτόν εκτελείται το σύνολο τον εργασιών ενός υπολογιστή και η ταχύτητα του, που μετριέται σε Hz είναι ο βασικότερος

παράγοντας που καθορίζει τον χρόνο που θα χρειαστεί ο επεξεργαστής για να τερματίσει τις εργασίες που του έχουν ανατεθεί. Συνδυάζοντας τον επεξεργαστή με μια μνήμη και συνδέοντας τα δυο, έχουμε ένα ολοκληρωμένο υπολογιστικό σύστημα.

Είναι λοιπόν εύλογο να προσπαθήσουμε να λύσουμε τα προβλήματα μας χωρίζοντας τα σε μικρότερα τμήματα και στη συνέχεια βάζοντας έναν μεγάλο αριθμό επεξεργαστών να επιλύσει τα τμήματα αυτά. Δουλεύοντας σε ένα κατανεμημένο σύστημα επεξεργασίας όπως του εργαστηρίου είναι προφανές ότι πρέπει να ορίσουμε έναν τρόπο επικοινωνίας μεταξύ των διαφορετικών επεξεργαστών. Ο πλέον διαδεδομένος τρόπος είναι η χρήση μιας βιβλιοθήκης που λέγεται MPI και χρησιμοποιείται κατά κόρον σε παράλληλα συστήματα. Βασική μονάδα μέτρησης αυτής της βελτίωσης λόγο παράλληλης επεξεργασίας είναι το Speedup το οποίο εκφράζει πόσο πιο γρήγορα λύθηκε το πρόβλημα από έναν αριθμό επεξεργαστών.

Στα πλαίσια της εργασίας για την παραλληλοποίηση του γενετικού αλγορίθμου επιλέχθηκε το μοντέλο των νησιών. Σε αυτό το μοντέλο κάθε επεξεργαστής έχει ένα δικό του πληθυσμό, μικρότερο σε μέγεθος του σειριακού. Ο κάθε επεξεργαστής – νησί αρχικά έχει μόνο αυτόν τον πληθυσμό για να βελτιώσει. Θα εισάγουμε λοιπόν μία ακόμα δυνατότητα στον παράλληλο γενετικό αλγόριθμο που είναι η μετανάστευση. Σε αυτό το βήμα, ανά τα τακτά χρονικά διαστήματα, ένα υποσύνολο του πληθυσμού, εγκαταλείπει το νησί στο οποίο βρίσκεται και μεταφέρεται σε κάποιο άλλο. Έτσι έχουμε έμμεσα μεγεθύνει το μέγεθος των νησιών και ο παράλληλος γενετικός στη συνέχεια προσπαθεί να λύσει τα ίδια προβλήματα χρονοπρογραμματισμού. Το αποτέλεσμα αυτής της προσπάθειας ήταν η δημιουργία ενός παράλληλου αλγορίθμου ο οποίος λύνει το πρόβλημα με τον ίδιο πληθυσμό, περίπου στο ένα τρίτο του χρόνου αλλά βρίσκει λύση σε λιγότερα προβλήματα. Βρίσκοντας λύση σε έξι από τα δεκατρία προβλήματα, ο αλγόριθμος αποδίδει αξιόλογα από άποψης ποιότητας και επιταχύνει την διαδικασία, όπως και αναμενόταν.

Καταλήγουμε λοιπόν μετά το τέλος των πειραμάτων στα εξής συμπεράσματα. Η επικοινωνία μεταξύ των υπολογιστών είναι ένας πάρα πολύ σημαντικός παράγοντας που δεν πρέπει να αμελείται σε καμία περίπτωση. Η επικοινωνία πρέπει διατηρείται στο ελάχιστο μεταξύ των επεξεργαστών και τα μηνύματα να μεγιστοποιούνται σε μέγεθος. Επιπλέον καταλήγουμε στο συμπέρασμα ότι ο σειριακός γενετικός έχει καλύτερες πιθανότητες να βρει λύσεις λόγο του πλήθους του γενετικού υλικού που διαθέτει. Επιπλέον οι παράμετροι του πληθυσμού, και τον πιθανοτήτων μετάλλαξης και αναπαραγωγής, καθορίζουν τον ποιότητα του αλγορίθμου και πρέπει να επιλέγονται με προσοχή. Καταλήγουμε επίσης στο γεγονός ότι αυτές οι τιμές όταν είναι πολύ μικρές καθυστερούν πολύ την δημιουργία διαφορετικών χαρακτηριστικών σε ένα πληθυσμό και κατά συνέπεια την σύγκλιση του αλγορίθμου σε υπαρκτή λύση.

Οι δυνατοί συνδυασμοί των παραμέτρων είναι πάρα πολλοί και αξίζει η μελέτη τους. Ιδανικός στόχος είναι η δημιουργία ενός μοντέλου επιλογής των παραμέτρων ανάλογα με την προβλεπόμενη δυσκολία του προβλήματος καθώς και συνδυασμός των γενετικών αλγορίθμων με άλλη τεχνική ώστε να οδηγηθούμε σε ακόμα καλύτερες λύσεις και ιδανικά στην τελική επίλυση των προβλημάτων χρονοπρογραμματισμού.

ΚΕΦΑΛΑΙΟ 2:ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

2.1)Εισαγωγή στην παράλληλη επεξεργασία

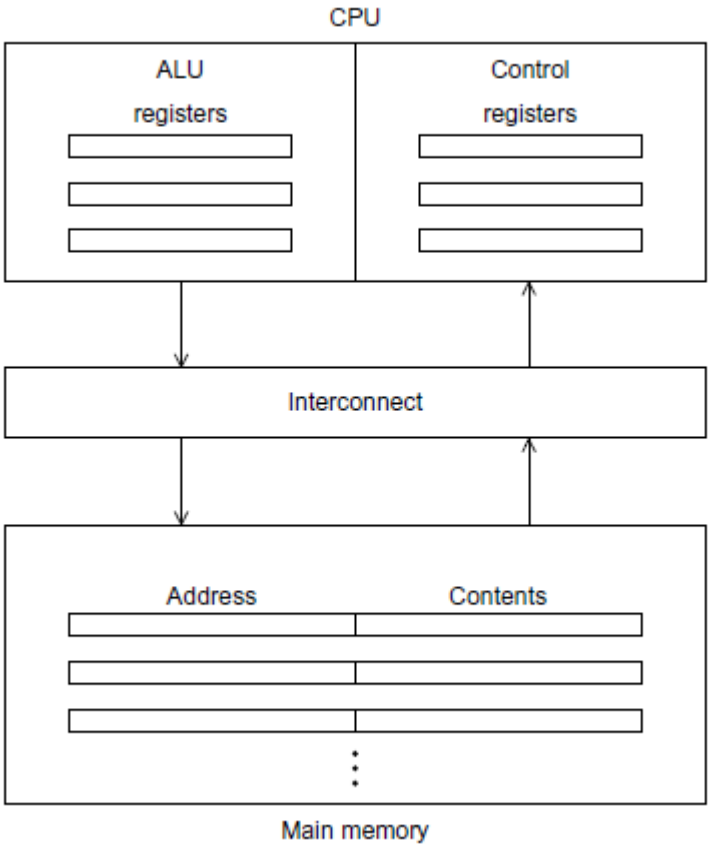
Τα παράλληλα συστήματα έχουν δημιουργηθεί σαν εξελίξεις των σειριακών συστημάτων και κατά συνέπεια πριν καταγράψουμε βασικά ζητήματα και ιδέες των παράλληλων συστημάτων κρίνεται απαραίτητη μια επανάληψη κάποιων εννοιών από τα σειριακά. Ο κλασσικός σχεδιασμός είναι ο λεγόμενος Von Neumann για τα συστήματα και αποτελείται από μια CPU (ή Core), μια μνήμη RAM και ένα δίκτυο διασύνδεσης μεταξύ αυτών των δυο ώστε να μπορούν να αλληλεπιδρούν. Η κύρια μνήμη αποτελείται από ένα σύνολο στοιχείων τα οποία αποθηκεύουν είτε δεδομένα, είτε εντολές και από διευθύνσεις οι οποίες χρησιμοποιούνται ώστε να υπάρχει πρόσβαση στα στοιχεία.(σχήμα 2.1)

Η Cpu γενικά αποτελείται από μια ALU που είναι υπεύθυνη για τις πράξεις και από μια μονάδα ελέγχου (control unit) που ασχολείται με τις εντολές που εκτελούνται. Οι εντολές που θα εκτελεστούν και κάποιες άλλες πληροφορίες όπως ο Program Counter κα καταχωρούνται σε μια ταχύτατα μονάδα αποθήκευσης κοντά στη Cpu που λέγεται cache. Τα δεδομένα μεταφέρονται μέσω του δικτύου διασύνδεσης το οποίο παραδοσιακά είναι ένα BUS δηλαδή ένα σύνολο από παράλληλα καλώδια. Όταν η CPU αναζητά δεδομένα τότε λέμε ότι κάνει fetch από την μνήμη.

Η βιβλιογραφία είναι πλούσια για αυτό το πολύ ενδιαφέρον θέμα. Στα πλαίσια της εργασίας θα παραμείνουμε σε αυτό το απλό σχεδιάγραμμα (2.1) το οποίο κάνει σαφές το μεγάλο πρόβλημα αυτής της αρχιτεκτονικής το οποίο είναι το λεγόμενο Von Neumann Bottleneck το οποίο ορίζει ότι λόγο του διαχωρισμού μεταξύ CPU και μνήμης η χρόνος στον οποίο θα ολοκληρωθεί μια διεργασία ορίζεται κατά κύριο λόγο από το σύστημα διασύνδεσης το οποίο είναι πολλές φορές πιο αργό από την Cpu.
Με δεδομένα 2013 υπάρχουν Cpu που εκτελούν εντολές τουλάχιστον 100 φορές πιο γρήγορα από ότι μπορούν να κάνουν fetch δεδομένα από την μνήμη. Αυτό είναι ένα μεγάλο πρόβλημα το οποίο μειώνει δραστικά την ταχύτητα ενός συστήματος και απαιτεί λύση. Έχουν γίνει πολλές προσπάθειες ώστε να γίνουν τα συστήματα πιο γρήγορα οι βασικότερες εκ των οποίων θα περιγραφούν στη συνέχεια.

Μια μεγάλη προσπάθεια γίνεται στο λειτουργικό σύστημα (OS). Το σύνολο των μοντέρνων λειτουργικών υποστηρίζουν πλέον multithreading δηλαδή ένα σύστημα στο οποίο το λειτουργικό πραγματοποιεί ταχύτατες εναλλαγές μεταξύ των running processes και δίνει την εντύπωση ότι όλα τρέχουν ταυτόχρονα. Αλλά αυτό δεν λύνει το πρόβλημα της ταχύτητας απλά φαίνεται στο χρήστη ότι το λύνει. Ένα άλλο βήμα προς την βελτίωση είναι το Instruction Level Parallelism. Αυτό το σύστημα επιτρέπει σε ένα επεξεργαστή να εκτελεί πολλές εντολές ταυτόχρονα. Οι δυο κύριες μέθοδοι είναι το pipelining και το multiple issue και εμφανίζονται πλέον σε όλες τις μοντέρνες Cpu.

Μια άλλη προσπάθεια που μας ενδιαφέρει στα πλαίσια της διπλωματικής είναι η παράλληλη επεξεργασία. Η αρχή πίσω από την οποία στηρίζονται τα παράλληλα συστήματα, είναι ότι, στην πλειοψηφία των περιπτώσεων ένα μεγάλο πρόβλημα μπορεί να διαχωριστεί σε πολλά μικρότερα και ανεξάρτητα τμήματα. Έτσι υπάρχει δυνατότητα να εκτελεστούν αυτά τα ανεξάρτητα κομμάτια σε διαφορετικούς επεξεργαστές και στο τέλος η λύση να προέλθει από την συνένωση των κομματιών. Με αυτή την προσέγγιση λοιπόν ευελπιστούμε ότι τα μικρότερα κομμάτια τα οποία θα εκτελεστούν παράλληλα, θα δώσουνε πιο γρήγορα αποτελέσματα από έναν σειριακό υπολογιστή.



Πίνακας 2.1 **[17]** Von Neumann CPU architecture

2.2) Παράλληλες Αρχιτεκτονικές

Διαχωρισμός των συστημάτων

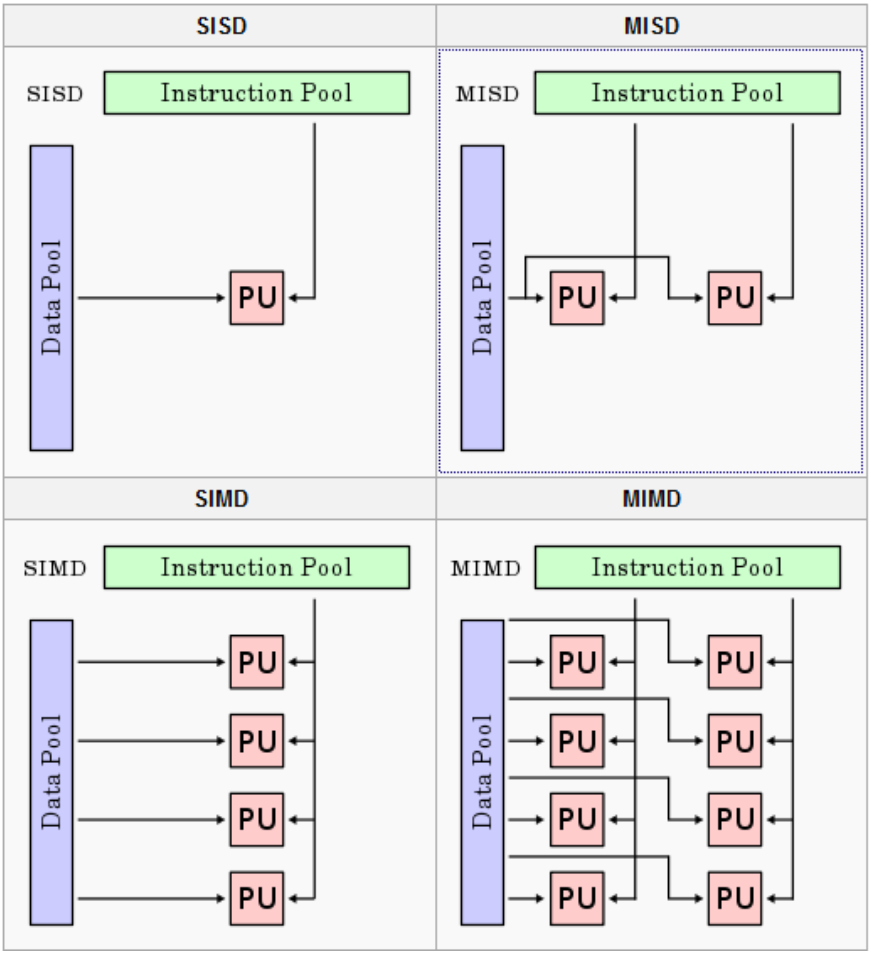
Γνωρίζοντας λοιπόν κάποια πράγματα για την αρχιτεκτονική του συστήματος στο οποίο εργάζεται, ο προγραμματιστής μπορεί να μεταβάλει τον κώδικα του ώστε να κερδίσει κάτι από το παράλληλο σύστημα. Τα παράλληλα ταξινομούνται σύμφωνα με τον Michael J Flynn (ταξινόμηση Flynn) στα εξής:

Single Instruction, Single Data stream (SISD)
Σειριακοί υπολογιστές που δεν έχουν κανένα στοιχείο παραλληλισμού και οι εντολές εκτελούνται μια την φορά π.χ. παλιά mainframes.

Single Instruction, Multiple Data streams (SIMD)
Υπολογιστές οι οποίοι χρησιμοποιούν πολλά δεδομένα για μια εντολή όπως οι GPU.

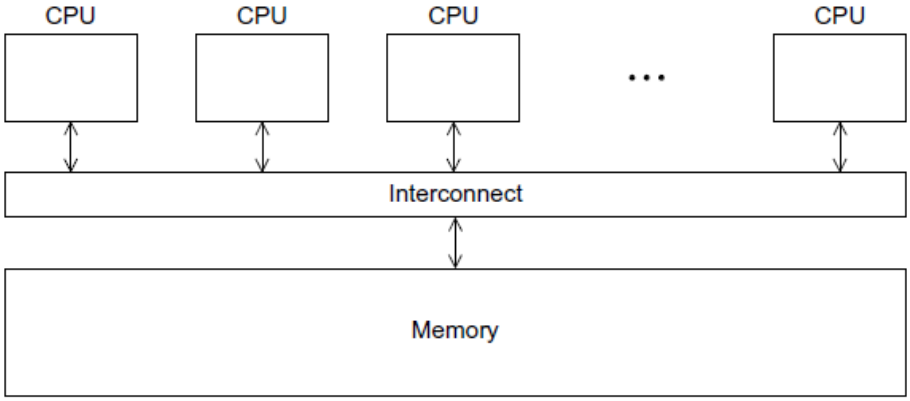
Multiple Instruction, Single Data stream (MISD)
Σχετικά σπάνια αρχιτεκτονική που κυρίως χρησιμοποιείται για συστήματα ανθεκτικά σε λάθη. Σε αυτά τα συστήματα πολλαπλές εντολές ενεργούν σε μοναδικά δεδομένα

Multiple Instruction, Multiple Data streams (MIMD)
Πολλαπλές εντολές πραγματοποιούνται ταυτόχρονα σε πολλά δεδομένα.

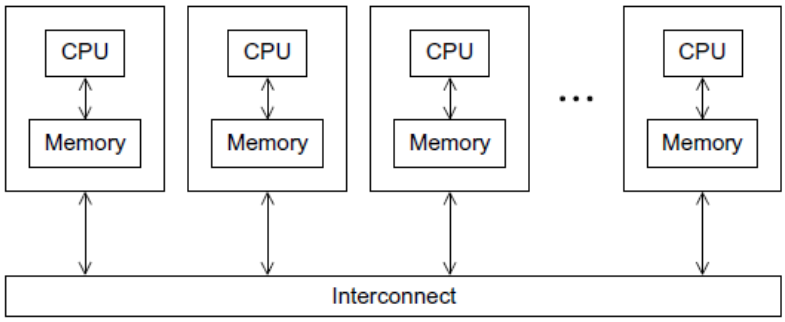


Σχήμα 2.2**[37]** Διαχωρισμοί Συστημάτων

Υπάρχει ένας ακόμα ενδιαφέρον διαχωρισμός των συστημάτων μνήμης στα παράλληλα συστήματα που πρέπει να γίνει κατανοητός. Κατανεμημένη μνήμη (distributed memory) και κοινή μνήμη (Shared memory). Στην πρώτη περίπτωση ο κάθε επεξεργαστής έχει δική του μνήμη στην οποία πρόσβαση έχει μόνον αυτός και κατά συνέπεια η επικοινωνία μεταξύ των Cpu γίνεται με Message Passing με κλασικότερο παράδειγμα τα Clusters. Στην άλλη κατηγορία η μνήμη είναι κοινή και όλοι έχουν πρόσβαση σε αυτή. Τα σχήματα κάνουν σαφέστερη την βασική αυτή λεπτομέρεια.



Σχήμα 2.3a**[17]** Shared Memory



Σχήμα 2.3b**[17]** Distributed Memory

Τέλος υπάρχουν οι εξής διαχωρισμοί. Τα συστήματα κοινής μνήμης χωρίζονται σε δυο κατηγορίες.
Αν όλοι οι επεξεργαστές συνδέονται απευθείας με την κεντρική μνήμη έχουμε το λεγόμενο Uniform Memory Access. Εάν οι επεξεργαστές βλέπουν ξεχωριστά block μνήμης και η επεξεργασία ενός block από άλλο επεξεργαστή γίνεται με ειδικό hardware τότε μιλάμε για Non-Uniform Memory Access**[17].**

2.3) Συστήματα επικοινωνίας

Συστήματα κοινής μνήμης

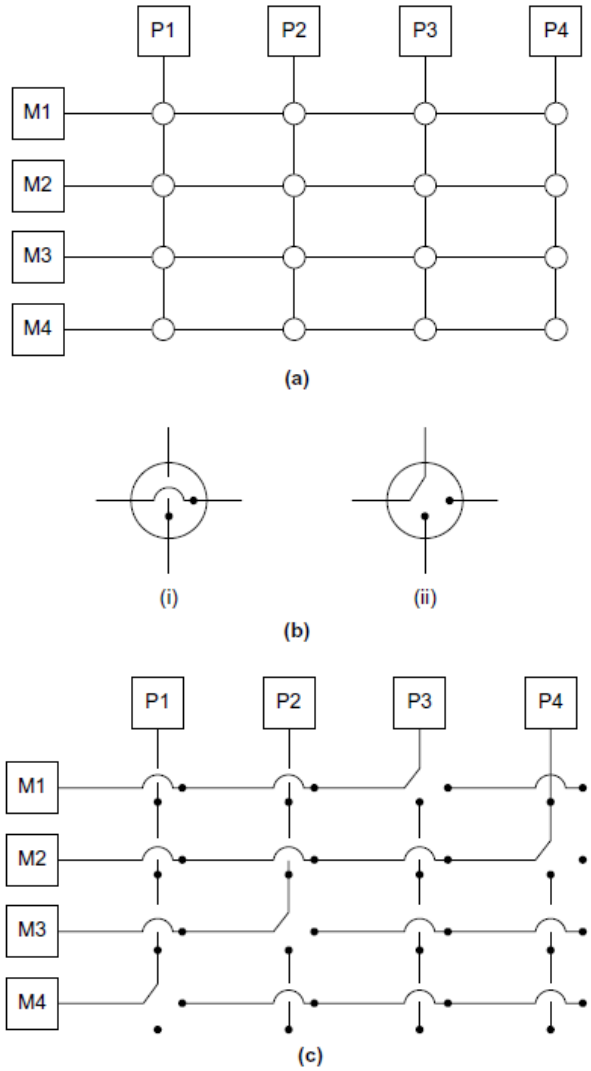
Το σύστημα διασύνδεσης παίζει πολύ σημαντικό ρόλο τόσο στα κατανεμημένα όσο και στα κοινής μνήμης συστήματα. Ακόμα και αν υποθέταμε ότι οι επεξεργαστές είχανε άπειρη επεξεργαστική ισχύ ένα αργό δίκτυο θα τους υποχρέωνε να δουλεύουν με την ταχύτητα του δικτύου.
Δυο είναι τα κύρια είδη συστημάτων διασύνδεσης που θα σχολιαστούν:

BUS

Το πρώτο είναι το είναι το bus το οποίο όπως αναφέρθηκε είναι ένα σύνολο από παράλληλα καλώδια. Βασικό στοιχείο είναι ότι όλες οι συσκευές είναι συνδεδεμένες στο bus. Το κυριότερο πλεονέκτημα του είναι η απλότητα του διότι χωρίς κόστος όλες οι συσκευές συνδέονται και επικοινωνούν. Στον αντίποδα όμως όταν ο αριθμός των διασυνδεδεμένων συσκευών αυξάνεται τόσο αυξάνεται και η πιθανότητα οι συσκευές να πρέπει να περιμένουν για την χρήση του bus και κατά συνέπεια η αναμενόμενη απόδοση πέφτει.

CROSSBAR

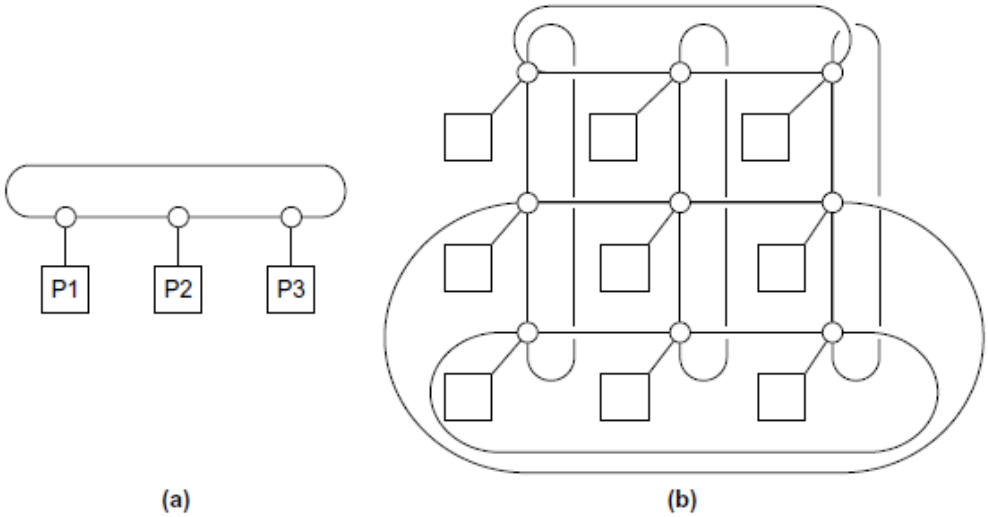
Για την επίλυση του προβλήματος του bus με τις πολλές συσκευές δημιουργήθηκε το Crossbar (Switched interconnection). Χρησιμοποιεί switches ώστε να ελέγχει τη ροή των δεδομένων μεταξύ πολλών επεξεργαστών και πολλών θέσεων μνήμης. Τα σχήματα **[11]** δείχνουν ένα σύστημα διασύνδεσης μεταξύ 4 Cpu και τεσσάρων συστημάτων μνήμης με crossbars. Μπορούμε να δούμε εύκολα ότι στην γενική περίπτωση μπορούμε να γράψουμε παράλληλα στις θέσεις μνήμης του συστήματος. Προσοχή μόνο χρειάζεται όταν δυο επεξεργαστές απαιτούν πρόσβαση στην ίδια θέση μνήμης (contention). Το πλεονέκτημα αυτό βέβαια αντισταθμίζεται από το αυξημένο κόστος των πολλών crossbars.



Σχήμα 2.4**[17]** four way interconnection

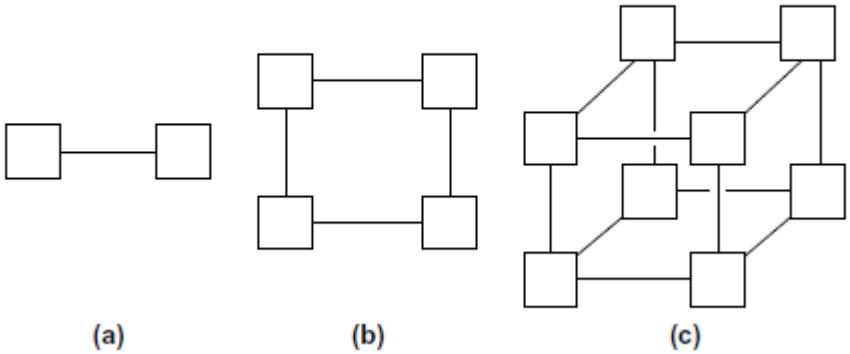
Συστήματα κατανεμημένης μνήμης

Σε αυτά τα συστήματα που ενδιαφέρουν περισσότερο στα πλαίσια της εργασίας υπάρχουν 2 κατηγορίες συστημάτων διασύνδεσης. Η άμεση και η έμμεση. Στην άμεση διασύνδεση κάθε Switch είναι άμεσα συνδεδεμένο με ένα ζευγάρι μνήμης-Cpu και τα Switches είναι συνδεδεμένα και μεταξύ τους. Τα επόμενα σχήματα δείχνουν ένα **ring** και ένα **toroidal mesh**. Στο σχήμα 2.5 (τετράγωνο = Cpu , κύκλος = Switch) βλέπουμε το βασικό πλεονέκτημα των συνδεσμολογιών αυτών έναντι του bus, ότι στην γενική περίπτωση μπορούμε να έχουμε πολλαπλές επικοινωνίες.



Σχήμα 2.5 **[17]** ring and toroidal mesh

Από το σύνολο των πολλαπλών συνδεσμολογιών της βιβλιογραφίας, αξίζει να αναφέρουμε 2 ακόμα. Η βέλτιστη συνδεσμολογία από πλευράς επικοινωνίας είναι αυτή, στην οποία όλα τα Switch συνδέονται με όλα τα Switch. Αυτό όμως το σύστημα απαιτεί $p^2/2+p/2$ καλώδια για την υλοποίηση του που εν γένει είναι μη πρακτικό. Περισσότερο ενδιαφέρον παρουσιάζει το hypercube το οποίο απαντάται συχνά λόγο της ευχρηστίας του διότι απαιτεί 1+log2(p) για την υλοποίηση του και έχει πολύ καλή απόδοση.



Σχήμα 2.6 **[17]** (1,2,3 Dimensional hypercubes)

Σύγκριση

Μια εύλογη απορία είναι, για πιο λόγο όλα τα MIMD συστήματα δεν είναι κοινής μνήμης αφού η πλειοψηφία των προγραμματιστών θεωρεί την αποστολή μηνυμάτων ακριβή και πλεονάζουσα εργασία και προτιμά απλά να ρυθμίζει και να συγχρονίζει τους επεξεργαστές μέσω μόνο των δομών δεδομένων.
Ο κυριότερος λόγο ύπαρξης των κατανεμημένων συστημάτων είναι ότι όσο αυξάνεται το μέγεθος του συστήματος αυξάνεται και η πολυπλοκότητα συγχρονισμού ενώ τα δίκτυα όπως το hypercube είναι πολύ απλά στην ρύθμιση. Έτσι μπορούμε να εξάγουμε με ασφάλεια το συμπέρασμα ότι, τα προβλήματα τα οποία απαιτούν πλήθος δεδομένων και μεγάλη επεξεργαστική ισχύς, τα κατανεμημένα συστήματα με χρήση βιβλιοθηκών όπως το MPI είναι η πλέον αποδοτική λύση και έτσι θα εργαστούμε στα πλαίσια αυτής της διπλωματικής.

2.4) Performance

2.4.1 Speedup-Efficiency

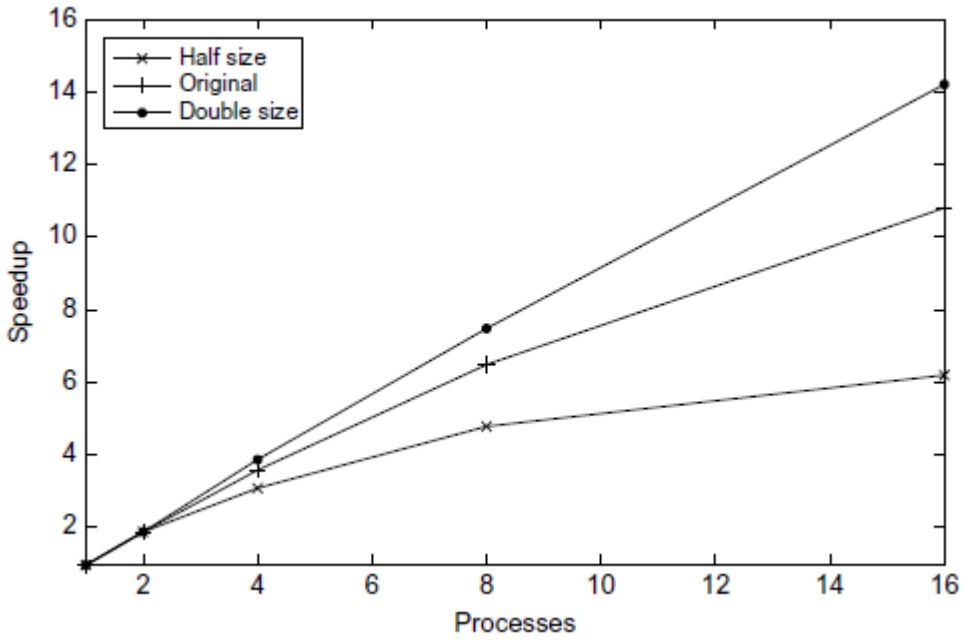
Είναι προφανές ότι ο λόγος που προσπαθούμε να γράψουμε παράλληλα προγράμματα είναι για να αυξήσουμε την απόδοση τους. Τι σημαίνει όμως αυτό. Υπάρχουν δυο μεγέθη για την μέτρηση αυτής της αφηρημένης έννοιας. Το **speedup** και η αποδοτικότητα. Ιδανικά θέλουμε να χωρίσουμε την διεργασία σε όμοια κομμάτια όσα και οι επεξεργαστές(p) (load balancing). Αν μπορούμε να το κάνουμε αυτό τότε το πρόγραμμα μας θα είναι N φορές πιο γρήγορο δηλαδή αν ονομάσουμε Tserial τον χρόνο που κάνει ο καλύτερος σειριακός κώδικας και Tparallel τον χρόνο που χρειάζεται ο παράλληλος κώδικας, αυτοί συνδέονται με την σχέση του speedup

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

Κατά συνέπεια το ανώτατο όριο είναι το γραμμικό speedup. Επιπλέον όσο το p αυξάνεται αναμένουμε το S να αυξάνεται ανάλογα. Δυστυχώς αυτό είναι μόνο θεωρητικό διότι η εισαγωγή δικτύων και πολλών επεξεργαστών δημιουργεί επιπλέον χρόνο T overhead (latency Δικτύου, mutex κλπ) το οποίο αυξάνει τον χρόνο T parallel. Εισάγουμε λοιπόν άλλο ένα μέγεθος που θα ονομάσουμε αποδοτικότητα (**efficiency)** που δίνεται από τον τύπο:

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

Τα μεγέθη αυτά τα χρησιμοποιούμε σε κάθε παράλληλο πρόγραμμα και μεταβάλλονται στην πλειοψηφία των περιπτώσεων ανάλογα με τον αριθμό επεξεργαστών που διαθέτουμε. πχ(σχήμα 4.7)



Σχήμα 2.1**[17]** Παράδειγμα μεταβολής Speedup~Processor count

2.4.2.Νομοι Amdahl - Gustafson

Δυστυχώς η προσπάθεια μας δεν γίνεται να έχει άπειρη βελτίωση. Από το 1960 ο Gene Amdahl κατέγραψε ένα πάρα πολύ σημαντικό συμπέρασμα το οποίο είναι γνωστό ως Amdahl's Law.
Κατέγραψε ότι, εάν όλο το πρόβλημα δεν γίνεται να γραφτεί σε απόλυτα παράλληλη μορφή τότε το συνολικό speedup έχει ανώτατο όριο και μάλιστα χαμηλό. Εάν για παράδειγμα υποθέσουμε ότι το 90% του κώδικα παραλληλοποιείται και το 10% δεν παραλληλοποιείται τότε το μέγιστο speedup δίνεται από:

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Κατά συνέπεια ακόμα και αν το p είναι άπειρο έχουμε μέγιστο speedup 10! Αυτό είναι ένα μεγάλο αγκάθι στην παράλληλη επεξεργασία. Αυτό όμως δεν σημαίνει ότι πρέπει να αφήνουμε την προσπάθεια. Ένα Speedup ακόμα και 2 πολλές φορές είναι πολύ μεγάλη βελτίωση στον χρόνο.

Επιπλέον ο νόμος Amdahl δεν λαμβάνει υπόψη του ένα σημαντικό ζήτημα το οποίο είναι το μέγεθος του προβλήματος και λαμβάνεται από τον νόμο Gustafson.

$$S(P) = P - \alpha \cdot (P - 1)$$

Αν P είναι ο αριθμός των επεξεργαστών, και α το τμήμα που δεν παραλληλοποιείται, τότε ο νόμος μας λέει ότι, S(p) είναι ο λόγος του χρόνου που χρειάζεται η σειριακή μηχανή προς τον χρόνο που χρειάζεται μία επεξεργαστική μονάδα του παράλληλου συστήματος και με λίγα λόγια λέει ότι, λόγο του αυξανόμενου μεγέθους των δεδομένων οι προγραμματιστές μπορούνε να χρησιμοποιήσουν την παραλληλία για να λύσουν στον ίδιο χρόνο μεγαλύτερα προβλήματα και κατά συνέπεια οι συνέπειες του νόμου του Amdahl περιορίζονται σε κάποιο βαθμό.

2.4.3 Scalability - Χρονισμοί

Ένα επιπλέον μέγεθος που πρέπει να έχουμε υπόψη είναι το scalability. Αυτό το βασικό μέγεθος μπορεί να γίνει κατανοητό με το εξής παράδειγμα. Έστω ότι έχουμε ένα πρόγραμμα με συγκεκριμένο αριθμό threads/Cpu και συγκεκριμένο μέγεθος εισόδου. Έστω ότι έχουμε speedup S. Αν τώρα αυξήσουμε των αριθμό των επεξεργαστών και μπορούμε να βρούμε μια είσοδο ώστε το πρόγραμμα να παραμένει με speedup S τότε λέμε ότι το πρόβλημα είναι **Scalable.**

Χρονισμοί

Τέλος δεν πρέπει να αμελήσουμε την σημασία της χρονομέτρησης των προγραμμάτων μας για να βρούμε τα T serial και T parallel.Υπάρχουν διάφοροι ορισμοί εδώ αλλά στα πλαίσια της διπλωματικής θα εργαστούμε με Wall time, δηλαδή τον χρόνο που πέρασε για εμάς από την αρχή μέχρι το τέλος του προγράμματος.

2.5) Σχεδιασμός παράλληλων προγραμμάτων

Σαν προγραμματιστές λοιπόν αναζητούμε έναν τρόπο να παραλληλοποιήσουμε ένα πρόγραμμα που γνωρίζουμε τον σειριακό του κώδικα. Γνωρίζουμε πχ ότι Θα προσπαθήσουμε να χωρίσουμε την εργασία σε ίσα κομμάτια. Γνωρίζουμε επίσης ότι θα πρέπει κάπως να ορίσουμε την επικοινωνία μεταξύ των Cpus. Δυστυχώς ακριβής μέθοδος δεν υπάρχει και η εμπειρία παίζει πολύ συχνά πολύ σημαντικό ρόλο. Υπάρχει όμως ένα σύνολο βημάτων που είναι γνωστό ως μεθοδολογία Foster **[11]** που θα χρησιμοποιηθεί σαν πρώτη σκέψη στην σχεδίαση του παράλληλου κώδικα.

Ορίζουμε λοιπόν τα εξής βήματα.

1. **Partitioning**. Διαχωρίζουμε τις διεργασίες και τα δεδομένα σε μικρότερες διεργασίες με στόχο να βρούμε τις διεργασίες που μπορούν να εκτελεστούν παράλληλα.
2. **Communication**.Ορίζουμε το σύνολο της επικοινωνίας που θα υπάρξει μεταξύ των Cpus.
3. **Aggregation**.Συνδυασμός των 2 παραπάνω βημάτων σε μεγαλύτερες διεργασίες.
4. **Mapping**.Ορίζουμε τις διεργασίες από τα προηγούμενα σε κάθε Cpu με στόχο την ελαχιστοποίηση της επικοινωνίας και το βέλτιστο Load Balancing.

Το βήμα 2 είναι αυτό που έχει ενδιαφέρον. Χρειαζόμαστε έναν τρόπο να επικοινωνούν οι Cpu μεταξύ τους και αυτό που θα χρησιμοποιήσουμε στα πλαίσια της εργασίας είναι το MPI.

2.6) Εισαγωγή στο MPI

2.6.1 Ορισμός του MPI

Το MPI (αρχικά από το Message Passing Interface) είναι ένα πρωτόκολλο επικοινωνίας υπολογιστών που χρησιμοποιείται ευρέως στις συστοιχίες υπολογιστών για παράλληλο προγραμματισμό. Αποτελείται από έναν αριθμό διεργασιών από τις οποίες η καθεμία δουλεύει με κάποια τοπικά δεδομένα. Κάθε διεργασία έχει τοπικές μεταβλητές και δεν υπάρχει κανένας τρόπος να αποκτήσει μια διεργασία άμεση πρόσβαση στη μνήμη μιας άλλης. Το μοίρασμα των δεδομένων μεταξύ διεργασιών γίνεται στέλνοντας και λαμβάνοντας ρητά τα δεδομένα. Οι διεργασίες δεν είναι απαραίτητο να τρέχουν σε διαφορετικό επεξεργαστή η καθεμία, αλλά αυτό συμβαίνει στην συντριπτική πλειοψηφία των περιπτώσεων, οπότε θα θεωρήσουμε ότι κάθε διεργασία αντιστοιχεί σε ένα και μόνο επεξεργαστή. Το MPI είναι μια βιβλιοθήκη συναρτήσεων (στη C/C++) ή ρουτινών (σε Fortran) που εισάγονται στον πηγαίο κώδικα ενός προγράμματος για να επιτρέψουν την επικοινωνία μεταξύ διεργασιών. Το πρότυπο MPI-1 ορίστηκε την άνοιξη του 1994 και καθορίζει τις προδιαγραφές που πρέπει να τηρούν οι διάφορες υλοποιήσεις του MPI, έτσι ώστε προγράμματα που χρησιμοποιούν το MPI να μπορούν να μεταγλωττιστούν και να εκτελεστούν σε κάθε πλατφόρμα που υποστηρίζει το πρότυπο. Όμως η αναλυτική υλοποίηση της βιβλιοθήκης αφήνεται σε ανεξάρτητες εταιρίες που είναι ελεύθερες να παράγουν βελτιστοποιημένες εκδόσεις για τα προϊόντα τους. Οι γλώσσες που υποστηρίζονταν αρχικά ήταν η C και η Fortran77.
Αργότερα ορίστηκαν τα πρότυπα MPI-2 και το MPI-3 (2012), στα οποία προστέθηκαν επιπλέον χαρακτηριστικά, όπως συναρτήσεις για C++, R και Fortran90, καθώς και εργαλεία για παράλληλο χειρισμό αρχείων. Οι στόχοι του MPI είναι η υψηλή απόδοση, η επεκτασιμότητα και η μεταφερσιμότητα **[16].**

2.6.2 Κλήσεις προγραμμάτων MPI

Τα προγράμματα MPI αποτελούνται από έναν αριθμό διεργασιών, οι οποίες εκτελούνται παράλληλα και επικοινωνούν μεταξύ τους μέσω κλήσεων βιβλιοθήκης (library calls). Αυτές οι κλήσεις μπορούν χονδρικά να χωριστούν σε τέσσερις τάξεις:

1)Κλήσεις που χρησιμοποιούνται για να αρχικοποιήσουν, διαχειριστούν και τελικά να τερματίσουν επικοινωνίες.
2)Κλήσεις που χρησιμοποιούνται για να επικοινωνήσουν μεταξύ ζευγαριών επεξεργαστών.
3)Κλήσεις που χρησιμοποιούνται για διαδικασίες επικοινωνίας μεταξύ ομάδων επεξεργαστών.
4)Κλήσεις που χρησιμοποιούνται για να δημιουργήσουν αυθαίρετους τύπους δεδομένων.

Η 1η τάξη των κλήσεων αποτελείται από κλήσεις για να ξεκινήσουν την επικοινωνία, να αναγνωρίσουν τον αριθμό των επεξεργαστών που χρησιμοποιείται, να δημιουργήσουν υποομάδες επεξεργαστών και να αναγνωρίσουν ποιος επεξεργαστής τρέχει μια συγκεκριμένη διεργασία.
Η 2η τάξη των κλήσεων, που ονομάζεται point-to-point communication operations, αποτελείται από διαφορετικούς τύπους διαδικασιών αποστολής και λήψης δεδομένων.

Η 3η τάξη των κλήσεων αποτελείται από συλλογικές διαδικασίες που προσφέρουν συγχρονισμό ή ορισμένους τύπους επικοινωνίας μεταξύ ομάδων επεξεργαστών και κλήσεις που εκτελούν διαδικασίες υπολογισμών.
Η 4η τάξη των κλήσεων προσφέρει ευελιξία στο χειρισμό πολύπλοκων δομών δεδομένων.

2.6.3 MPI C bindings

Όπως αναφέρθηκε, υπάρχουν υλοποιήσεις του MPI για C, και Fortran. Στη συνέχεια θα ασχοληθούμε με τις υλοποιήσεις για C καθώς είναι αυτές που χρησιμοποιήθηκαν στην εργασία.
Για να προγραμματίσουμε με χρήση MPI, είναι απαραίτητο να κάνουμε τη δήλωση «*#include "mpi.h"*» στο αντίστοιχο τμήμα του πηγαίου κώδικα του προγράμματος. Στο αρχείο κεφαλίδας mpi.h εμπεριέχονται οι ορισμοί των σταθερών και των δομών δεδομένων καθώς και τα πρωτότυπα των συναρτήσεων του MPI στη C, οι συναρτήσεις του MPI ξεκινούν με το πρόθεμα MPI_.

2.6.4 Τύποι Δεδομένων του MPI

Οι βασικοί τύποι δεδομένων του MPI είναι οι ίδιοι με τους βασικούς τύπους δεδομένων της κάθε γλώσσας που υποστηρίζει, απλώς έχουν διαφορετικά ονόματα. Στον παρακάτω πίνακα φαίνονται οι τύποι δεδομένων του MPI για C. Οι τύποι δεδομένων BYTE και PACKED δεν αντιστοιχούν σε κάποιο τύπο της C. Το BYTE παριστάνει 8 δυαδικά ψηφία και χρησιμοποιείται όταν θέλουμε να αποστείλουμε μηνύματα με bit, όπου η τιμή κάθε bit παριστάνει της τιμή μιας σημαίας (flag). Αυτός ο τύπος μπορεί να χρησιμοποιηθεί για την αποστολή πληροφοριών ελέγχου στον παραλήπτη. Αν και μπορούμε να αποστείλουμε πληροφορίες ελέγχου και με άλλους τύπους δεδομένων, το BYTE είναι μικρότερο σε μέγεθος και έτσι η μετάδοση γίνεται γρηγορότερα. Ο τύπος PACKED χρησιμοποιείται για την αποστολή και λήψη συνεπτυγμένων μηνυμάτων.

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

Τύπο δεδομένων στο MPI **[16]**

2.7) Χρήση του MPI

2.7.1 Αρχικοποίηση του MPI

Η αρχικοποίηση μιας εφαρμογής MPI γίνεται με τη συνάρτηση MPI_Init() και το πρότυπο της είναι
int MPI_Init(int *argc, char ***argv) για τη C.

Όλα τα προγράμματα του MPI πρέπει να έχουν μια κλήση σε αυτή τη συνάρτηση. Η συνάρτηση καλείται μια φορά, συνήθως στην αρχή του προγράμματος και πρέπει να κληθεί πριν από κάθε άλλη κλήση συνάρτησης MPI. Το argc (argument vector) είναι ο αριθμός των παραμέτρων που δίνονται στη γραμμή εντολών και το argv ένας πίνακας χαρακτήρων που περιέχει αυτές τις παραμέτρους. Μετά την κλήση της MPI_Init, όλες οι διεργασίες που δημιουργήθηκαν θα έχουν ένα αντίγραφο αυτών των παραμέτρων. Εναλλακτικά, αν δε θέλουμε να περάσουμε ορίσματα στη C χρησιμοποιούμε MPI_Init(NULL,NULL).

2.7.2 Τερματισμός του MPI

Ο τερματισμός του MPI γίνεται με τη συνάρτηση MPI_Finalize(). Το πρότυπο της είναι
int MPI_Finalize(void) για τη C και Η MPI_Finalize() καθαρίζει όλες τις δομές δεδομένων του MPI και ακυρώνει τις λειτουργίες που δεν ολοκληρώθηκαν. Πρέπει να κληθεί από όλες τις διεργασίες, αλλιώς το πρόγραμμα θα κολλήσει. Η συνάρτηση αυτή δε δέχεται παραμέτρους και πρέπει να είναι η τελευταία που καλείται στο πρόγραμμα.

2.7.3 Communicators

Ο communicator είναι ένας χειριστής (handler) του MPI που αντιπροσωπεύει μια ομάδα διεργασιών που μπορούν να επικοινωνήσουν μεταξύ τους. Ένα όνομα communicator απαιτείται ως όρισμα σε όλες τις διαδικασίες point-to-point και collective επικοινωνιών. Μπορούν να υπάρξουν πολλοί communicators σε ένα πρόγραμμα, ενώ μια διεργασία μπορεί να είναι μέλος σε παραπάνω από έναν communicators. Σε κάθε communicator, οι διεργασίες αριθμούνται διαδοχικά, αρχίζοντας από το μηδέν. Αυτός ο αναγνωριστικός αριθμός λέγεται τάξη (rank) της διεργασίας σε αυτόν τον communicator. Από τα παραπάνω συμπεραίνουμε πως αν μια διεργασία συμμετέχει σε παραπάνω από έναν communicators, το rank της σε καθέναν μπορεί να είναι, και συνήθως είναι, διαφορετικό. Για τα περισσότερα προγράμματα MPI, αρκεί ένας βασικός communicator, ο MPI_COMM_WORLD.

Αυτός ο communicator δημιουργείται αυτόματα από το MPI και περιέχει όλες τις διεργασίες, οπότε αν τον χρησιμοποιήσουμε μπορούμε να επικοινωνήσουμε με οποιονδήποτε διεργασία. Επίσης μπορούμε να ορίσουμε επιπλέον communicators που αποτελούνται από υποσύνολα των διαθέσιμων διεργασιών, έτσι ώστε να οργανώσουμε τις διεργασίες σε διαφορετικές τοπολογίες με σκοπό την απλοποίηση των αλγορίθμων ορισμένων προγραμμάτων.

Μέγεθος του Communicator

Κάθε διεργασία μπορεί να αναγνωρίσει το μέγεθος (size), δηλαδή τον αριθμό των διεργασιών ενός communicator στον οποίο ανήκει, με μια κλήση στη συνάρτηση MPI_Comm_size. Το πρότυπο της συνάρτησης αυτής είναι:
`int` MPI_Comm_size(MPI_Comm comm, int *size) για τη C και comm είναι ο communicator το μέγεθος του οποίου επιστρέφεται στο size. Σαν παράμετρο πρέπει να περάσουμε τη διεύθυνση μνήμης του size.

Τάξη διεργασίας

Με τη συνάρτηση MPI_Comm_rank μπορούμε να βρούμε την τάξη της τρέχουσας διαδικασίας σε ένα communicator.

Το πρότυπο της συνάρτησης είναι `int` MPI_Comm_rank(MPI_Comm comm, int *rank) στη C όπου comm είναι ο communicator στον οποίο ανήκει η διεργασία, ενώ στη μεταβλητή rank επιστρέφεται η τάξη της διεργασίας.

Οι τιμές που επιστρέφονται από τις συναρτήσεις MPI_Comm_size και MPI_Comm_rank συχνά χρησιμοποιούνται για να διαιρέσουμε ένα πρόβλημα ανάμεσα στις διεργασίες. Έτσι δεν είναι ανάγκη να ξέρουμε τη στιγμή που γράφουμε το πρόβλημα πόσους επεξεργαστές έχουμε διαθέσιμους, αλλά μπορούμε να πάρουμε την απόφαση τη στιγμή που θα εκτελέσουμε το πρόγραμμα. Για παράδειγμα, αν υποθέσουμε πως η MPI_Comm_size μας επιστρέφει την τιμή 5, ξέρουμε ότι κάθε διεργασία έχει να λύσει το 1/5 ενός υποτιθέμενου προβλήματος, αν φυσικά έχουμε γράψει τον κώδικα κατάλληλα. Στη συνέχεια, κάθε διεργασία μπορεί να υπολογίσει την τάξη της με χρήση της MPI_Comm_rank ώστε να αποφασίσει ποιο κομμάτι του προβλήματος θα επιλύσει.

2.8) Μεταφορά δεδομένων στο MPI

Το κύριο χαρακτηριστικό του message passing είναι πως η μεταφορά των δεδομένων από μια διεργασία σε άλλη απαιτεί λειτουργίες και από τις δυο διεργασίες. Αυτή η διαδικασία πραγματοποιείται με τις συναρτήσεις MPI_Send και MPI_Recv.

2.8.1 Αποστολή δεδομένων

Η αποστολή δεδομένων από μια διεργασία σε άλλη γίνεται με την κλήση της MPI_Send από την διεργασία που στέλνει τα δεδομένα. Το πρότυπο της MPI_Send είναι
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm).

Κάθε κλήση στην MPI_Send πρέπει να αντιστοιχεί σε μια αντίστοιχη κλήση στην MPI_Recv στην λαμβάνουσα διεργασία. Οι τρεις πρώτες παράμετροι της συνάρτησης χρησιμοποιούνται για να προσδιορίσουν τα αποστελλόμενα δεδομένα. Το buf αντιστοιχεί στη διεύθυνση του πρώτου στοιχείου που αποστέλλεται, το count στον αριθμό των στοιχείων που αποστέλλονται και το datatype στον τύπο δεδομένων που αποστέλλονται. Το datatype θα έχει ως τιμή έναν από τους τύπους δεδομένων του MPI που αναφέρθηκαν σε προηγούμενη παράγραφο. Το dest είναι ένας ακέραιος που αντιστοιχεί στην τάξη της διεργασίας που θα λάβει τα δεδομένα. Το tag είναι ένας ακέραιος που χρησιμοποιείται για να διακρίνει πολλαπλά μηνύματα.

2.8.2 Λήψη Δεδομένων

Η λήψη δεδομένων γίνεται με την κλήση της συνάρτησης MPI_Recv από τη διεργασία που λαμβάνει τα δεδομένα. Το πρότυπο της συνάρτησης είναι
`int` MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status).

Οι παράμετροι της συνάρτησης είναι οι ίδιες με την προηγούμενη συνάρτηση, εκτός από την προσθήκη status. Στο C binding, το MPI_Status είναι ένας ορισμός τύπου για μια δομή (struct) που αποθηκεύει πληροφορίες σχετικά με το μήνυμα που λήφθηκε. Αποτελείται από 3 πεδία, τα MPI_SOURCE, MPI_TAG και MPI_ERROR, τα οποία περιέχουν την πηγή, το tag και τον κωδικό σφάλματος αντίστοιχα. Με τη συνάρτηση MPI_Recv μπορούμε να χρησιμοποιήσουμε μια μεταβλητή «μπαλαντέρ» (wildcard) για την πηγή ή το tag του μηνύματος, τις MPI_ANY_SOURCE και MPI_ANY_TAG, ώστε να λάβουμε το μήνυμα από οποιαδήποτε πηγή ή με οποιοδήποτε tag και να καθορίσουμε την πηγή και το tag από τα αντίστοιχα πεδία της status.

Κάτι που πρέπει να τονιστεί είναι ότι οι συναρτήσεις MPI_Send και MPI_Recv αναστέλλουν και οι δυο την εκτέλεση του προγράμματος μέχρι να εκτελεστούν (blocking calls). Για παράδειγμα, αν επιχειρήσουμε να λάβουμε δεδομένα που δεν έχουν αποσταλεί ακόμη, η διεργασία θα ανασταλεί μέχρι να αποσταλούν και μετά θα συνεχίσει την εκτέλεση της. Αυτή η λειτουργία μπορεί να οδηγήσει το πρόγραμμα σε αδιέξοδο(deadlock), όταν 2 ή περισσότερες διεργασίες περιμένουν την άλλη να εκτελεστεί για να προχωρήσουν και πρέπει ο προγραμματιστής να φροντίσει ώστε να μην δημιουργηθεί.

2.9) Εξειδικευμένες συναρτήσεις MPI

Οι βασικές συναρτήσεις που παρουσιάστηκαν παραπάνω επαρκούν για την διεκπεραίωση των περισσότερων λειτουργιών, άρα είναι εφικτό να χρησιμοποιήσουμε μόνο αυτές στον κώδικα μας. Το πρότυπο MPI όμως περιλαμβάνει πάνω από 100 επιπλέον συναρτήσεις, που δεν είναι πάντα απαραίτητες, αλλά μπορούν να αυξήσουν την απόδοση ή τη λειτουργικότητα του προγράμματος.

Broadcast communications

Η συνάρτηση MPI_Bcast προσφέρει ένα μηχανισμό για να διανείμουμε την ίδια πληροφορία μέσα σε ένα communicator και πρότυπο της είναι:
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm).
Επίσης υπάρχει η συνάρτηση MPI_Reduce, η οποία συλλέγει δεδομένα από όλες τις προκαθορισμένες διεργασίες με βάση έναν τελεστή που επιλέγουμε.
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, introot, MPI_Comm comm).
Η συνάρτηση MPI_Gather συλλέγει και αυτή δεδομένα, αλλά αντίθετα από την MPI_Reduce, δεν εφαρμόζει σε αυτά κάποιο τελεστή αλλά τα αποθηκεύει όλα και έχει πρότυπο:
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) .
Η συνάρτηση MPI_Scatter επιτελεί την αντίθετη διαδικασία, δηλαδή στέλνει διαφορετικά δεδομένα σε όλες τις διεργασίες ενός communicator και έχει πρότυπο:
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm) .

Non-blocking Send και Receive

Όπως είχαμε αναφέρει, οι συναρτήσεις MPI_Send και MPI_Receive αναστέλλουν την εκτέλεση του προγράμματος μέχρι να εκτελεστούν. Υπάρχουν όμως και εκδόσεις αυτών των συναρτήσεων που δεν αναστέλλουν την εκτέλεση του προγράμματος αλλά είναι πιο περίπλοκες στη χρήση τους, οι MPI_Isend και MPI_Irecv. Επίσης υπάρχουν διάφορες παραλλαγές της MPI_Send, που επιτρέπουν συγχρονισμό της αποστολής με τη λήψη (synchronous send) ή επιτρέπουν το buffering.

Ανταλλαγή δεδομένων μεταξύ δυο διεργασιών

Τέλος θέλουμε να στείλουμε δεδομένα από μια διεργασία σε μια άλλη και μετά η άλλη διεργασία να στείλει και αυτή δεδομένα στην αρχική, μπορούμε να χρησιμοποιήσουμε τη συνάρτηση MPI_Sendrecv. Αυτή η συνάρτηση έχει πρότυπο

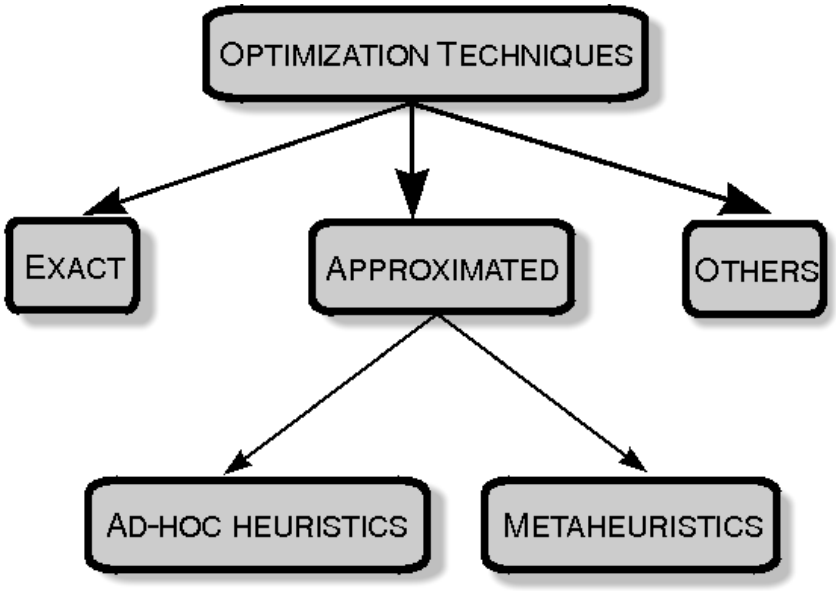
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)

ΚΕΦΑΛΑΙΟ 3 :ΠΡΟΒΛΗΜΑΤΑ ΒΕΛΤΙΣΤΟΠΟΙΗΣΗΣ

3.1)Εισαγωγή στα προβλήματα βελτιστοποίησης

Στα μαθηματικά και στην επιστήμη των υπολογιστών, ονομάζουμε πρόβλημα βελτιστοποίησης το πρόβλημα της αναζήτησης της βέλτιστης λύσης από ένα σύνολο πιθανών λύσεων **[13]**

Υπάρχουν αρκετές τεχνικές στην βιβλιογραφία και το επόμενο διάγραμμα κάνει έναν βασικό διαχωρισμό **[18]**.



Σχήμα 3.1**[18]** Διαχωρισμός τεχνικών αναζήτησης

Υπάρχει λοιπόν η πρώτη κατηγορία των ακριβών αλγορίθμων (exact algorithms) οι οποίοι εγγυώνται να βρουν την βέλτιστη λύση σε ένα πρόβλημα. Δυστυχώς το σύνηθες μέγεθος των προβλημάτων καθιστά πρακτικά αδύνατη τη χρήση αυτών των αλγορίθμων στους οποίους ο χρόνος αναζήτησης αυξάνεται εκθετικά ανάλογα με το μέγεθος του προβλήματος. Κατά συνέπεια, τις τελευταίες δεκαετίες υπάρχει αυξημένο ενδιαφέρον για προσεγγιστικές τεχνικές στις οποίες κάποιο μικρό σφάλμα είναι αποδεκτό στην αναζήτηση λύσης με σκοπό να μειωθεί αισθητά ο χρόνος εύρεσης αποδεκτής λύσης **[18]**

Αυτοί οι αλγόριθμοι χωρίζονται σε δυο κατηγορίες.

- Η πρώτη κατηγορία , τα ευρετικά (***heuristics)***, είναι ένα σύνολο αλγορίθμων που έχουν στόχο να λύσουν μεγάλα προβλήματα σε λίγο χρόνο ,χρησιμοποιώντας κάποια τεχνική συντόμευση στην αναζήτηση της λύσης. Συνήθως χρησιμοποιούν κάποια επιπλέον γνώση για το πρόβλημα (πχ πρόβλεψη) ή όπως θα δούμε στην εργασία μας έχουν τμήματα τα οποία χρησιμοποιούν στοχαστικούς τελεστές για να οδηγηθούν προς τις λύσεις.

- Η δεύτερη κατηγορία τα μεταευρετικά (***metaheuristics)*** είναι συνδυασμός τεχνικών της πρώτης σε ένα ανώτερο επίπεδο όπου γίνεται επιλογή για το εάν ένα ευρετικό θα δώσει αξιόπιστες λύσεις

Τα μεταευρετικά χωρίζονται επιπλέον σε δυο κατηγορίες ανάλογα με τον τρόπο που προσεγγίζουν τις λύσεις.

Η πρώτη κατηγορία ονομάζεται **T*rajectory based*** όπου εργαζόμαστε με μία λύση και προσπαθούμε να αναζητήσουμε καλύτερες στην γειτονιά της στο χώρο του προβλήματος.
Η δεύτερη κατηγορία ονομάζεται ***Population based*** και με αυτήν θα εργαστούμε στα πλαίσια της εργασίας. Εργαζόμαστε με ένα σύνολο από πιθανές λύσεις και σε κάθε βήμα του επαναληπτικού αλγορίθμου ο πληθυσμός βελτιώνεται σαν σύνολο με σκοπό την αναζήτηση της λύσης.

Οι ακριβείς αλγόριθμοι όπως αναφέραμε έχουν το πλεονέκτημα ότι βρίσκουν πάντα το ολικό ακρότατο. Όταν προσπαθούν να λύσουν ***NP-Hard*** ή ***NP-Complete*** προβλήματα τότε ο χρόνος εκτέλεσης αυξάνεται εκθετικά ανάλογα με το μέγεθος του προβλήματος προς λύση. Χωρίς πολλές μαθηματικές λεπτομέρειες θα αναφέρω απλά ότι όταν μιλάμε για ***NP-Hard*** και ***NP-Complete*** εννοούμε προβλήματα τα στα οποία η εύρεση της λύσης σε λογικό χρονικό περιθώριο από μια ντετερμινιστική μηχανή δεν είναι δυνατή. Στον αντίποδα, τα ***heuristics*** και τα ***metaheuristics*** βρίσκουν γρήγορες λύσεις σε αυτά τα προβλήματα αλλά στην πλειοψηφία τους οι λύσεις αυτές δεν είναι βέλτιστες. Μια υποκατηγορία αυτών των μεταευρετικών είναι και οι λεγόμενοι εξελικτικοί αλγόριθμοι με τους οποίους θα εργαστούμε στα πλαίσια της εργασίας μας. Στην επόμενη παράγραφο θα ασχοληθούμε με κάποιους βασικούς αλγοριθμικούς-μαθηματικούς ορισμούς που θα μας απασχολήσουν.

3.2)Μαθηματικοί Ορισμοί

Ορισμός 1:

Ένα πρόβλημα βελτιστοποίησης ορίζεται από ένα ζεύγος (S,f) οπου S ≠0 και αντιπροσωπεύει τον χώρο των λύσεων ή χώρο αναζήτησης ενός προβλήματος και f είναι ένα κριτήριο ποιότητας
(Objective function) που ορίζεται σαν

$$F: S \rightarrow R$$

Κατά συνέπεια η επίλυση ενός προβλήματος βελτιστοποίησης αποτελείται από την εύρεση ενός συνόλου μεταβλητών έστω $i^* \in S$ ώστε :

$$f(i^*) \leq f(i),\ \forall i \in S$$

Υποθέτοντας ότι στην γενική περίπτωση οι έννοιες του ελαχίστου και του μεγίστου είναι αντίθετες, τότε ανάλογα με το πεδίο ορισμού S χωρίζουμε τα προβλήματα σε:

- Δυαδικά $S \subseteq B$ **(binary)**
- Διακριτά $S \subseteq N$ **(discrete)**
- Συνεχή $S \subseteq R$ **(complete)**
- Μεικτά $S \subseteq B \cup N \cup R$ **(mixed)**

Αναζητώντας λοιπόν τη λύση σε ένα πρόβλημα βελτιστοποίησης είναι προφανές ότι πρέπει να ορίσουμε κριτήριο απόστασης για τον χώρο της αναζήτησης για να μπορέσουμε να ορίσουμε το κόστος.

Ορισμός 2:

Δυο λύσεις είναι κοντά η μία στην άλλη όταν βρίσκονται στην ίδια γειτονιά. Ορίζουμε λοιπόν μια γειτονιά στο χώρο S

$$N : S \rightarrow S$$

Τέτοια ώστε για κάθε λύση $i \in S$ ορίζεται ένα σετ $Si \subseteq S$ ώστε εάν το **i** είναι στην γειτονία του **j** τότε ισχύει και αντίστροφα και το **j** είναι στην γειτονιά του **i** δηλαδή $i: j \in Si$ εάν $i \in Sj$.

Γενικά, στα πολύπλοκα προβλήματα η αντικειμενική συνάρτηση παρουσιάζει μια λύση η οποία είναι βέλτιστη μόνο για μια συγκεκριμένη γειτονιά. Ορίζουμε λοιπόν την έννοια του τοπικού ακρότατου.

Ορισμός 3:

Ορίζουμε λοιπόν για ένα πρόβλημα (S,f) και $Si \subseteq S$ την γειτονιά της λύσης i ∈ Si τότε βρισκόμαστε σε τοπικό ακρότατο αν ικανοποιείται η εξής ανισότητα

$$f(i^*) \leq f(i), \ \forall i \in Si$$

Ορισμός 4:

Στα πραγματικά προβλήματα βελτιστοποίησης, συνήθως υπάρχει και μια σειρά από περιορισμούς **(Constraints)** οι οποίοι μειώνουν το μέγεθος του χώρου S. Δεδομένου ενός προβλήματος (S,f) ορίζουμε $M = \{i \in S \mid gk(i) \geq 0, \forall k \in [1, \ldots, q]\}$ την περιοχή δυνατών λύσεων. Οι συναρτήσεις $gk : S \rightarrow R$ ονομάζονται περιορισμοί (**Constraints)** και ανάλογα με τις τιμές i υπάρχουν οι εξής ονομασίες

$$Satisfied: \Leftrightarrow gk(i) \geq 0$$
$$Active: \Leftrightarrow gk(i) = 0$$
$$Inactive: \Leftrightarrow gk(i) > 0,$$
$$Violated: \Leftrightarrow gk(i) < 0$$

3.3)Το πρόβλημα χρονοπρογραμματισμού των εξετάσεων

Το πρόβλημα του χρονοπρογραμματισμού των εξετάσεων (**Exam timetabling Problem**), που θα αναφέρεται σαν **ETP** στην συνέχεια της εργασίας, είναι μια από τις πιο σημαντικές δραστηριότητες που λαμβάνουν χώρα σε όλα τα ακαδημαϊκά ιδρύματα του κόσμου. Στις επόμενες σελίδες θα γίνει μια σύντομη εισαγωγή στο πρόβλημα και στην έρευνα που γίνεται για την επίλυση του κυρίως τα τελευταία 10 χρόνια. Στην συνέχεια, θα εμβαθύνουμε περισσότερο στο πρόβλημα το οποίο πραγματεύεται αυτή η διπλωματική.

Ορισμός του προβλήματος

Τα προβλήματα του χρονοπρογραμματισμού εμφανίζονται συχνά και σε πολλές μορφές. Χαρακτηριστικά παραδείγματα από τη βιβλιογραφία βρίσκονται σε:

- Ακαδημαϊκά περιβάλλοντα **[1]**
- Διαχείριση νοσηλευτικού προσωπικού **[2]**
- Αθλητικά περιβάλλοντα **[3]**
- Προβλήματα μεταφορών **[4]**

Οι Burke, Kingston, και De Werra (2004) **[1]** έδωσαν έναν πολύ καλό ορισμό για το πρόβλημα του χρονοπρογραμματισμού.

Ένα πρόβλημα χρονοπρογραμματισμού είναι ένα πρόβλημα 4 παραμέτρων :

- ***T*** *ένα πεπερασμένο σύνολο χρόνοθυρίδων* ***(timeslot)***
- ***P*** *ένα πεπερασμένο σύνολο πόρων* ***(resource)***
- ***M*** *ένα πεπερασμένο σύνολο από εξετάσεις* ***(meetings-exams)***
- ***C*** *ένα πεπερασμένο σύνολο από περιορισμούς* ***(constraints)***

Το πρόβλημα πλέον είναι να εκχωρήσουμε χρόνο και πόρους στο σύνολο των εξετάσεων ικανοποιώντας τους περισσότερους δυνατούς περιορισμούς. Όπως θα δούμε και στη συνέχεια αυτό είναι ένα πολύ δύσκολο πρόβλημα και μάλιστα είναι **NP-Complete** με 5 διαφορετικούς τρόπους οπότε το να το λύσουμε με ακριβείς μεθόδους λόγου του μεγέθους του είναι πρακτικά αδύνατο.**[19]**
Όπως είδαμε, ένα ETP μπορεί να οριστεί σαν την εκχώρηση ενός συνόλου από εξετάσεις **E {e1,e2…},** στο ταξινομημένο σύνολο χρόνων **T {t1,t2..}** σε αίθουσες συγκεκριμένης χωρητικότητας (πόρος)
C {C1,C2..}, ικανοποιώντας παράλληλα ένα σύνολο από περιορισμούς (constraints). Αυτά τα constraints αναλύονται σε δύο κατηγορίες.

HARD CONSTRAINTS

Με αυτό τον ορισμό, ο οποίος μεταφράζεται σαν αυστηρή δέσμευση, ονομάζουμε το σύνολο των δεσμεύσεων που δεν πρέπει να παραβιαστούν σε καμία περίπτωση.
Μια λύση που ικανοποιεί όλους τους σκληρούς περιορισμούς ονομάζεται υλοποιήσιμη **(feasible).**
Υπό πιο αυστηρή μαθηματική σκοπιά μπορούμε να πούμε για το πρόβλημα των επικαλυπτόμενων μαθημάτων ότι αν **D**ij είναι ο αριθμός των μαθητών που είναι γραμμένοι στις εξετάσεις **i,j** και **Xi ∈ T** τότε

xi ≠ xj για κάθε **i, j** στο **E , i ≠ j** και **Dij > 0**

SOFT CONSTRAINTS

Στον αντίποδα, υπάρχουν και οι λεγόμενοι μαλακοί περιορισμοί, δηλαδή περιορισμοί που επιθυμούμε να ικανοποιήσουμε αλλά δεν είναι υποχρεωτικό για την ύπαρξη της λύσης.

Το πρόβλημα συνήθως βρίσκεται σε αυτούς του περιορισμούς οι οποίοι πολλές φορές δεν είναι εφικτό να ικανοποιηθούν πλήρως ή και πολλές φορές είναι αντίθετοι ή και αμοιβαία αποκλειόμενοι με άλλους περιορισμούς.

Ακολουθούν δείγματα από την βιβλιογραφία όπου αναφέρονται παραδείγματα constrains **[5]**

PRIMARY HARD CONSTRAINTS

1)Να μην υπάρχουν εξετάσεις στις οποίες μαθητές θα πρέπει να είναι ταυτόχρονα σε 2 αίθουσες.
2)Οι πόροι να επαρκούν για την απαραίτητη εργασία, δηλαδή πχ ο αριθμός των μαθητών να μην υπερβαίνει την μέγιστη χωρητικότητα της αίθουσας.

Είναι προφανές ότι αυτοί οι περιορισμοί δεν είναι δυνατόν να παραβιαστούν για φυσικούς λόγους. Πρέπει λοιπόν η πιθανή λύση να τους ικανοποιεί υποχρεωτικά δηλαδή σύμφωνα με τον ορισμό μας **Gk(x)>=0**

PRIMARY SOFT CONSTAINTS

1)Οι εξετάσεις να είναι συνεχόμενες.
2)Μόνο εξετάσεις ίδιας διάρκειας μπορούν να βρίσκονται στην ίδια αίθουσα ταυτόχρονα.
3)Οι εξετάσεις που έχουν σύγκρουση (conflict) να διασκορπιστούν όσο καλύτερα γίνεται στα πρόγραμμα.
4)Ομαδοποίηση εξετάσεων που πρέπει να γίνουν την ίδια μέρα και ώρα στο ίδιο μέρος.
5)Ανώτατο όριο μαθητών σε κάθε χρονικό πλαίσιο (timeslot).
6)Κάποια μαθήματα να μην μπορούν να μπουν σε συγκεκριμένη ώρα.
7)Εξετάσεις ίδιας χρονικής διάρκειας να προγραμματίζονται την ίδια ώρα.

Μια τελευταία παρατήρηση είναι, ότι σαν πρόβλημα παρουσιάζει κοινά στοιχεία με το **traveling salesman problem** το οποίο επίσης δεν λύνεται με ακριβής μεθόδους.

3.4) Τεχνικές Προσέγγισης - Επίλυσης

Υπάρχουν πολλοί τρόποι προσέγγισης αυτού του προβλήματος. Υπάρχει πλήθος βιβλιογραφίας **[5](**p 8-11)και έρευνας σε παγκόσμιο επίπεδο. Υπάρχουν μάλιστα και ειδικές γλώσσες περιγραφής του προβλήματος όπως **ECLiP, TTL, UniLang, EASYLOCAL++** και πολλές άλλες. Θα γίνει λοιπόν μια γρήγορη ανασκόπηση των διαφόρων τεχνικών που έχουν χρησιμοποιηθεί από ερευνητές και έπειτα θα γίνει ανάλυση της προσπάθειας επίλυσης με χρήση γενετικού αλγορίθμου, που είναι και το αντικείμενο της διπλωματικής εργασίας. Για κάθε διαφορετική στρατηγική υπάρχει πληθώρα υλικού στο **[5]**(Chapter2).

- **Τεχνικές βασισμένες στην αναζήτηση σε γράφους-Graph based sequential Techniques**

Μία από τις πρώτες δημοσιεύσεις των Welsh and Powell **[7]** του 1967 ήταν μεγάλο βήμα για την επίλυση του ETP. Γεφύρωσε το χάσμα μεταξύ των τεχνικών χρωματισμού γραφημάτων και ETP και οδήγησε την περαιτέρω έρευνα σε ευρετικά γραφημάτων με καλά αποτελέσματα. Ανέφερε ότι σε ένα ETP, οι εξετάσεις μπορούν να αναπαρασταθούν σαν κορυφές ενός γραφήματος και οι αυστηροί περιορισμοί σαν τις ακμές που ενώνουν αυτές τις κορυφές. Ο χρωματισμός των κορυφών ώστε 2 διπλανές κορυφές να μην έχουν το ίδιο χρώμα αποτελεί το λεγόμενο πρόβλημα χρωματισμού γράφου **(Graph Coloring)** και είναι ισοδύναμο με την εκχώρηση timeslot για την εξέταση του κάθε μαθήματος. Ανάλογα με τον τύπο του ευρετικού που χρησιμοποιείται έχουμε διαφορετική στρατηγική επίλυσης του προβλήματος
(πχ neural met ,fuzzy logic)**[20][21].** Παραθέτουμε τέλος έναν πίνακα με τους βασικότερους τύπους ευρετικών που έχουν χρησιμοποιηθεί σε συνδυασμό με graph techniques για την επίλυση του προβλήματος.**[33]**

Heuristics	Ordering Strategy
Saturation Degree	Οι εξετάσεις με τα λιγότερα δυνατά timeslots προγραμματίζονται πρώτες
Largest Degree	Οι εξετάσεις με τις περισσότερες συγκρούσεις προγραμματίζονται πρώτες
Largest Weighted Degree	Ίδιο με LD αλλά λαμβάνει υπόψη και τον αριθμό των μαθητών
Largest Enrolment	Οι εξετάσεις με τους περισσότερους εγγεγραμμένους προγραμματίζονται πρώτες
Random Ordering	Τυχαία ταξινόμηση
Color Degree	Οι εξετάσεις με τις περισσότερες συγκρούσεις στο timeslot προγραμματίζονται πρώτες

- **Τεχνικές εκπλήρωσης περιορισμών -Constraint based techniques**

Ο λογικός προγραμματισμός περιορισμών **[8]** έχει τις πηγές του στο πεδίο της τεχνητής νοημοσύνης. Αυτές οι μέθοδοι προσέλκυσαν το ενδιαφέρον των μελετητών λόγο της ευκολίας και της ευελιξίας που παρουσιάζουν. Οι εξετάσεις μοντελοποιούνται σαν μεταβλητές σε πεπερασμένους τομείς. Στην συνέχεια αυτές οι μεταβλητές παίρνουν τιμές σειριακά και αφού αντιπροσωπεύουν μαθήματα και ώρες έχουμε λύση στο πρόβλημα. Παρόλο που γενικά είναι πολύ ακριβές υπολογιστικά αυτές οι λύσεις έχουν τουλάχιστον διδακτική αξία και έχουν οδηγήσει στην κατασκευή πολύ ισχυρών και εξειδικευμένων γλωσσών (**ECLiPSE, CHIP, OPL)**. Πολλές δημοσιεύσεις υπάρχουν και σε αυτό τον τομέα και μπορούν να βρεθούν στο **[5] (p 13-14).**

- **Τεχνικές βασισμένες σε τοπική αναζήτηση-Local search techniques**

Είναι μια κατηγορία τεχνικών οι οποίες αναζητούν λύσεις σε μια 'γειτονιά' μιας ήδη γνωστής μερικής λύσης. Η όλη διαδικασία οδηγείται εν γένει από μια γνωστή objective function η οποία χρησιμοποιείται για να αξιολογήσει της λύσεις που έχουν βρεθεί ήδη.
Ανάλογα με τον τρόπο που αναζητούν λύση στο χώρο καταστάσεων αυτά τα υπερευρετικά μπορούν να διαχωριστούν στις εξής δυο ομάδες.

1) Tabu Search. Αυτές οι τεχνικές ορίζουν μια νέα δομή που ονομάζεται λίστα tabu. Εκεί αποθηκεύονται περιορισμοί του χώρου οι οποίοι δίνονται από το πρόβλημα ,και επίσης αποθηκεύονται πρόσφατα αξιολογημένες λύσεις. Αν μια νέα πιθανή λύση παραβιάζει κάποιον κανόνα ή ο αλγόριθμος την έχει προσπελάσει ξανά στο πρόσφατο παρελθόν τότε αυτή θεωρείται tabu και ο αλγόριθμος συνεχίζει την αναζήτηση αλλού **[22].**

2)Simulated Annealing. Η βασικά ιδέα είναι ότι, όταν προχωράμε γρήγορα προς την λύση τότε η πιθανότητα να σταματήσουμε σε τοπικό ακρότατο αυξάνεται ταχύτατα**[23]**. Έτσι οι αλγόριθμοι αυτοί χρησιμοποιούν στοχαστικά μοντέλα για να περάσουν από τη μία κατάσταση στην άλλη και εν γένει βρίσκουν γρήγορα λύσεις που προφανώς είναι μη βέλτιστες ,αλλά είναι κοντά σε αυτές. Περισσότερα είδη και σχετική έρευνα μπορούμε να δούμε στο **[5]**(p18-19)

- **Αναζητήσεις μεταβλητούς γειτονιάς-Variable neighborhood search**

Με αυτή τη νέα σχετικά μεθοδολογία, οι αλγόριθμοι μελετάνε λύσεις σε μια κοντινή γειτονιά, αλλά ταυτόχρονα εξετάζουν γειτονιές που βρίσκονται μακριά από την λύση που έχουν μέχρι εκείνη τη στιγμή και αλλάζουν γειτονιά μόνο εάν βρεθεί βελτίωση. Οι πιο γνωστές μέθοδοι αυτής της κατηγορίας είναι **Kempe chain,** και **iterated local search.**

- **Αλγόριθμοι Αναζήτηση πολλαπλών στόχων**

Σε αυτού του είδους τις αναζητήσεις, δίνουμε αριθμητικό βάρος στον κάθε περιορισμό και με κάθε παραβίαση περιορισμού, το άθροισμα αυτών των βαρών, το οποίο είναι η ποινή μας για την λύση αυξάνεται. Σκοπός μας είναι η ελαχιστοποίηση αυτού του αθροίσματος. Η έρευνα σε αυτούς τους αλγορίθμους δεν είναι τόσο εκτεταμένη και υπάρχει αρκετός χώρος για νέους ερευνητές. **[12]**

- **Population Based Algorithms.**

Αυτή η κατηγορία μας αφορά ιδιαίτερα στα πλαίσια της εργασίας. Χωρίζεται σε 2 υποκατηγορίες.

1)Ant Algorithms. Αυτή η κατηγορία προσπαθεί να αντιγράψει τον τρόπο με τον οποίο τα μυρμήγκια βρίσκουν την συντομότερη διαδρομή προς την τροφή τους αφήνοντας πίσω τους ίχνη φερορμονών. Τα έντομα αρχικά ταξιδεύουν τυχαία αλλά μόλις βρουν τροφή επιστρέφουν στη φωλιά τους αφήνοντας ίχνος.

Όσο προχωράει ο αλγόριθμος τα μυρμήγκια σταματάνε να κινούνται τυχαία και κινούνται πάνω στα ίχνη που άφησαν τα άλλα μυρμήγκια. Επειδή αυτό το ίχνος εξασθενεί με το χρόνο, μετά από λίγο μόνο η συντομότερη διαδρομή παραμένει προς τον στόχο**[24].**

2)Γενετικοί και μιμητικοί αλγόριθμοι. Στα πλαίσια αυτής της διπλωματικής θα εργαστούμε με γενετικό αλγόριθμο. Κατά συνέπεια, η ανάλυση του θα γίνει σε μεταγενέστερο κεφάλαιο αναλυτικότερα. Σαν εισαγωγή μόνο θα αναφερθεί ότι οι γενετικοί αλγόριθμοι είναι ένα ισχυρό ευρετικό που πολλές φορές δίνει ταχύτατες λύσεις εκεί που πολλά ευρετικά δεν μπορούν. Πλήρης ανάλυση θα υπάρξει σε επόμενο κεφάλαιο.

3.5) University of Toronto Benchmark Data

Στα πλαίσια της διπλωματικής εργασίας, θα αναπτύξουμε προγράμματα ώστε να λύνουν το Carter Dataset. Τα Carter Benchmark Data εμφανίστηκαν στη βιβλιογραφία το 1996 στο **[25].**
Σε αυτό το άρθρο έγινε για πρώτη φορά η παρατήρηση ότι μέχρι τότε, στα περισσότερα papers, οι διάφοροι ερευνητές αναφέρονταν σε τυχαία προβλήματα και σπάνια σε κάποιο πρακτικό παράδειγμα. Αλλάζοντας λοιπόν αυτή τη φιλοσοφία, βρέθηκαν 13 πραγματικά προβλήματα.
Δέκα από διάφορα Πανεπιστήμια του Καναδά, 1 από το LSE , 1 από το King Fahd University of Petroleum and Minerals και 1 από το Purdue και ο στόχος ήταν οι ερευνητές να δοκιμάζουν τις λύσεις τους πάνω στο ίδιο πρόβλημα ώστε να υπάρχει μέτρο σύγκρισης.

Ο στόχος είναι απλός κατά τη δημιουργία του timetable. Προσπαθούμε να ελαχιστοποιήσουμε το κόστος που δημιουργείται από το timetable. Το κόστος αυτό εξάγεται από τον εξής συλλογισμό. Έστω ένας τυχαίος μαθητής A ο οποίος έχει N μαθήματα τα οποία εξετάζονται στα timeslots N1,N2… κλπ. Εάν για τα Ni, Nj υπάρχει i=j τότε τα μαθήματα έχουν επικάλυψη, τότε αυτό το πρόγραμμα είναι infeasible και αυτό το μάθημα δίνει ένα πολύ μεγάλο κόστος στο πρόγραμμα (400.000.000). Για όλες τις άλλες περιπτώσεις όπου $i \neq j$, τότε όσο πιο 'μακριά' βρίσκονται τα μαθήματα τόσο καλύτερο είναι το πρόγραμμα για τον τυχαίο μαθητή και αυτό εκφράζεται από μία φθίνουσα συνάρτηση κόστους ανάλογα με την απόσταση (16 για ένα κενό,8 για δύο κενά κλπ). Ο μέσος όρος του κόστους όλων των φοιτητών του πανεπιστημίου είναι αυτό που προσπαθούμε να ελαχιστοποιήσουμε.

Από την μορφή του προβλήματος δεν θα μας απασχολήσουν 2 ζητήματα:

1)Το μέγεθος της αίθουσας. Το πρόβλημα που καλούμαστε να επιλύσουμε λέγεται uncapacitated δηλαδή σε μια τυχαία αίθουσα χωράνε όσοι φοιτητές θέλουμε χωρίς περιορισμό χώρου.

2)Το γεγονός ότι 2 εξετάσεις που είναι σε διαδοχικές μέρες (τελευταία μιας μέρας και η πρώτη της επόμενης), και δύο εξετάσεις που είναι σε 2 διαδοχικές ώρες θα μας δίνουν την ίδια ποινή.

Carter benchmarks.

Problem	Institution	Periods	No. of examinations	No. of students	Density of conflict matrix
car-f-92 I	Carleton University, Ottawa	32	543	18419	0.14
car-s-91 I	Carleton University, Ottawa	35	682	16925	0.13
ear-f-83 I	Earl Haig Collegiate Institute, Toronto	24	190	1125	0.27
hec-s-92 I	Ecole des Hautes Etudes Commerciales, Montreal	18	81	2823	0.42
kfu-s-93	King Fahd University of Petroleum and Minerals, Dharan	20	461	5349	0.06
lse-f-91	London School of Economics	18	381	2726	0.06
pur-s-93 I	Purdue University, Indiana	43	2419	30029	0.03
rye-s-93	Ryerson University, Toronto	23	486	11483	0.08
sta-f-83 I	St Andrew's Junior High School, Toronto	13	139	611	0.14
tre-s-92	Trent University, Peterborough, Ontario	23	261	4360	0.18
uta-s-92 I	Faculty of Arts and Sciences, University of Toronto	35	622	21266	0.13
ute-s-92	Faculty of Engineering, University of Toronto	10	184	2749	0.08
yor-f-83 I	York Mills Collegiate Institute, Toronto	21	181	941	0.29

Σχήμα 3.2 Πληροφορίες των προβλημάτων **[10]**

ΚΕΦΑΛΑΙΟ 4 ΕΠΙΛΥΣΗ

Μέρος Α: Γενετικοί αλγόριθμοι

4.1) Γενικά περί εξελικτικών αλγορίθμων

Οι εξελικτικοί αλγόριθμοι (evolutionary algorithms) αποτελούν τεχνικές αναζήτησης και βελτιστοποίησης που έχουν προέλευση και έμπνευση από τον κόσμο της βιολογίας. Ειδικότερα, χρησιμοποιούν την ιδέα της φυσικής επιλογής και της επιβίωσης του καλύτερου (survival of the fittest), όπως την είχε ορίσει ο Δαρβίνος. Οι εξελικτικοί αλγόριθμοι μιμούνται τις φυσικές διαδικασίες της επιλογής ή αναπαραγωγής, **(selection, reproduction)**, της μετάλλαξης **(mutation)** και της διασταύρωσης **(crossover)** και τις χρησιμοποιούν ως μηχανισμούς ή τελεστές αναζήτησης (search operators), ώστε να βρουν καλύτερες λύσεις πιο γρήγορα σε προβλήματα βελτιστοποίησης. Οι υποψήφιες λύσεις του προβλήματος βελτιστοποίησης έχουν το ρόλο των ατόμων **(individuals)** ενός πληθυσμού και η «συνάρτηση ικανότητας» **(fitness function)** καθορίζει το περιβάλλον μέσα στο οποίο «ζούνε» οι λύσεις. Η εξέλιξη του πληθυσμού λαμβάνει χώρα μετά την επαναλαμβανόμενη εφαρμογή αυτών των τελεστών.

Οι εξελικτικοί αλγόριθμοι έχουν χρησιμοποιηθεί για να λύσουν με επιτυχία πολλά προβλήματα από διάφορα επιστημονικά πεδία, όπως η μηχανική, η οικονομία, η ρομποτική, η φυσική και η χημεία. Αυτή η επιτυχία τους οφείλεται στο γεγονός πως οι γενετικοί αλγόριθμοι, σε αντίθεση με τις κλασικές τεχνικές βελτιστοποίησης, δεν κάνουν καμιά υπόθεση σχετικά με τη μορφή του τοπίου της fitness function **(*fitness landscape*) [26].** Μακράν η πιο χρησιμοποιούμενη τεχνική γενετικού προγραμματισμού είναι οι γενετικοί Αλγόριθμοι. Στη συνέχεια θα γίνεται αναφορά αποκλειστικά σε αυτούς, καθώς ο αλγόριθμος που αναπτύχθηκε ως μέρος της διπλωματικής εργασίας ανήκει στην κατηγορία των γενετικών αλγορίθμων.

4.2) Γενετικοί Αλγόριθμοι (Genetic Algorithms)

Ο πρώτος που συνέλαβε την ιδέα ενός γενετικού αλγορίθμου ήταν ο John Holland του πανεπιστημίου του Michigan, Ann Arbor στα τέλη της δεκαετίας του 1960. Έκτοτε, αυτός και οι φοιτητές του έχουν συνεισφέρει πολλά στην ανάπτυξη τους. Το σκεπτικό με το οποίο λειτουργούν δεν είναι δύσκολο στην αντίληψη και στην συνέχεια θα γίνει παρουσίαση της λειτουργίας ενός GA μέσω ενός παραδείγματος. Πριν όμως παρουσιαστεί το παράδειγμα θα γίνει μια σύντομη ανασκόπηση όρων από την βιολογία οι οποίοι χρειάζονται για την κατανόηση των GA.

Ορολογία από τη βιολογία

Όλοι οι ζωντανοί οργανισμοί αποτελούνται από κύτταρα και κάθε κύτταρο περιέχει ένα σύνολο από **χρωμοσώματα** τα οποία γενικά είναι αλυσίδες DNA, δηλαδή πληροφορίας για τον οργανισμό. Κάθε χρωμόσωμα μπορεί να υποδιαιρεθεί σε γονίδια **(genes).** Σε γενικές γραμμές κάθε γονίδιο αντιπροσωπεύει ένα χαρακτηριστικό του οργανισμού πχ το χρώμα των ματιών. Τα διαφορετικά αποτελέσματα της αλλαγής ενός γονιδίου (πχ χρώμα ματιών καφέ, πράσινο κλπ) ονομάζονται **alleles** και κάθε γονίδιο είναι τοποθετημένο σε συγκεκριμένη θέση **(locus)** σε ένα χρωμόσωμα. Οι περισσότεροι οργανισμοί έχουν πολλά χρωμοσώματα σε ένα κύτταρο, το σύνολο των οποίων ονομάζεται **genome.**

Αρχικός πληθυσμός και κωδικοποίηση

Πρώτο βήμα είναι η δημιουργία ενός αρχικού πληθυσμού. Ο αρχικός πληθυσμός είναι ένα σύνολο από πιθανές λύσεις στο πρόβλημα. Προκύπτει λοιπόν το ερώτημα της κωδικοποίησης των αρχικών λύσεων για να αντιπροσωπεύουν συγκεκριμένες λύσεις στο πρόβλημα. Οι γενετικοί αλγόριθμοι χωρίζονται σε 2 κατηγορίες ανάλογα με την κωδικοποίηση τιμών των μεταβλητών του συστήματος που συνιστούν το γονιδίωμα ενός ατόμου (genotype). Στους *Γ.Α. δυαδικής κωδικοποίησης* **(binary-coded)** και στους *Γ.Α. πραγματικών παραμέτρων* **(real-parameter).**

4.2.1 Δυαδική κωδικοποίηση

Στη δυαδική κωδικοποίηση, κάθε μεταβλητή αναπαρίσταται με μια σειρά δυαδικών αριθμών (binary string) σταθερού μήκους. Για παράδειγμα, αν οι μεταβλητές ενός ατόμου είναι οι **x1=8 και x2=10** και χρησιμοποιούμε 5 bits για την αναπαράσταση της καθεμιάς, τότε ο γενότυπος του ατόμου θα είναι **01000 01010**. Σε αυτή την αναπαράσταση θεωρήσαμε πως το κατώτερο και το ανώτερο όριο των μεταβλητών είναι 0 και 31 αντίστοιχα, τότε έχουμε ακριβώς 32 διαφορετικές πιθανές λύσεις. Φυσικά, μπορούμε να αλλάξουμε τα όρια και το μήκος της σειράς ώστε να αναπαραστήσουμε διαφορετικού πλήθους ακέραιους ή δεκαδικούς αριθμούς. Μπορούμε να ελέγξουμε την ακρίβεια των τιμών, αλλά πάντα αυτή θα είναι πεπερασμένη. Ο λόγος που κωδικοποιούνται οι μεταβλητές σε μια σειρά bits είναι να επιτύχουμε μια αναπαράσταση όσο πιο κοντά στη φυσική βιολογική του χρωμοσώματος. Δηλαδή θεωρούμε ότι κάθε bit της σειράς αναπαριστά ένα γονίδιο που επηρεάζει το φαινότυπο του ατόμου.

4.2.2 Πραγματικές παράμετροι

Σε αυτή την αναπαράσταση κάθε μεταβλητή του συστήματος αντιστοιχίζεται σε ένα πραγματικό αριθμό (πχ του τύπου double στη C/C++), δηλαδή δεν έχουμε κάποιου είδους κωδικοποίηση. Όπως φαίνεται, η αναπαράσταση σε αυτή τη μορφή είναι ευκολότερη και πιο γρήγορη από υπολογιστικής άποψης, καθώς η τιμή κάθε μεταβλητής δε χρειάζεται μετατροπή από το δυαδικό στο δεκαδικό σύστημα για να υπολογιστεί οπότε οι μεταβλητές μπορούν να χρησιμοποιηθούν κατευθείαν για να υπολογιστεί η τιμή της συνάρτησης κόστους. Τα προβλήματα με αυτή την αναπαράσταση εμφανίζονται στην εφαρμογή των τελεστών αναζήτησης. Γενικά, η αναπαράσταση σε δυαδική κωδικοποίηση βοηθά στην υλοποίηση και στην οπτικοποίηση κατά τη διάρκεια της εφαρμογής των τελεστών.

4.3) Τελεστές και λειτουργία του γενετικού αλγορίθμου

4.3.1 Συνάρτηση κόστους (fitness function)

Το επόμενο στάδιο του GA είναι η αξιολόγηση του πληθυσμού. Αυτό γίνεται από την λεγόμενη συνάρτηση αξιολόγησης ή συνάρτηση κόστους**(fitness function).** Η συνάρτηση αυτή έχει σαν σκοπό την αξιολόγηση κάθε λύσης του πληθυσμού ενός γενετικού αλγορίθμου, αντιστοιχίζοντας μια τιμή ικανότητας σε κάθε άτομο (fitness value). Στις περισσότερες περιπτώσεις που δεν έχουμε περιορισμούς, η συνάρτηση fitness επιλέγεται ως ίδια με τη συνάρτηση στόχου του προβλήματος.

4.3.2 Κύκλος ενός γενετικού αλγορίθμου

Όπως αναφέραμε γενετικός αλγόριθμος ξεκινά την αναζήτηση με ένα τυχαίο πληθυσμό ατόμων και αντιστοιχίζεται μια τιμή ικανότητας σε κάθε άτομο. Στην συνέχεια ο πληθυσμός τροποποιείται από τρεις βασικούς τελεστές και ένα νέος, πιθανώς καλύτερος, πληθυσμός δημιουργείται. Αυτός ο κύκλος του αλγορίθμου ονομάζεται γενιά (generation). Υπάρχουν διάφορα κριτήρια τερματισμού του γενετικού αλγορίθμου. Μπορούμε να επιλέξουμε αρχικά τον μέγιστο αριθμό γενιών και ο αλγόριθμος να σταματήσει μόλις ο αριθμός αυτός συμπληρωθεί. Εναλλακτικά, μπορούμε να εξετάζουμε σε κάθε γενιά το μέγιστο fitness του πληθυσμού και ο αλγόριθμος να σταματήσει μόλις προσεγγίσει τη ζητούμενη τιμή, δηλαδή

όταν συγκλίνει. Τέλος, μπορούμε να θέσουμε εξαρχής το χρονικό διάστημα για το οποίο θα τρέξει ο αλγόριθμος. Στη συνέχεια θα αναπτυχθούν αναλυτικά οι τρεις βασικοί τελεστές του γενετικού αλγορίθμου.

4.3.3 Τελεστής αναπαραγωγής

Ο κύριος στόχος του τελεστή αναπαραγωγής είναι να δημιουργήσει αντίγραφα των καλών λύσεων και να εξαλείψει τις κακές λύσεις, ενώ το μέγεθος του πληθυσμού διατηρείται σταθερό. Υπάρχουν διάφοροι τρόποι για να επιτευχθεί αυτό:

- **tournament selection,**
- **proportionate selection**
- **ranking selection.**

Στο **tournament selection**, παίζονται τουρνουά, δηλαδή γίνεται σύγκριση μεταξύ N τυχαία επιλεγμένων μελών και η καλύτερη λύση εισάγεται στον πληθυσμό. Αν επαναληφθεί με προσοχή, κάθε λύση θα συμμετέχει σε ακριβώς δυο τουρνουά. Η καλύτερη λύση του πληθυσμού θα κερδίσει και τις δυο φορές οπότε θα έχει δυο αντίγραφα στο νέο πληθυσμό, ενώ η χειρότερη θα χάσει και τις δυο και δε θα έχει κανένα αντίγραφο. Οι υπόλοιπες λύσεις θα έχουν μεταξύ μηδέν και δυο αντιγράφων στο νέο πληθυσμό. Αυτή η μέθοδος έχει αποδειχθεί ότι έχει καλύτερη ή ίση σύγκλιση και καλύτερο ή ίσο χρόνο υπολογισμού σε σχέση με τις άλλες (**[9]**, p 85).

Στο proportionate selection κάθε λύση αντιστοιχεί σε αντίγραφα ανάλογα με την τιμή fitness που έχει. Ο μηχανισμός που λειτουργεί η μέθοδος είναι παρόμοιος με τη λειτουργία μιας ρουλέτας, στην οποία ο τροχός είναι χωρισμένος σε N κομμάτια, όπου N το μέγεθος του πληθυσμού. Το κάθε κομμάτι χωρίζεται αναλογικά με βάση τη τιμή κόστους fi κάθε λύσης. Σε κάθε άτομο αντιστοιχεί πιθανότητα pi=fi/Σfi. Στη συνέχεια, ο τροχός γυρίζει N φορές και κάθε φορά επιλέγει μια λύση ανάλογα με τη θέση του δείκτη. Στατιστικά, τα αντίγραφα κάθε ατόμου θα είναι N*pi. Έτσι οι καλύτερες λύσεις αναμένεται να έχουν περισσότερα αντίγραφα στο νέο πληθυσμό. Όμως η μέθοδος αυτή παρουσιάζει μεγάλη διακύμανση στα αποτελέσματα της ([**9],** p 86-87).

Το **ranking selection** είναι μια παραλλαγή της προηγούμενης μεθόδου, όπου τα άτομα κάθε πληθυσμού ταξινομούνται ανάλογα με την τιμή fitness τους από το χειρότερο στο καλύτερο. Μετά σε κάθε λύση αντιστοιχίζεται μια τιμή fitness ίση με τον αριθμό της κατάταξης τους, δηλαδή 1 για το χειρότερο και N για το καλύτερο και τέλος εφαρμόζεται με αυτές τις τιμές η προηγούμενη μέθοδος.

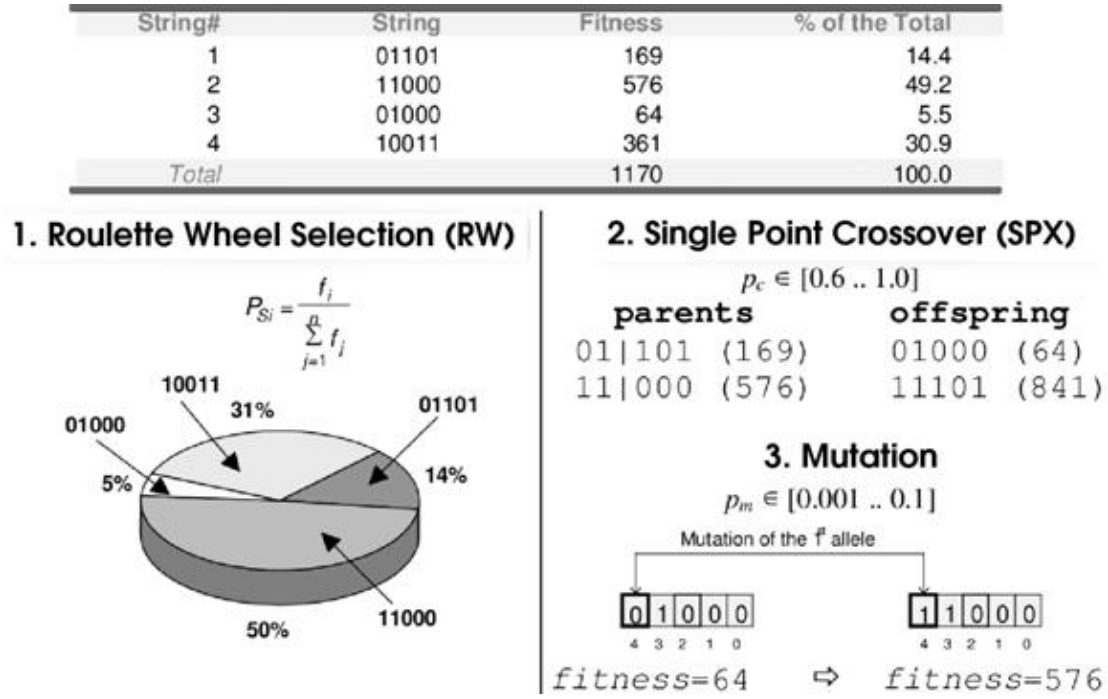
4.3.4 Τελεστής διασταύρωσης (crossover)

Ο τελεστής διασταύρωσης εφαρμόζεται στο νέο πληθυσμό που δημιουργείται από την εφαρμογή του τελεστή αναπαραγωγής. Με λίγη σκέψη βλέπουμε πως ο τελεστής αναπαραγωγής δεν μπορεί να δημιουργήσει νέες λύσεις στον πληθυσμό, μόνο να δημιουργήσει περισσότερα αντίγραφα των καλών λύσεων. Η δημιουργία νέων λύσεων πραγματοποιείται από τον τελεστή διασταύρωσης και τον τελεστή μετάλλαξης. Η λειτουργία του τελεστή διασταύρωσης εξαρτάται από την αναπαράσταση που θα χρησιμοποιήσουμε για τις μεταβλητές. Για την περίπτωση της δυαδικής κωδικοποίησης υπάρχουν διάφορες τεχνικές, αλλά συνήθως επιλέγονται δυο σειρές από bits (που ανήκουν σε δυο λύσεις) και ένα μέρος της μιας σειράς ανταλλάσσεται με κάποιο μέρος της άλλης σειράς ώστε να δημιουργηθούν δυο νέες σειρές από bits, δηλαδή δυο νέες λύσεις. Όπως στη βιολογία, αναμένουμε πως μια ή και οι δυο νέες λύσεις που θα δημιουργηθούν από δυο καλές λύσεις θα είναι καλύτερες από τους γονείς τους. Στην περίπτωση του single-point crossover επιλέγεται ένα σημείο και τα bits δεξιά του ανταλλάσσονται μεταξύ των strings. Στο multi-point crossover ακολουθείται η ίδια διαδικασία μόνο που το γονιδίωμα χωρίζεται σε περισσότερα του ενός σημεία **([9],** p 89-90).

4.3.5 Τελεστής μετάλλαξης

Ο τελεστής μετάλλαξης είναι ο δεύτερος τελεστής μέσω του οποίο επιτυγχάνεται η δημιουργία νέων λύσεων, δηλαδή ουσιαστικά η αναζήτηση του αλγορίθμου. Όπως και με τον προηγούμενο τελεστή, η λειτουργία του τελεστή διασταύρωσης εξαρτάται από την αναπαράσταση που θα χρησιμοποιήσουμε για τις μεταβλητές μας. Στην περίπτωση της αναπαράστασης με δυαδική κωδικοποίηση, ο τελεστής σαρώνει κάθε σειρά από strings και μεταβάλλει το 1 σε 0 ή το 0 σε 1 με πιθανότητα μετάλλαξης pm. Δηλαδή για κάθε στοιχείο παράγεται ένας τυχαίος αριθμός ανάμεσα στο 0 και στο 1. Αν αυτός ο αριθμός είναι μεγαλύτερος από το 1-pm ή μικρότερος από το pm (ανάλογα με την υλοποίηση) τότε η μετάλλαξη εκτελείται.

Όπως έχει αναφερθέι, κάθε bit αναπαριστά ένα γονίδιο στο γονιδίωμα ενός ατόμου, συνεπώς η μετάλλαξη εκτελείται με παρόμοιο τρόπο με τον οποίο συμβαίνει και η βιολογική μετάλλαξη, με αλλαγή γονιδίων. Αυτός ο τελεστής είναι πάρα πολύ σημαντικός αν και συχνά παραμελείται. Βασικός του ρόλος είναι να μην 'κολλάει' ο αλγόριθμος σε τοπικά ακρότατα και αυτό επιτυγχάνεται από την εισαγωγή ενός στοιχείου 'τύχης' που υπάρχει σε αυτό τον τελεστή.



Σχήμα 4.1 Παράδειγμα των τελεστών **[18]**

Σε αυτό το σημείο ο νέος πληθυσμός αξιολογείται ξανά και ο κύκλος του γενετικού αλγορίθμου συνεχίζεται. Αποδεικνύεται ότι, **[26]** ο γενετικός αλγόριθμος οδηγεί το σύνολο του πληθυσμού με τέτοιο τρόπο ώστε το μέσο κόστος του πληθυσμού να μειώνεται.

```
Procedure Genetic Algorithm
begin
    t ← 0
    Αρχικοποίησε το P(t)
    Αξιολόγησε το P(t)
    while ( not συνθήκη τερματισμού) do
    begin
        t ← t+1
        Επιλογή του P(t) από το P(t–1)
        Τροποποίηση του P(t)
        Αξιολόγηση του P(t)
    end
end
```

Σχήμα 4.2 δομή ενός G.A.**[14]**

Μέρος Β : Σειριακή επίλυση

4.4) Παρουσίαση της Σειριακής λύσης του Προβλήματος

Στις παραγράφους που ακολουθούν θα παρουσιάσουμε τον τρόπο προσέγγισης που ακολουθήσαμε για την επίλυση του προβλήματος μας σε ANSI –C **[28] [10]**
Ο αλγόριθμός μας χωρίζεται σε δύο μέρη. Σε πρώτο μέρος γίνεται κατανομή τών μαθήματα στα timetables χρησιμοποιώντας γνώση για το πρόβλημα με **(Largest Degree)** όπως θα παρουσιαστεί στη συνέχεια. Αν για ένα τυχαίο μάθημα δεν υπάρχει timeslot που να μην δημιουργεί σύγκρουση τότε ο αλγόριθμος δεν θα σταματήσει αλλά θα το κάνει allocate στην τύχη. Αυτό σημαίνει ότι το πρόγραμμα που έχει δημιουργηθεί δεν είναι feasible αλλά αναμένουμε ότι θα έχει έναν πολύ μικρό αριθμό συγκρούσεων. Σε σύγκριση δηλαδή με έναν αλγόριθμο που κάνει allocate στην τύχη ο δικός μας αρχικός πληθυσμός είναι πιο κοντά στη γειτονιά μίας υπαρκτής λύσης. Αυτό είναι σαφές από το γεγονός ότι ένα τυχαίο timetable θα έχει N (πχ 50) conflicts κατά μέσο όρο ενώ εμείς στην εργασία αναμένουμε ότι θα έχει M<<N (πχ 3-4). Στην συνέχεια ο GA με τους τελεστές που περιγράφτηκαν πιο πάνω θα προσπαθήσει να βρει feasible λύσεις οι οποίες έχουν πολύ μικρότερο κόστος από τις non-feasible. Χρησιμοποιήθηκε πιο πάνω η λέξη γνώση. Τι σημαίνει όμως αυτό? Η γνώση στο πρόβλημα ETP παίρνει την μορφή ευρετικού και σκοπός είναι να καθοδηγήσει την αρχική δημιουργία του πληθυσμού.

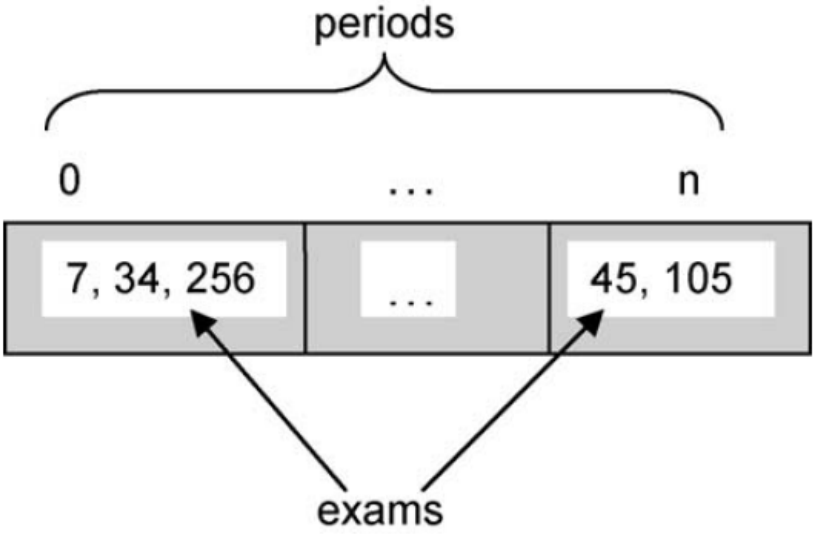
Πιο συγκεκριμένα εντοπίστηκαν στη βιβλιογραφία τα εξής heuristics :

1)Largest Degree: Ο αριθμός των συγκρούσεων κάθε εξέτασης λαμβάνεται υπόψη. Οι εξετάσεις με τις περισσότερες συγκρούσεις είναι οι πιο δύσκολες να προγραμματιστούν.
2)Largest Enrolment: Τα μαθήματα με τους περισσότερους μαθητές προγραμματίζονται πρώτα.
3)Largest weighted degree: Οι εξετάσεις με τον μεγαλύτερο αριθμό μαθητών με συγκρούσεις προγραμματίζονται πρώτες
4)Saturation Degree: Ο αριθμός timeslots που μια εξέταση μπορεί να προγραμματιστεί λαμβάνεται υπόψη. Εξετάσεις με μικρό αριθμό είναι πιο δύσκολες και άρα προηγούνται.
5)Highest Cost: Το κόστος δίνεται από συγκεκριμένη συνάρτηση που δίνεται από τον αλγόριθμό στο σχήμα 3 σελίδα 461 **[10].**

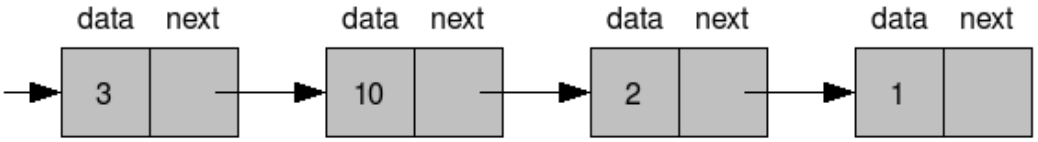
Στατιστικά στοιχεία δίνονται στο**[10]**(p 462-465)

Πριν γίνει καταγραφή του γενετικού αλγορίθμου πρέπει να σημειωθεί ότι πριν ξεκινήσει ο γενετικός, πρέπει να δημιουργηθεί ο Conflict Matrix από τα δεδομένα. Διαβάζοντας τα στοιχεία των μαθητών ο αλγόριθμος δημιουργεί έναν πίνακα [courses x courses] όπου τα στοιχεία είναι 1 αν τα μαθήματα μ, ν έχουν κοινούς μαθητές ή 0 αλλιώς.

Το πρώτο βήμα για την επίλυση του προβλήματος είναι να βρεθεί ένας τρόπος αναπαράστασης του χρωμοσώματος ο οποίος δίνεται από το εξής σχήμα (3.3). Όπως φαίνεται λοιπόν θα έχουμε ένα χρωμοσώματα με μήκους Ν όσες είναι κάθε φορά και οι εξετάσεις και κάθε γονίδιο θα είναι μια σειρά από τα μαθήματα που είναι προς εξέταση σε αυτό το timeslot.



Σχήμα 4.3 **[10]** Αναπαράσταση χρωμοσώματος



Σχήμα 4.4 παράδειγμα linked list

Σχηματική αναπαράσταση του χρωμοσώματος σε C. Από τα Δεδομένα του προβλήματος οδηγούμαστε σε χρήση linked list για την αποθήκευση των μαθημάτων. Το κάθε χρωμόσωμα, είναι ένας πίνακας μεγέθους timeslots, του οποίου κάθε στοιχείο δείχνει σε ένα μάθημα. Έπειτα κάθε μάθημα δείχνει στο επόμενο κλπ. Η χρήση Linked List ενδείκνυται λόγο της απλότητας της στην εισαγωγή και διαγραφή στοιχείων.

Αφού παρουσιάστηκε ο τρόπος με τον οποίο έγινε η αναπαράσταση του γονιδίου το επόμενο βήμα είναι ορισμός του κύριου μέρους του αλγορίθμου. Ο πρώτος τελεστής είναι το selection. Στην διπλωματική αυτή η χρήση **tournament selection** υλοποιήθηκε λόγο της ευκολίας μετάβασης της υλοποίησης της στο παράλληλο περιβάλλον**[27]**. Στο tournament selection απλά ‘κοιτάμε’ την προηγούμενη γενιά και δημιουργούμε ένα tournament μεγέθους Ν και στην συνέχει παίρνουμε τον

καλύτερο. Ο επόμενος τελεστής που παρουσιάστηκε ήταν το crossover. Με χρήση Linked list αυτός ο τελεστής υλοποιείται εύκολα ως εξής. Αφού ορίσουμε το στοιχείο στο οποίο θα γίνει το crossover (το timeslot δηλαδή) τότε απλά ορίζουμε τον κάθε δείκτη να δείχνει στην λίστα που έδειχνε ο άλλος γονέας.

Ο επόμενος τελεστής είναι ο Mutate. Θα μπορούσαμε απλά να πάμε σε ένα τυχαίο μάθημα και να αλλάξουμε την τιμή του. Αυτός είναι ένας απλός και σωστός τρόπος αλλά υπάρχει κάτι πιο απλό. Στον κώδικα μας ο τελεστής mutate απλά διαγράφει ένα στοιχείο από την λίστα. Ο λόγος είναι ο εξής. Λόγο της συγκεκριμένης επιλογής χρωμοσώματος ο τελεστής crossover δημιουργεί προγράμματα που είναι infeasible από την άποψη ότι μετά το crossover υπάρχουν διπλά μαθήματα και μερικά χαμένα μαθήματα από το timetable. Αυτό υποχρεώνει τον προγραμματιστή να ορίσει μια επιπλέον συνάρτηση για να επισκευάζει τα γονίδια (repair). Δουλειά αυτής της συνάρτησης είναι να κάνει scan όλα τα timetables από τα προηγούμενα βήματα και να διαγράψει τα πολλαπλά μαθήματα και να κάνει ξαναβάλει στο πρόγραμμα αυτά που χάθηκαν. Έτσι ο τελεστής mutate διαγράφοντας ένα στοιχείο υποχρεώνει τον την συνάρτηση επισκευής να το ξαναβάλει στο timetable.

4.5) Λεπτομέρειες του προγράμματος

Υπάρχουν πέντε βασικές μεταβλητές από τις οποίες εξαρτώνται τα αποτελέσματα του γενετικού αλγόριθμου. Η πρώτη μεταβλητή είναι το μέγεθος του πληθυσμού. Είναι προφανές ότι μεγαλύτερα μεγέθη δίνουν μια πιο μεγάλη δεξαμενή γενετικού υλικού στον αλγόριθμο για να εργαστεί και αυξάνουν την πιθανότητα εύρεσης λύσης. Η επόμενη μεταβλητή που έχει σημασία είναι το μέγεθος του τουρνουά. Προφανώς υπάρχει άμεση εξάρτηση από την προηγούμενη μεταβλητή αλλά προφανώς το τουρνουά πρέπει να κοιτάει έναν, μικρό μεν σημαντικό δε, αριθμό ατόμων για να επιλέξει τον καλύτερο.
Οι επόμενες δυο πάρα πολύ σημαντικές παράμετροι είναι

1. Η πιθανότητα δύο γονείς να αποκτήσουν παιδιά. Δηλαδή δύο άτομα του νέου πληθυσμού, τι πιθανότητα υπάρχει να αντικατασταθούν από τα παιδιά τους.
2. Η πιθανότητα μετάλλαξης ενός γονιδίου. Δηλαδή ένα τυχαίο γονίδιο, τι πιθανότητα έχει να αλλάξει πριν την επόμενη γενιά.

Τέλος δεν πρέπει να αγνοούμε ότι ο γενετικός είναι μια επαναληπτική διαδικασία, κάθε βήμα της οποίας ονομάζεται γενιά (generation) και ο αριθμός των γενεών είναι μια πολύ σημαντική μεταβλητή του αλγορίθμου.

Η σωστή επιλογή των παραπάνω αριθμών είναι μια δύσκολη επιλογή και αξίζει να αναφερθεί ότι από αυτές της μεταβλητές επηρεάζεται άμεσα η ποιότητα των λύσεων και η ταχύτητα του αλγορίθμου. Ανάλογα με το πρόβλημα είναι στην ευχέρεια του χρήστη να ανοίξει το genetic.c και να εργαστεί με διαφορετικές μεταβλητές ώστε να οδηγήσει πιθανότατα τον κώδικα σε πιο γρήγορη σύγκλιση. Αξίζει ένα επιπλέον σχόλιο για την συνάρτηση srand(). Γράφτηκε στο κεφάλαιο ο τρόπος με τον οποίο γίνονται allocate τα μαθήματα στη διπλωματική, ο οποίος εισάγει ένα στοιχείο τύχης στην διαδικασία. Η Srand είναι μια συνάρτηση παραγωγής ψευδοτυχαίων αριθμών η οποία παίρνει σαν seed έναν αριθμό για να παράγει μια ακολουθία ψευδοτυχαίων αριθμών. Αυτό σημαίνει ότι ανάλογα με την ώρα (αφού κάνουμε seed με time) που εκτελούμε το πρόγραμμα οι ποιότητα των λύσεων αλλάζει πολύ. Επίσης σημαίνει ότι ίδιο seed δίνει ίδια ακολουθία αριθμών και κατά συνέπεια ίδια timetables. Αυτό φάνηκε στον παράλληλο κώδικα καθώς για επεξεργαστές στο ίδιο die (ίδιο blade) παρατηρήθηκαν ίδιοι πληθυσμοί.

Ο σκελετός του αλγορίθμου μας είναι πλήρως βασισμένος σε αυτόν που δόθηκε στο προηγούμενο κεφάλαιο. Το πρώτο βήμα είναι η χρήση του tournament selection για την δημιουργία πληθυσμού. Στην συνέχεια οι γνωστοί τελεστές των γενετικών κάνουν την απαραίτητη διεργασία και μόλις τελειώσουν η repair επισκευάζει τα timetables. Τέλος το κύκλος τερματίζει με εξαγωγή του μέσου και του καλύτερου κόστους του κύκλου. Ο πλήρης κώδικας μπορεί να βρεθεί στο παράρτημα Α.

Μέρος Γ : Παράλληλη επίλυση

4.6) Μοντέλα παράλληλων γενετικών

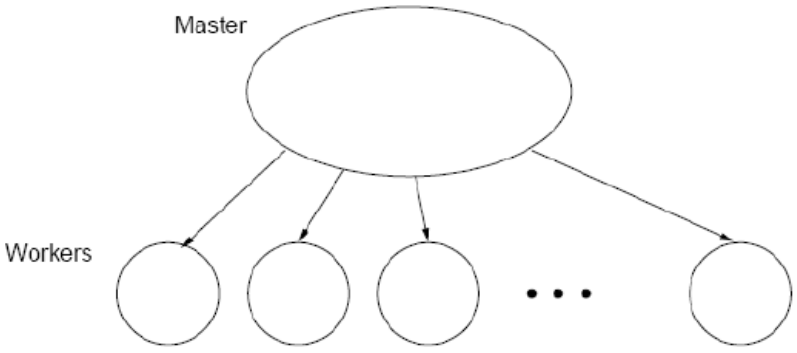
Είδαμε ότι οι γενετικοί αλγόριθμοι είναι ένα πολύ ισχυρό εργαλείο και δίνει αξιόλογες λύσεις σε μεγάλα προβλήματα σε μικρό χρονικό διάστημα. Όπως όμως μπορούμε να κρίνουμε και από τους χρόνους της σειριακής εκτέλεσης, αύξηση του μεγέθους του προβλήματος οδηγεί και στην κατακόρυφη αύξηση του χρόνου που απαιτείται από τον γενετικό μας αλγόριθμο. Αυτό είναι ένα πρόβλημα και θα είναι εύλογο να προσπαθήσουμε να βελτιώσουμε τον χρόνο εκτέλεσης του αλγορίθμου με χρήση παράλληλου προγραμματισμού. Τα σειριακά μοντέλα γενετικών στην πλειοψηφία τους κρατάνε τον πληθυσμό στην δική τους προσωπική μνήμη. Κατά συνέπεια κατά την εφαρμογή των τελεστών οποιοδήποτε μέρος του πληθυσμού μπορεί να αλληλεπιδράσει με οποιοδήποτε άλλο μέλος (panmixia). Τα πράγματα είναι λίγο πιο περίπλοκα στους παράλληλους γενετικούς όπου υπάρχουν τρείς βασικές κατηγορίες παράλληλων και ο τρόπος σκέψης και υλοποίησης διαφέρει από τύπο σε τύπο**[12][18]**.

Αυτοί είναι οι:

1. **Master-Slave**
2. **Island Model ή Coarse Grained**
3. **Diffusion ή Fine Grained**

4.6.1 Γενετικοί αλγόριθμοι Master Slave

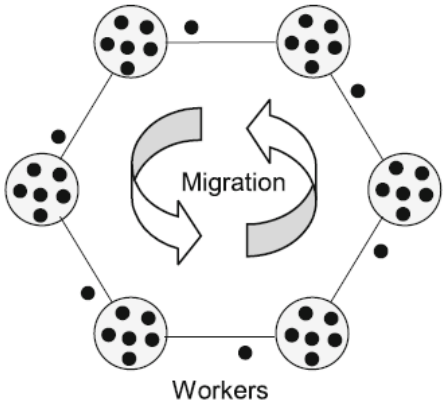
Σε αυτή τη μέθοδο ο αλγόριθμος χρησιμοποιεί έναν μοναδικό πληθυσμό και ο υπολογισμός της συνάρτησης fitness ή και η εφαρμογή των γενετικών τελεστών γίνεται παράλληλα. Όπως και σε έναν σειριακό γενετικό αλγόριθμο, κάθε άτομο μπορεί να ανταγωνιστεί ή να διασταυρώσει με οποιαδήποτε άλλο, δηλαδή η εφαρμογή των τελεστών επιλογής και διασταύρωσης γίνεται καθολικά. Οι γενετικοί αλγόριθμοι αυτού του τύπου λέγονται master-slave γιατί οι επεξεργαστές χωρίζονται σε ρόλους. Ένας αναλαμβάνει το ρόλο του συντονιστή (master) και οι υπόλοιποι το ρόλο των εργατών (slaves). Το πιο σύνηθες τμήμα που παραλληλοποιείται είναι ο υπολογισμός της συνάρτησης fitness επειδή η τιμή αυτή δεν εξαρτάται από άλλα μέλη του πληθυσμού. Ο master υποδιαιρεί τον πληθυσμό, τον αποστέλλει στους Slaves και αυτοί υλοποιούν το τμήμα που τους αντιστοιχεί.



Σχήμα 4.1 master-slave model**[12]**

4.6.2 Island Model ή Coarse Grained

Η δεύτερη κατηγορία είναι αυτή που μας ενδιαφέρει στα πλαίσια της εργασίας. Αυτή η κατηγορία βασίζεται στο φαινόμενο της δημιουργίας πληθυσμών με σχετική απομόνωση σε ξεχωριστές περιοχές (islands). Σε αυτή την κατηγορία κάθε νησί –Cpu έχει έναν δικό του σχετικά μεγάλο πληθυσμό αλλά μικρότερο του πληθυσμού του σειριακού. Σε κάθε νησί στην ουσία εκτελείται ο σειριακός γενετικός ανεξάρτητα από το τι συμβαίνει στα άλλα νησιά. Η λέξη ανεξάρτητα δεν είναι απόλυτα ακριβής. Άμα απλά υποδιαιρέσουμε τον πληθυσμό της σειριακής εκτέλεσης υπάρχει πιθανότητα ο υποπληθυσμός του κάθε επεξεργαστή να είναι πολύ μικρός και να μην οδηγούμαστε πουθενά στην αναζήτηση. Υπάρχει λοιπόν ένας επιπλέον τελεστής στους παράλληλους γενετικούς που λέγεται μετανάστευση **(migration).** Με τη χρήση αυτού του τελεστή, ανά τακτά διαστήματα, υπάρχει λίγη επικοινωνία μεταξύ των Cpus. Όταν έρθει η ώρα του τελεστή κάθε νησί 'στέλνει ' σε γειτονικό του ένα στοιχείο του πληθυσμού του και αντίστοιχα λαμβάνει έναν στοιχείο από γειτονικό του νησί.



Σχήμα 4.2 island model **[18]**

Οι αλγόριθμοι αυτής της κατηγορίας λέγονται και distributesGAs (dGAs) διότι συχνότερα υλοποιούνται σε distributed συστήματα. Λέγονται επίσης και πολλαπλών πληθυσμών (multiple deme) λόγο της ύπαρξης των υποπληθυσμών σε κάθε νησί, καθώς και fine grained λόγο του μεγάλου μεγέθους του υποπληθυσμού. Αυτή η κατηγορία είναι και η πιο δημοφιλής και επελέγη για την υλοποίηση της διπλωματικής εργασίας για τρείς βασικούς λόγους.

1)Αποτελούν μια επέκταση του σειριακού γενετικού αλγορίθμου. Το μόνο που απαιτείται είναι η εκτέλεση του σειριακού σε κάθε νησί και η περιοδική αποστολή και λήψη δεδομένων από άλλα συστήματα.

2)Η υλοποίηση της ρουτίνας migrate είναι πάρα πολύ απλή καθώς κάθε νησί υλοποιείται σε δικό του χώρο στη μνήμη ανεξάρτητα από άλλους πληθυσμούς.

3)Η μεγάλη διαθεσιμότητα παράλληλων distributed συστημάτων οδήγησε στην δημιουργία ελεύθερων βιβλιοθηκών όπως το MPI που είναι εύκολα στην εκμάθηση και χρήση.

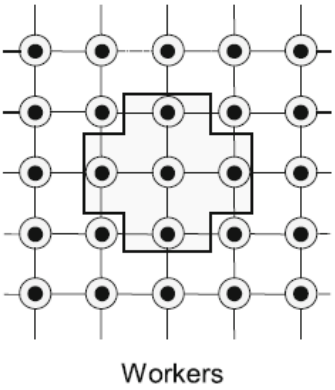
Μετανάστευση σε island model γενετικούς

Είναι υποχρεωτικό να γίνει καταγραφή κάποιων λεπτομερειών της πολύ σημαντικής λειτουργίας της μετανάστευσης στους dGAs. Η μετανάστευση εξαρτάται από τις εξής παραμέτρους:

- **Τοπολογία**
  Η τοπολογία της σύνδεσης είναι πολύ σημαντικός παράγοντας για την απόδοση του αλγορίθμου. Γενικά στα παράλληλα η επικοινωνία μεταξύ μονάδων είναι ακριβή διαδικασία η οποία όπου είναι δυνατόν αποφεύγεται. Αν λοιπόν η τοπολογία έχει πυκνή διασύνδεση δηλαδή πολλά νησιά είναι μεταξύ τους συνδεδεμένα ο αλγόριθμος θα καθυστερήσει περισσότερο αλλά υπάρχει μεγαλύτερο diversity στον πληθυσμό και επιπλέον οι πολύ καλές λύσεις εξαπλώνονται πολύ γρήγορα στο σύμπλεγμα των νησιών (take over rate).
  Τα αντίθετα ακριβώς συμβαίνουν στις περιπτώσεις με αραιότερη σύνδεση.
- **Συχνότητα**
  Η συχνότητα είναι μια πολύ σημαντική παράμετρος διότι η αύξηση του αριθμού των μηνυμάτων συνεπάγεται και αύξηση του χρόνου εκτέλεσης του αλγορίθμου. Είναι στην κρίση του προγραμματιστή να επιλέγει σωστή συχνότητα.
- **Ρυθμός**
  Ο ρυθμός εκφράζει το πλήθος των ατόμων που θα μεταναστεύσουν σε κάθε κύκλο από νησί σε νησί και υπάρχουν 2 βασικές μέθοδοι. Με χρήση ομοιόμορφης μεθόδου κάθε νησί στέλνει έναν συγκεκριμένο αριθμό ατόμων σε κάθε κύκλο στα άλλα νησιά. Αντίθετα με μή ομοιόμορφη μέθοδο ο αριθμός αυτός δεν είναι σταθερός αλλά μεταβάλλεται κατά την εκτέλεση του αλγορίθμου.
- **Μέθοδος αντικατάστασης**
  Η τελευταία παράμετρος ορίζει τον τρόπο που θα αντικατασταθούν τα άτομα που έχουν έλθει από τα γειτονικά νησιά. Συνήθως αυτό γίνεται στοχαστικά με άλλα τυχαία άτομα του πληθυσμού ώστε το σύνολο των ατόμων στον πληθυσμό παραμένει σταθερό.

4.6.3 Diffusion ή Fine Grained

Το τελευταίο μοντέλο που λέγεται επίσης και cellular (cGAs) διαφέρει άρδην από το προηγούμενο. Σε αυτό το μοντέλο υπάρχει μόνο ένας πληθυσμός αλλά ορίζεται μια δομή τέτοια ώστε οι αλληλεπιδράσεις μεταξύ ατόμων να ορίζονται αυστηρά από τους γείτονες του. Στο σχήμα 5.3 φαίνεται μια γειτονιά (σταυρός, τόρος, τετράγωνο κ.α.) στην οποία επενεργεί μία υπολογιστική μονάδα. Τα στοιχεία που έχει επιλέξει μπορούν να διασταυρωθούν μόνο με γειτονικά στοιχεία της γειτονιάς και η επικάλυψη μεταξύ των γειτονιών είναι αυτή που επιτρέπει στις καλές λύσεις να εξαπλώνονται. Λόγο του μικρό αριθμού ατόμων του επιλεγμένου υποπληθυσμού αυτές οι μέθοδοι λέγονται Fine Grained και συνήθως υλοποιούνται στο GRID ή σε συστήματα κοινής μνήμης για εκμετάλλευση της ταχύτητας πρόσβασης στην κοινή μνήμη.



Σχήμα 4.3 Cellular Genetic Model

ΚΕΦΑΛΑΙΟ 5 : ΑΠΟΤΕΛΕΣΜΑΤΑ

5.1) Αποτελέσματα – σχόλια σειριακής υλοποίησης

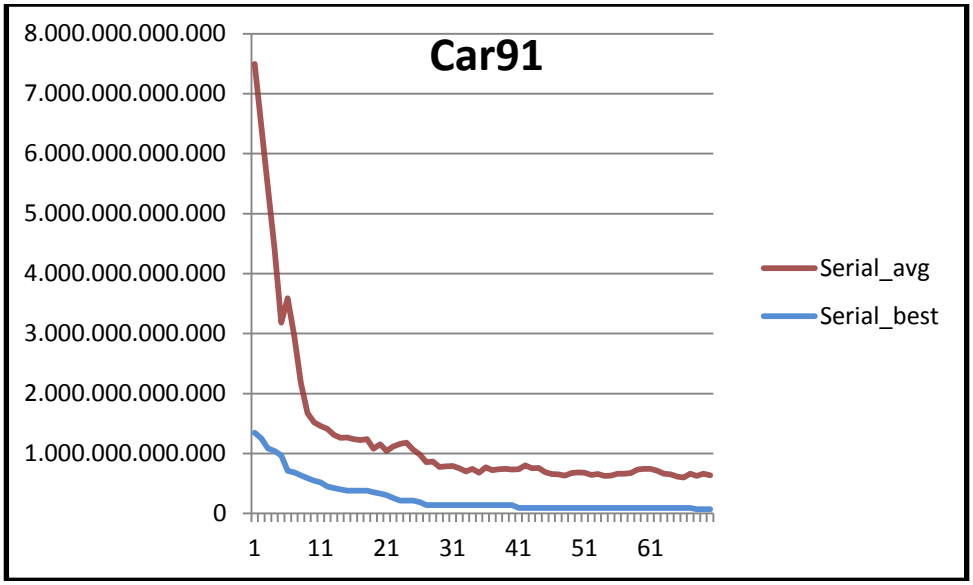
Στους υπολογιστές του εργαστηρίου λύθηκαν όλα τα προβλήματα του carter. Εργαστήκαμε με HS22 blades ,E5530-Cpu που συνδέονται μεταξύ τους με infiniband. Τα timetables που δημιουργήθηκαν βρίσκονται στους φακέλους που συνοδεύουν την εργασία. Πριν ακολουθήσουν τα διαγράμματα με το μέσο και το καλύτερο κόστος πρέπει να γίνει καταγραφή δυο προτύπων που παρουσιάστηκαν. Πέραν του προβλήματος της αλλαγής ποιότητας ανάλογα με την ώρα, παρατηρήθηκε ότι η αύξηση των μαθημάτων οδηγεί σε κατακόρυφη αύξηση του χρόνου εκτέλεσης.

Στον πίνακα που ακολουθεί βλέπουμε συγκεντρωμένα τα αποτελέσματα του αλγορίθμου για τα προβλήματα του Carter. Πραγματοποιήθηκε ένα σύνολο από δοκιμές και για το κάθε πρόβλημα και χρησιμοποιήθηκαν σαν μεταβλητές:

Population size = 800
Mutation probability rate=0.5%
Crossover probability rate=40%
Generations =70
Tournament size=20

Ακολουθεί ο πίνακας με τα αποτελέσματα που πήραμε από τη σειριακή εκτέλεση του αλγορίθμου. Στην συνέχεια ακολουθεί ένα από τα τυπικά διαγράμματα που παρουσιάζουν την σχέση μεταξύ μέσου και βέλτιστου κόστους κατά την εκτέλεση του αλγορίθμου.

Problem name	Best Cost Found	Runtime (s)	Feasible	Best after injection
Car91	70904.66	19845	No	7.55
Car92	108590.29	12982	No	6.06
Ear83	44.92	2285	Yes	43.81
Hec92	13.12	734	Yes	12.37
Kfu93	20.09	6879	Yes	19.02
Lse91	293486.75	2284	No	15.69
Pur93 *	9.53	38067	Yes	-
Rye92	15.69	14714	Yes	14.24
Sta83	162.92	1454	Yes	160.23
Tre92	10.49	2122	Yes	10.94
Uta92	5.80	16202	Yes	5.04
Ute92	30.13	1352	Yes	29.02
Yor83	1275284.63	1303	No	44.49



Διάγραμμα κόστους-γενεών

Όπως βλέπουμε και σε αυτό το τυπικό διάγραμμα της εκτέλεσης τόσο το μέσο όσο και το καλύτερο κόστος κάθε γενιάς, στη γενική περίπτωση, πέφτει με το πέρασμα του χρόνου.
Τα σύνολο των διαγραμμάτων μπορεί να βρεθεί στο παράρτημα Δ.

5.2) Αποτελέσματα – σχόλια παράλληλης υλοποίησης

Οι τελεστές των παράλληλων γενετικών είναι ακριβώς ίδιοι με τους τελεστές του σειριακού. Η μόνη νέα συνάρτηση είναι η migrate η οποία υλοποιείται με τον εξής τρόπο. Επειδή θέλουμε να αποφύγουμε την επικοινωνία, είναι ασύμφορο να στείλουμε τα δεδομένα ακέραιο – ακέραιο. Έτσι δημιουργούμε ένα chunk με όλα τα μαθήματα όπου, ο αποστολέας δημιουργεί έναν μονοδιάστατο πίνακα από το δυσδιάστατο timetable με τα μαθήματα και μόλις δει το τέλος της σειράς γράφει το μάθημα με αρνητικό πρόσημο. Έτσι ο παραλήπτης ξέρει πού να αλλάξει γραμμή. Όπως αναφέρθηκε ο βασικότερος λόγος ύπαρξης των παράλληλων συστημάτων είναι η επιτάχυνση ή αλλιώς speedup. Στην περίπτωση τον παράλληλων γενετικών όμως τα πράγματα είναι πιο πολυσύνθετα. Αρχικά επειδή είναι μια στοχαστική διαδικασία δεν μπορούμε απλά να διαιρέσουμε τον παράλληλο χρόνο με τον σειριακό και να πάρουμε το speedup**. [30]**
Πιο σωστό είναι να κάνουμε έναν μεγάλο αριθμό από runs και στη συνέχεια με χρήση του κεντρικού οριακού θεωρήματος να πάρουμε το speedup από την διαίρεση των δύο μέσων όρων.
Στη συνέχεια τίθεται το πολύ βασικό θέμα της ποιότητας της λύσης. Είναι λογικό για να μετρήσουμε το speedup ότι χρειαζόμαστε λύσεις περίπου ίδιας ποιότητας. Αυτό όμως δεν είναι εφικτό στα πλαίσια της εργασίας και κατά συνέπεια θα χρησιμοποιήσουμε έναν πιο χαλαρό ορισμό που θα είναι ο εξής. Σε πρώτο στάδιο τρέχουμε τον σειριακό κώδικα στην παράλληλη μηχανή με έναν processor και καταγράφουμε τον χρόνο που έκανε να τερματίσει τις 70 γενιές και την ποιότητα της λύσης. Στην συνέχεια θα τρέξουμε τον παράλληλο κώδικα στην ίδια μηχανή και θα μετρήσουμε τον χρόνο που έκανε να τελειώσει τις 70 γενιές και θα καταγράψουμε και την ποιότητα της λύσης . Θα ορίσουμε λοιπόν σαν βελτίωση στον χρόνο (time improvement) τον λόγο μεταξύ των χρόνων του σειριακού και του παράλληλου κώδικα ανεξάρτητα από την διαφορά στην ποιότητα των λύσεων.

Πριν παραθέσουμε τα αποτελέσματα στα οποία του παράλληλου γενετικού αλγορίθμου θα καταγράψουμε τα βασικότερα ερωτήματα που παραμένουν αναπάντητα στους παράλληλους γενετικούς. Όπως κατεγράφη, υπάρχουν 3 επιπλέον μεταβλητές που παίζουν σημαντικό ρόλο στους παράλληλους γενετικούς.

- Το μέγεθος του πληθυσμού που θα μεταναστεύσει.
- Η χρονική στιγμή που θα μεταναστεύσει ο πληθυσμός .
- Η τοπολογία των ‘νησιών’.

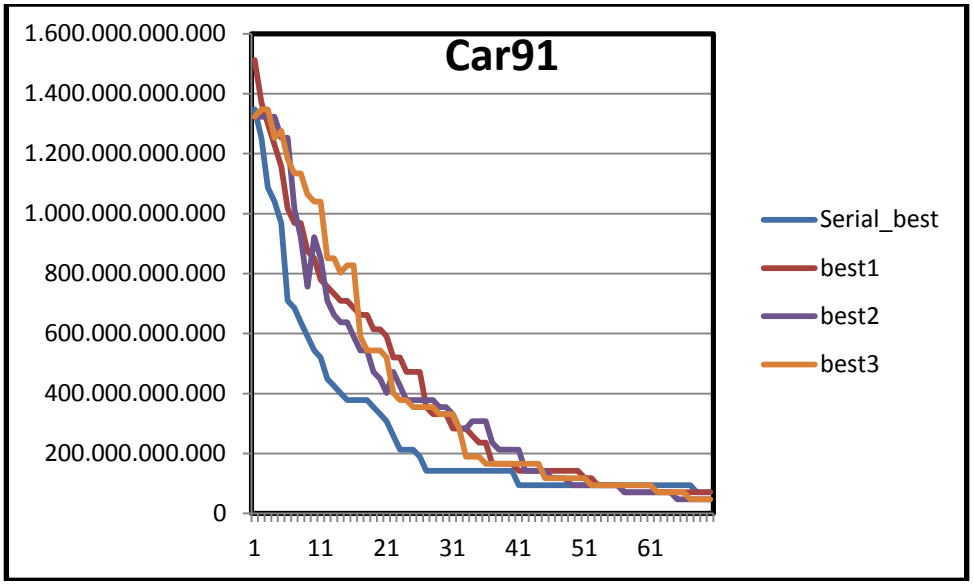
Από αυτούς το τελευταίο κομμάτι είναι συνήθως σταθερό κατά την διάρκεια της εκτέλεσης του κώδικα. Στα πλαίσια της διπλωματικής εμείς χρησιμοποιήσαμε νησιά που στέλνουν μόνο προς μια κατεύθυνση και συγκεκριμένα στον επόμενο processor. Τα άλλα δυο όμως έχουν πολύ ενδιαφέρον και παραμένουν αναπάντητα για τους μελετητές των pGA**[29]**. Για την παρούσα εργασία εργαστήκαμε με αποστολή του 10% του πληθυσμού στο γειτονικό νησί όπου ο καλύτερος πάει πάντα συνοδευόμενος από ένα τυχαία επιλεγμένο πλήθος επιπλέον χρωμοσωμάτων. Προκύπτουν λοιπόν τα εξής ερωτήματα τα οποία είναι αναπάντητα για τους pGA.

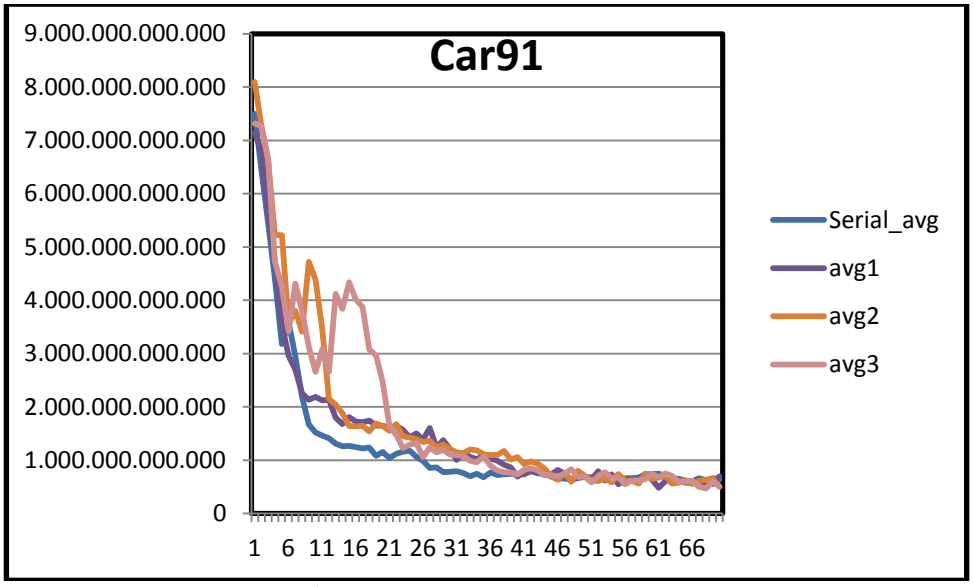
Ποίο είναι το σωστό μέγεθος των Demes ώστε ο παράλληλος κώδικας να συμπεριφέρεται σαν τον σειριακό?
Ποιο είναι το σωστό διάστημα που θα επιβάλουμε μετανάστευση?
Ποιο είναι το σωστό μέγεθος του πληθυσμού που θα μεταναστεύσει και πώς επιλέγεται αυτός?

Δυστυχώς η, κατά άλλα πλούσια βιβλιογραφία, πάνω στους παράλληλους γενετικούς δεν παρέχει απαντήσεις για αυτά τα ερωτήματα. Υπάρχουν όμως τα εξής σημεία που δέχεται, και κατά συνέπεια τα δεχόμαστε και εμείς στα πλαίσια της εργασίας.

- Η αύξηση του μέσου fitness σε έναν μικρό πληθυσμό συμβαίνει πιο γρήγορα από ότι σε έναν μεγαλύτερο για δεδομένο αριθμό γενεών.
- Επιπλέον, ειδικά στα απομονωμένα νησιά, ο αλγόριθμος συγκλίνει σε χειρότερες λύσεις από ότι τα σειριακά panmictic μοντέλα. Σε περιπτώσεις χαμηλού migration rate τα νησιά εξερευνούν διαφορετικές περιοχές του χώρου των λύσεων και κατά συνέπεια τα αποτελέσματα είναι περίπου ίδια με τα απομονωμένα νησιά καθώς οι 'μετανάστες' δεν έπαιξαν σημαντικό ρόλο στην πορεία του αλγορίθμου. Όταν το migration rate όμως αυξηθεί περεταίρω τα αποτελέσματα του pGA πλησιάζουν ή και ξεπερνούν τα αποτελέσματα του σειριακού panmictic μοντέλου. Κατά συνέπεια αντιλαμβανόμαστε ότι υπάρχει μια οριακή τιμή για το migration rate κάτω από την οποία ο pGA δεν αποδίδει καλά και καλούμαστε συνήθως να την βρούμε.
- Τέλος όσων αφορά το ερώτημα της χρονικής στιγμής του migration αξίζει μόνο να τονίσουμε ότι στην αρχή της εκτέλεσης του αλγορίθμου, γενικά, δεν υπάρχουν αρκετά 'κομμάτια' καλών λύσεων σε οποιονδήποτε πληθυσμό. Κατά συνέπεια δεν αξίζει να σπαταληθούν οι πολύ σημαντικοί πόροι της ακριβής επικοινωνίας στην αρχή του αλγορίθμου διότι το κέρδος από την επικοινωνία είναι ουσιαστικά ασήμαντο.

Problem name	Best Cost Found	Runtime (s)	Feasible	Improvement
Car91	47272.71	14208	No	1.39
Car92	65156	4813	No	2.69
Ear83	47.91	1430	Yes	1.59
Hec92	13.03	514	Yes	1.4
Kfu93	149580.02	5374	No	1.28
Lse91	17.17	1575	Yes	1.45
Pur93 *	13328.61	13258	No	2.87
Rye92	34850.31	10085	No	1.45
Sta83	160.22	525	Yes	2.82
Tre92	11.4	811	Yes	2.6
Uta92	37624.57	5821	No	2.78
Ute92	30.9	637	Yes	2.12
Yor83	425125.71	905	No	1.43





Διαγράμματα κόστους-γενεών

Τα παραπάνω διαγράμματα είναι από την λύση ενός από τα προβλήματα σε 3 επεξεργαστές και σύγκριση των αντίστοιχων τιμών με την σειριακή εκτέλεση.

5.3) Συμπεράσματα-μελλοντικές κατευθύνσεις

Μελετώντας και επιλύοντας τα προβλήματα του Carter καταλήγουμε στα εξής συμπεράσματα. Καταρχάς ο σειριακός αλγόριθμος που κρατάει όλο τον πληθυσμό σε ένα pool έχει πλεονέκτημα λύσεων σε σχέση με τον παράλληλο που έχει διαιρέσει τον πληθυσμό σε υποπληθυσμούς. Το επιπλέον γενετικό υλικό που υπάρχει στον σειριακό είναι μεγάλο πλεονέκτημα και ο παράλληλος, τουλάχιστον με την υλοποίηση της migrate όπως είναι στα πλαίσια της εργασίας δεν είναι δυνατόν να φτάσει σε βέλτιστες λύσεις ίδιας ποιότητας. Αυτό βέβαια αντισταθμίζεται από τον επιπλέον χρόνο που κάνει ο σειριακός για να τερματίσει. Επιπλέον παρατηρήθηκε και στα πειράματα μας κάτι που αναφέρθηκε στη βιβλιογραφία. Παρατηρήσαμε ότι σε μικρότερους πληθυσμούς το μέσο κόστος πέφτει πιο γρήγορα από ότι το μέσο κόστος στον σειριακό αλγόριθμο. Στον αντίποδα, ο καλύτερος κάθε γενιάς είναι καλύτερος στον σειριακό

από ότι στον παράλληλο. Από πλευράς νέας έρευνας τα προβλήματα του χρονοπρογραμματισμού παρουσιάζουν πολύ ενδιαφέρον και αξίζει η μελέτη τους, τουλάχιστον μέχρι να βρεθεί η βέλτιστη μέθοδος επίλυσης. Οι γενετικοί όμως είναι το πιο ενδιαφέρον κομμάτι. Είναι ένα πολύ ισχυρό εργαλείο το οποίο μπορεί να δώσει σε πολλά μεγάλα προβλήματα του χώρου. Ο ακριβής τρόπος λειτουργίας όμως δεν έχει μοντελοποιηθεί πλήρως και στο προηγούμενο κεφάλαιο αναφέρθηκαν μερικά ζητήματα που απαιτούν λύση.

Επιπλέον παρόλο υπάρχει πολύ ενδιαφέρον από άποψης διπλωματικών στο CSL για το τμήμα ΗΜΤΥ και η διπλωματική μπορεί κάλλιστα να συνεχιστεί από κάποιον άλλο φοιτητή με τους εξής τρόπους. Είναι προφανές ότι με την χρήση παράλληλης επεξεργασίας έχει επιτευχθεί μια βελτίωση στο χρόνο εκτέλεσης του αλγορίθμου. Σύμφωνα με τον πίνακα της προηγούμενη παραγράφου υπάρχει σημαντική βελτίωση στο χρόνο εκτέλεσης. Κοιτώντας όμως τους χρόνους εκτέλεσης του αλγορίθμου παρατηρούμε ότι ίσως υπάρχουν σημεία που μπορούν να βελτιωθούν. Για παράδειγμα η συνάρτηση της επισκευής με αυτή την μορφή της έχει μια πολυπλοκότητα $O(n^2)$ το οποίο καθυστερεί τον αλγόριθμο αρκετά. Είναι εφικτό να γραφτεί σε μορφή O(n) και να υπάρξει αισθητή βελτίωση.

Ειδικά για το μεγαλύτερο από τα προβλήματα το runtime είναι περίπου 2 μέρες κάτι που μπορεί να χαρακτηριστεί ως ασύμφορο. Στη συνέχεια η συνάρτηση αποστολής μπορεί να βελτιστοποιηθεί περεταίρω με τον εξής τρόπο. Στην εργασία τα δεδομένα στέλνονται από επεξεργαστή σε επεξεργαστή σε chunk μεγέθους όσα είναι τα μαθήματα. Αυτό απαιτεί επικοινωνία για κάθε χρωμόσωμα προς αποστολή. Πιθανότατα υπάρχει τρόπος γραφής δημιουργώντας chunks που είναι ακόμα μεγαλύτερα και να στέλνεται ένα μήνυμα κάθε φορά από Cpu σε Cpu.Επίσης θα μπορούσε να σταλεί άλλος ένας float αριθμός με το κόστος από τον αντί να υπολογίζεται εκ νέου. Αυτό βέβαια θα απαιτούσε πιο ακριβή μελέτη αν η αποστολή του ενός αριθμού είναι πιο φθηνή από τον υπολογισμό του κόστους από τον νέο υπολογιστή.

Αυτά αφορούν το προγραμματιστικό κομμάτι. Υπάρχει όμως και μια σειρά από ενδιαφέροντα θέματα που αξίζει να μελετηθούν περεταίρω. Όπως αναφέρθηκε υπάρχει μια σειρά από ευρετικά για την μελέτη του προβλήματος. Μια πολύ ενδιαφέρουσα ιδέα είναι να υλοποιηθούν αυτά τα ευρετικά και να συγκριθούν τα αποτελέσματα με τα αποτελέσματα της διπλωματικής. Μια τέτοια σύγκριση απαιτεί βέβαια την χρήση πολλών στατιστικών στοιχείων και δεδομένων οπότε είναι επιτακτική η χρήση SPSS για T-Test και ANOVA ώστε να γίνει κατανοητό αν τα αποτελέσματα έχουν στατιστικό νόημα.

Παράρτημα Α: Σειριακός Κώδικας

Θα γίνει τώρα καταγραφή των βασικών βημάτων ώστε ο αρχάριος χρήστης να τρέξει ένα πρόβλημα του Carter. Εκτελώντας το .exe μας παρουσιάζεται το shell. Η πρώτη επιλογή (1) είναι για φόρτωση του αρχείου το .crs που είναι τα μαθήματα. Η επιλογή (2), η οποία είναι χρήσιμη μόνο μετά την (1) είναι αναζήτηση του αριθμό των μαθητών που έχουν πάρει ένα μάθημα. Η επιλογή (3) είναι για φόρτωση του αρχείου .stu που είναι τα μαθήματα που έχουν πάρει όλοι οι φοιτητές του ιδρύματος. Μετά την εκτέλεση της υπάρχουν οι επιλογές (4) και (5) που είναι για αναζήτηση των μαθημάτων που έχει πάρει ένας τυχαίος φοιτητής καθώς και για την αναζήτηση του συνολικού αριθμού των μαθημάτων στο ίδρυμα. Έπειτα και αφού έχουν φορτωθεί τα δυο αρχεία του προβλήματος, με την επιλογή (6) το εκτελέσιμο φτιάχνει τον conflict matrix που είναι απαραίτητος για την υλοποίηση του προβλήματος ώστε να παρατηρούνται οι συγκρούσεις. Στην συνέχει η επιλογή (7) εξάγει όλη τη λίστα των μαθητών για να την δει ο χρήστης.
Η επιλογή (8) είναι αυτή που ενεργοποιεί το κύριο μέρος του αλγορίθμου δηλαδή τον γενετικό που επιλύει το πρόβλημα. Τέλος η επιλογή (9) αφού έχει τερματίσει η (8) κάνει export το αρχείο με το πρόγραμμα που έχει δημιουργηθεί.

Η main είναι υπεύθυνη για την δημιουργία shell και export του timetable

```
Int main(){

      printf("\nWELCOME TO THE THESIS OF GURU HKOMINOS\n");
            printf(" \n  to run choices are 1-3-6-8 \n\n\n");
            printf("\nwhat would you like to do\n\n");
          while(1){
                    switch(choices()){
case 1: ptr_to_crs=readcourseslist(&main_number_of_courses) break;
case 2: printf("dwse ena courseid pros anazitisi\n");
scanf("%d",&crs_to_find);
printcourses(ptr_to_crs,crs_to_find); break;
case 3: ptr_to_std=load_student();break;
case 4: printf("dwse ena id pros anazitisi\n");
scanf("%d",&studentid_to_find);
printstudent(ptr_to_std,studentid_to_find);break;

case 5: printf("%d\n",main_number_of_courses);break;
case 6: ptrtoconflictmatrix=create_matrix(ptr_to_std,main_number_of_courses); break;
case 7: printallstudents(ptr_to_std); break;
case 8: printf("enter timeslotes\n\n");
                scanf("%d",&timeslots);
                 ptrtoptr=&ptr_popsize;
float start=clock();
ptrtothebest=create_timetable(ptrtoconflictmatrix,timeslots,main_number_of_courses,ptr
toptr,ptr_to_std);
float stop=clock();
float duration=(stop-start)/CLOCKS_PER_SEC;   break;
case 9: printf("print the best timetable ");
       exportbesttimetable(ptrtothebest,timeslots,duration); break
case 0: exit(0);
default: exit(0);
          }

      }
return 0;}
```

```
int choices(void){
int choice=0;
printf( "1)----   Load the student list         ----\n"
        "2)----   Print a course enrollement    ----\n"
        "3)----   Load student List             ----\n"
                          "4)----   Print student courses         ----\n"
                   "5)----   Print global courses          ----\n"
                   "6)----   Create conflict matrix        ----\n"
                   "7)----   Print all students            ----\n"
                   "8)----   Create genetic timetable      ----\n"
                   "9)----   Export best timetable         ----\n"
                   "0)----   Exit                          ----\n");

scanf("%d",&choice);
fflush(stdin);
return choice;
}

//function to the best timetable to external txt.
void exportbesttimetable(struct chromosome *ptr,int timeslots,float duration){

      FILE *fp ;
      char filename[20];
      int i;
    printf("Eisagete output file name <name.txt>");
    gets(filename);
    fflush(stdin);
    if((fp=fopen(filename,"w"))==NULL)
                    {
                    fprintf(stderr,"Error opening file %s",filename);
                    getchar();
                    }
    else
                    {
                          printf("File opened  succesfully  ");
                    }
    fprintf(fp,"best cost  %lf = and duration was %f \n\n" ,ptr->cost,duration);
    for(i=0;i<timeslots;i++)
        {
            struct courseintb *temp=ptr->timetable[i];
             do
             {
             fprintf(fp,"%4d %3d \n",temp->course,i);
              temp=temp->next;
             }while(temp!=NULL);
        }

fclose(fp);
}
```

Οι επόμενες συναρτήσεις είναι υπεύθυνες για την ανάγνωση των εξωτερικών αρχείων. crs και .stu

```
#include "functions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


//function to read courses file
struct course *readcourseslist(int *globalcoursesaddress)
```

```
{
        FILE *fp ;
        char *token;
        int courses=0,total_enrolments=0;
        char filename[40],buffer[256];
        printf("Eisagogi arxeiou mathimatwn <------.crs>\n\n\n");
        gets(filename);
        fflush(stdin);
        if((fp=fopen(filename,"r"))==NULL)
        {
                fprintf(stderr,"Error opening file %s",filename);
                getchar();
        }else
        {
             printf("File loaded succesfully containing ");
        }
        LINK root=NULL;
        LINK new=NULL;
        LINK current=NULL;
        while (fgets(buffer,256, fp) != NULL)
        {
                if(root==NULL)
                {
                     new=(LINK)malloc(sizeof(COURSE));
                     if(new==NULL)
                          {
                          printf("memory error");
                          getchar();
                          exit(0);
                          }
                      new->next=root;
                      token=strtok(buffer," ");
                      new->courses_num=atoi(token);
                      token = strtok(NULL, " ");
                      new->enrolment=atoi(token);
                      total_enrolments+=new->enrolment;
                      root=new;

                }
                else
                {
                    current=root;
                    while(current->next!=NULL)
                    {
                           current=current->next;
                    }
                    LINK new=NULL;
                    new=(LINK)malloc(sizeof(COURSE));
                             if(new==NULL)
                             {
                                         printf("error");
                                         getchar();
                                         exit(0);
                             }
                     current->next=new;
                     new->next=NULL;
                     token=strtok(buffer," ");
                     new->courses_num=atoi(token);
                     token = strtok(NULL, " ");
```

```
                    new->enrolment=atoi(token);
                    total_enrolments+=new->enrolment;
                  //printf("course  me  id  %d   exei  tosous  %d\n",new->courses_num  ,new-
>enrolment);
                }

        courses++;

        }

        printf("%d lines and %d total enrol\n ",courses,total_enrolments);
        *globalcoursesaddress=courses;
        fclose(fp);
        return (root);

}


struct student *load_student(void){
             FILE *fp2 ;
             char *token;
          int filelines=0,i=0;
          char filename[40],buffer[256];
             printf("Eisagwgi arxeiou mathitwn<------.stu > \n\n\n");
             gets(filename);
             fflush(stdin);
      if((fp=fopen(filename,"r"))==NULL)
             {
                    fprintf(stderr,"Error opening file %s",filename);
                    getchar();
          }
             else
             {
                    printf("File loaded succesfully containing ");
             }
          LINK2 root=NULL;
             LINK2 new=NULL;
             LINK2 current=NULL;
             while (fgets(buffer,256, fp2) != NULL)
             {
                    i++;
                    filelines+=1;
                    if(root==NULL)
                     {
              new=(LINK2)malloc(sizeof(STUDENT));
                        if(new==NULL)
                        {
                            printf("memory error");
                            getchar();
                        }
                    new->next=root;
                    new->student_id=i;
                    token=strtok(buffer," ");
                    int courseid=0;
                    while (token!=NULL)
                        {
                           new->courses[courseid]=atoi(token);
                           new->courses[courseid+1]=0;
                           new->courses[courseid+2]=0;
```

```
                        courseid++;
                        token=strtok(NULL," ");}
                    root=new;
                    }
                else
                {
                        current=root;
                    while(current->next!=NULL)
                    {
                        current=current->next;
                    }
                LINK2 new=NULL;
                new=(LINK2)malloc(sizeof(STUDENT));
                            if(new==NULL)
                                    {
                                            printf("error");
                                    }
            current->next=new;
            new->student_id=i;
            new->next=NULL;
            token=strtok(buffer," ");
            int courseid=0;
            while (token!=NULL)
            {
                        new->courses[courseid]=atoi(token);
                        new->courses[courseid+1]=0;
                        new->courses[courseid+2]=0;
                        token=strtok(NULL," ");
                        courseid++;

            }
                }
        fflush(stdin);
                }
             printf("%d number of lines  \n",filelines);
             fclose(fp2);
        return(root);

}
//Gia debugging anazitisi mathimatos

void printcourses(struct course *ptr,int crs_to_find){
       int found=0;
       do{
              if(ptr->courses_num==crs_to_find)
              {
                            printf("course  num  is  %d    enrolment  is  %d\n",  ptr-
>courses_num,ptr->enrolment);
                            found=1;
              }
              else
              {
                    ptr=ptr->next;
              }
         }while (ptr!=NULL&&found==0);
```

```
if (found==0)
	{
		printf("Student not found!\n");
	 }
}


// Gia debugging anazitisi foititi

void printstudent(struct student *ptr,int Idlookup){
	int found=0;
	do{
		if(ptr->student_id==Idlookup)
		{
			found=1;
		}
		else
		{
			ptr=ptr->next;
		}
		}while((found==0)&&(ptr!=NULL));
    if(found==1){
	int i=0;
	printf("O student me arithmo id  %d exei parei ta eksis mathimata \n ",ptr-
>student_id);
	do{
		printf("%d\n",ptr->courses[i]);
		i++;
	 }while(ptr->courses[i]!=0);
    }
    else
    {
	printf("Student not found!\n");
    }
}

// kirios gia debugging
void printallstudents(struct student *ptr){
do{
		int i=0;
		printf("o student me arithmoid  %d exei parei ta eksis mathimata \n
",ptr->student_id);

		do{
			printf("%d\n",ptr->courses[i]);
			i++;
			}while(ptr->courses[i]!=0);
		 ptr=ptr->next;

	}while(ptr!=NULL);
}
```

Η επόμενη συνάρτηση είναι υπεύθυνη για την δημιουργία του πίνακα των συγκρούσεων

```
#include "functions.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int *ptr_to_table=NULL;
```

```
int *pinakas_deiktwn;
float matrix_density;


//creaton of conflict matrix.kathe seira tou pinaka einai ena course.prosoxi dioti i C
vazei tous pinakes apo to 0 ara to course 1 einai to row 0
//scanaroume dld oli ti lista ton mathitwn kai vleoume ta mathimata pou exoun.
int** create_matrix(struct student *ptr,int size){
       float numberofones=0;
       struct student *root=ptr;
    int found=0;
    int i,j,course_scaned=1;
    int** theArray;
    theArray = Make2DDoubleArray(size,size);
    for(i=0; i<size; i++)
    {
       for(j=0; j< size; j++)
              {
                     theArray[i][j] = 0;

              }
    }

    do{
               do{
                              found+=hascourse(course_scaned,theArray[course_scaned-
1],ptr->courses);
                                     ptr=ptr->next;

                      }while(ptr!=NULL);
               ptr=root;
               course_scaned++;
    }while(course_scaned<size+1);

    for(i=0; i<size; i++)
       {
              for(j=0; j< size; j++)
                     {
                            if (theArray[i][j]==1)
                                   {
                                   numberofones+=1;
                                   }
                     }
}


       matrix_density=numberofones/(size*size);
       printf("the matrix density is %5f\n\n",matrix_density);
return theArray;
}

int hascourse(int course,int *theArray,int courses []){
       int i=0,found=0;
       do{
         if(courses[i]==course)
         {
              found=1;
         }
              i++;
```

```
	}while(courses[i]!=0);
if(found==1)
	{
	i=0;
	do{
		theArray[courses[i]-1]=1;
		i++;
		}while(courses[i]!=0);
	}
	return found;
}



int** Make2DDoubleArray(int arraySizeX, int arraySizeY)//make2dintarray stin ousia
{
	int** theArray;
	int i;
	theArray = (int**) malloc(arraySizeX*sizeof(int*));
	for ( i = 0; i < arraySizeX; i++)
		theArray[i] = (int*) malloc(arraySizeY*sizeof(int));
		return theArray;
}
```

Το τελευταίο κομμάτι είναι η καρδιά του γενετικού

```
#include "functions.h"
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "timetabling.h"
#include <string.h>
#include <math.h>


struct	chromosome*	create_timetable(int**	conflictmatrix,int	timeslots,int
courses,struct chromosome **ptr,struct student *ptrtostudentlist){

struct chromosome *ptrtothebest=NULL; //tha kratithei o kaliteros gia to export
int POPSIZE=250;	//obvious
int TOURNAMENTSIZE=10;		//obvious. kala noumera einai POPSIZE/10++
int NUMBEROFGENERATIONS=5;	//arithmos genewn
int CROSSOVERPROB=40;		 //pithanotita 2 goneis na kanoun paidi .sta 100 dld
0.25
int MUTATIONPROB=100;		//pithanotita na iparksei mutation .sinistatai heavy
mutation .sta 10000 dld 0.01
	int i,k;
	srand(time(NULL));
	int* sortedmatrix=(int *)malloc(sizeof(int)*courses);
	fillcourses(conflictmatrix,sortedmatrix,courses);
	*ptr=initiatepop(conflictmatrix,sortedmatrix,courses,POPSIZE,timeslots,ptrtostu
dentlist);
	struct chromosome **initialpop=createptrtable(*ptr,POPSIZE);
	struct chromosome **nextgeneration=(LINK3 *)malloc(POPSIZE*sizeof(LINK3));
	ptrtothebest=replicatebest(initialpop[0],timeslots);
	float averagecost=0.0;
	float generationbest=5000000.0;
```

```
	for(k=0;k<POPSIZE;k++)
	{
		averagecost+=(initialpop[k]->cost);
		if(initialpop[k]->cost<generationbest)
			{
		  generationbest=initialpop[k]->cost;
			}
	  }
	printf("Average cost of initial population is %lf and best cost of initial
population is %lf\n" ,averagecost/(POPSIZE*1.0),generationbest);


for(i=0;i<NUMBEROFGENERATIONS;i++){
	averagecost=0.0;
	srand(time(NULL));
	generationbest=5000000.0;
	 for(k=0;k<POPSIZE;k++)
		{

nextgeneration[k]=selectfromtournament(initialpop,TOURNAMENTSIZE,POPSIZE,timeslots);
		//printf("after round %d selected organism organism with cost %lf\n",
i,nextgeneration[k]->cost);
		}

	    printf("crossingover\n");
	    reproducegenes(nextgeneration,POPSIZE,CROSSOVERPROB,timeslots);
	      printf("mutating\n");
	      mutate(nextgeneration,POPSIZE,MUTATIONPROB,timeslots,conflictmatrix);
	      printf("repairing\n");

	repair(nextgeneration,courses,timeslots,POPSIZE,conflictmatrix,ptrtostudentlist
);
		deletepreviousgeneration(initialpop,nextgeneration,POPSIZE,timeslots);
		printf("generation now is %d ",i);
		for(k=0;k<POPSIZE;k++)
			{
				if((initialpop[k]->cost)<(ptrtothebest->cost))
				{
					struct chromosome *temp=ptrtothebest;
				ptrtothebest=replicatebest(initialpop[k],timeslots);
				printf("best cost found %lf \n",ptrtothebest->cost);
				myfree(temp,timeslots);
				  }
				if(initialpop[k]->cost<generationbest)
				{
			   generationbest=initialpop[k]->cost;
			  }
			 averagecost+=(initialpop[k]->cost);
			}
		printf("average	cost	is	%lf	bestcost	is
%lf\n",averagecost/(POPSIZE*1.0),generationbest);
		export(averagecost/(POPSIZE*1.0),generationbest);
	}
return ptrtothebest;
}


//dimiourgia pinaka  pou periexei otn arithmw ston sigkrousewn kathe mathimatos
void fillcourses(int** conflictmatrix,int* sortedmatrix,int courses){
```

```
int i;
	for(i=0;i<courses;i++)
	{
		sortedmatrix[i]=clashnumber(conflictmatrix[i],courses);
	}
}

//arithmos sigkrousewn
int clashnumber(int *conflictrow,int courses){
	int i,sum=0;
	for(i=0;i<courses;i++)
	{
		sum+=conflictrow[i];
	}
	return sum;
}

//dunuiyrgia linkedlist me ton arxiko plithismo
struct chromosome *initiatepop(int **conflictmatrix ,int *sortedmatrix ,int courses
,int POPSIZE,int timeslots,struct student *ptrtostudentlist)
{
	LINK3 root=NULL;
	struct chromosome **ptrtoroot=&root;
	int i;
		for(i=0;i<POPSIZE;i++)
			{
			int* yourownmatrix=(int *)malloc(sizeof(int)*courses);
			memcpy(yourownmatrix,sortedmatrix,courses*sizeof(int));
			printf("creating individual %d\n",i);

	create_individual(conflictmatrix,yourownmatrix,courses,timeslots,ptrtoroot,ptrt
ostudentlist);
			free(yourownmatrix);
			}
	return root;
}

//dimourgia timetable simfwna me to kritirio
void  create_individual(int  **conflictmatrix  ,int  *sortedmatrix  ,int  courses,int
timeslots,struct chromosome** ptrtoroot,struct student *ptrtostudentlist){
	   if(*ptrtoroot==NULL){
					LINK3 new=NULL;
				new=(LINK3)malloc(sizeof(CHROMOSOME));
				if(new==NULL){printf("error");getchar();}
				new->next=*ptrtoroot;
				*ptrtoroot=new;
				new->timetable=Make2dintArray(timeslots);
				filltable(new-
>timetable,conflictmatrix,courses,sortedmatrix,timeslots);
				new->cost=calculatecost(ptrtostudentlist,new-
>timetable,timeslots);
					}
		else {
			 LINK3 new=NULL;
			   LINK3 current=NULL;
				current=*ptrtoroot;
			  while(current->next!=NULL)
			  {
				current=current->next;
```

```
                                        }
                                        new=(LINK3)malloc(sizeof(CHROMOSOME));
                                                if(new==NULL)
                                                                {
                                                                   printf("error");
                                                                   getchar();
                                                                }
                                           current->next=new;
                                           new->next=NULL;
                                           new->timetable=Make2dintArray(timeslots);
                                           filltable(new-
>timetable,conflictmatrix,courses,sortedmatrix,timeslots);
                                           new->cost=calculatecost(ptrtostudentlist,new-
>timetable,timeslots);

                }
}



//safes
struct courseintb** Make2dintArray(int arraySizeX) {
struct courseintb** theArray;
int i;
theArray = (struct courseintb**) malloc(arraySizeX*sizeof(COURSEINTB));
for(i=0;i<arraySizeX;i++)
{
      theArray[i]=NULL;
}
   return theArray;
}

//gemisma tou timetable tou kathe individual
void  filltable(struct  courseintb  **timetable,int**  conflictmatrix,int  courses,int
*sortedmatrix,int timeslots){
      int i;
      for(i=0;i<courses;i++){
            int coursetoallocateatposition=0,*ptr_coursetoalloc;
            ptr_coursetoalloc=&coursetoallocateatposition;
      allocate(courses,conflictmatrix,sortedmatrix,ptr_coursetoalloc,timeslots,timeta
ble);
                                  }
}

//safes
void     allocate(int     courses,int     **conflictmatrix,int     *sortedmatrix,int
*coursetoallocate,int timeslots,struct courseintb **timetable){
    int max=0,i,maxposition=0;
    for(i=0;i<courses;i++)
              {
                     if(sortedmatrix[i]>max)
                     {
                          max=sortedmatrix[i];
                          maxposition=i;
                     }
                      *coursetoallocate=maxposition+1;
            }
       sortedmatrix[maxposition]=0;
      positioninwhichtoalloc(maxposition,conflictmatrix,timeslots,timetable);
```

```
}


//stin ousia einai coursetoalloc.to proto orisma eiani to mathima pou thes na kaneis
alloc -1
void positioninwhichtoalloc(int maxposition,int **conflictmatrix,int timeslots,struct
courseintb** timetable){
int allocated=0,isthereconflict,tries=0;
int coursetoalloc=maxposition+1;
int position=rand()%timeslots;
while(allocated<1&&(tries<((timeslots/2)+2))){
                    isthereconflict=0;
                            if(timetable[position]==NULL){
                            struct                courseintb                *temp=(struct
courseintb*)malloc(sizeof(COURSEINTB));
                            temp->course=coursetoalloc;
                            temp->next=NULL;
                            timetable[position]=temp;
                            allocated=1;
                            }else{
                                    struct courseintb *temp=timetable[position];
                                            do{
                                                    int tempcourse=temp->course;

        if(conflictmatrix[coursetoalloc-1][tempcourse-1]==1)
                                                                                    {

        isthereconflict=1;
                                                                                    }
                                            temp=temp->next;
                                            }while(temp!=NULL);
                                            if(isthereconflict==1)
                                                                    {

        position=rand()%timeslots;

        tries++;

        //printf(" rand2 has given %d\n",position);
                                                                            }
                                                                    else{
                                                                            struct
courseintb *temp=(struct courseintb*)malloc(sizeof(COURSEINTB));
                                                                            temp-
>course=coursetoalloc;
                                                                            temp-
>next=timetable[position];

        timetable[position]=temp;

        allocated=1;

isthereconflict=0;
                                                                    }
                                    }
                            if(tries>=(timeslots/2)){

        tries=forceallocation(coursetoalloc,conflictmatrix,timeslots,timetable);
```

```
                                }
        }

}

int   forceallocation(int   coursetoalloc,int   **conflictmatrix,int   timeslots,struct
courseintb** timetable){
       int allocated=0,isthereconflict,tries=0;
    int position=0,maxtries=5000;
            tries=0;
            position=0;
            while(allocated<1&&(tries<timeslots)){
                           isthereconflict=0;
                                  if(timetable[position]==NULL){
                                  struct         courseintb         *temp=(struct
courseintb*)malloc(sizeof(COURSEINTB));
                                  temp->course=coursetoalloc;
                                  temp->next=NULL;
                                  timetable[position]=temp;
                                  allocated=1;
                                  }else{
                                        struct                          courseintb
*temp=timetable[position];
                                              do{
                                                    int         tempcourse=temp-
>course;

       if(conflictmatrix[coursetoalloc-1][tempcourse-1]==1)

              {

              isthereconflict=1;

              }
                                              temp=temp->next;
                                              }while(temp!=NULL);
                                              if(isthereconflict==1)
                                                                                  {

       position++;
       tries++;
       //printf(" rand2 has given %d\n",position);
              }

       else{

       struct courseintb *temp=(struct courseintb*)malloc(sizeof(COURSEINTB));
       temp->course=coursetoalloc;
              temp->next=timetable[position];
       timetable[position]=temp;
              allocated=1;
              isthereconflict=0;
                           }
                                        }

         if(position>=timeslots){
              position=rand()%timeslots;
              struct courseintb *temp=(struct courseintb*)malloc(sizeof(COURSEINTB));
              temp->course=coursetoalloc;
```

```
                temp->next=timetable[position];
                timetable[position]=temp;
                allocated=1;
               isthereconflict=0;
          }

       }

    return maxtries;
}



//pinakas deiktwn pros to kathe chromosome
struct chromosome  **createptrtable(struct chromosome *ptr,int POPSIZE){
    struct chromosome **ptrtable=NULL;
       int i=0;
       ptrtable=(struct chromosome **)malloc(POPSIZE*sizeof(LINK3));
       do{
              ptrtable[i]=ptr;
              ptr=ptr->next;
              i++;
       }while(ptr!=NULL);
return ptrtable;
}

//safes
double calculatecost(struct student *ptr,struct courseintb **timetable,int timeslots){
       double cost=0.0;
       int studentcount=0;
       do{
              cost+=costforstudent(ptr->courses,timetable,timeslots);
              studentcount++;
              ptr=ptr->next;
       }while(ptr!=NULL);

return cost/(studentcount);
}


//safes
double costforstudent(int *student,struct courseintb **timetable,int timeslots){
       double studentcost=0.0;
       int temp=0,temp2=0,distance=0,distance2=0,i=0,j=0;
       while(student[i]!=0){
              temp=student[i];
              j=i;
              while(student[j+1]!=0){
         temp2=student[j+1];
         distance=lookforcourse(temp,timetable,timeslots);
         distance2=lookforcourse(temp2,timetable,timeslots);
         studentcost+=distancecost(distance,distance2);
         j++;
              }
i++;
       }
       return studentcost;
}
```

```
//gia debug
int lookforcourse(int coursetofind,struct courseintb **timetable,int timeslots){
	int timeslot=0,i;
	for(i=0;i<timeslots;i++){
		struct courseintb *temp=timetable[i];
		    while(temp!=NULL){

				if(coursetofind==temp->course){
					timeslot=i;
					goto tab;
				}
			temp=temp->next;
			 }

		    }
tab:

	return timeslot;
}


//gia debug
void countcourses(struct courseintb **timetable,int timeslots){
	int sum=0,i;
	for(i=0;i<timeslots;i++){
		struct courseintb *temp=timetable[i];
				while(temp!=NULL){
					sum+=temp->course;
					temp=temp->next;
				 }
	}
//printf("course sum is %d\n",sum);
}
////kostos analoga to distance


double distancecost(int A,int B){
	double cost=0;
	int C=abs(B-A);
	switch (C){
	case 0:cost=400000000.0;break;
	case 1:cost=16.0;break;
	case 2:cost=8.0;break;
	case 3:cost=4.0;break;
	case 4:cost=2.0;break;
	case 5:cost=1.0;break;
	default:cost=0.0;break;
	}
	return cost;
}
//tournament selection
LINK3	selectfromtournament(struct	chromosome	**previeousgeneration,int
tournamentsize,int POPSIZE,int timeslots){
	struct chromosome *best=NULL,*testsubject=NULL,*newvalue=NULL;
	int i;
	best=previeousgeneration[rand()%POPSIZE];

	for(i=0;i<tournamentsize;i++)
	{
```

```
            testsubject=previeousgeneration[rand()%POPSIZE];
            if((testsubject->cost)<=(best->cost)){
                  best=testsubject;

            }
      }
      newvalue=replicatebest(best,timeslots);
      return newvalue;
}
//anaparagogi
void reproducegenes(struct chromosome **generation,int POPSIZE,int crossoverprob,int
timeslots){
      int randomtable[POPSIZE],i,counter=0,erased=0;
      for(i=0;i<POPSIZE;i++){
                                    randomtable[i]=(rand()%100);
                                    if(randomtable[i]<=crossoverprob)
                                          {
                                          counter++;
                                          }
                                    }
            if(counter%2!=0){
            int j=0;
            do{
                  if(randomtable[j]<=crossoverprob){
                                    randomtable[j]=2*crossoverprob;
                                    erased=10;
                                    }
                  j++;
            }while(erased<1);
      }


   struct chromosome* parent1=NULL;
   struct chromosome* parent2=NULL;
   int p=0,parent2found=0;
   while(p<POPSIZE){
                     if(randomtable[p]<=crossoverprob)
                     {
                           int j=p+1;
                           parent2found=0;
                           parent1=generation[p];
                                 while(parent2found==0)
                                 {
                                             if (randomtable[j]<=crossoverprob)
                                                   {
                                                         parent2found=1;
                                                         parent2=generation[j];
                                                               }
                                             j++;
                                       }
                  crossover(parent1,parent2,timeslots);
         p=j-1;//worst bug of all time
          }
p++;
   }
}

//anaparagogi dio genes pou orizontai opos prin
```

```
void crossover(struct chromosome *parent1,struct chromosome *parent2,int timeslots){
       int crossoverposition=0;
       while (crossoverposition==0){crossoverposition=rand()%timeslots;}
       int i;
       for(i=0;i<crossoverposition;i++)
           {
              struct courseintb *temp=NULL;
           temp=parent1->timetable[i];
              parent1->timetable[i]=parent2->timetable[i];
              parent2->timetable[i]=temp;
              }
}


//mutate
void    mutate(struct    chromosome**    generation,int    POPSIZE,int    mutationprob,int
timeslots,int **conflictmatrix){
       int k;
       for(k=0;k<POPSIZE;k++)
       {
       mutateslot(generation[k],conflictmatrix,timeslots,mutationprob);
       }

}


void    mutateslot(struct    chromosome*    genetorescedule,int    **conflictmatrix,int
timeslots,int mutationprob){
       int i;
       for(i=0;i<timeslots;i++){
              struct       courseintb       *todelete,*previous,*temp=genetorescedule-
>timetable[i];
              int test=rand()%10000;
              if (test<mutationprob){
                     todelete=genetorescedule->timetable[i];
                     genetorescedule->timetable[i]=genetorescedule->timetable[i]->next;
                     free(todelete);
              }
              else{
                     previous=temp;
                     temp=temp->next;
                     while(temp!=NULL){
                            test=rand()%10000;
                            if(test<mutationprob){
                                   todelete=temp;
                                   previous->next=todelete->next;
                                   temp=temp->next;
                                   free(todelete);
                            }
                            else{
                                   previous=temp;
                                   temp=temp->next;
                     }
                       }
            }
   }
}
```

```
void repair(struct chromosome **nextgen,int courses,int timeslots,int POPSIZE,int
**conflictmatrix,struct student *ptrtostudent){
	int i;
	for(i=0;i<POPSIZE;i++)
		{
		repeargene(nextgen[i],courses,timeslots,conflictmatrix);
		nextgen[i]->cost=calculatecost(ptrtostudent,nextgen[i]-
>timetable,timeslots);
		countcourses(nextgen[i]->timetable,timeslots);
		//printf("new timetable cost is %lf \n",nextgen[i]->cost);
		}
}


void repeargene(struct chromosome *ptrtogene,int courses,int timeslots,int
**conflictmatrix){
	int i;
	for(i=1;i<courses+1;i++){
		repearcourse(i,ptrtogene->timetable,timeslots,conflictmatrix);
	}
}

void repearcourse(int coursetorepear,struct courseintb **timetable,int timeslots,int
**conflictmatrix){

	int *deletiontb=(int*)malloc(sizeof(int)*60),k;
	for(k=0;k<60;k++){
	deletiontb[k]=-1;
	}
	int count=scanttb(coursetorepear,deletiontb,timeslots,timetable);
    switch(count){
    case 0:positioninwhichtoalloc(coursetorepear-
1,conflictmatrix,timeslots,timetable);break;
    case 1:;break;
    default:deletemultiple(coursetorepear,deletiontb,count,timetable);break;
    }
    free(deletiontb);
}


int scanttb(int coursetorepear,int *deletiontb,int timeslots,struct courseintb
**timetable){
	int sum=0,i,pos=0;
	for(i=0;i<timeslots;i++){
		struct courseintb *temp=timetable[i];
		do{
			if(temp->course==coursetorepear){
				sum++;
				deletiontb[pos]=i;
				pos++;
			}
			temp=temp->next;
		}while(temp!=NULL);
	}
	return sum;
}
```

```
void deletemultiple(int coursetorepear,int *deletiontb,int count,struct courseintb
**timetable){
	int survivor=(rand()%count),i=0,tries=0; //esto oti exoume 3 fores to mathima
vrei.prepei na dialeksoume stin tixi enan na zisei.ton survivor kai svinoume ta alla
	struct courseintb *previous,*temp,*todelete=NULL;
while(deletiontb[i]!=-1){
                            if(tries==survivor){
                                  i++;
                                  tries++;
                                  continue;
                            }
                            else
                            {
                              temp=timetable[deletiontb[i]];
                              if(timetable[deletiontb[i]]->course==coursetorepear){
                                   todelete=timetable[deletiontb[i]];
                                   timetable[deletiontb[i]]=timetable[deletiontb[i]]-
>next;
                                   free(todelete);
                              }
                              else
                              {
                                  previous=temp;
                                  temp=temp->next;
                             while(temp!=NULL){
                                                 if(temp->course==coursetorepear)
                                                 {
                                                       todelete=temp;
                                                       temp=temp->next;
                                                       previous->next=todelete->next;
                                                       free(todelete);
                                                 }
                                                 else{
                                                       previous=temp;
                                                       temp=temp->next;
                                                 }
                                            }
                                       }
                             }

                             i++;
                             tries++;
                           }
}


struct chromosome* replicatebest(struct chromosome* pointer,int timeslots){
struct chromosome *best=(struct chromosome*)malloc(sizeof(CHROMOSOME));
int i;
best->timetable=Make2dintArray(timeslots);
best->cost=pointer->cost;
for(i=0;i<timeslots;i++){
	struct courseintb *temppointer=pointer->timetable[i];
	do{
		struct courseintb *temp=(struct courseintb*)malloc(sizeof(COURSEINTB));
		temp->course=temppointer->course;
		temp->next=best->timetable[i];
		best->timetable[i]=temp;
```

```
        temppointer=temppointer->next;
      }while(temppointer!=NULL);
}
return best;
}


void   deletepreviousgeneration(struct   chromosome   **initialpop,struct   chromosome
**nextgen,int POPSIZE,int timeslots){
      int i=0;
      for(i=0;i<POPSIZE;i++){
            struct chromosome *temp=initialpop[i];
          initialpop[i]=nextgen[i];
            myfree(temp,timeslots);

      }
}


void myfree(struct chromosome* tofree,int timeslots){
      int i;
      for(i=0;i<timeslots;i++){
            struct courseintb *temp,*temp2=tofree->timetable[i];

            do{
           temp=temp2;
           temp2=temp2->next;
           free(( struct courseinttb *)temp);
              }while(temp2!=NULL);

      }

}

void export( float averagecost,float bestcost){
      FILE *fp ;
      FILE *fp2;
          if((fp=fopen("bestdata.txt","a"))==NULL)
                        {
                        fprintf(stderr,"Error opening file data .txt");
                        exit(0);
                        }
          else
                        {
                              //printf("File opened  succesfully  ");
                    fprintf(fp," %lf \n" ,bestcost);
                        }
      fclose(fp);

      if((fp2=fopen("averagedata.txt","a"))==NULL)
                              {
                              fprintf(stderr,"Error opening file data .txt");
                              getchar();
                              }
                else
                              {
                                    //printf("File opened  succesfully  ");
                           fprintf(fp2," %lf \n",averagecost);
                              }
```

```
	fclose(fp2);

}

//header file functions.h
#ifndef FUNCTIONS_H_
#define FUNCTIONS_H_

struct course
{
     int enrolment;
     int courses_num;
      struct course *next;
};

struct student{
	int student_id;
	int courses[16];
	struct student *next;
};


typedef struct course COURSE;
typedef COURSE *LINK;
typedef struct student STUDENT;
typedef STUDENT *LINK2;
typedef struct sorted PROSTAKSINOMISI;

struct course *readcourseslist(int *a);
void printcourses( struct course *ptr,int crs_to_find);
struct student *load_student(void);
void printstudent(struct student *ptr,int Idlookup);
int** create_matrix(struct student *ptr,int courses_num);
int** Make2DDoubleArray(int arraySizeX, int arraySizeY);
int hascourse(int course,int *theArray,int courses[]);
void printallstudents(struct student *ptr);


#endif

//header file timetabling.h
#ifndef TIMETABLING_H_
#define TIMETABLING_H_


struct courseintb{
	int course;
	struct courseintb *next;

};

typedef struct courseintb COURSEINTB;
typedef COURSEINTB *LINK4;

struct chromosome{
	struct courseintb **timetable;
	double cost;
	struct chromosome *next;
};
```

```
typedef struct chromosome CHROMOSOME;
typedef CHROMOSOME *LINK3;


void export(float averagecost,float bestcost);
struct chromosome* create_timetable(int** conflictmatrix,int timeslots,int
courses,struct chromosome **ptr,struct student *ptrtostudentlist);
void fillcourses(int **conflictmatrix,int *sortedmatrix,int courses);
int clashnumber(int *conflictrow,int courses);
struct chromosome *initiatepop(int **conflictmatrix ,int *sortedmatrix ,int courses
,int popsize,int timeslots,struct student *ptrtostudentlist);
double costforstudent(int *student,struct courseintb **timetabl,int timeslots);
struct courseintb** Make2dintArray(int arraySizeX);
void filltable(struct courseintb **timetable,int** conflictmatrix,int courses,int
*sortedmatrix,int timeslots);
void allocate(int courses,int **conflictmatrix,int *sortedmatrix,int
*coursetoallocate,int timeslots,struct courseintb **timetable);
void positioninwhichtoalloc(int maxposition,int **conflictmatrix,int timeslots,struct
courseintb** timetable);
void create_individual(int **conflictmatrix ,int *sortedmatrix ,int courses,int
timeslots,struct chromosome** ptrtoroot,struct student *ptrtostudentlist);
struct chromosome  **createptrtable(struct chromosome *ptr,int POPSIZE);
double calculatecost(struct student *ptr,struct courseintb **timetabl,int timeslots);
int lookforcourse(int coursetofind,struct courseintb **timetable,int timeslots);
double distancecost(int A,int B);
struct chromosome* replicatebest(struct chromosome* pointer,int timeslots);
void myfree(struct chromosome* tofree,int timeslots);
struct chromosome* selectfromtournament(struct chromosome
**previeouschromosomeration,int tournamentsize,int POPSIZE,int timeslots);
void reproducegenes(struct chromosome **generation,int POPSIZE,int crossoverprob,int
timeslots);
void crossover(struct chromosome *parent1,struct chromosome *parent2,int timeslots);
void mutate(struct chromosome** generation,int POPSIZE,int mutationprob,int
timeslots,int **conflictmatrix);
void mutateslot(struct chromosome* genetorescedule,int **conflictmatrix,int
timeslots,int mutationprob);
void repair(struct chromosome **nextgen,int courses,int timeslots,int POPSIZE,int
**conflictmatrix,struct student* ptrtostudent);
void repeargene(struct chromosome *ptrtogene,int courses,int timeslots,int
**conflictmatrix);
void repearcourse(int coursetorepear,struct courseintb **timetable,int timeslots,int
**conflictmatrix);
void deletepreviousgeneration(struct chromosome **initialpop,struct chromosome
**nextgen,int POPSIZE,int timeslots);
int scantimeslot(int *timetablerow,int coursetofind);
void deletemultiple(int coursetorepear,int *deletiontb,int count,struct courseintb
**timetable);
void countcourses(struct courseintb **timetable,int timeslots);
int scanttb(int coursetorepear,int *deletiontb,int timeslots,struct courseintb
**timetable);
int forceallocation(int coursetoalloc,int **conflictmatrix,int timeslots,struct
courseintb** timetable);
#endif
```

ΠΑΡΑΡΤΗΜΑ Β :Παρουσίαση Συναρτήσεων του κώδικα (API)

- **int choices(void)**: Συνάρτηση που δέχεται την αριθμητική επιλογή του χρήστη από το πληκτρολόγιο και την επιστρέφει σε ακέραιο.

- **void exportbesttimetable(struct chromosome *ptr,int timeslots,float duration)**: Συνάρτηση που δέχεται σαν όρίσματα το timetable προς export καθώς και τον αριθμό των timeslots και την διάρκεια που πήρε στον κώδικα να εκτελεστεί και τα κάνει export.

- **struct course *readcourseslist(int *globalcoursesaddress)**: Συνάρτηση που δέχεται σαν όρισμα την διεύθυνση μιας ακέραιης τιμής ώστε να αποθηκεύσει τον συνολικό αριθμό των μαθημάτων και επιστρέφει τον root Της Linked list με τα μαθήματα.

- **struct student *load_student(void)**: Συνάρτηση που επιστρέφει τον root της linked list των μαθητών.

- **void printcourses(struct course *ptr,int crs_to_find)**: Συνάρτηση που δέχεται τον Root της linked list Των μαθημάτων και έναν ακέραιο Id προς αναζήτηση και αν υπάρχει στην Ram το αντίστοιχο μάθημα εκτυπώνει τα enrollments.

- **void printstudent(struct student *ptr,int Idlookup)**: Συνάρτηση που δέχεται τον root Της linked list των μαθητών και το Id ενός μαθητή και αν αυτός υπάρχει εκτυπώνει τα μαθήματα που έχει πάρει αυτός.

- **void printallstudents(struct student *ptr)**: Συνάρτηση που παίρνει τον root της λίστας των μαθητών και τους εκτυπώνει όλους.

- **int** create_matrix(struct student *ptr,int size)**: Συνάρτηση που δέχεται την λίστα με τους μαθητές και των αριθμό των timeslots Και επιστρέφει ένα δείκτη προς τον Πίνακα των συγκρούσεων [timeslots * timeslots].

- **int hascourse(int course,int *theArray,int courses [])**: Βοηθητική συνάρτηση όπου δέχεται σαν ορίσματα ένα μάθημα ,μια σειρά του Conflict matrix και τα μαθήματα ενός μαθητή και γράφει 1 στον conflict matrix άμα δει Conflict .Τέλος επιστρέφει 1 όταν βρει το μάθημα που ζητούσε.

- **int**** Make2DDoubleArray(int arraySizeX, int arraySizeY)**: Συνάρτηση που δημιουργεί έναν πίνακα X*Y από τα ορίσματα που δέχεται και επιστρέφει τον δείκτη προς τον πίνακα.

- **struct chromosome* create_timetable(int** conflictmatrix,int timeslots,int courses,struct chromosome **ptr,struct student *ptrtostudentlist)**: Η βασική συνάρτηση της διπλωματικής. Δέχεται σαν όρισμα τον πίνακα των συγκρούσεων, τα timeslots ,τον αριθμό των μαθημάτων, μια διεύθυνση για την αποθήκευση της λίστας με τα χρωμοσώματα, τον root της λίστας με τους μαθητές και επιστρέφει δείκτη προς το καλύτερο timetable που βρέθηκε.

- **void fillcourses(int** conflictmatrix,int* sortedmatrix,int courses)**:Αυτή η συνάρτηση δέχεται τον CM σαν πρώτο όρισμα και τον αριθμό των μαθημάτων σαν

τρίτο.Σκόπός της είναι να επιστρέψει στο δεύτερο όρισμα ένα ταξινομημένο πίνακα με τα μαθήματα όπως εκφράζονται από το Largest Degree.

- **int clashnumber**(int *conflictrow,int courses):Βοηθητική συνάρτηση στην Fillcourses όπου παίρνει σαν όρισμα μια σειρά του CF και τον αριθμό των μαθημάτων και επιστρέφει το πόσα μη μηδενικά στοιχεία βρήκε.

- **struct** chromosome ***initiatepop**(int **conflictmatrix ,int *sortedmatrix ,int courses ,int POPSIZE,int timeslots,struct student *ptrtostudentlist):Βασική συνάρτηση οπου παίρνει σαν ορίσματα τον CM, τον πίνακα του largest degree,τα μαθήματα, το μέγεθος του πληθυσμού, τα timeslots και τον root της student list και επιστρέφει τον Root μιας Linked list με τα χρωμοσώματα που έχουν δημιουργηθεί.

- **void create_individual**(int **conflictmatrix ,int *sortedmatrix ,int courses,int timeslots,struct chromosome** ptrtoroot,struct student *ptrtostudentlist): Βοηθητική συνάρτηση που στόχος της είναι να δημιουργήσει ένα χρωμόσωμα. Δέχεται σαν ορίσματα τον CM, τον πίνακα του Largest Degree, τα μαθήματα, τα timeslots ,τον Root Της λίστας τον χρωμοσωμάτων και τον root της student list.

- **struct** courseintb** **Make2dintArray**(int arraySizeX): Συνάρτηση η οποία δέχεται σαν όρισμα ένα μέγεθος και επιστρέφει έναν πίνακα δεικτών προς δομές τύπου Courseintb μεγέθους arraySizeX.

- **void filltable**(struct courseintb **timetable,int** conflictmatrix,int courses,int *sortedmatrix,int timeslots): Συνάρτηση που έχει σαν σκοπό να γεμίσει το άδειο timetable ενός χρωμοσώματος. Δέχεται σαν ορίσματα το timetable, τον CD,τα μαθήματα, τον πίνακα του LD και τα timeslots.

- **void allocate**(int courses,int **conflictmatrix,int *sortedmatrix,int *coursetoallocate,int timeslots,struct courseintb **timetable): Συνάρτηση που προσπαθέι να κάνει allocate τα μαθήματα όπως περιγράφτηκε στο κείμενο της διπλωματικής.

- **void positioninwhichtoalloc**(int maxposition,int **conflictmatrix,int timeslots,struct courseintb** timetable): Συνάρτηση που προσπαθεί να κάνει allocate σε ένα timetable το μάθημα που δίνεται στο πρώτο όρισμα.Τα αλλα ορίσματα είναι ο CM,τα timeslots ,και το timetable που προσπαθούμε να βάλουμε το μάθημα.

- **int forceallocation**(int coursetoalloc,int **conflictmatrix,int timeslots,struct courseintb** timetable): Ίδια με την προηγούμενη με μοναδική διαφορά ότι βάζει υποχρεωτικά το μάθημα στο timetable ακόμα και αν βρει συγκρούσεις.

- **struct** chromosome ****createptrtable**(struct chromosome *ptr,int POPSIZE): Συνάρτηση που κατασκευάζει έναν πίνακα δεικτών προς τα χρωμοσώματα και επιστρέφει ένα δείκτη προς αυτό τον πίνακα. Σαν ορίσματα δέχεται τον root των χρωμοσωμάτων και το μέγεθος του πληθυσμού.

- **double calculatecost**(struct student *ptr,struct courseintb **timetable,int timeslots): Συνάρτηση υπολογισμού κόστους που επιστρέφει το κόστος και δέχεται σαν

ορίσματα τον Root των μαθητών, ένα timetable για να υπολογίσει το κόστος του και τα timeslots.

- **double costforstudent**(**int** *student,**struct** courseintb **timetable,**int** timeslots): Συνάρτηση που υπολογίζει το κόστος για έναν μαθητή και δέχεται σαν ορίσματα τον πίνακα των μαθημάτων του μαθητή, το timetable και τα timeslots.

- **int lookforcourse**(**int** coursetofind,**struct** courseintb **timetable,**int** timeslots): Συνάρτηση αναζήτηση μαθήματος(όρισμα 1) σε ένα timetable(όρισμα 2).Τελευταίο όρισμα είναι τα timeslots.

- **void countcourses**(**struct** courseintb **timetable,**int** timeslots): Συνάρτηση που μετράει τα μαθήματα που βρίσκονται σε ένα timetable(1).

- **double distancecost**(**int** A,**int** B): Συνάρτηση που επιστρέφει το κόστος δυο μαθημάτων τύπου double που βρίσκονται στα timeslots A,B που είναι τα ορίσματα της.

- LINK3 **selectfromtournament**(**struct** chromosome **previeousgeneration,**int** tournamentsize,**int** POPSIZE,**int** timeslots): Συνάρτηση που κάνει επιλογή από τον πίνακα της προηγούμενης γενιάς(1) ,έναν αριθμό από χρωμοσώματα (2) από τον συνολικό πληθυσμό (3)για την επόμενη γενιά. Τελευταίο όρισμα είναι τα timeslots.

- **void reproducegenes**(**struct** chromosome **generation,**int** POPSIZE,**int** crossoverprob,**int** timeslots): Συνάρτηση που υλοποιεί την αναπαραγωγή μεταξύ γονιδίων. Δέχεται σαν ορίσματα τον πίνακα της γενιάς, το μέγεθος του πληθυσμού, μια πιθανότητα αναπαραγωγής και τα timeslots.

- **void crossover**(**struct** chromosome *parent1,**struct** chromosome *parent2,**int** timeslots): Βοηθητική συνάρτηση που υλοποιεί την αναπαραγωγή μεταξύ των δυο γονέων που είναι τα ορίσματα της (1)(2).Δέχεται σαν τελευταίο όρισμα τα timeslots.

- **void mutate**(**struct** chromosome** generation,**int** POPSIZE,**int** mutationprob,**int** timeslots,**int** **conflictmatrix): Υλοποίηση της Mutate(delete) ενός μαθήματος από ένα timetable.Δέχεται σαν ορίσματα τον πίνακα με τα χρωμοσώματα, το μέγεθος του πληθυσμού, μια πιθανότητα μετάλαξης και τον CM.

- **void mutateslot**(**struct** chromosome* genetorescedule,**int** **conflictmatrix,**int** timeslots,**int** mutationprob): Βοηθητική συνάρτηση για την Mutate.Δέχεται ένα συγκεκριμένο χρωμόσωμα προς μετάλλαξη (1),τον CM,τα timeslots, και την πιθανότητα μετάλλαξης.

- **void repair**(**struct** chromosome **nextgen,**int** courses,**int** timeslots,**int** POPSIZE,**int** **conflictmatrix,**struct** student *ptrtostudent): Συνάρτηση που θα δει όλα τον πίνακα των χρωμοσωμάτων και θα επισκευάσει τα πάντα. Δέχεται σαν επιπλέον ορίσματα τα μαθήματα, τα timeslots, το μέγεθος του πληθυσμού, τον CM και τον Root της student list.

- **void repeargene**(**struct** chromosome *ptrtogene,**int** courses,**int** timeslots,**int** **conflictmatrix): Βοηθητική συνάρτηση για την επισκευή ενός συγκεκριμένου χρωμοσώματος. Δέχεται σαν ορίσματα το χρωμόσωμα, τα μαθήματα, τα timeslots και τον CM.

- **void repearcourse**(**int** coursetorepear,**struct** courseintb **timetable,**int** timeslots,**int** **conflictmatrix): Συνάρτηση που αναλαμβάνει την επισκευή ενός

συγκεκριμένου μαθήματος. Δέχεται σαν όρισμα το μάθημα προς επισκευή, το timetable Που πρέπει να επισκευαστεί, τα timeslots και τον CM.

- **int scanttb**(int coursetorepear,int *deletiontb,int timeslots,struct courseintb **timetable): Συνάρτηση που δέχεται ένα μάθημα(1) , ένα πίνακα που περιέχει τα timeslots που βρέθηκε αυτό το μάθημα(2), τα timeslots και το timetable και επιστρέφει έναν ακέραιο για το πόσες φορές βρέθηκε το μάθημα στον πίνακα.

- **void deletemultiple**(int coursetorepear,int *deletiontb,int count,struct courseintb **timetable): Συνάρτηση διαγραφής των πολλαπλών μαθημάτων. Δέχεται ένα μάθημα(1) ένα πίνακα που περιέχει τα timeslots που βρέθηκε αυτό το μάθημα(2), πόσα μαθήματα πολλαπλά υπάρχουν(3), στο timetable(4).

- **struct** chromosome* **replicatebest**(**struct** chromosome* pointer,**int** timeslots): Συνάρτηση που δημιουργεί ένα ακριβές αντίγραφο ενός χρωμοσώματος (1). Δεχεται τα timeslots σαν δεύτερο όρισμα και επιστρέφει έναν δείκτη προς το αντίγραφο.

- **void deletepreviousgeneration**(**struct** chromosome **initialpop,**struct** chromosome **nextgen,**int** POPSIZE,**int** timeslots): Συνάρτηση που διαγράφει πλήρως την προηγούμενη γενιά(1) και μεταφέρει τους pointers της νέας γενιάς(2) στον πίνακα της προηγούμενης. Δέχεται επίσης το μέγεθος του πληθυσμού (3) και τα timeslots.

- **void myfree**(**struct** chromosome* tofree,**int** timeslots): Συνάρτηση που ελευθερώνει ένα ολόκληρο χρωμόσωμα(1) από τη μνήμη. Δέχεται σαν δεύτερο όρισμα τα timeslots.

- **void export**(**float** averagecost,**float** bestcost): Συνάρτηση που εξάγει σε εξωτερικό αρχείο το μέσο και το καλύτερο κόστος κάθε γενιάς.

Παράρτημα Γ: Παράλληλες Ρουτίνες.

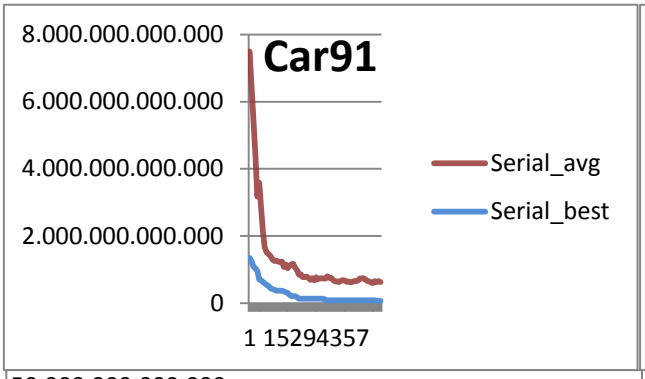
Όπως παρουσιάστηκε και στο κείμενο ο Παράλληλος γενετικός δεν έχει πολλές διαφορές στη δομή του και το μόνο που χρειάζεται από τον προγραμματιστή είναι η υλοποίηση της συνάρτησης για τη μετανάστευση. Κατά συνέπεια θα δοθεί μόνο αυτή στο παράρτημα και ο ενδιαφερόμενος αναγνώστης μπορεί να βρει περισσότερες λεπτομέρειες στον πηγαίο κώδικα . Πριν δοθεί η συνάρτηση απλά θα αναφέρω τον τρόπο λειτουργίας της. Βασιζόμενοι στις συναρτήσεις Mpi_Send Και Mpi_Receive υλοποιήσαμε το εξής. Η συνάρτηση στέλνει όλο το πρόγραμμα των εξετάσεων ενός γονιδίου με ένα μήνυμα και δεν στέλνει το κόστος. Αυτό συμβαίνει για την ελαχιστοποίηση σε κάποιο βαθμό της επικοινωνίας. Αν τα στέλναμε ακέραιο-ακέραιο θα καθυστερούσαμε πολύ περισσότερο. Έτσι λοιπόν δημιουργούμε το chunk των ακεραίων με τον εξής τρόπο. Κατασκευάζουμε έναν πίνακα με αριθμό θέσεων courses που είναι γνωστό μέγεθος και στον παραλήπτη. Έπειτα διαβάζουμε το timetable προς αποστολή και όταν φτάσουμε στο τέλος της σειράς του μαθήματος αλλάζουμε το πρόσημο πριν το βάλουμε στον πίνακα για να ξέρει ο παραλήπτης να δημιουργήσει το ακριβώς ίδιο timetable. Το κόστος του νέου timetable υπολογίζεται εκ νέου στον παραλήπτη και έτσι μπορεί ο χρήστης να ελέγξει για πιθανά λάθη.

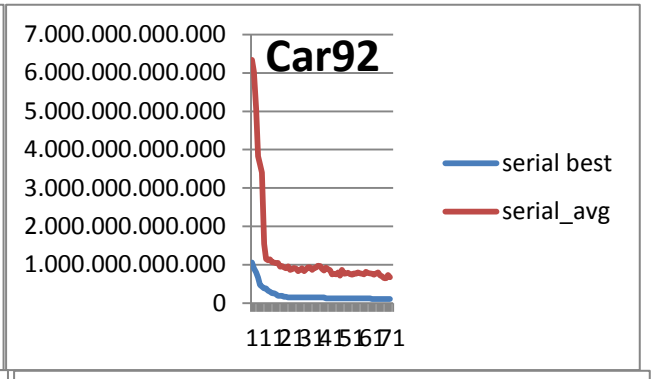
```
void SendBestChromosome(int timeslots,int courses,struct chromosome *tosend){
      int *bufferout=(int *)malloc(sizeof(int)*courses);
            int i,j=0;
            for(i=0;i<timeslots;i++){
                  struct courseintb *temp=tosend->timetable[i];
                  while(temp!=NULL){
                        bufferout[j++]=temp->course;
                        temp=temp->next;
                        if(temp==NULL){
```

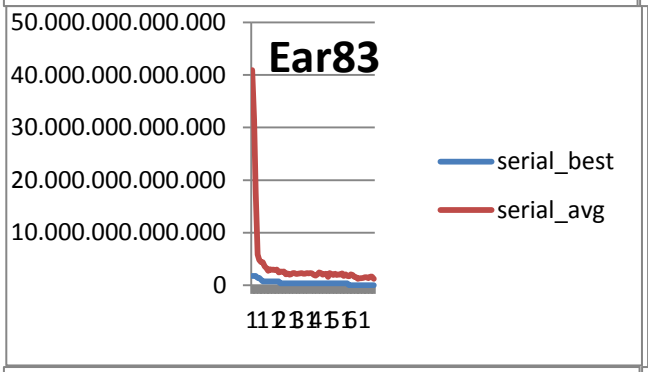
```
                                int reverse=bufferout[j-1];
                                bufferout[j-1]=-reverse;

                        }
                }

        }
        MPI_Send(bufferout,courses,MPI_INT,0,12,MPI_COMM_WORLD);
        free(bufferout);

}



void SendChromosome(int timeslots,int courses,struct chromosome **initialpop,int
POPSIZE,int my_rank,int commsz,struct student *ptrtostudentlist,int bestposition,int
bestflag){

        int destination;
        struct chromosome* tosend=NULL;
        if(my_rank==commsz-1){
      destination=1;
        }
        else{
         destination=my_rank+1;
        }
        if(bestflag<1){
                tosend=replicatebest(initialpop[bestposition],timeslots);
        }
        else
        {
        tosend=replicatebest(initialpop[rand()%POPSIZE],timeslots);
        }
        int *bufferout=(int *)malloc(sizeof(int)*courses);
        int i,k,j=0;
        for(i=0;i<timeslots;i++){
                struct courseintb *temp=tosend->timetable[i];
                while(temp!=NULL){
                        bufferout[j++]=temp->course;
                        temp=temp->next;
                        if(temp==NULL){
                                int reverse=bufferout[j-1];
                                bufferout[j-1]=-reverse;

                        }
                }

        }
MPI_Send(bufferout,courses,MPI_INT,destination,10,MPI_COMM_WORLD);

//fprintf(stderr,"i send to %d a timetable with cost %f\n",destination,tosend->cost );
myfree(tosend,timeslots);
free(bufferout);

}

void ReceiveChromosome(int timeslots,int courses,struct chromosome **initialpop,int
POPSIZE,int my_rank,int commsz,struct student *ptrtostudentlist){
        int receivefrom;
```

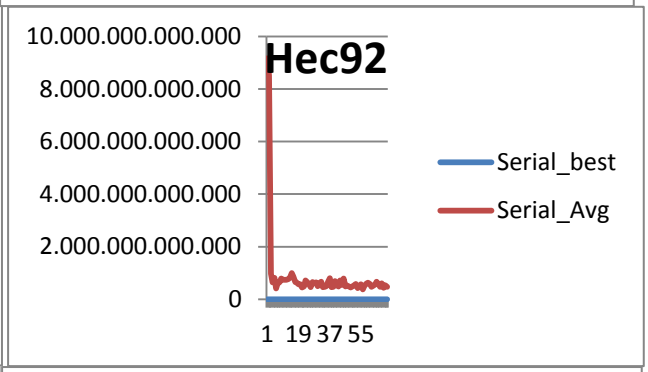
```
       int k;
       int sum=0;
       if(my_rank==1){
        receivefrom=commsz-1;
       }
       else{
        receivefrom=my_rank-1;
       }
    int j,row=0;
       int *bufferin=(int *)malloc(sizeof(int)*courses);
       MPI_Recv(bufferin,courses,MPI_INT,receivefrom,10,MPI_COMM_WORLD,MPI_STATUS_IGNO
RE);

       struct chromosome *received=(LINK3)malloc(sizeof(CHROMOSOME));
       received->timetable=Make2dintArray(timeslots);
       for(j=0;j<courses;j++){
              if(bufferin[j]<0){
                     putin(row,abs(bufferin[j]),received->timetable,timeslots);
                     row++;
              }
              else{
                     putin(row,bufferin[j],received->timetable,timeslots);
              }
       }
received->cost=calculatecost(ptrtostudentlist,received->timetable,timeslots);
//fprintf(stderr,"i received from %d a timetable with cost %f
\n",receivefrom,received->cost);
int position=rand()%POPSIZE;
struct chromosome* random=initialpop[position];
if(random->cost>received->cost){
       struct chromosome* delete=random;
       initialpop[position]=received;
       myfree(random,timeslots);

}
}


void putin(int row,int coursetoput,struct courseintb **timetable,int timeslots){
       struct courseintb *temp2,*temp=(struct courseintb*)malloc(sizeof(COURSEINTB));
       temp->course=coursetoput;
       temp2=timetable[row];
       timetable[row]=temp;
       temp->next=temp2
```

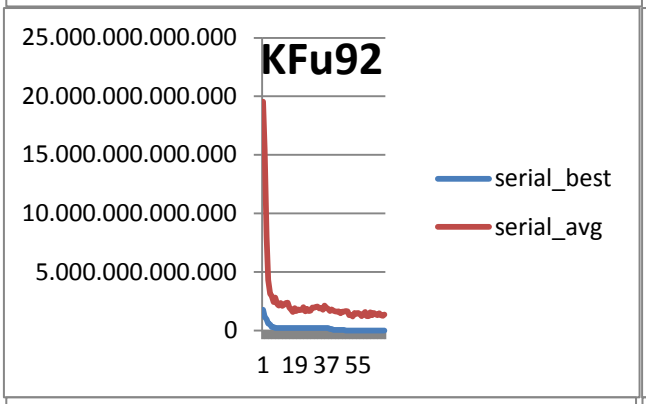
ΠΑΡΑΡΤΗΜΑ Δ: Διαγράμματα σειριακής σύγκλισης

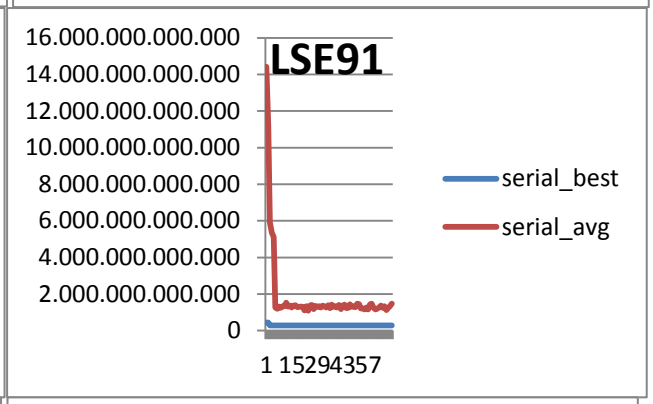


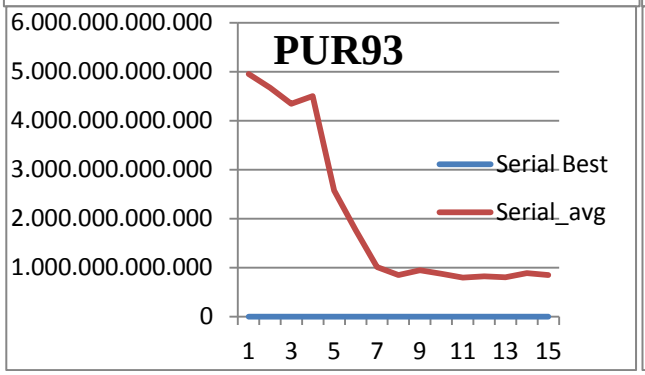


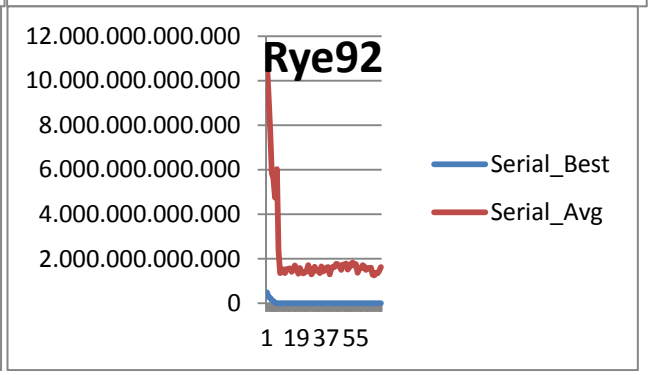


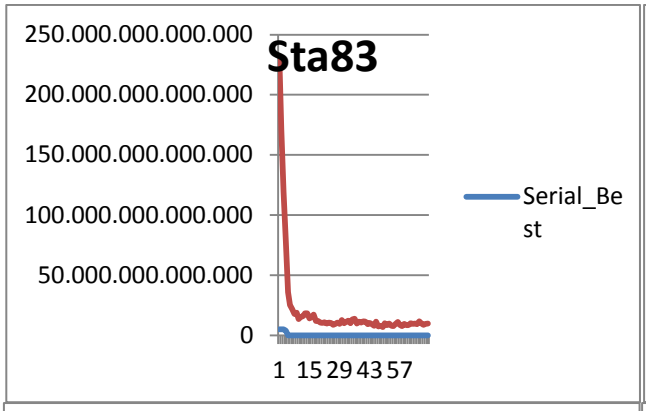


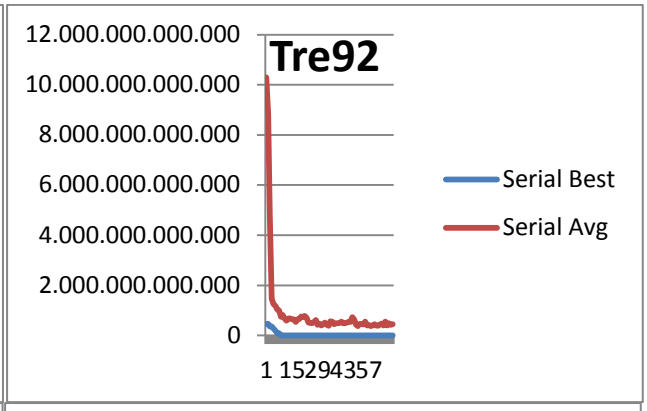


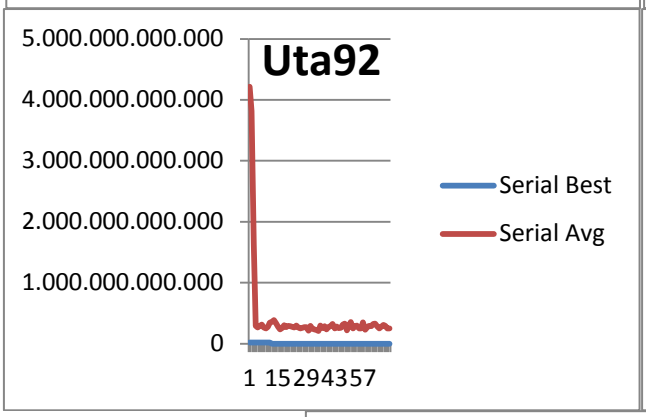


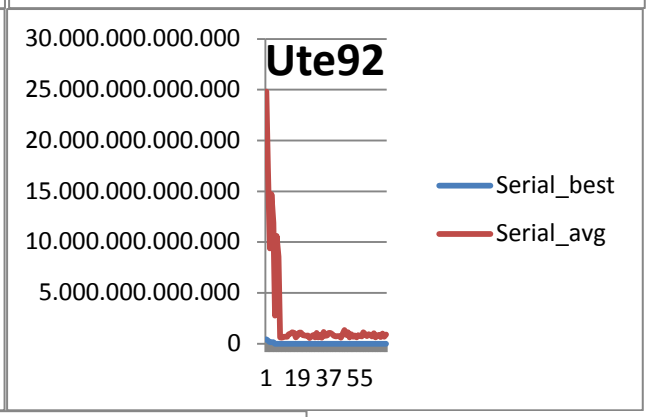


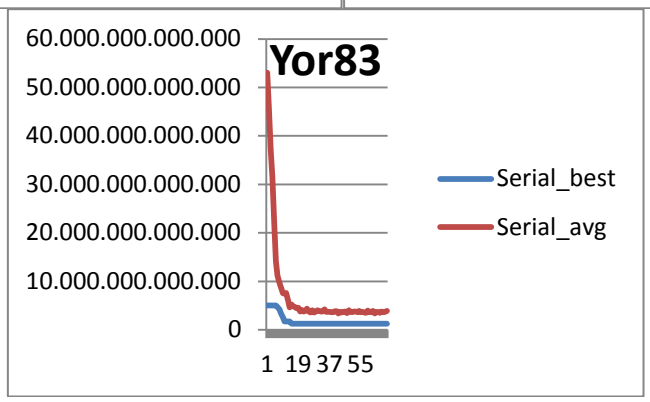



Sta83
250.000.000.000.000
200.000.000.000.000
150.000.000.000.000
100.000.000.000.000
50.000.000.000.000
0
1 15 29 43 57
Serial_Best

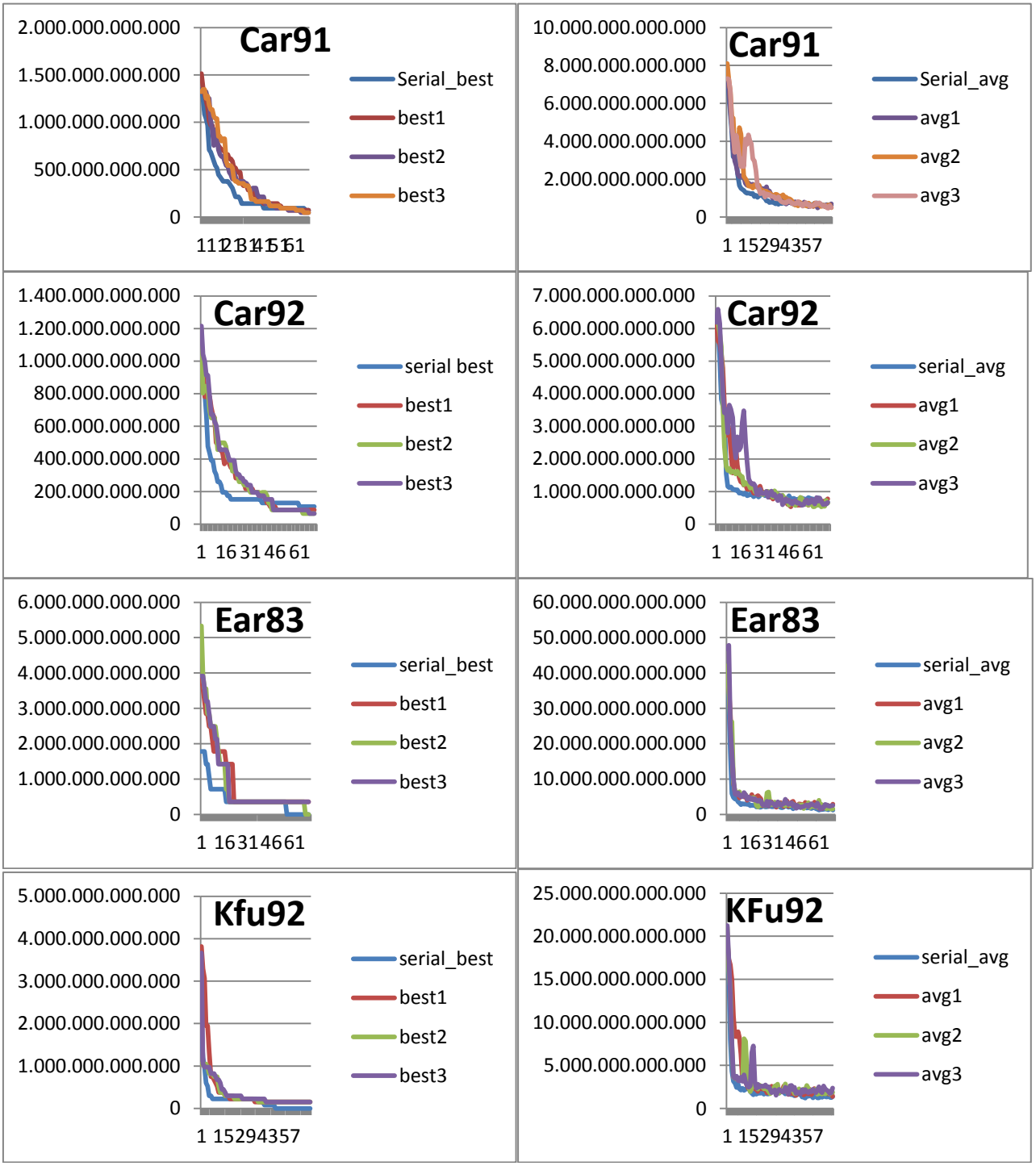

Tre92
12.000.000.000.000
10.000.000.000.000
8.000.000.000.000
6.000.000.000.000
4.000.000.000.000
2.000.000.000.000
0
1 15294357
Serial Best
Serial Avg

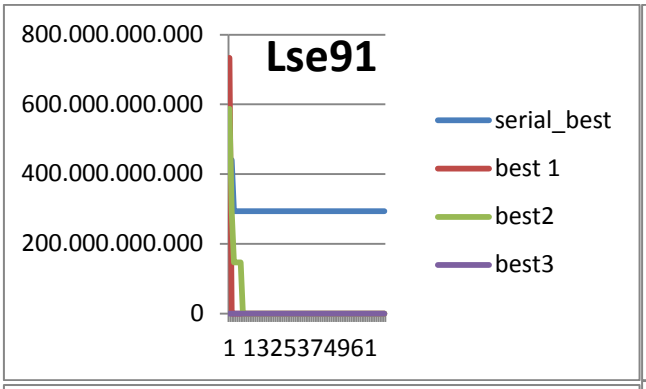

Uta92
5.000.000.000.000
4.000.000.000.000
3.000.000.000.000
2.000.000.000.000
1.000.000.000.000
0
1 15294357
Serial Best
Serial Avg

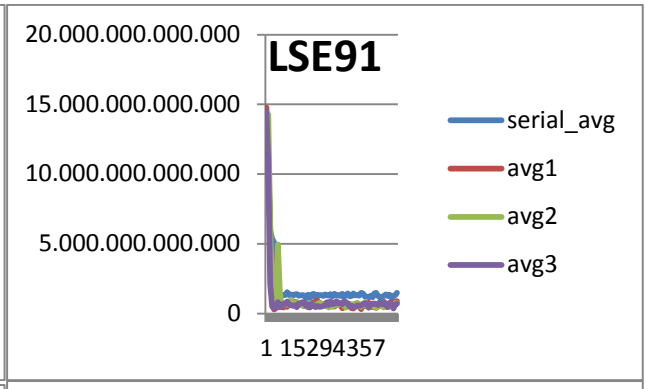

Ute92
30.000.000.000.000
25.000.000.000.000
20.000.000.000.000
15.000.000.000.000
10.000.000.000.000
5.000.000.000.000
0
1 19 37 55
Serial_best
Serial_avg

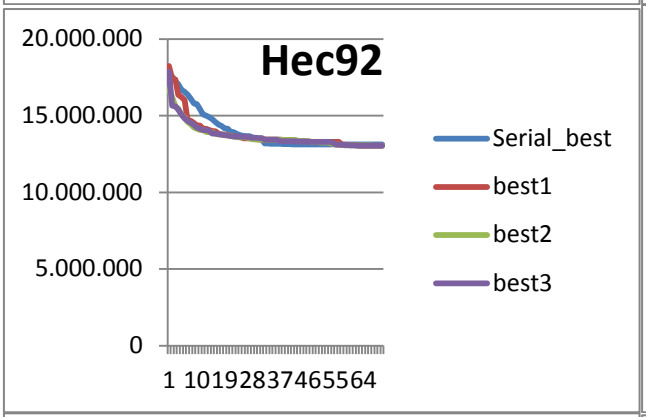

Yor83
60.000.000.000.000
50.000.000.000.000
40.000.000.000.000
30.000.000.000.000
20.000.000.000.000
10.000.000.000.000
0
1 19 37 55
Serial_best
Serial_avg

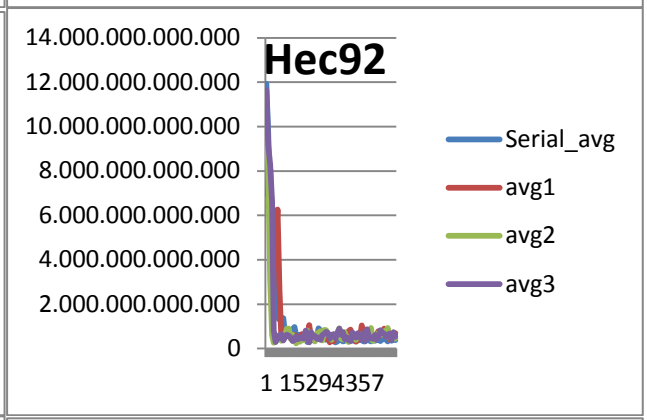
Παράρτημα Ε :Διαγράμματα Παράλληλης Σύγκλισης

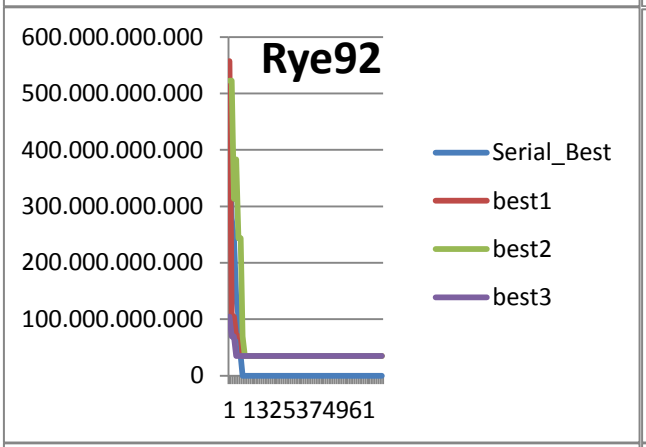


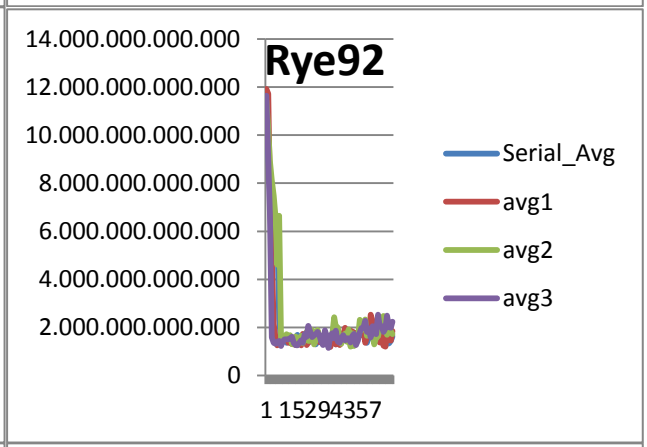

Lse91
800.000.000.000
600.000.000.000
400.000.000.000
200.000.000.000
0
1 1325374961
serial_best
best 1
best2
best3

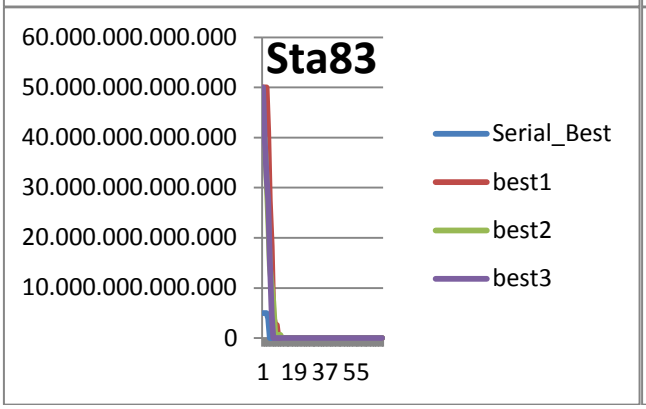

LSE91
20.000.000.000.000
15.000.000.000.000
10.000.000.000.000
5.000.000.000.000
0
1 15294357
serial_avg
avg1
avg2
avg3

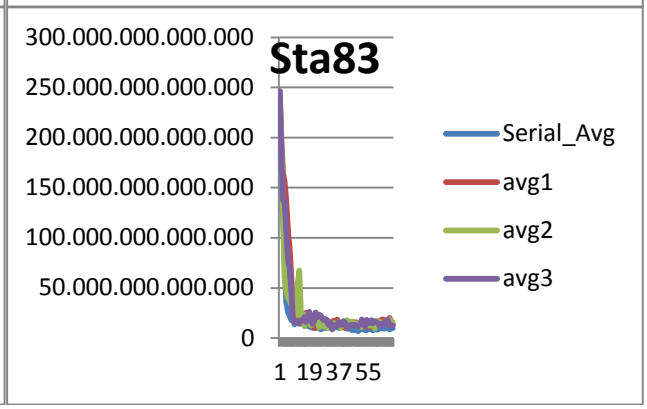

Hec92
20.000.000
15.000.000
10.000.000
5.000.000
0
1 10192837465564
Serial_best
best1
best2
best3

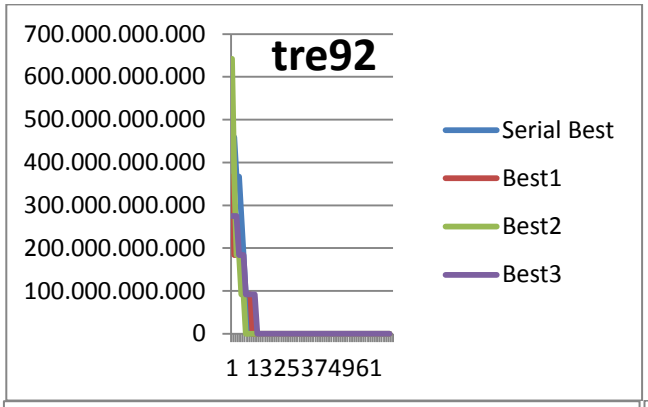

Hec92
14.000.000.000.000
12.000.000.000.000
10.000.000.000.000
8.000.000.000.000
6.000.000.000.000
4.000.000.000.000
2.000.000.000.000
0
1 15294357
Serial_avg
avg1
avg2
avg3

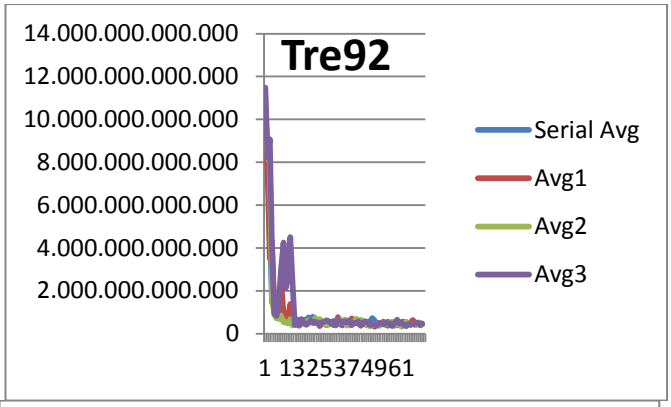

Rye92
600.000.000.000
500.000.000.000
400.000.000.000
300.000.000.000
200.000.000.000
100.000.000.000
0
1 1325374961
Serial_Best
best1
best2
best3

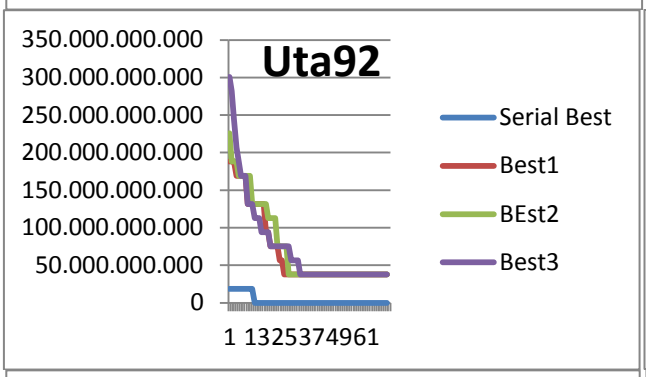

Rye92
14.000.000.000.000
12.000.000.000.000
10.000.000.000.000
8.000.000.000.000
6.000.000.000.000
4.000.000.000.000
2.000.000.000.000
0
1 15294357
Serial_Avg
avg1
avg2
avg3

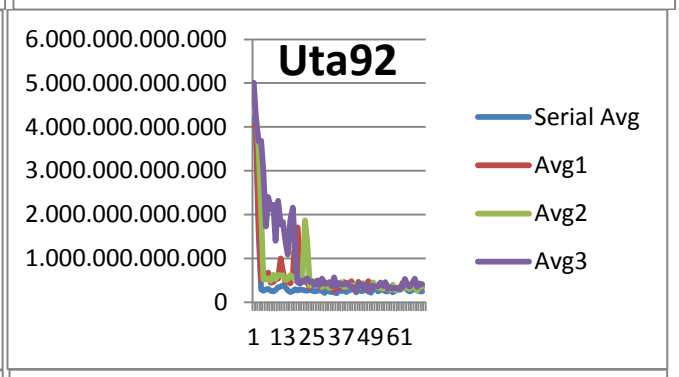

Sta83
60.000.000.000.000
50.000.000.000.000
40.000.000.000.000
30.000.000.000.000
20.000.000.000.000
10.000.000.000.000
0
1 19 37 55
Serial_Best
best1
best2
best3

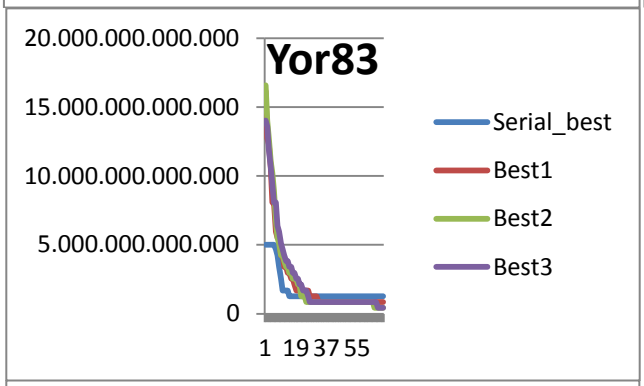

Sta83
300.000.000.000.000
250.000.000.000.000
200.000.000.000.000
150.000.000.000.000
100.000.000.000.000
50.000.000.000.000
0
1 193755
Serial_Avg
avg1
avg2
avg3

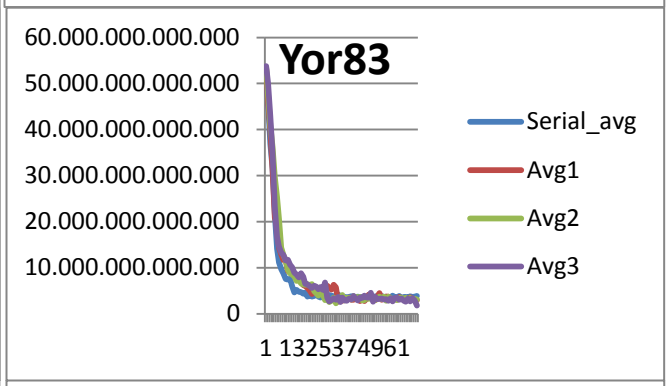


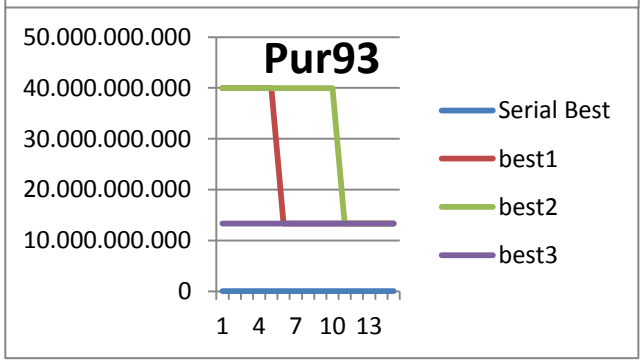


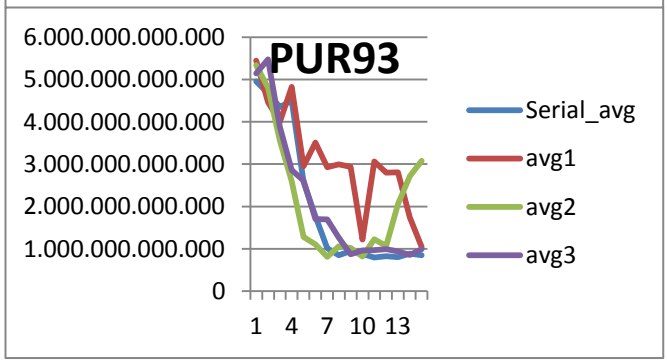












ΠΗΓΕΣ

[1] E.K. Burke, J.H. Kingston and D. deWerra (2004). *Applications to timetabling.*
[2] E.K. Burke, P. De Causmaecker, G. Vanden Berghe and H. Van Landeghem (2004). *The state of the art of nurse rostering*. Journal of Scheduling, 7(6): 441-499.
[3] K. Easton, G. Nemhauser and M. Trick (2004) Sports scheduling. In: J. Leung (ed.3)

Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapter 52. CRC Press.
[4] D.B. Leake (1996). *Case Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI Press/MIT Press.
[5] R. Qu, E. K. Burke, B. McCollum, L.T.G. Merlot, and S.Y. Lee. *A Survey of Search Methodologies and Automated System Development for Examination Timetabling*. Journal of Scheduling
[6] Amr Soghier, *Novel Hyper-heuristic Approaches in Exam Timetabling* PhD @ to The University of Nottingham July 2012
[7] D.J.A.Welsh and M.B. Powell (1967). *The upper bound for the chromatic number of a graph and its application to timetabling problems*. The Computer Journal 11:41-47.
[8] P. Van. Hentenryck (1989). *The OPL Optimization Programmin Language*. The MIT Press
[9] Kalyanmoy Deb, *Multi-Objective Optimization using Evolutionary Algorithms*, WILEY
[10] N. Pillay a, W. Banzhaf b *An informed genetic algorithm for the examination timetabling problem*
[11]I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA, (1995).
[12] Αλογάριαστος Κωσταντίνος *Παραλληλοποίηση γενετικών αλγορίθμων πολλαπλών στόχων (2008)*
[13] http://en.wikipedia.org/wiki/Optimization_problem
[14]Σπυρίδων Λυκοθανάσης *Γενετικοί Αλγόριθμοι και Εφαρμογές* Παν.Πατρών
[15] A. Tanenbaum *Modern Operating Systems (3rd Edition)* (2011)
[16] M. Snir, S. W.Otto, S Lederman, D Walker, J Dongarra *MPI The Complete Reference* Mit Press 1991
[17] P.Pacheco *An introduction to Parallel Programming (2011) Mkp*
[18] Gabriel Luque-Enrique Alba *Parallel Genetic Algorithms. Theory and Real World Applications* 2011 Springer
[19] T.B. Cooper and J.H. Kingston (1996). The *complexity of timetable construction problems*. In: E.K. Burke and P. Ross (eds.) *Practice and Theory of Automated Timetabling: Selected Papers from the 1st International Conference*. Lecture Notes στin Computer Science, vol. 1153, 283-295.
[20] H. Asmuni, E.K. Burke, J. Garibaldi and B. McCollum(2005). *Fuzzy multiple ordering criteria for examination timetabling*
[21] P.H. Corr, B. McCollum, M.A.J McGreevy, P.McMullan (2006). *A new neural network Based construction Heuristic for the examination time tabling problem*
[22] F.Glover, M Laguna. *Tabu Search (1993)*
[23] E.H.L Aarts, J.Korst, M Michiels: *Simulated Annealing* (2005)
[24] M. Dorigo, C. Blumm *Ant Colony optimization theory (2005)*
[25] M.W. Carter, G. Laporte, S. Lee, (1996) *Examination timetabling: Algorithmic strategies and applications, Journal of the Operational Research* Society 47(3), 373 – 383
[26] Melanie Mitchell *An Introduction to Genetic algorithms* Mit Press 1999
[27] Ευταξιόπουλος Χαρίλαος: *Μελέτη εκτέλεσης αλγορίθμων στο πλέγμα Υπολογιστών 2009* Π.Πατρών
[28] B. Kernighan, D Ritchie : *The C programming language* 1988 Prentice Hall
[29] E. Cantu Paz *A Survey of Parallel Genetic Algorithms UIUC* Calculators Paralleles, 1998
[30]Barr, R.S., Hickman, B.L.: Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions. ORSA Journal on Computing 5(1), 2–18 (1993)
[31] http://en.wikipedia.org/wiki/Flynn%27s_taxonomy
[32]M.W. Carter, G. Laporte and S.Y.Lee (1996). *Examination timetabling. Algorithmic strategies and applications.* Journal of Operational Research Society, 47