ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Παράλληλοι υπολογισμοί σε πολυπύρηνα επεξεργαστικά συστήματα με το περιβάλλον Cilk

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Ασημίνας Βουρονίκου

Βόλος, Οκτώβριος 2012

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΔΙΚΤΥΩΝ

Παράλληλοι υπολογισμοί σε πολυπύρηνα επεξεργαστικά συστήματα με το περιβάλλον Cilk

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Της

Ασημίνας Βουρονίκου

Επιβλέποντες:

Παναγιώτα Τσομπανοπούλου	Παναγιώτης Μποζάνης
Επίκουρη καθηγήτρια Πανεπιστήμιο Θεσσαλίας	Αναπληρωτής καθηγητής Πανεπιστήμιο Θεσσαλίας

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 01/10/2012

(Υπογραφή)

.....

ΚΥΡΙΟΣ ΕΠΙΒΛΕΠΩΝ
Επίκουρη καθηγήτρια Πανεπιστήμιο Θεσσαλίας

(Υπογραφή)

.....

ΔΕΥΤΕΡΕΥΩΝ ΕΠΙΒΛΕΠΩΝ
Αναπληρωτής καθηγητής Πανεπιστήμιο Θεσσαλίας

(Υπογραφή)

.....

Βουρονίκου Ασημίνα

Διπλωματούχος Μηχανικός Ηλεκτρονικών Υπολογιστών, Τηλεπικοινωνιών και Δικτύων Πανεπιστημίου Θεσσαλίας

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω όλους όσους με στήριξαν και βοήθησαν με τον δικό τους τρόπο για την ολοκλήρωση αυτής της διπλωματικής. Πρώτα από όλα θα ήθελα να ευχαριστήσω την κυρία Τσομπανοπούλου για όλη την καθοδήγηση και την στήριξη απέναντι μου κατά την διάρκεια της εκπόνησης αυτής της εργασίας .Πραγματικά η συνεργασία μας ήταν άψογη σε όλα τα επίπεδα .

Ακόμη θέλω να ευχαριστήσω τους γονείς μου που είναι πάντα δίπλα μου και σε αποτυχίες και σε επιτυχίες και με στηρίζουν όλη μου την ζωή. Άλλωστε δεν θα τα κατάφερνα χωρίς αυτούς. Επίσης θέλω να ευχαριστήσω τον αδελφό μου Μάκη γιατί με τον δικό του τρόπο με κάνει πάντα να χαμογελάω και να ξεπερνάω τις δυσκολίες.

Σε αυτό το σημείο θέλω να ευχαριστήσω την φίλη μου Σίσσυ Κωνσταντινίδου για όλη την συμπαράσταση και την αγάπη όλων αυτών των ετών ,για όλα όσα ζήσαμε στην κοινή αυτή φοιτητική μας ζωή(και γιατί ήταν πάντα πιο συνεπής από μένα και πήγαινε στις διαλέξεις για σημειώσεις).Ακόμη θέλω να ευχαριστήσω όλους μου τους φίλους για την χαρά τους στην δική μου χαρά και για την λύπη τους στην λύπη μου, πάντα έκαναν την πραγματικότητα πιο εύκολη.

Τέλος θέλω να ευχαριστήσω ιδιαιτέρως τον Πέτρο Καλό για όλη την βοήθεια σε τεχνικά και μη θέματα γιατί χωρίς αυτόν δεν θα υπήρχε αυτή η διπλωματική. Για όλη την υπομονή και την επιμονή που έχει δείξει όλο αυτόν το καιρό και πάνω από όλα που ήταν και είναι πάντα δίπλα μου όποτε τον χρειαστώ.

Στην οικογένεια μου και στον Πέτρο

Περίληψη

Ο σκοπός της διπλωματικής εργασίας ήταν η μελέτη του περιβάλλοντος παραλλήλου προγραμματισμού Cilk για την παραλληλοποίηση διαφόρων αλγορίθμων. Για το σκοπό αυτό πραγματοποιήθηκαν μετρήσεις σε έναν ,δυο, τρεις και τέσσερις επεξεργαστές.

Συγκεκριμένα, έγινε μελέτη του αλγορίθμου σχεδιασμού αυτοτετραγωνιζομενων fractal με opengl , του αλγορίθμου πλημμύρας(floodfill) για γέμισμα με Opengl ,του αλγορίθμου ταξινόμησης συγχωνεύσεως και του υπολογισμού της ακολουθίας Fibonacci. Επίσης, καταγράφηκαν οι χρόνοι και η απόδοση των αλγορίθμων για διάφορες εισόδους. Ακόμη παρατίθεται ανάλυση της αποδοτικότητας του Cilk για την παραλληλοποίηση του κάθε αλγορίθμου και επισκόπηση της γενικής λειτουργίας του.

Λέξεις Κλειδιά: << **cilk, spawn ,sync ,opengl, Floodfill , fractal, συγχώνευση , Fibonacci , παράλληλος ,απόδοση, Χρονοβελτίωση**>>

Abstract

The purpose of this thesis was to study Cilk a parallel programming environment for the parallelization of various algorithms particularly those relating to computer graphics. For this purpose measurements were performed in one, two, three and four processors.
Specifically, the study has been made in the design algorithm of self-squared fractals in opengl, in the floodfill algorithm for filling in opengl ,in the merge sort algorithm and in the calculation of fibonacci sequence algorithm. Also were recorded times and the performance of algorithms for various inputs. Furthermore it is given Cilk performance analysis for the parallelization of each algorithm and general overview of its general operation.

Keywords: << **cilk, spawn, sync, opengl, Floodfill, fractal, merge, Fibonacci , parallel, efficiency,speedup**>>

Πίνακας περιεχομένων

1

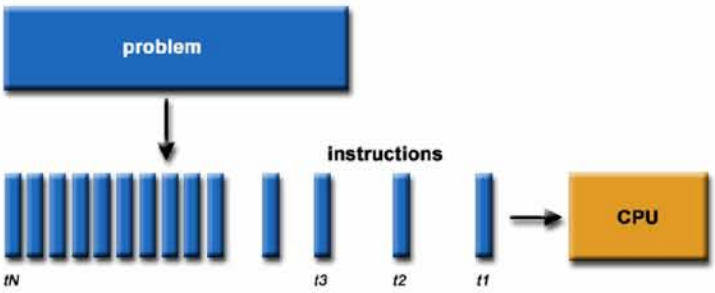
Εισαγωγή

Παράλληλος προγραμματισμός

Η παρούσα διπλωματική εστιάζει στον χώρο του παράλληλου προγραμματισμού .Ο στόχος της παραλληλοποίησης ενός αλγορίθμου είναι η μείωση του χρόνου ολοκλήρωσης των υπολογισμών με σκοπό την χρονοβελτίωση. Ιδανικά ο χρόνος μειώνεται στο μισό με τον διπλασιασμό των χρησιμοποιούμενων επεξεργαστών δηλαδή υπάρχει γραμμική χρονοβελτίωση.

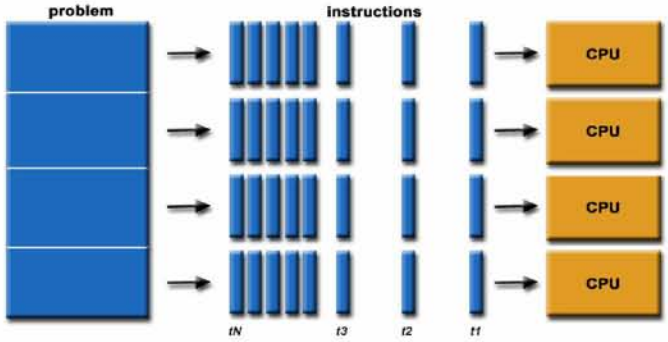
1.1 Τι είναι παράλληλη επεξεργασία;

Συνήθως, το λογισμικό γράφεται με στόχο να εκτελείται ***ακολουθιακά (σειριακά)***. Ο υπολογιστής θεωρείται ότι έχει μια Κεντρική Μονάδα Επεξεργασίας (CPU).Το πρόβλημα επιλύεται εκτελώντας μια ακολουθία εντολών (instructions). Η εκτέλεση μιας εντολής προϋποθέτει την ολοκλήρωση των προηγούμενης εντολής. Κάθε χρονική στιγμή (*t1, t2, t3, ..., tN*) εκτελείται μόνο μια εντολή.



Εικόνα 1

Στην απλούστερη μορφή της, η ***παράλληλη επεξεργασία*** είναι η ταυτόχρονη χρήση πολλαπλών υπολογιστικών πόρων για την επίλυση ενός προβλήματος. Εδώ ο υπολογιστής θεωρείται ότι διαθέτει πολλαπλές CPUs .Το πρόβλημα διασπάται σε τμήματα που μπορούν να επιλυθούν ταυτόχρονα. Κάθε τμήμα επιλύεται εκτελώντας μια ακολουθία εντολών. Οι εντολές κάθε τμήματος εκτελούνται ταυτόχρονα σε διαφορετικές CPUs. Για τις εντολές κάθε τμήματος ισχύουν οι περιορισμοί της ακολουθιακής εκτέλεσης.



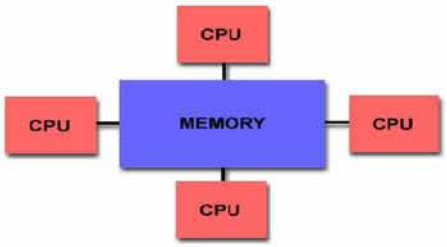
Εικόνα 2

Στην επιστήμη υπολογιστών **παράλληλος προγραμματισμός** λέγεται η ανάπτυξη εφαρμογών οι οποίες εκμεταλλεύονται την ύπαρξη πολλαπλών επεξεργαστικών μονάδων σε έναν πολυεπεξεργαστή ή πολυυπολογιστή με σκοπό να αυξήσουν τις υπολογιστικές επιδόσεις και να μειωθεί ο χρόνος εκτέλεσης που απαιτείται για την ολοκλήρωση της εφαρμογής. Συνεπώς, ο παράλληλος προγραμματισμός μπορεί να θεωρηθεί ως ειδική περίπτωση ταυτόχρονου προγραμματισμού, όπου η εκτέλεση γίνεται πραγματικά παράλληλα και όχι ψευδοπαράλληλα.

1.2 Αρχιτεκτονικές Παράλληλων Υπολογιστών

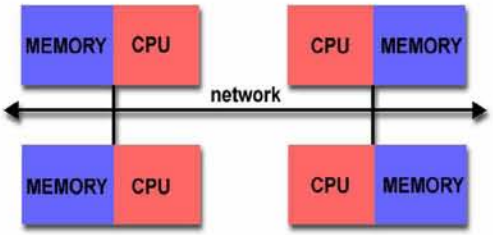
Τα παράλληλα συστήματα διακρίνονται σε:

- *πολυεπεξεργαστές κοινής μνήμης*, όπου πολλαπλοί επεξεργαστές επικοινωνούν με μία κοινή μνήμη ενιαίου χώρου διευθύνσεων



Εικόνα 3

- σε *πολυυπολογιστές κατανεμημένης μνήμης*, όπου πολλαπλά πακέτα επεξεργαστή-ιδιωτικής μνήμης, με τον δικό του χώρο διευθύνσεων το καθένα, διασυνδέονται και επικοινωνούν μεταξύ τους



Εικόνα 4

και στις δύο περιπτώσεις η επικοινωνία γίνεται μέσω ενός «δικτύου διασύνδεσης».

1.3 Μοντέλα Παράλληλου Προγραμματισμού

Τα βασικότερα μοντέλα παράλληλου προγραμματισμού για πολυεπεξεργαστές και πολυυπολογιστές είναι το *μοντέλο κοινού χώρου διευθύνσεων* (π.χ. πολλαπλές διεργασίες ή νήματα, OpenMP) και το *μοντέλο μεταβίβασης μηνυμάτων* (π.χ. PVM, MPI), αντιστοίχως. Στο πρώτο οι επεξεργαστικές μονάδες ανταλλάσσουν πληροφορίες προσπελαύνοντας κοινόχρηστες μεταβλητές στην κοινή μνήμη, ενώ στο δεύτερο ανταλλάσσοντας μηνύματα. Κάθε προγραμματιστικό μοντέλο μπορεί να εφαρμοστεί και σε σύστημα μιας αρχιτεκτονικής που δεν είναι η φύση του (π.χ. πολυνηματικό πρόγραμμα σε πολυυπολογιστή ή πρόγραμμα MPI σε πολυεπεξεργαστή) αλλά συνήθως με χαμηλότερες επιδόσεις.

1.4 Κατανόηση του Προβλήματος και του Προγράμματος

- Αναμφισβήτητα το πρώτο βήμα στην ανάπτυξη παράλληλου λογισμικού είναι η κατανόηση του προβλήματος. Επίσης, αν υπάρχει ακολουθιακός κώδικας, πρέπει και αυτός να γίνει πλήρως κατανοητός.

-***Το βασικότερο που πρέπει να κάνουμε πριν ξεκινήσουμε την ανάπτυξη της παράλληλης λύσης είναι να αποφασίσουμε αν το πρόβλημα παραλληλοποιείται.*** Συνήθως ένας αλγόριθμος παραλληλοποιείται διασπώντας τον σε πολλά τμήματα τα οποία ανατίθενται σε ξεχωριστά νήματα ή διεργασίες και έτσι εκτελούνται παράλληλα σε διαφορετικές CPUs.

Παρόλα αυτά δεν είναι βέβαιο ότι ένας αλγόριθμος, υλοποιημένος σε κάποιο πρόγραμμα, μπορεί πάντα να παραλληλοποιηθεί. Εξαρτήσεις από προηγούμενους υπολογισμούς είναι το σημαντικότερο εμπόδιο για την παραλληλοποίηση.

Ωστόσο ακόμα και στην περίπτωση μη παραλληλοποιήσιμου προγράμματος η χρήση διαφορετικών νημάτων ή διεργασιών μπορεί να βοηθήσει σημαντικά αν ο προγραμματιστής επιθυμεί να επικαλύψει υπολογισμούς με επικοινωνίες (π.χ. ένα νήμα να συνεχίζει τους υπολογισμούς όσο το άλλο αναμένει είσοδο από το δίκτυο ή από τον χρήστη· αυτό έχει νόημα ακόμα και αν το πρόγραμμα εκτελείται ψευδοπαράλληλα σε μονοεπεξεργαστικό, σειριακό υπολογιστή).

-Έπειτα πρέπει να γίνει εύρεση των θερμών σημείων (hotspots) του προγράμματος. Βρίσκουμε τα πιο χρονοβόρα τμήματα του προγράμματος που συνήθως περιλαμβάνουν βρόχους. Εστιάζουμε στα θερμά σημεία και αφήνουμε τα υπόλοιπα τμήματα του προγράμματος για παραπέρα βελτιώσεις.

-Εύρεση των σημείων συμφόρησης (bottlenecks) του προγράμματος. Προσπαθούμε να αντιληφθούμε εάν μέσα στα θερμά σημεία υπάρχουν τμήματα κώδικα που εισάγουν σημαντικές καθυστερήσεις. Για παράδειγμα οι υπερβολικές αναφορές στη μνήμη για τον υπολογισμό εκφράσεων ή οι υπερβολικά σύνθετες εκφράσεις είναι σημεία που μπορούν να επιβραδύνουν το πρόγραμμα. Ίσως χρειαστεί αναδόμηση του προγράμματος ή αλλαγή αλγορίθμου για την ελαχιστοποίηση των σημείων συμφόρησης.

-Εύρεση των εμποδίων παραλληλισμού. Το συχνότερο εμπόδιο είναι η επαναληπτική ή δομική ***εξάρτηση δεδομένων (data dependence)***, όπως αναλύθηκε παραπάνω.

- Εύρεση εναλλακτικών αλγορίθμων: πολλές φορές είναι το σημαντικότερο ζήτημα. Πολύ συχνά απλούστεροι, ακολουθιακοί αλγόριθμοι παρουσιάζουν μεγαλύτερες δυνατότητες παραλληλισμού από ότι αλγόριθμοι που χρησιμοποιούν σύνθετες δομές δεδομένων ή πολύπλοκους υπολογισμούς με περίπλοκες εξαρτήσεις δεδομένων.

1.5 Μέθοδος παραλληλοποίησης

Η παραλληλοποίηση ενός σειριακού αλγορίθμου, προκειμένου να εκμεταλλευτούμε την αύξηση των υπολογιστικών επιδόσεων που προσφέρει ο παραλληλισμός, διακρίνεται σε **τέσσερα βήματα** που εκτελούνται διαδοχικά:

1. Τεμαχισμός (partition-decomposition):
   Είναι η διάσπαση του ολικού υπολογισμού σε επιμέρους εργασίες που εκτελούνται ταυτόχρονα . Εξαρτάται από το πρόβλημα και εναπόκειται στον προγραμματιστή.

Μπορεί να είναι αναδρομικός ή συναρτησιακός. Το πλήθος των επιμέρους διεργασιών θα πρέπει να είναι τουλάχιστον ίσο με το πλήθος των επεξεργαστών στο σύστημα που θα χρησιμοποιήσουμε.

2. Συγχώνευση (agglomeration):
Είναι η συγχώνευση μικρών επιμέρους-εργασιών σε μία εργασία, με σκοπό τη μείωση επικοινωνιακού κόστους. Αυξάνοντας το μέγεθος των διεργασιών ελαττώνεται η επικοινωνία αλλά ταυτόχρονα ελαττώνεται και η ευελιξία και ο βαθμός παραλληλισιμότητας. Επιμέρους διεργασίες που δεν μπορούν να εκτελεστούν ταυτόχρονα, πρέπει να συγχωνεύονται. Μπορούμε να αποφύγουμε επιπλέον επικοινωνία με εκτέλεση των ίδιων υπολογισμών σε κάθε επεξεργαστή.

3. Την επικοινωνία των οντοτήτων(communication).
Εδώ γίνεται ο καθορισμός του τρόπου συντονισμού και επικοινωνίας μεταξύ των οντοτήτων εκτέλεσης, π.χ. φράγματα εκτέλεσης, αποστολή ή λήψη μηνυμάτων κλπ.

Τύποι επικοινωνίας:
- Τοπική ή γενική
- Δομημένη ή μη
- Στατική ή δυναμική
- Σύγχρονη ή ασύγχρονη

Η επικοινωνία μεταξύ των επιμέρους διεργασιών θα πρέπει να είναι:
- Όμοια σε συχνότητα και όγκο.
- Όσο δυνατόν τοπική.
- Σύγχρονη.
- Επικαλυπτόμενη από υπολογισμούς.

Η επικοινωνία δεν πρέπει να αναστέλλει τις σύγχρονες εκτελέσεις των επιμέρους διεργασιών.

4. Την αντιστοίχηση οντοτήτων σε επεξεργαστές(mapping).
Μπορεί να είναι στατική, δηλαδή προκαθορισμένη από τον προγραμματιστή, ή δυναμική, δηλαδή ρυθμιζόμενη από το σύστημα κατά τον χρόνο εκτέλεσης. Δίνεται έμφαση στην ταυτόχρονη εργασία και στο ελάχιστο επικοινωνιακό κόστος.

Δύο βασικές στρατηγικές για να αναθέτουμε διεργασίες σε επεξεργαστές:
- Αναθέστε διεργασίες που μπορούν να εκτελεσθούν ταυτόχρονα σε διαφορετικούς (πολυ)επεξεργαστές.
- Αναθέστε διεργασίες που επικοινωνούν συχνά στον ίδιο (πολυ)επεξεργαστή.

Πρόβλημα: Οι δύο στρατηγικές συγκρούονται.

Η βέλτιστη λύση ανάθεσης διεργασιών σε επεξεργαστές είναι NP-complete πρόβλημα → διάφορες μέθοδοι χρησιμοποιούνται για να συμβιβαστούν οι δύο στρατηγικές.

1.6 Δείκτες μέτρησης παράλληλης συμπεριφοράς

- Χρονοβελτίωση (Speedup) : $S_p = T_1 / T_p$
- Απόδοση (Efficiency) : $E_p = T_1 / (p * T_p)$

Όπου T_1 = χρόνος εκτέλεσης σε ένα επεξεργαστή.
T_p = χρόνος παράλληλης εκτέλεσης σε p επεξεργαστές.

1.7 Γιατί Παράλληλη Επεξεργασία;

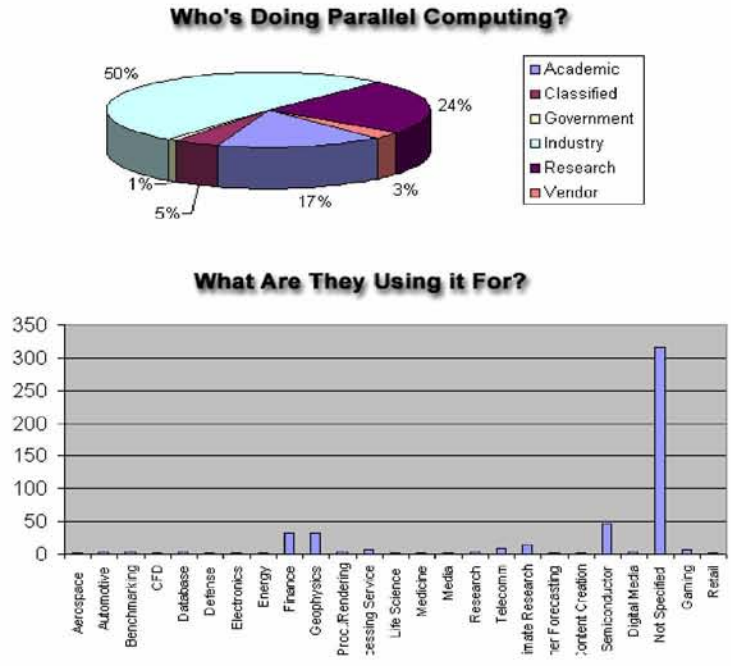
Οι κύριοι λόγοι χρήσης της παράλληλης επεξεργασίας είναι:

- Μείωση του χρόνου εκτέλεσης
- Επίλυση μεγαλυτέρων προβλημάτων
- Παροχή ταυτόχρονης επεξεργασίας

Μερικοί άλλοι λόγοι:

- Χρήση απομακρυσμένων πόρων μέσω τοπικών δικτύων και Διαδικτύου.
- Μείωση κόστους, χρήση πολλών "φθηνών" πόρων αντί λίγων ακριβών (πχ υπέρ-υπολογιστών).
- Κατανομή δεδομένων σε πολλούς υπολογιστές ώστε να ξεπεραστούν περιορισμοί στο μέγεθος της μνήμης.

Είναι πλέον φανερό ότι οι τάσεις στο σχεδιασμό επεξεργαστών, κατανεμημένων συστημάτων και δικτύων υποδεικνύουν ότι (ακόμη και στο επίπεδο του απλού σταθμού εργασίας) ***η παράλληλη επεξεργασία είναι το μέλλον***.



Εικόνα 5

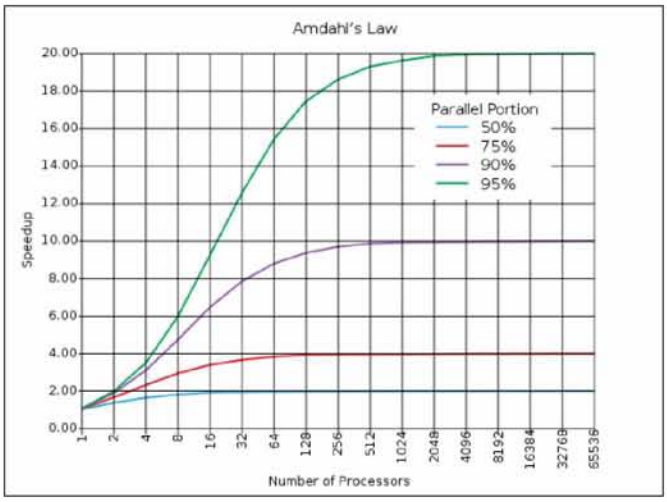
1.8 Όρια και Κόστη Παράλληλου Προγραμματισμού

1.8.1 Νόμος Amdahl:

- Ο Νόμος του Amdahl υπολογίζει τη μέγιστη δυνατή χρονοβελτίωση για μια συγκεκριμένη εφαρμογή ως εξής: Έστω ότι ο ακολουθιακός κώδικας αποτελείται από δύο τμήματα, ένα που μπορεί να παραλληλοποιηθεί και ένα που δεν μπορεί. Άρα, *Τακ* = 1 = P + (1 - P). Προσοχή: το P *δεν* αναφέρεται σε αριθμό γραμμών αλλά σε ποσοστό επί του μοναδιαίου χρόνου εκτέλεσης, δηλαδή στο *ποσοστό χρόνου εκτέλεσης που αντιστοιχεί στα θερμά σημεία του προγράμματος*. Έστω επίσης ότι ο παραλληλοποιημένος κώδικας εκτελείται σε σύστημα με N επεξεργαστές, και έχει χρόνο εκτέλεσης *Τπαρ* = P/N+ (1 - P). Η μέγιστη αναμενόμενη χρονοβελτίωση είναι:

$$Smax = \frac{Τακ}{Τπαρ} = \frac{P + (1 - P)}{P/N + (1 - P)} = \frac{1}{P/N + (1 - P)} \text{ και } \frac{1}{1 - P} \text{ αν } N \to \text{απειρο}$$

Αν δεν υπάρχει παραλληλοποίηση , τότε P = 0 και Smax = 1 (καμία χρονοβελτίωση). Αν υπάρχει πλήρης παραλληλοποίηση, τότε P = 1 και η μέγιστη χρονοβελτίωση S μπορεί να αυξηθεί αυθαίρετα, ανάλογα με την υλοποίηση. Αν τα θερμά σημεία αντιστοιχούν στο 50% του χρόνου εκτέλεσης, τότε P = 0.5 και η μέγιστη χρονοβελτίωση είναι 2/(N+1) < 2!



Εικόνα 6

Γρήγορα γίνονται φανερά τα όρια στη κλιμάκωση του παραλληλισμού. Στην εικόνα φαίνονται οι μέγιστες επιταχύνσεις για διάφορα P και N. Είναι προφανές ότι για P < 0.75 η χρήση άνω των 16 ή 32 επεξεργαστών είναι ανώφελη, ενώ η μέγιστη χρονοβελτίωση περιορίζεται στο 8. Αντίστοιχα για P < 0.90 χρειαζόμαστε περίπου 128 επεξεργαστές για να επιτύχουμε χρονοβελτίωση 16.

Όμως αυτή το εκ πρώτης όψεως αποθαρρυντικό συμπέρασμα δε λαμβάνει υπ' όψη του το γεγονός ότι οι ενδιαφέρουσες εφαρμογές έχουν συνήθως θερμά σημεία των οποίων το ποσοστό υπερβαίνει το 0.8 και αυτό ο ποσοστό *αυξάνει* με το μέγεθος του προβλήματος. Για παράδειγμα ένας πολλαπλασιασμός πινάκων έχει $T_{\alpha\kappa} = P + (1 - P) = M^3$ υπολογισμό + $3M^2$ I/O όπου MXM το μέγεθος των πινάκων. Από αυτά αν θεωρήσουμε το I/O τμήμα ως ακολουθιακό και τον υπολογισμό ως παραλληλοποιήσιμο, τότε για διάφορα M έχουμε:

M = 10^3 P = 10^9 ή 99.70% (1 - P) = 3*10^6 ή 0.030%
M = 10^4 P = 10^12 ή 99.97% (1 - P) = 3*10^8 ή 0.003%
M = 10^6 P = 10^18 ή 99.99% (1 - P) = 3*10^12 ή 0.001%

Τα προβλήματα των οποίων το P αυξάνεται μαζί με το μέγεθός τους είναι περισσότερο ***κλιμακώσιμα*** (***scalable***) από οτι προβλήματα με σταθερό P.

- Τα προβλήματα των οποίων το P αυξάνεται μαζί με το μέγεθός τους είναι περισσότερο ***κλιμακώσιμα*** (***scalable***) από οτι προβλήματα με σταθερό P.
- Από την άλλη πλευρά ο νόμος του Amdahl μεροληπτεί υπέρ του παραλληλισμού γιατί δεν λαμβάνει υπ' όψη του την επιβάρυνση της παράλληλης εφαρμογής που προκύπτει από τη διαχείριση των εργασιών και των δεδομένων.

1.8.2 Πολυπλοκότητα:

- Γενικά οι παράλληλες εφαρμογές είναι αρκετά πιο σύνθετες από τις αντίστοιχες ακολουθιακές. Όχι μόνο πρέπει να παρακολουθούμε πολλαπλές ροές εντολών που εκτελούνται ταυτόχρονα αλλά και πολλαπλές ομάδες δεδομένων που υφίστανται ταυτόχρονη επεξεργασία. Σε γενικές γραμμές, οι παράλληλες εφαρμογές είναι πολύ πιο περίπλοκες από ό, τι οι αντίστοιχες σειριακές εφαρμογές, ίσως μια τάξη μεγέθους. Το κόστος αυτής της πολυπλοκότητας είναι εμφανές σε σχεδόν όλες τις φάσεις ανάπτυξης της εφαρμογής:
    - Σχεδιασμό
    - Συγγραφή
    - Εκσφαλμάτωση
    - Βελτιστοποίηση
    - Συντήρηση
- Η τήρηση καλών πρακτικών ανάπτυξης λογισμικού είναι ιδιαίτερα σημαντική - ιδιαίτερα αν τη λογισμικό που θα αναπτυχθεί θα χρησιμοποιηθεί από άλλους χρήστες.

1.8.3 Φορητότητα:

- Χάρη στη προτυποποίηση αρκετών APIs, όπως το MPI, POSIX, OpenMP και HPF, η φορητότητα των παράλληλων εφαρμογών είναι πολύ μεγαλύτερη από οτι πριν μια δεκαετία. Όμως...
- Όλα τα συνήθη προβλήματα συμβατότητας των ακολουθιακών εφαρμογών ισχύουν και εδώ. Για παράδειγμα, διάφοροι κατασκευαστές προσφέρουν "βελτιωμένες" βιβλιοθήκες με ειδικές συναρτήσεις, η χρήση των οποίων μειώνει τη φορητότητα.
- Αν και υπάρχει προτυποποίηση των APIs, μπορεί υπάρχουν διαφορές στην υλοποίηση οι οποίες επηρεάζουν την απόδοση ώστε να απαιτείται προσαρμογή του κώδικα.
- Τα λειτουργικά συστήματα και τα συστήματα εκτέλεσης επηρεάζουν σημαντικά την φορητότητα.
- Η αρχιτεκτονική είναι ο βασικότερος περιορισμός στη φορητότητα.

1.8.4 Απαιτήσεις σε πόρους:

- Ο πρωταρχικός στόχος της παράλληλης επεξεργασίας είναι η μείωση του χρόνου εκτέλεσης (***wall clock time***) μιας εφαρμογής. Όμως αυτό μπορεί να σημαίνει τη συνολική χρήση περισσότερου χρόνου επεξεργασίας (***CPU time***). Για παράδειγμα η εκτέλεση μιας ακολουθιακής εφαρμογής μπορεί να απαιτεί 4 ώρες σε ένα επεξεργαστή ενώ μιας παράλληλης 1.5 ώρα σε 4 επεξεργαστές. Το wall clock time μειώθηκε από 4 σε 1.5 αλλά το CPU time αυξήθηκε από 4 σε 6.
- Οι απαιτήσεις σε μνήμη μπορεί επίσης να αυξηθούν, λόγω της ανάγκης για δημιουργία αντιγράφων ορισμένων δεδομένων για κάθε παράλληλη εργασία, καθώς επίσης και για την χρήση των βιβλιοθηκών και του συστήματος εκτέλεσης.
- Ειδικά για παράλληλες εφαρμογές σχετικά μικρού μεγέθους, είναι αρκετά συνηθισμένο να έχουμε *μείωση αντί αύτη αύξηση* της απόδοσης, σε σύγκριση με την ακολουθιακή εφαρμογή. Η επιβάρυνση δημιουργίας, διαχείρισης και συγχρονισμού εργασιών, καθώς και η διανομή, επικοινωνία και συλλογή δεδομένων, επιφέρει μεγαλύτερη καθυστέρηση στην παράλληλη εφαρμογή από το κέρδος της όποιας χρονοβελτίωσης.

1.8.5 Κλιμάκωση:

- Η δυνατότητα επιτυχούς κλιμάκωσης μιας παράλληλης εφαρμογής εξαρτάται από αρκετούς παράγοντες. Η απλή πρόσθεση περισσότερων επεξεργαστών συνήθως δεν αρκεί, εκτός ίσως από τις περιπτώσεις φυσικού παραλληλισμού.
- Συνήθως οι παράλληλοι αλγόριθμοι έχουν ορισμένα όρια στη κλιμάκωση που επιτρέπουν. Επομένως, αρχικά η πρόσθεση νέων επεξεργαστών παρουσιάζει σημαντική βελτίωση της απόδοσης, ενώ στη συνέχεια οι επιπλέον επεξεργαστές δεν αποδίδουν τα αναμενόμενα και σταδιακά μπορεί να έχουμε μείωση της απόδοσης. Οι συνηθέστεροι λόγοι είναι τρείς:
    - υπερβολική αύξηση της διαχειριστικής επιβάρυνσης της εφαρμογής
    - υπερβολική αύξηση της επικοινωνίας δεδομένων ή συγχρονισμού
    - υπερβολική μείωση της κοκκιότητας, δηλαδή του υπολογισμού ανά επεξεργαστή

- Η αρχιτεκτονική και το υλικό παίζουν σημαντικό ρόλο στη κλιμάκωση. Παραδείγματα:
    - Το εύρος ζώνης του διαύλου μνήμης και τα επίπεδα/το μέγεθος της κρυφής μνήμης
    - Το εύρος ζώνης του συστήματος επικοινωνίας
    - Το μέγεθος της διαθέσιμης μνήμης, τοπικής ή και μοιραζόμενης
    - Τεχνολογία και ταχύτητα επεξεργαστών

- Η υλοποίηση των διαφόρων APIs επίσης μπορεί να επηρεάσει την κλιμάκωση λόγω περιορισμών που επιβάλλονται.

2

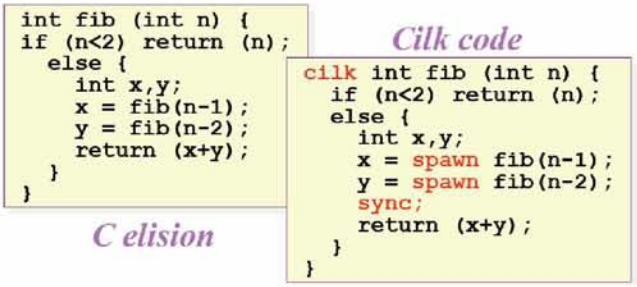
Το περιβάλλον Cilk

2.1 Η γλώσσα προγραμματισμού Cilk

Η Cilk (προφέρεται “silk”) είναι μια αλγοριθμική, πολυνηματική γλώσσα βασισμένη στην Ansi C για παράλληλο προγραμματισμό που αναπτύχθηκε στο Εργαστήριο του MIT για την Επιστήμη Υπολογιστών.
Η Cilk καθιστά εύκολο τον προγραμματισμό ιδιόμορφων εφαρμογών παράλληλα, ιδίως σε σύγκριση με τα data-parallel ή message-passing programming συστήματα . Πολλές Cilk εφαρμογές τρέχουν σχεδόν τόσο γρήγορα όσο συγκρίσιμα προγράμματα σε C σε έναν επεξεργαστή, και κλιμακώνουν πολύ καλά σε πολλούς επεξεργαστές.
Το Cilk επιτρέπει σε έναν προγραμματιστή να συγκεντρωθεί στη δόμηση ενός προγράμματος για να αναδείξει τον παραλληλισμό και να εκμεταλλευτεί την τοπικότητα, αφήνοντας το runtime σύστημα με την ευθύνη να προγραμματίσει τον υπολογισμό για να τρέξει αποτελεσματικά σε μια δεδομένη πλατφόρμα. Ο προγραμματιστής Cilk δεν χρειάζεται να ανησυχεί για τα πρωτόκολλα και την εξισορρόπηση φορτίου, τις οποίες αναλαμβάνει το αποδεδειγμένα αποτελεσματικό runtime σύστημα του Cilk .
Σε αντίθεση με άλλες πολυνηματικές γλώσσες, η Cilk είναι αλγοριθμική, και με αυτό ο runtime scheduler του συστήματος εγγυάται αποδεδειγμένα αποτελεσματική και προβλέψιμη απόδοση.

Η βασική γλώσσα Cilk αποτελείται από την γλώσσα C με την προσθήκη τριών νέων λέξεων-κλειδιών – cilk , sync , και spawn -για να δείξει τον παραλληλισμό και τον συγχρονισμό.
Εάν οι λέξεις-κλειδιά, cilk , sync , και spawn διαγραφούν από ένα Cilk πρόγραμμα, το αποτέλεσμα είναι ένα συντακτικά και σημασιολογικά σωστό πρόγραμμα σε C, το οποίο καλούμε την ***C έκθλιψη(elision)*** του προγράμματος Cilk. Η Cilk είναι πιστή επέκταση της C ,για αυτό μια C έκθλιψη ενός Cilk προγράμματος παρέχει μια νόμιμη εκτέλεση της παράλληλης σημασιολογίας.

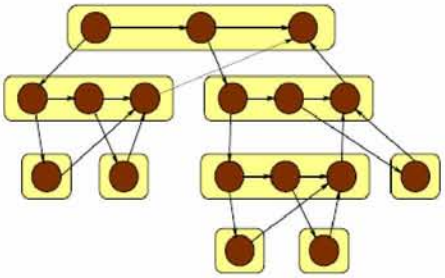
C elision της συνάρτησης υπολογισμού Fibonacci και Cilk εκδοχή



Εικόνα 7

Η λέξη-κλειδί cilk προσδιορίζει τον ορισμό μιας διαδικασίας Cilk (Cilk procedure), η οποία είναι η παράλληλη εκδοχή μιας συνάρτησης σε C, και η οποία έχει λίστα ορισμάτων(argument list) και σώμα(body) ακριβώς όπως μια συνάρτηση σε C. Μια διαδικασία Cilk μπορεί να «δημιουργήσει» (spawn) υποδιαδικασίες (subprocedures) παράλληλα και να τις συγχρονίσει με την ολοκλήρωση τους. Όπως και στη C, το μεγαλύτερο μέρος της εργασίας σε μια διαδικασία Cilk εκτελείται σειριακά.
Ο παραλληλισμός δημιουργείται όταν η κλήση της Cilk διαδικασίας είναι αμέσως πριν από την λέξη-κλειδί spawn. Μια spawn είναι μια παράλληλη εκδοχή μιας κλήσης συνάρτησης σε C, και όπως και με μια κλήση συνάρτησης σε C, η εκτέλεση προχωρά στο παιδί όταν μια διαδικασία Cilk «έχει δημιουργήσει»(spawned).

Για μια κλήση συνάρτησης C, ωστόσο, ο γονέας(parent) ανακόπτεται (suspended) μέχρι το παιδί του (child) να επιστρέψει , ενώ για μια Cilk spawn , ο γονέας μπορεί να συνεχίσει να εκτελείται παράλληλα με το παιδί. Πράγματι ο πατέρας μπορεί να συνεχίσει να παράγει παιδιά δημιουργώντας ένα υψηλό επίπεδο παραλληλισμού. Ο χρονοπρογραμματιστής του Cilk αναλαμβάνει να αναθέσει τις spawned διαδικασίες στους επεξεργαστές του υπολογιστή.



Εικόνα 8

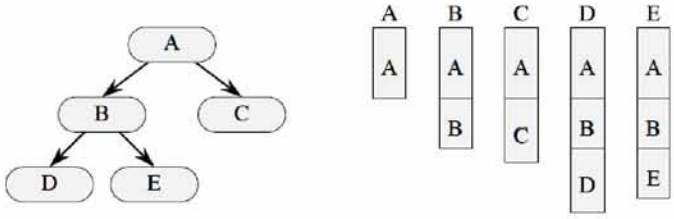
Το μοντέλο πολυνηματικού υπολογισμού του Cilk. Κάθε διαδικασία, εμφανίζεται ως ένα στρογγυλεμένο ορθογώνιο, χωρίζεται σε ακολουθίες των νημάτων της, που εμφανίζονται ως κύκλοι. Η προς τα κάτω ακμή δείχνει την spawned subprocedure . Μια οριζόντια ακμή δείχνει τη συνέχιση στο νήμα που διαδέχεται. Μια ανοδική ακμή δείχνει την επιστροφή μιας τιμής σε μια διαδικασία γονέα. Και οι τρεις τύποι των ακμών είναι εξαρτήσεις που περιορίζουν την σειρά με την οποία τα νήματα μπορούν να προγραμματιστούν.

Μια διαδικασία Cilk δεν μπορεί να χρησιμοποιήσει με ασφάλεια τις τιμές επιστροφής των παιδιών που έχει δημιουργήσει μέχρι να εκτελεστεί μια δήλωση sync. Αν όλα τα παιδιά της δεν έχουν ολοκληρωθεί, όταν εκτελείται μια sync, η διαδικασία ανακόπτεται και δεν συνεχίζει μέχρι όλα τα παιδιά της να έχουν ολοκληρώσει. Η sync δήλωση είναι μια τοπικό «φράγμα»(barrier), σε αντίθεση με τα global «φράγματα» που χρησιμοποιούνται μερικές φορές, για παράδειγμα, στο message-passing programming. Στη Cilk, μια sync δήλωση περιμένει μόνο για τα παιδιά της τη διαδικασίας να ολοκληρωθούν. Όταν όλα τα παιδιά έχουν επιστρέψει, η εκτέλεση της διαδικασίας

συνεχίζει στο σημείο αμέσως μετά την sync δήλωση.

Ένας προγραμματιστής Cilk χρησιμοποιεί τις λέξεις-κλειδιά spawn και sync για να εκθέσει τον παραλληλισμό σε ένα πρόγραμμα. Το Cilk runtime σύστημα αναλαμβάνει την ευθύνη για τον προγραμματισμό των διαδικασιών αποτελεσματικά.

Το runtime σύστημα της Cilk υποστηρίζει τη σημασιολογία της C για stack –allocated αποθήκευση. Ένας δείκτης σε μια τοπική μεταβλητή μπορεί να περάσει σε μια υπορουτίνα , αλλά ένας δείκτης σε μια τοπική μεταβλητή, δεν μπορεί να επιστραφεί, δεδομένου ότι οι τοπικές μεταβλητές αποδεσμεύονται αυτόματα σε μια επιστροφή. Η Cilk υποστηρίζει αυτά τα χαρακτηριστικά της σημασιολογίας της C ακριβώς, ενώ επιτρέπει υποδιαδικασίες . Το Cilk χρησιμοποιεί μια στοίβα κάκτου για τη Stack allocation αποθήκευση.



Εικόνα 9

Μια στοίβα κάκτου. Η διαδικασία A δημιουργεί την B και την Γ, και η B δημιουργεί την D και την E. Το αριστερό μέρος της εικόνας δείχνει το δέντρο παιδί, και το δεξί μέρος της εικόνας δείχνει την όψη της στοίβας από τις πέντε διαδικασίες. (Η στοίβα μεγαλώνει προς τα κάτω.)

Το Cilk runtime σύστημα εφαρμόζει μια αποδεδειγμένα αποτελεσματική πολιτική χρονοπρογραμματισμού με βάση τυχαίο work stealing. Κατά τη διάρκεια της εκτέλεσης ενός προγράμματος Cilk, όταν ένας επεξεργαστής μείνει χωρίς εργασία, ζητά από άλλο επεξεργαστή που επιλέγεται τυχαία δουλειά για να κάνει. Σε τοπικό επίπεδο, ένας επεξεργαστής εκτελεί διαδικασίες με την συνηθισμένη σειριακή σειρά (ακριβώς όπως η C), διερευνώντας το παιδί - δέντρο πρώτα σε βάθος. Όταν μία διαδικασία - παιδί δημιουργηθεί, ο επεξεργαστής αποθηκεύει τοπικές μεταβλητές του πατέρα στο κάτω μέρος μιας στοίβας και αρχίζει τις εργασίες για το παιδί. (Εδώ, χρησιμοποιούμε τη σύμβαση ότι η στοίβα μεγαλώνει προς τα κάτω, και ότι τα στοιχεία εισάγονται και εξάγονται από το κάτω μέρος της στοίβας.) Όταν το παιδί τερματίσει την λειτουργία του και επιστρέψει , τα δεδομένα εξάγονται από το κάτω μέρος της στοίβας (ακριβώς όπως C) και ο πατέρας επανέρχεται. Όταν ένας άλλος επεξεργαστής ζητά εργασία, ωστόσο, η εργασία «κλέβεται» από την κορυφή της στοίβας.

2.2 Intel Cilk

Στην παρούσα διπλωματική παρουσιάζεται η τεχνολογία του Cilk με τον Intel® C++ Compiler. Αυτή η τεχνολογία επιτρέπει την πρόσθεση παραλληλισμού σε νέα ή υπάρχοντα C ή C++ προγράμματα.

2.2.1 Λέξεις -κλειδιά

Σε αντιστοιχία με την γλώσσα Cilk χρησιμοποιούνται τρείς λέξεις κλειδιά για να προσδιοριστούν οι εργασίες που μπορούν να εκτελεστούν παράλληλα:

_Cilk_spawn (ή, cilk_spawn, εάν το πρόγραμμα περιλαμβάνει το <cilk/cilk.h>) το οποίο υποδηλώνει την κλήση σε μια συνάρτηση-«παιδί» που μπορεί να διεξάγεται παράλληλα με την καλούσα (η «γονική»). Η λέξη-κλειδί επιτρέπει, αλλά δεν απαιτεί παράλληλη λειτουργία. Το Cilk καθορίζει δυναμικά ποιές πράξεις εκτελούνται παράλληλα όταν πολλοί επεξεργαστές είναι διαθέσιμοι.

Η cilk_spawn μπορεί να κληθεί με τους ακόλουθους τρόπους:

```
type var = cilk_spawn func(args);      // η func() επιστρέφει τιμή
var = cilk_spawn func(args);           // η func() επιστρέφει τιμή
cilk_spawn func(args);                 // η func() μπορεί να επιστρέψει void
```

όπου func είναι το όνομα μιας συνάρτησης η οποία μπορεί να λειτουργεί παράλληλα με το τρέχον νήμα-κλώνο. Ένα νήμα – κλώνος(strand) είναι μια σειριακή ακολουθία εντολών χωρίς κανένα παράλληλο έλεγχο. Η εκτέλεση της ρουτίνας που περιέχει την cilk_spawn μπορεί να εκτελεστεί παράλληλα με το τμήμα της μετά την ρουτίνα.

args είναι τα ορίσματα της συνάρτησης που έχει δημιουργηθεί. Τα ορίσματα αυτά αποτιμώνται πριν από την δημιουργία της συνάρτησης. Πρέπει να διασφαλιστεί ότι τα ορίσματα της κλήσης κατ'αναφορά και της κλήσης κατά διεύθυνση έχουν διάρκεια ζωής τουλάχιστον μέχρι το επόμενο cilk_sync καθώς διαφορετικά η συνάρτηση μπορεί να «ζήσει» περισσότερο από τη μεταβλητή και να προσπαθήσει να τη χρησιμοποιήσει αφού έχει καταστραφεί. Αυτό είναι ένα κλασσικό παράδειγμα συνθήκης ανταγωνισμού(data race).

Δεν είναι δυνατή η δημιουργία μια συνάρτησης ως όρισμα σε μια άλλη συνάρτηση:

```
g (cilk_spawn f ());       // Δεν επιτρέπεται
```

Η σωστή προσέγγιση είναι η δημιουργία μιας συνάρτησης που καλεί τόσο την f () όσο και την g (). Αυτό επιτυγχάνεται εύκολα χρησιμοποιώντας ένα C + + lambda:

```
cilk_spawn [&]{ g (f()); }  ();
```

Να σημειωθεί ότι το ανωτέρω είναι διαφορετικό από το ακόλουθο:

```
cilk_spawn g (f ());
```

_Cilk_sync (ή, cilk_sync, εάν το πρόγραμμα περιλαμβάνει το <cilk/cilk.h>) το οποίο υποδηλώνει ότι όλα τα παιδιά που δημιουργήθηκαν πρέπει να ολοκληρωθούν προτού συνεχίσει η διαδικασία.). Η λέξη-κλειδί δεν επηρεάζει την δημιουργία παράλληλών νημάτων σε άλλες συναρτήσεις.

Η σύνταξη είναι ως εξής:

```
cilk_sync;
```

η cilk_sync μόνο συγχρονίζει τα παιδιά που δημιουργήθηκαν από αυτή τη συνάρτηση . Παιδιά από άλλες συναρτήσεις δεν επηρεάζονται. Υπάρχει μια σιωπηρή cilk_sync στο τέλος κάθε συνάρτησης και κάθε μπλοκ try που περιέχει ένα cilk_spawn.

_Cilk_for (ή, cilk_for, εάν το πρόγραμμα περιλαμβάνει το <cilk/cilk.h>) το οποίο υποδηλώνει ένα βρόχο για τον οποίο όλες οι επαναλήψεις μπορούν να εκτελεστούν παράλληλα.

Η γενική σύνταξη του cilk_for:

```
cilk_for (declaration; conditional expression; increment expression)
  body
```

Η δήλωση (declaration) πρέπει να δηλώνει και να αρχικοποιεί μια μόνο μεταβλητή, που ονομάζεται "μεταβλητή ελέγχου"(control variable). Η συντακτική μορφή του constructor δεν έχει σημασία. Αν ο τύπος της μεταβλητής έχει ένα προεπιλεγμένο constructor, δεν είναι απαραίτητη ρητή αρχική τιμή.
Η υπό όρους έκφραση (conditional expression)πρέπει να συγκρίνει τη μεταβλητή ελέγχου σε μια " έκφραση τερματισμού " χρησιμοποιώντας ένα από τους παρακάτω τελεστές σύγκρισης
< < = != >= >

• Η έκφραση τερματισμού και η μεταβλητή ελέγχου μπορούν να εμφανίζονται σε οποιαδήποτε πλευρά του τελεστή σύγκρισης, αλλά η μεταβλητή ελέγχου δεν μπορεί να εμφανιστεί εντός της έκφρασης τερματισμού. Η τιμή της έκφρασης τερματισμού δεν πρέπει να αλλάξει από τη μία επανάληψη στην επόμενη.
• Η έκφραση αύξησης (increment expression) πρέπει να προσθέτει ή να αφαιρεί από τη μεταβλητή ελέγχου χρησιμοποιώντας ένα από τους μετά από υποστηριζόμενους τελεστές:

+=
-=
++ (prefix or postfix)
-- (prefix or postfix)

Το Cilk runtime μετατρέπει ένα cilk_for βρόχο σε μία αποτελεσματική διαίρει και βασίλευε αναδρομική διέλευση κατά τη διάρκεια των επαναλήψεων βρόχου.

Παράδειγμα:

```
cilk_for (int i = begin; i < end; i += 2)
  f(i);
cilk_for (T::iterator i(vec.begin()); i != vec.end(); ++i)
 g(i);
```

Στην C,αλλά όχι στην C++, η μεταβλητή ελέγχου μπορεί να δηλωθεί διαφορετικά :

```
int i;

cilk_for (i = begin; i < end; i += 2)
f(i);
```

Η σειριακή εκδοχή ενός έγκυρου Cilk προγράμματος έχει την ίδια συμπεριφορά με το C/C++ πρόγραμμα, όπου η σειριοποίηση του cilk_for προκύπτει αν αντικατασταθεί το cilk_for με το for.Για αυτό ο βρόχος cilk_for πρέπει να είναι ένας έγκυρος C/C++ for βρόχος, αλλά οι βρόχοι cilk_for έχουν διάφορους περιορισμούς σε σχέση με τους C/C++ for βρόχους.

Για να παραλληλοποίηση ενός βρόχου χρησιμοποιώντας την τεχνική του διαίρει και βασίλευε, το runtime σύστημα πρέπει να υπολογίσει τον συνολικό αριθμό των επαναλήψεων και να είναι σε θέση να προ-υπολογιστεί την τιμή της μεταβλητής έλεγχου του βρόχου σε κάθε επανάληψη. Για να ενεργοποιηθεί ο υπολογισμός , η μεταβλητή ελέγχου πρέπει να ενεργεί ως ακέραιος με σεβασμό στην προσθήκη, αφαίρεση, και την σύγκριση, ακόμη κι αν είναι ένας καθορισμένος από τον χρήστη τύπο.

Επιπλέον, ένας cilk_for βρόχος έχει τους ακόλουθους περιορισμούς, που δεν ισχύουν σε ένα τυπικό C/C++ βρόχο. Ο μεταγλωττιστής θα αναφέρει σφάλμα ή προειδοποίηση σε περίπτωση που παραβιαστούν τα παρακάτω:

• Πρέπει να υπάρχει ακριβώς μια μεταβλητή ελέγχου βρόχου, καθώς και στη ρήτρα του βρόχου αρχικοποίησης πρέπει να εκχωρείται η τιμή. Η παρακάτω φόρμα δεν υποστηρίζεται:

```
cilk_for (unsigned int i, j = 42, j <1, i++, j++)
```

• Η μεταβλητή ελέγχου βρόχου δεν πρέπει να τροποποιηθεί μέσα στο σώμα βρόχου. Η παρακάτω φόρμα δεν υποστηρίζεται:

cilk_for (unsigned int i = 1; j <16;++i) i = f ();

• Οι τιμές τερματισμού και αύξησης παίρνουν τιμή πριν από την έναρξη του βρόχου και δεν θα πρέπει να επαναξιολογούνται σε κάθε επανάληψη. Καταυτόν τον τρόπο, τροποποιώντας οποιαδήποτε τιμή μέσα στο σώμα του βρόχου δεν θα προστίθενται ή θα αφαιρούνται επαναλήψεις. Η παρακάτω φόρμα δεν υποστηρίζεται:
cilk_for (unsigned int i = 1; i<x; ++ i) x = f ();

• Στη C++, η μεταβλητή ελέγχου πρέπει να δηλώνεται στην κεφαλίδα του βρόχου, όχι έξω από το βρόχο.
Η ακόλουθη μορφή υποστηρίζεται για C, αλλά όχι C++:
int i; cilk_for (i = 0; i <100; i++)

• Μια δήλωση break ή return δεν θα λειτουργήσει μέσα στο σώμα του βρόχου cilk_for, ο compiler θα παραγάγει ένα μήνυμα λάθους.

• Ένα goto μπορεί να χρησιμοποιηθεί μέσα στο σώμα ενός βρόχου cilk_for μόνο αν ο στόχος (target) είναι εντός του βρόχου . Ο compiler θα Χρονοβελτίωση ένα μήνυμα λάθους, εάν υπάρχει μια goto μεταφορά προς ή από ένα cilk_for σώμα βρόχου.

• Ένα βρόχος cilk_for δεν μπορεί να κάνει το «wrap around» Για παράδειγμα, στην C/C++ μπορείτε να γράψετε:
for (unsigned int i = 0; i != 1; i + = 3);
Και αυτό σημαίνει την εκτέλεση του βρόχου 2.863.311.531 φορές. Ένας τέτοιος βρόχος παράγει απρόβλεπτα αποτελέσματα στο Cilk όταν μετατρέπεται σε ένα cilk_for βρόχο.

• Ένας cilk_for βρόχος δεν μπορεί να είναι ένα άπειρος βρόχος όπως:
cilk_for (unsigned int i = 0; i != i, i += 0);

Η δήλωση cilk_for διαιρεί το βρόχο σε κομμάτια που περιέχουν μία ή περισσότερες επαναλήψεις βρόχου. Κάθε κομμάτι εκτελείται σειριακά, και έχει δημιουργηθεί σαν ένα κομμάτι κατά την εκτέλεση του βρόχου. Ο μέγιστος αριθμός των επαναλήψεων σε κάθε κομμάτι είναι το grain size. Σε ένα βρόχο με πολλές επαναλήψεις, ένα σχετικά μεγάλο grain size μπορεί να μειώσει σημαντικά το overhead. Εναλλακτικά, με ένα βρόχο που έχει λίγες επαναλήψεις, ένα μικρό grain size μπορεί να αυξήσει τον παραλληλισμό του προγράμματος και έτσι να βελτιώσει την απόδοση, όπως ο αριθμός των επεξεργαστών αυξάνεται.

2.2.2 Ρύθμιση του grain size

Χρησιμοποιήστε το cilk grainsize pragma για να καθορίσετε το μέγεθος του grain για ένα cilk_for βρόχο:

#pragma cilk grainsize = expression

Για παράδειγμα, μπορείτε να γράψετε:
#pragma cilk grainsize = 1
cilk_for (int i=0; i<IMAX; ++i){ }

Η προεπιλεγμένη τιμή ορίζεται ως εξής, εάν το ακόλουθο pragma δεν ήταν σε ισχύ:
#pragma cilk grainsize = min (512, N / (8 * p))
όπου N είναι ο αριθμός των επαναλήψεων βρόχου, και το ρ είναι ο αριθμός των νημάτων που δημιουργούνται κατά την διάρκεια του τρέχοντος προγράμματος. Αυτός ο τύπος θα παράγει παραλληλισμό τουλάχιστον 8 και το πολύ 512. Για βρόχους με λίγες επαναλήψεις (λιγότερο από 8 * εργάτες) το grain size θα ρυθμιστεί στο 1, και κάθε επανάληψη του βρόχου μπορούν να διεξάγεται παράλληλα. Για βρόχους με περισσότερες από (4096 * p) επαναλήψεις, το grain

size θα ρυθμιστεί στο 512.

Εάν καθορίσετε ένα grain size στο μηδέν, ο τύπος που θα χρησιμοποιηθεί είναι ο προεπιλεγμένος. Το αποτέλεσμα είναι απροσδιόριστο, αν προσδιοριστεί ένα grain size μικρότερο από το μηδέν.
Πρέπει να σημειωθεί ότι η έκφραση στο pragma αξιολογείται κατά το χρόνο εκτέλεσης. Εδώ είναι ένα παράδειγμα που καθορίζει το grain size με βάση τον αριθμό των εργαζομένων:
pragma cilk grainsize = n/(4 * __cilkrts_get_nworkers())

2.2.3 Build και εκτέλεση του προγράμματος

Για να γίνει build το πρόγραμμα:

•Για το Windows OS *: Μπορεί να χρησιμοποιηθεί είτε το ICL εργαλείο γραμμής εντολών ή να γίνει μεταγλώττιση στο Microsoft Visual * Studio . Εάν χρησιμοποιηθεί Visual Studio *, πρέπει να είναι βέβαιο ότι έχει επιλεγεί Χρήση Intel C++ (Use Intel C++) από το μενού.

• Για το Linux OS *: Μπορεί την γίνει χρήση της εντολής icc.

Linux* OS:
icc example.cpp -o example

Windows* Command Line:
icl example.cpp

Windows Visual Studio*:
Build το Release configuration.

Για να γίνει εκτέλεση του προγράμματος:

Αν δεν υπάρχουν συνθήκες ανταγωνισμού , το παράλληλο πρόγραμμα παράγει το ίδιο αποτέλεσμα , με το σειριακό πρόγραμμα. Η διόρθωση τυχόν συνθηκών ανταγωνισμού μπορεί να γίνει με *reducers*, locks , ή recode.

2.2.4 Θέτοντας τον αριθμό των νημάτων

Εξ ορισμού, ο αριθμός των νημάτων έχει τεθεί ίσος με τον αριθμό των πυρήνων του συστήματος . Στις περισσότερες περιπτώσεις, η προεπιλεγμένη τιμή λειτουργεί καλά. Μπορεί να γίνει αύξηση ή μείωση του αριθμού των νημάτων κάτω από τον έλεγχο του προγράμματος ή μέσω του περιβάλλοντος . Μπορεί να γίνει χρήση λιγότερων νημάτων από τον αριθμό των πυρήνων του επεξεργαστή .

Μπορείτε να γίνει έλεγχος του αριθμού των νημάτων, θέτοντας την μεταβλητή περιβάλλοντος CILK_NWORKERS.

Windows* OS: set CILK_NWORKERS=4

Linux* OS: export CILK_NWORKERS=4

Ακόμη μπορεί να τεθεί ο αριθμός των νημάτων μέσω του προγράμματος. Με την κλήση __cilkrts_set_param ("nworkers", "N"), όπου το N είναι ένας αριθμός που εκφράζεται σε δεκαδικό, ή με αρχικό 0x ή 0, σε δεκαεξαδικό ή οκταδικό. Αυτή η κλήση για να οριστεί ο αριθμός των νημάτων υπερισχύει της τιμής που έχει οριστεί στην μεταβλητή περιβάλλοντος CILK_NWORKERS. Για να χρησιμοποιηθεί το cilkrts_set_param __ , πρέπει να περιλαμβάνεται #include <cilk/cilk_api.h> στο πρόγραμμα.

Παράδειγμα:

```
If(0!= __cilkrts_set_param("nworkers","4")){
   printf("Failed to set worker count\n");
   return 1;
}
```

2.2.5 Preprocessor Macros

__cilk

Αυτή η μακροεντολή ορίζεται αυτόματα από τον Intel compiler και βρίσκεται στον αριθμό έκδοσης Cilk. Η τιμή σε αυτή την έκδοση είναι 200, αναφερόμενοι στην έκδοση γλώσσας 2.0.

2.2.6 Μοντέλο Εκτέλεσης Cilk

Ο Cilk προγραμματισμός μπορεί να φαίνεται μια απόκλιση από τον παραδοσιακό σειριακό προγραμματισμό. Ορισμένες βασικές έννοιες απαιτούν εξήγηση για να μπορούν να γραφούν Cilk προγράμματα που εκτελούνται και κλιμακώνονται καλά.

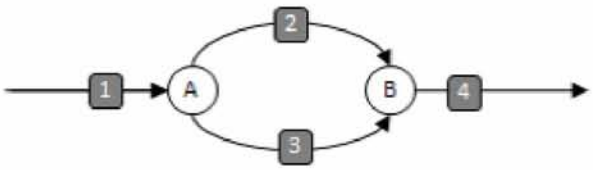
Οι ακόλουθες έννοιες-κλειδιά:

• strands: η δομή ενός προγράμματος Cilk γίνεται καλύτερα κατανοητή ως ένα γράφος από strands που συνδέει παράλληλα σημεία έλεγχου.

•work , span , parallelism : η αναμενόμενη απόδοση ενός προγράμματος Cilk μπορεί να αναλυθεί από την άποψη του έργου, της διάρκειας, και του παραλληλισμού.

Το strand περιγράφει ένα σειριακό τμήμα του προγράμματος. Ακριβέστερα, ο ορισμός ενός strand είναι «κάθε ακολουθία των οδηγιών χωρίς παράλληλες δομές ελέγχου."

Ένα πρόγραμμα Cilk περιέχει δύο είδη των παράλληλων δομών ελέγχου - spawn και sync. (Ένας παράλληλος βρόχος, δηλαδή, το _Cilk_for, είναι απλώς ένας βολικός συμβολισμός για την αποσύνθεση ενός προβλήματος σε spawns και syncs.) Το ακόλουθο δείχνει 4 strands (1, 2, 3, 4), ένα spawn (A) και ένα sync (B). Σε αυτό το σχήμα, μόνο τα strands (2) και (3) μπορεί να εκτελούνται παράλληλα.

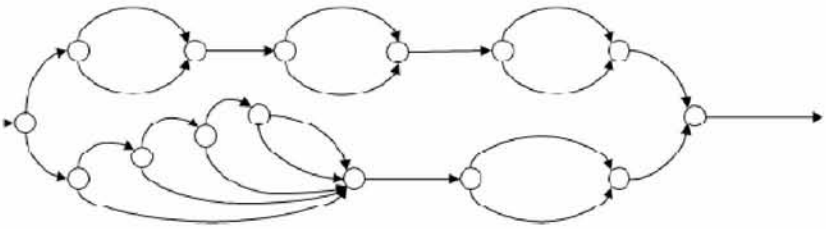


Εικόνα 10

Το διάγραμμα αντιπροσωπεύει έναν κατευθυνόμενο άκυκλο γράφο (DAG), το οποίο αντιπροσωπεύει την σειριακή/ παράλληλη δομή ενός προγράμματος Cilk.

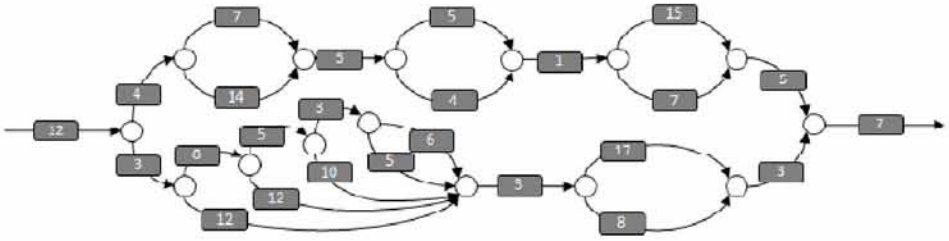
Μόλις είναι διαθέσιμη μια μέθοδος που περιγράφει τη σειριακή/παράλληλη δομή ενός Cilk προγράμματος , μπορεί να ξεκινήσει η ανάλυση για την απόδοση και κλιμάκωση του προγράμματος.

Έστω ένα πιο σύνθετο πρόγραμμα Cilk, που φαίνεται στο διάγραμμα που ακολουθεί:



Εικόνα 11

Αυτό το ΚΑΓ αντιπροσωπεύει παράλληλη δομή κάποιου προγράμματος Cilk.Για να κατασκευαστεί ένα πρόγραμμα Cilk που έχει αυτό το ΚΑΓ πρέπει να προστεθούν ταμπέλες στα strands για να δηλωθεί ο αριθμός των χιλιοστών του δευτερολέπτου που χρειάζεται να εκτελεστεί κάθε strand:



Εικόνα 12

Work (έργο)

Το συνολικό ποσό του χρόνου επεξεργαστή που απαιτείται για την ολοκλήρωση του προγράμματος είναι το άθροισμα όλων των αριθμών. Η τιμή αυτή ορίζεται ως το έργο(work).

Span(Διάρκεια)

Το span, μερικές φορές ονομάζεται μήκος κρισίμου μονοπατιού , είναι το πιο ακριβό μονοπάτι που εκτείνεται από την αρχή έως το τέλος του προγράμματος.

Μπορεί να χρησιμοποιηθεί το έργο και διάρκεια για να γίνει πρόβλεψη για το πώς ένα πρόγραμμα Cilk θα επιταχύνει και θα κλιμακώσει σε πολλαπλούς επεξεργαστές.

Κατά την ανάλυση ενός προγράμματος Cilk, θα πρέπει να εξεταστεί ο χρόνος λειτουργίας του προγράμματος σε διάφορο αριθμό επεξεργαστών. Οι ακόλουθοι συμβολισμοί είναι χρήσιμοι:

- T (P) είναι ο χρόνος εκτέλεσης του προγράμματος σε P επεξεργαστές.
- T (1) είναι το work
- T (∞) είναι το span

Στους 2 επεξεργαστές, ο χρόνος εκτέλεσης δεν μπορεί ποτέ να είναι μικρότερος από το T(1)/2. Σε γενικές γραμμές, το έργο μπορεί να εκφραστεί ως:

T(P) >= T (1) / P

Ομοίως, για P επεξεργαστές, ο χρόνος εκτέλεσης δεν είναι ποτέ μικρότερος από το χρόνο εκτέλεσης σε έναν άπειρο αριθμό επεξεργαστών. Ως εκ τούτου, το span μπορεί να διατυπωθεί ως εξής:

T (P) >= T (∞)

2.2.7 Χρονοβελτίωση και Παραλληλισμός

Εάν ένα πρόγραμμα εκτελείται δύο φορές πιο γρήγορα σε 2 επεξεργαστές, τότε η χρονοβελτίωση είναι 2. Ως εκ τούτου, η χρονοβελτίωση σε P επεξεργαστές μπορεί να διατυπωθεί ως εξής:

T(1)/T(P)

Μία ενδιαφέρουσα συνέπεια του τύπου της χρονοβελτίωσης είναι ότι αν T(1) αυξάνει ταχύτερα από T(P), τότε η χρονοβελτίωση αυξάνεται καθώς αυξάνεται το έργο. Μερικοί αλγόριθμοι, για παράδειγμα, κάνουν επιπλέον έργο , για να επωφεληθούν από επιπλέον επεξεργαστές. Τέτοιοι αλγόριθμοι είναι τυπικά πιο αργοί από ένα αντίστοιχο σειριακό αλγόριθμο σε έναν ή δύο επεξεργαστές, αλλά αρχίζουν να δείχνουν χρονοβελτίωση σε τρεις ή περισσότερους επεξεργαστές. Η κατάσταση αυτή δεν είναι συνηθισμένη, αλλά αξίζει να σημειωθεί.

Η μέγιστη δυνατή χρονοβελτίωση θα συμβεί με έναν άπειρο αριθμό των επεξεργαστών. Ως παραλληλισμός ορίζεται η υποθετική χρονοβελτίωση σε έναν άπειρο αριθμό επεξεργαστών. Ως εκ τούτου, μπορεί ο παραλληλισμός να ορίζεται ως:

T (1) / T (∞)

2.2.8 Χειρισμός εξαίρεσης

Το Cilk προσπαθεί να αναπαράγει, όσο το δυνατόν, τη σημασιολογία της C++ για τον χειρισμό εξαίρεσης. Αυτό γενικά απαιτεί τον περιορισμό του παραλληλισμού, ενώ εκκρεμούν εξαιρέσεις, και τα προγράμματα δεν θα πρέπει να εξαρτώνται από τον παραλληλισμό κατά το χειρισμό εξαίρεσης. Η λογική χειρισμού εξαίρεσης είναι η εξής:

- Εάν παρουσιαστεί εξαίρεση και δεν πιαστεί σε spawned παιδί, στη συνέχεια, η εξαίρεση «ξαναπετιέται» στο γονέα στο επόμενο σημείο sync.

- Σε περίπτωση που ο γονέας ή άλλο παιδί «πετάει» επίσης μια εξαίρεση, τότε η πρώτη εξαίρεση που θα είχε πεταχτεί στη σειριακή σειρά εκτέλεσης υπερισχύει. Οι λογικά-αργότερες εξαιρέσεις απορρίπτονται. Δεν υπάρχει μηχανισμός για την συλλογή πολλαπλών εξαιρέσεων που «πετάγονται» παράλληλα.

Το πέταγμα μια εξαίρεσης δεν ακυρώνει τα υπάρχοντα παιδιά ή αδέλφια του strand στο οποίο η εξαίρεση πετιέται. Αυτά τα strands θα τρέξουν κανονικά μέχρι το τέλος.
Εάν ένα μπλοκ try περιέχει ένα cilk_spawn και/ή ένα cilk_sync, τότε υπάρχει μια σιωπηρή cilk_sync ακριβώς πριν την είσοδο του μπλοκ try και μια σιωπηρή cilk_sync στο τέλος του μπλοκ try. Μια sync εκτελείται αυτόματα πριν την έξοδο από ένα μπλοκ try, μπλοκ συνάρτησης , ή το σώμα cilk_for μέσω εξαίρεσης. Μια συνάρτηση δεν έχει ενεργά παιδιά

όταν ξεκινά η εκτέλεση ενός catch μπλοκ. Αυτά τα σιωπηρά syncs υπάρχουν για να εξασφαλίζουν ότι τα ίδια τα catch μπλοκ τρέχουν όπως θα τρέξει και μια σειριακή εκτέλεση του προγράμματος . Σιωπηρά syncs μπορεί να περιορίσουν τον παραλληλισμό του προγράμματός . Η sync πριν από ένα try μπλοκ , ειδικότερα, μπορεί πρόωρα να συγχρονίσει μία spawn που εμφανίζεται πριν από το μπλοκ try, όπως στο ακόλουθο παράδειγμα:

```
void func() {
  cilk_spawn f();
   try {              // oops! implicit sync prevents parallel execution of f()
     cilk_spawn g();
      h();
  }
   catch (...) {
   // Handle exceptions from g() or h(), but not f()
   }
}
```

2.2.9 Cilk Runtime System API

Τα Cilk προγράμματα απαιτούν το runtime σύστημα και τις βιβλιοθήκες, οι οποίες συνδέονται αυτόματα σε κάθε πρόγραμμα που χρησιμοποιεί Cilk. Το σύστημα εκτέλεσης παρέχει ένα μικρό αριθμό από προσπελάσιμες από το χρήστη λειτουργίες για τον έλεγχο των λεπτομερειών της συμπεριφοράς του προγράμματος.

__cilkrts_set_param

int __cilkrts_set_param(const char* name, const char* value);

Αυτή η λειτουργία επιτρέπει στον προγραμματιστή να ελέγχει διάφορες παραμέτρους του συστήματος εκτέλεσης του Cilk.
Αναγνωρίζονται τα ακόλουθα ορίσματα ονόματος:

nworkers: Η τιμή του ορίσματος καθορίζει τον αριθμό των νημάτων που χρησιμοποιούνται. Αν η συνάρτηση δεν καλείται τότε η τιμή θα προκύψει από την μεταβλητή περιβάλλοντος CILK_NWORKERS αν υπάρχει .Αλλιώς ο αριθμός των νημάτων είναι προκαθορισμένος στον αριθμό των διαθέσιμων πυρήνων. . Αυτή η συνάρτηση είναι αποτελεσματική μόνο πριν από την πρώτη κλήση σε μια συνάρτηση που περιέχει ένα cilk_spawn ή cilk_for. Η τιμή επιστροφής είναι μηδέν για την επιτυχία και μια μη μηδενική τιμή σε αποτυχία.

__cilkrts_get_nworkers

int __cilkrts_get_nworkers (void);

Η συνάρτηση αυτή επιστρέφει τον αριθμό των νημάτων που χρησιμοποιούνται από τις εργασίες Cilk και παγώνει αυτόν τον αριθμό, ώστε να μην μπορεί να αλλάξει από το cilkrts_set_param __. Αν κληθεί σε σειριακό κώδικα, θα επιστρέψει 1.Τα αναγνωριστικά των νημάτων δεν είναι κατ 'ανάγκη σε μια συνεχόμενη σειρά, έτσι το αναγνωριστικό για ένα συγκεκριμένο νήμα μπορεί να είναι μεγαλύτερο από την τιμή επιστροφής του cilkrts_get_nworkers __.

__cilkrts_get_worker_number

int cilkrts_get_worker_number __ (void);

Η συνάρτηση cilkrts_get_worker_number __ επιστρέφει ένα μικρό ακέραιο που δείχνει το νήμα του Cilk στο οποίο η συνάρτηση εκτελείται.

__cilkrts_get_total_workers

int cilkrts_get_total_workers __ (void);

Το Cilk runtime σύστημα μπορεί να διαθέτει περισσότερα νήματα από όσα είναι ενεργά σε μια δεδομένη στιγμή. Η συνάρτηση __cilkrts_get_total_workers επιστρέφει ένα περισσότερο από ό, τι το μέγιστο αριθμητικό ID που θα μπορούσε να ανατεθεί σε ένα νήμα. Τυπικά, αυτός ο αριθμός είναι μια μικρή αύξηση πάνω από τον αριθμό των πραγματικών νημάτων που ανατέθηκαν στην εκτέλεση εργασιών Cilk. Μπορεί να δημιουργηθεί ένας πίνακας μεγέθους __cilkrts_get_total_workers (), με κάθε στοιχείο να δεικτοδοτείται από ένα αναγνωριστικό νήματος . Όπως και με __cilkrts_get_nworkers, η συνάρτηση αυτή παγώνει τον αριθμό των νημάτων ώστε να μην μπορεί να είναι αλλάξει μέσω της cilkrts_set_param __. Αν κληθεί σε σειριακό κώδικα ,η __cilkrts_get_total_workers θα επιστρέψει 1.

2.2.10 Reducers

Το Cilk παρέχει τους reducers για να αντιμετωπίσει το πρόβλημα της πρόσβασης σε μη τοπικές μεταβλητές σε παράλληλο κώδικα. Εννοιολογικά, ένας reducer είναι μια μεταβλητή που μπορεί να χρησιμοποιηθεί με ασφάλεια από πολλαπλά strands που τρέχουν παράλληλα. Το runtime εξασφαλίζει ότι κάθε νήμα έχει πρόσβαση σε ένα ιδιωτικό αντίγραφο της μεταβλητής, εξαλείφοντας έτσι την πιθανότητα ανταγωνισμών χωρίς να απαιτούνται locks. Όταν τα νήματα συγχρονίζονται, τα αντίγραφα του reducer συγχωνεύονται (ή "μειώνονται ") σε μια μεταβλητή. Το runtime δημιουργεί αντίγραφα μόνο όταν χρειάζεται , ελαχιστοποιώντας έτσι το γραφειοκρατικό κόστος (overhead).

2.2.10.1 Χρησιμοποιώντας τους reducers

Στο παρακάτω σειριακό πρόγραμμα καλείται συνεχώς η compute και αθροίζονται τα αποτελέσματα στην μεταβλητή total.

```
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i;           // return a value computed from i
}

int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    unsigned int total = 0;

    // Compute the sum of integers 1..n
    for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i);
    }

    // the sum of the first n integers should be n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;

    if (total == correct)
            std::cout << "Total (" << total
                      << ") is correct" << std::endl;
    else
            std::cout << "Total (" << total
                      << ") is WRONG, should be "
                      << correct << std::endl;
    return 0;
}
```

Κώδικας 1

Η μετατροπή αυτού του προγράμματος σε ένα πρόγραμμα Cilk και η αλλαγή του for σε cilk_for έχει ως αποτέλεσμα ο βρόχος να τρέξει παράλληλα, αλλά δημιουργεί ένα ανταγωνισμό δεδομένων (data race) σχετικά με την μεταβλητή total. Για την επίλυση του ανταγωνισμού, μπορεί να μετατραπεί η total σε reducer. Συγκεκριμένα ο reducer_opadd , ορίστηκε για τους τύπους που έχουν τον συσχετιστικό τελεστή "+" .Οι αλλαγές φαίνονται παρακάτω :

```
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i;           // return a value computed from i
```

```
}

int main(int argc, char* argv[])
{
    unsigned int n = 1000000;
    cilk::reducer_opadd<unsigned int> total;

    // Compute 1..n
    cilk_for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i);
    }

    // the sum of the first N integers should be n * (n+1) / 2
    unsigned int correct = (n * (n+1)) / 2;

    if (total.get_value() == correct)
            std::cout << "Total (" << total.get_value()
                      << ") is correct" << std::endl;
    else
            std::cout << "Total (" << total.get_value()
                      << ") is WRONG, should be "
                      << correct << std::endl;
    return 0;
}
```

Κώδικας 2

Οι ακόλουθες αλλαγές σε ένα σειριακό κώδικα δείχνουν πως χρησιμοποιείται ένας reducer

1. Συμπερίληψη του κατάλληλου reducer αρχείου κεφαλίδων (header file).
2. Δήλωση της reducer μεταβλητής ως reducer_kind<TYPE> και όχι ως TYPE.
3. Εισαγωγή παραλληλισμού , σε αυτή την περίπτωση μετατρέποντας το for βρόχο σε cilk_for βρόχο.
4. Ανάκτηση της τελικής τιμής του reducer με την μέθοδο get_value() αφού ο cilk_for βρόχος ολοκληρωθεί.

Σημείωση : Οι Reducers είναι αντικείμενα .Για αυτό δεν μπορούν να αντιγραφούν απευθείας .Τα αποτελέσματα είναι απρόβλεπτα αν αντιγραφεί ένα reducer αντικείμενο χρησιμοποιώντας την memcpy(). Αντί αυτού μπορεί να χρησιμοποιηθεί copy constructor.

2.2.11 Locks

Τα locks μπορούν να χρησιμοποιηθούν για την επίλυση συνθηκών ανταγωνισμού .
Υπάρχουν διάφορα είδη locks, όπως οι εξής:
• tbb :: mutex αντικείμενα από την βιβλιοθήκη Intel ® Threading Building Blocks
• system locks σε Windows ή Linux * * λειτουργικά συστήματα
• ατομικές εντολές που είναι ουσιαστικά μικρής διάρκειας locks τα οποία προστατεύουν μια read-modify-write ακολουθία εντολών
Το παρακάτω πρόγραμμα περιλαμβάνει μια συνθήκη ανταγωνισμού στο άθροισμα, επειδή η δήλωση sum = sum + i τόσο διαβάζει και γράφει την μεταβλητή sum:

```
#include <cilk/cilk.h>

int main()
{
    int sum = 0;
    cilk_for (int i=0; i<10; ++i)
    {
        sum = sum + i;                  // THERE IS A RACE ON SUM
    }
}
```

Κώδικας 3

Χρησιμοποιώντας ένα lock για να λυθεί η συνθήκη ανταγωνισμού :

```
#include <cilk/cilk.h>
#include <mutex.h>
#include <iostream>

int main()
{
    tbb::mutex mut;
    int sum = 0;
    cilk_for (int i=0; i<10; ++i)
    {
        mut.lock();
        sum = sum + i;          // PROTECTED WITH LOCK
        mut.unlock();
    }
    std::cout << "Sum is " << sum << std::endl;

    return 0;
}
```

Κώδικας 4

2.2.11.1 Ζητήματα για τη χρήση των Locks

Το Cilk αναγνωρίζει τους μηχανισμούς κλειδώματος που αναφέρονται εδώ:
• Η Intel ® Threading Building Blocks βιβλιοθήκη παρέχει το tbb :: mutex για την δημιουργία τμημάτων κρίσιμου κώδικα όπου είναι ασφαλής η ενημέρωση και η πρόσβαση σε κοινόχρηστη μνήμη ή σε άλλους κοινόχρηστους πόρους με ασφάλεια.
Τα εργαλεία Intel ® Parallel Studio αναγνωρίζουν τα locks και δεν αναφέρουν μια συνθήκη ανταγωνισμού σε μια πρόσβαση στη μνήμη που προστατεύεται από την tbb :: mutex

• Windows OS *: τα αντικείμενα CRITICAL_SECTION Windows παρέχουν σχεδόν την ίδια λειτουργικότητα με τα tbb :: mutex αντικείμενα. Τα Intel ® Parallel Studio εργαλεία δεν θα αναφέρουν μια συνθήκη ανταγωνισμού σε μια πρόσβαση στη μνήμη που προστατεύεται από EnterCriticalSection (), TryEnterCriticalSection (), ή LeaveCriticalSection ().

• Linux OS *: * Τα Posix Pthreads mutexes (pthead_mutex_t) παρέχουν σχεδόν την ίδια λειτουργικότητα με τα tbb :: mutex αντικείμενα. Τα Intel ® Parallel Studio εργαλεία δεν θα αναφέρουν μια συνθήκη ανταγωνισμού σε μια πρόσβαση στη μνήμη που προστατεύεται από pthread_mutex_lock (), pthread_mutex_trylock (), ή pthread_mutex_unlock ().

•Τα Intel ® Parallel Studio εργαλεία αναγνωρίζουν ατομικές εντολές υλικού , διαθέσιμες στους C/C++ προγραμματιστές μέσω compiler intrinsics.

Τα locks μπορούν να δημιουργήσουν προβλήματα απόδοσης σε παράλληλα προγράμματα. Επιπλέον, ενώ τα locks μπορούν να επιλύσουν συνθήκες ανταγωνισμού στα δεδομένα, τα προγράμματα που χρησιμοποιούν locks είναι συχνά μη-ντετερμινιστικά. Συνίσταται η αποφυγή locks όποτε αυτό είναι δυνατό.

3

Παρουσίαση και ανάλυση των αλγόριθμων

Στην συνέχεια παρουσιάζονται οι αλγόριθμοι που υλοποιήθηκαν παράλληλα με το περιβάλλον Cilk. Παρατίθενται μια περιγραφή του εκάστοτε αλγορίθμου και τα αποτελέσματα της παραλληλοποίησης του. Όλοι οι υπολογισμοί έγιναν στον επεξεργαστή "Intel(R) Core(TM) i3 CPU M 330 @ 2.13GHz" με 2 φυσικά νήματα και αλλά 2 με hyper threading και είναι αποτέλεσμα μέσου ορού δέκα επαναλήψεων.

3.1 Γινόμενο διανυσμάτων

Σαν πρώτη υλοποίηση παρατίθεται ένα πολύ απλός αλγόριθμος αυτός του πολλαπλασιασμού διανυσμάτων. Η παράθεση του γίνεται καθώς είναι ένας πλήρως παραλληλοποιήσιμος αλγόριθμος και είναι εύκολο να δειχθούν τα αποτελέσματα του Cilk.

Παρατίθεται ο κώδικας σε Intel Cilk:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <cilk/cilk.h>
#include <cilktime.h>
#include <cilk/cilk_api.h>



double* matrix_mult(double* A,double* B,int size){
int i;
double* C;
   C = malloc(sizeof(double)*size);
    cilk_for(i=0;i<size;i++){
      C[i]=(pow(pow(A[i]*B[i],A[i]*B[i]),B[i]),B[i]);
     }

     return(C);

}

 int main (int argc, char *argv[]) {
  int i,size;
  double *A,*B,*C;

  unsigned long long ticks;
  unsigned long long start_ticks,end_ticks;

  if (0!= __cilkrts_set_param("nworkers",argv[2])) return 0;


  printf("\n This is matrix multiplication program!\n");

  size = atoi(argv[1]);
  A = malloc(sizeof(double)*size);
  B = malloc(sizeof(double)*size);
  C = malloc(sizeof(double)*size);

  cilk_for(i=0;i<size;i++){
    A[i]= i*1000;
    B[i]= (size-i)*1000;
  }

   start_ticks = cilk_getticks();
   C = matrix_mult(A,B,size);
   end_ticks = cilk_getticks();
  ticks = end_ticks - start_ticks;
  printf("Calculation succeeded in %lf  milliseconds.\n",cilk_ticks_to_seconds(ticks)*1000);

    return(0);
}
```

Κώδικας 5

Η σειριακή εκδοχή προκύπτει πολύ απλά με την αντικατάσταση του βρόχου cilk_for με τον απλό βρόχο for. Ακολουθούν οι μετρήσεις σε ένα ,δυο, τρεις και τέσσερις επεξεργαστές για διάφορες εισόδους.

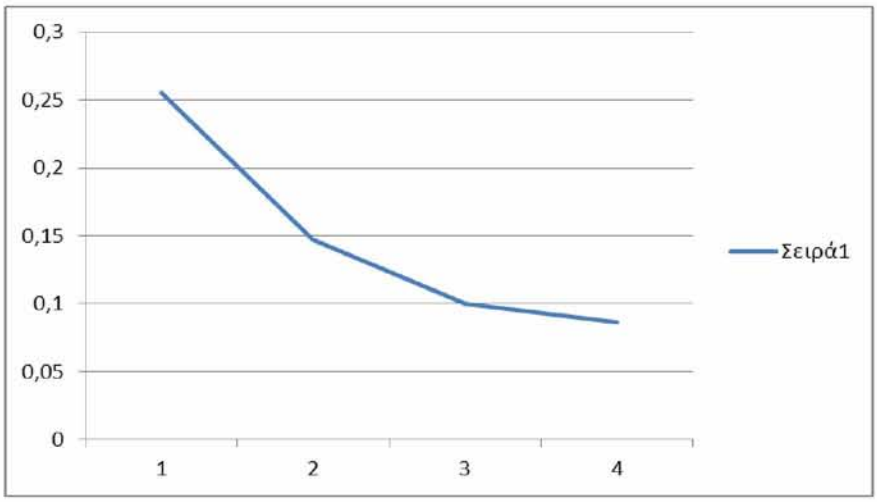
Μετρήσεις

Πολλαπλασιασμός διανυσμάτων μεγέθους 50000000 στοιχείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	0,255 sec	-	-
2	0,147 sec	1,734	0,867
3	0,100 sec	2,550	0,850
4	0,086 sec	2,965	0,741

Πίνακας 1

Διάγραμμα



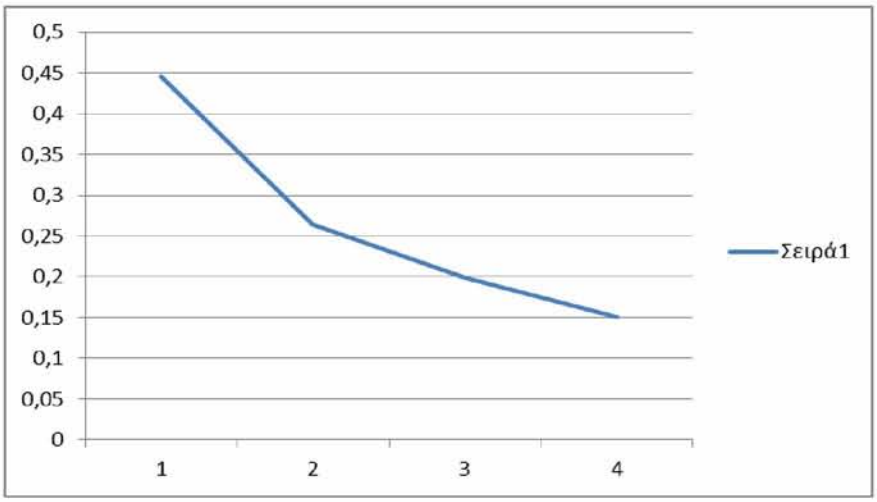
Διάγραμμα 1

Πολλαπλασιασμός διανυσμάτων μεγέθους 90000000 στοιχείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	0,445 sec	-	-
2	0,264 sec	1,685	0,842
3	0,198 sec	2,247	0,749
4	0,151 sec	2,947	0,736

Πίνακας 2

Διάγραμμα



Διάγραμμα 2

3.2 Ακολουθία Φιμπονάτσι (Fibonacci Sequence)

Στα Μαθηματικά, οι *Αριθμοί Φιμπονάτσι* είναι οι αριθμοί της παρακάτω ακέραιης ακολουθίας:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$$

Εξ ορισμού, οι πρώτοι δύο αριθμοί Φιμπονάτσι είναι το 0 και το 1, και κάθε επόμενος αριθμός είναι το άθροισμα των δύο προηγούμενων.

Σε μαθηματικούς όρους, η ακολουθία F_n των αριθμών Φιμπονάτσι ορίζεται από τον αναδρομικό τύπο:

$$F_n = F_{n-1} + F_{n-2}$$

με $F_0 = 0$ και $F_1 = 1$.

ακολουθία μπορεί να επεκταθεί και σε αρνητικό δείκτη n χρησιμοποιώντας αναδιαταγμένη την αναδρομική σχέση

$$F_{n-2} = F_n - F_{n-1},$$

που παράγει την ακολουθία των αρνητικών αριθμών Φιμπονάτσι και ικανοποιεί τη σχέση:

$$F_{-n} = (-1)^{n+1} F_n.$$

Σε ψευδοκώδικα ο αναδρομικός αλγόριθμος υπολογισμού της ακολουθίας Fibonacci είναι ο εξής :

```
long fib (int n) {

   if (n == 0 || n == 1) {
return n;
   }
   else {
      return fib (n - 1) + fib (n - 2);
   }
}
```

Κώδικας 6

Ακολουθεί παράθεση του παράλληλου κώδικα σε Intel Cilk :

```
#include <cilk/cilk.h>
#include <stdio.h>

#include <cilktime.h>
#include <cilk/cilk_api.h>

int fibonacci(int n){

  int temp;
  if ( n == 0 )    return 0;
   else if ( n == 1 )  return 1;
   else{
    temp= cilk_spawn fibonacci(n-1);

      return (temp + fibonacci(n-2));
   }

}

int main (int argc, char *argv[]){
  unsigned long long ticks;
  unsigned long long start_ticks,end_ticks;
  unsigned long long int n;
  if (0!= __cilkrts_set_param("nworkers",argv[2])) return 0;

  n = atoll(argv[1]);

  printf("\nNumbers of the sequence  %ld\n",n);

  start_ticks= cilk_getticks();
  fibonacci(n);
   cilk_sync;
  end_ticks = cilk_getticks();
  ticks = end_ticks - start_ticks;
  printf("Calculation succeeded in %lf  milliseconds.\n",cilk_ticks_to_seconds(ticks)*1000);

  return 0;
```

Κώδικας 7

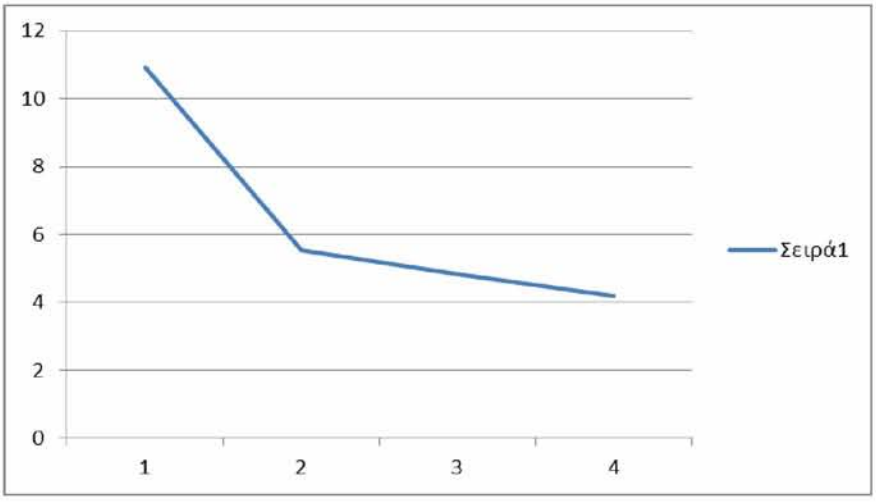
Μετρήσεις

Φιμπονάτσι του αριθμού 40

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	10,904 sec	-	-
2	5,540 sec	1,96	0,984
3	4,823 sec	2,26	0,753
4	4,189 sec	2,60	0,650

Πίνακας 3

Διάγραμμα



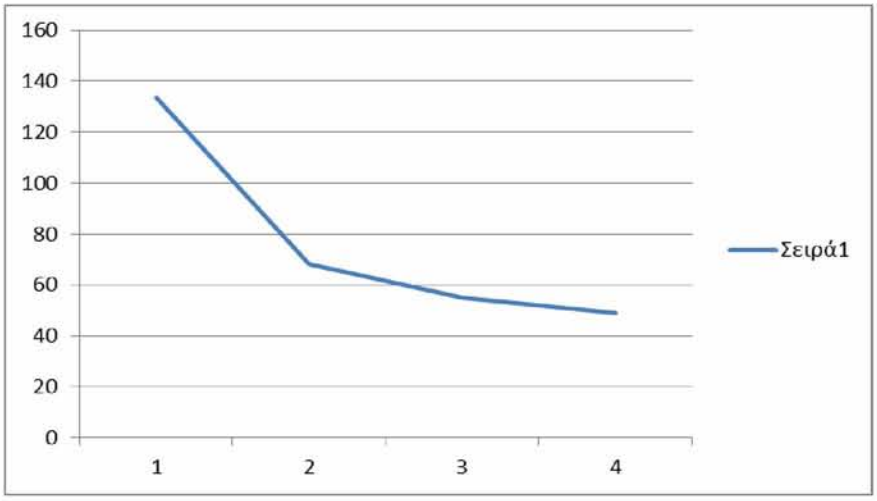
Διάγραμμα 3

Φιμπονάτσι του αριθμού 45

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	133,442 sec	-	-
2	68,206 sec	1,956	0,978
3	54,916 sec	2,429	0,809
4	49,150 sec	2,714	0,678

Πίνακας 4

Διάγραμμα



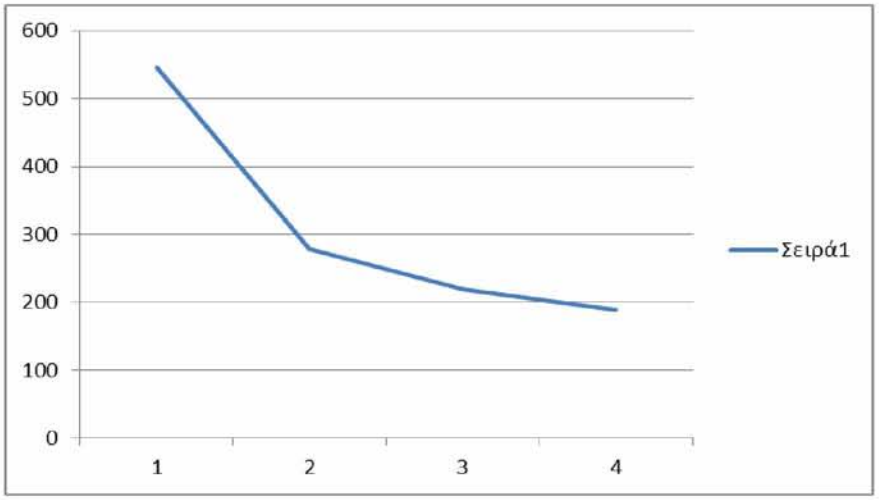
Διάγραμμα 4

Φιμπονάτσι του αριθμού 48

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	546,129 sec	-	-
2	278,693 sec	1,959	0,979
3	219,590 sec	2,487	0,829
4	189,340 sec	2,880	0,721

Πίνακας 5

Διάγραμμα



Διάγραμμα 5

Χάριν συντομίας αλλά και επειδή δεν κρίνεται απαραίτητο δεν παρατίθεται ο πηγαίος κώδικας στους ακόλουθους αλγορίθμους αλλά μόνο οι μετρήσεις και τα συμπεράσματα.

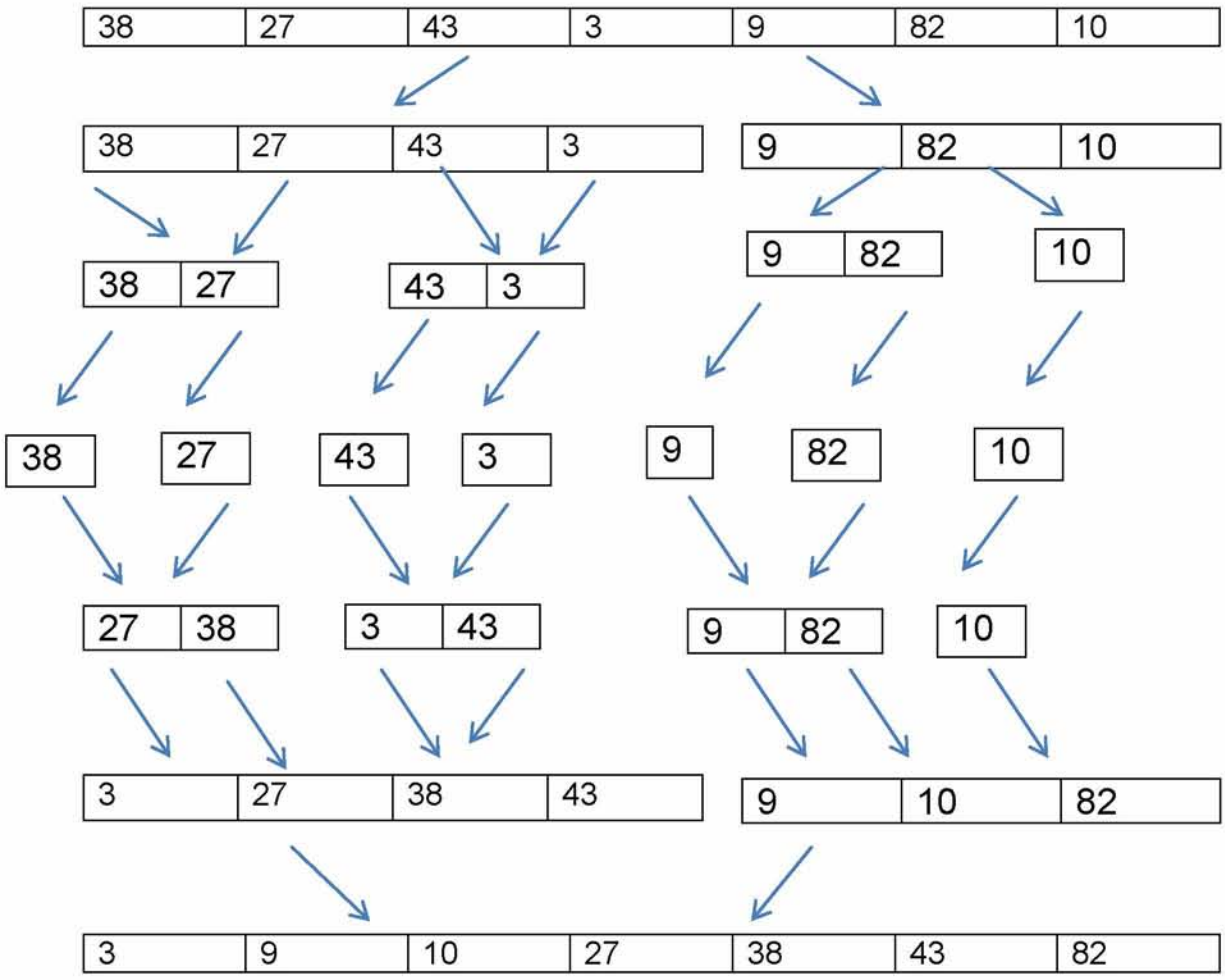
3.3 Ταξινόμηση συγχωνεύσεως (Merge Sort)

Η **Ταξινόμηση συγχωνεύσεως (Merge sort)** είναι ένας αλγόριθμος ταξινόμησης βασισμένος στη σύγκριση. Στις περισσότερες υλοποιήσεις του, παράγεται μία ευσταθή ταξινόμηση, που σημαίνει ότι η υλοποίηση διατηρεί τη σειρά των ίσων στοιχείων από την είσοδο, στην ταξινομημένη έξοδο. Η ταξινόμηση με συγχώνευση είναι αλγόριθμος Διαίρει και Βασίλευε.

Η ταξινόμηση συγχωνεύσεως λειτουργεί ως εξής

1. Διαιρείται η μη ταξινομημένη λίστα σε *n* υπολίστες, με την καθεμιά να περιέχει από 1 στοιχείο (η λίστα του ενός στοιχείου θεωρείται ταξινομημένη).
2. Επαναληπτικά, συγχωνεύονται οι υπολίστες, ώστε να παραχθούν νέες υπολίστες, μέχρι να απομείνει 1 μόνον υπολίστα (η οποία θα είναι ταξινομημένη.)

Παράδειγμα



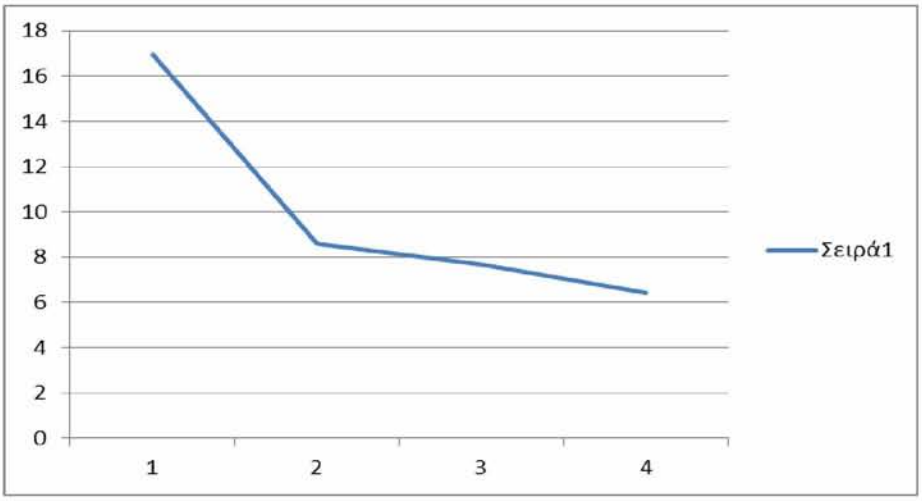
Μετρήσεις

Συγχώνευση 100000000 στοιχείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	16,944 sec	-	-
2	8,588 sec	1.978	0,986
3	7,678 sec	2,206	0,735
4	6,418 sec	2,640	0,660

Πίνακας 6

Διάγραμμα



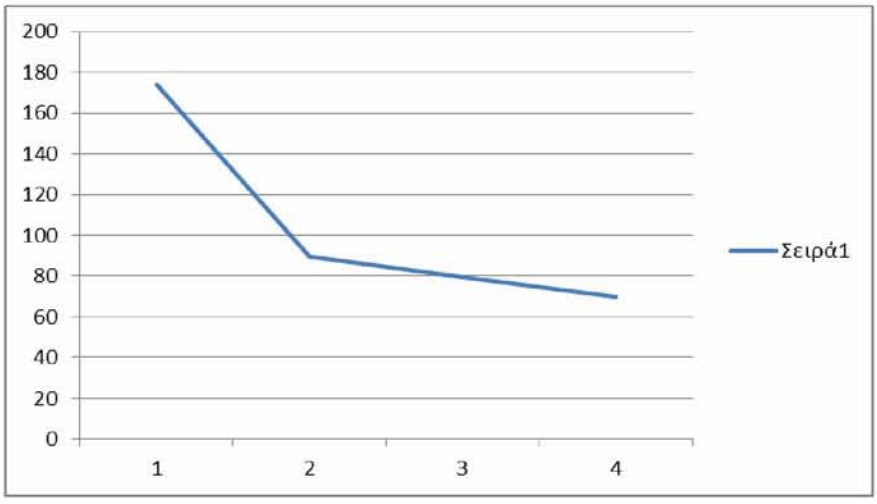
Διάγραμμα 6

Συγχώνευση 1000000000 στοιχείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	173,704 sec	-	-
2	89,837 sec	1,933	0.966
3	79,653 sec	2,180	0,726
4	70,020 sec	2,480	0,620

Πίνακας 7

Διάγραμμα

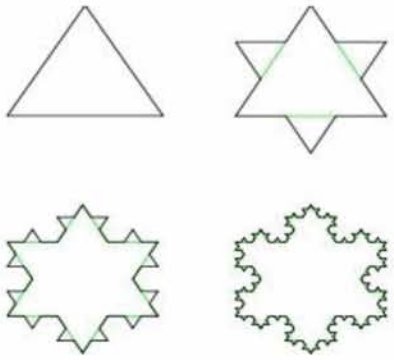


Διάγραμμα 7

3.4 Fractals

3.4.1 ΜΕΘΟΔΟΙ ΓΕΩΜΕΤΡΙΑΣ FRACTAL

Τα περισσότερα αντικείμενα περιγράφονται μέσω μεθόδων Ευκλείδειας γεωμετρίας δηλαδή τα σχήματα των αντικειμένων περιγράφονται με εξισώσεις. Αυτές οι μέθοδοι είναι επαρκείς για την περιγραφή κατασκευασμένων αντικειμένων: αυτών που έχουν ομαλές επιφάνειες και συνήθη σχήματα. Ωστόσο τα φυσικά αντικείμενα, όπως να βουνά και τα σύννεφα, έχουν ακανόνιστα ή αποκομμένα χαρακτηριστικά και οι Ευκλείδειες μέθοδοι δεν παρέχουν ρεαλιστικές περιγραφές τέτοιων αντικείμενων . Τα φυσικά αντικείμενα μπορούν να περιγραφούν με αληθοφάνεια με μεθόδους γεωμετρίας fractal(fractal-geometry methods),όπου χρησιμοποιούνται διαδικασίες και όχι εξισώσεις για να μοντελοποιήσουμε αντικείμενα. Όπως θα μπορούσαμε να περιμένουμε τα αντικείμενα που ορίζονται από διαδικασίες έχουν χαρακτηριστικά που είναι αρκετά διαφορετικά από αντικείμενα που περιγράφονται με εξισώσεις. Οι περιγραφές γεωμετρίας fractal για αντικείμενα χρησιμοποιούνται συχνά σε πολλά πεδία για να περιγράψουν και να εξηγήσουν τα χαρακτηριστικά των φυσικών φαινομένων .Στα γραφικά υπολογιστών χρησιμοποιούμε μεθόδους fractal για να παράγουμε προβολές φυσικών αντικειμένων και οπτικοποιήσεις διάφορων μαθηματικών και φυσικών συστημάτων.
Ένα αντικείμενο fractal έχει δυο βασικά χαρακτηριστικά : άπειρη λεπτομέρεια σε κάθε σημείο και μια συγκεκριμένη αυτό-ομοιότητα (self-similarity) ανάμεσα στα μέρη του αντικειμένου και στα γενικά χαρακτηριστικά του. Οι ιδιότητες αυτό-ομοιότητας ενός αντικειμένου μπορούν να πάρουν διάφορες μορφές, ανάλογα με την περιγραφή που επιλέγουμε για το fractal. Περιγράφουμε ένα αντικείμενο fractal με μια διαδικασία που ορίζει μια επαναλαμβανόμενη πράξη για την παραγωγή της λεπτομέρειας στα υπό-μέρη του αντικειμένου. Τα φυσικά αντικείμενα περιγράφονται με διαδικασίες που θεωρητικά επαναλαμβάνονται άπειρο πλήθος φορών. Οι προβολές των γραφικών, φυσικά, παράγονται με ένα πεπερασμένο αριθμό από βήματα.



Εικόνα 13

3.4.2 Κατηγοριοποίηση των fractal

Τα ***αυτό-όμοια (self-similar) fractal*** έχουν μέρη που είναι εκδοχές ολόκληρου του αντικειμένου και έχουν υποστεί σμίκρυνση.
Τα ***αυτό –συσχετισμένα (self – affine) fractal*** έχουν μέρη που έχουν σχηματιστεί με διαφορετικές παραμέτρους κλιμάκωσης Sx, Sy και Sz σε διαφορετικές κατευθύνσεις

συντεταγμένων.
Τα ***αμετάβλητα σύνολα fractal (invariant fractal sets)*** σχηματίζονται με μη-γραμμικούς μετασχηματισμούς . Αυτή η κατηγορία fractal περιλαμβάνει τα αυτό-τετραγωνιζόμενα (self-squaring) fractal όπως το σύνολο Mandelbrot (που σχηματίζεται με συναρτήσεις τετραγώνησης σε μιγαδικό χώρο), και τα αυτό-αντίστροφα(self-inverse) fractal που κατασκευάζονται με διαδικασίες αντιστροφής.

3.4.3 Αυτό –Τετραγωνιζόμενα Fractal

Μια άλλη μέθοδος παραγωγής αντικειμένων fractal είναι να εφαρμόσουμε επαναλαμβανόμενα μια συνάρτηση μετασχηματισμού σε σημεία σε μιγαδικό χώρο. Στις δύο διαστάσεις ένας μιγαδικός περιγράφεται ως z = x+i*y όπου x και y είναι πραγματικοί αριθμοί και i^2= -1. Σε τρισδιάστατο και τετραδιάστατο χώρο, τα σημεία περιγράφονται με το τετραδικό σύστημα του Hamilton. Μια συνάρτηση f(z) τετραγώνισης μιγαδικού είναι μια που περιλαμβάνει τον υπολογισμό z^2 και μπορούμε να χρησιμοποιήσουμε μερικές αυτό-τετραγωνιζόμενες συναρτήσεις για να παράγουμε σχήματα fractal.
Ανάλογα με την αρχική θέση που επιλέχθηκε για την επανάληψη η επαναλαμβανόμενη εφαρμογή μιας αυτό-τετραγωνιζόμενης συνάρτησης θα παράγει μία από τις εξής τρείς περιπτώσεις:

- Η μετασχηματισμένη θέση μπορεί να αποκλίνει στο άπειρο.
- Η μετασχηματισμένη θέση μπορεί να συγκλίνει μπορεί να συγκλίνει σε ένα πεπερασμένο οριακό σημείο που ονομάζεται σημείο έλξης.
- Η μετασχηματισμένη θέση παραμένει στο σύνορο κάποιας περιοχής.

Για παράδειγμα η non-fractal πράξη τετραγώνισης f(z) = z^2 στο μιγαδικό επίπεδο μετασχηματίζει θέσεις ανάλογα με την σχέση τους με το μοναδιαίο κύκλο. Οποιοδήποτε σημείο z του οποίου το μέτρο |z| είναι μεγαλύτερο του 1 μετασχηματίζεται μέσα από μια ακολουθία θέσεων που τείνουν στο άπειρο. Ένα σημείο με |z|<1 μετασχηματίζεται προς την αρχή των συντεταγμένων. Σημεία που αρχικά βρίσκονται μέσα στον κύκλο ,|z|=1, παραμένουν στον κύκλο. Εάν ο μετασχηματισμός z^2 δεν παράγει fractal ,μερικές μιγαδικές πράξεις τετραγώνισης παράγουν μια καμπύλη fractal ως το σύνορο ανάμεσα σε αυτές τις θέσεις που κινούνται προς το άπειρο και αυτές που τείνουν προς ένα πεπερασμένο όριο. Ένα κλειστό σύνορο fractal που παράχθηκε με μια πράξη τετραγώνισης ονομάζεται σύνολο Julia (Julia set).
Γενικά μπορούμε να εντοπίσουμε το σύνορο του fractal μιας συνάρτησης τετραγώνισης ελέγχοντας τη συμπεριφορά επιλεγμένων θέσεων. Εάν μια θέση μετασχηματιστεί ,έτσι ώστε είτε να αποκλίνει στο άπειρο είτε να συγκλίνει σε ένα σημείο έλξης ,τότε μπορούμε να δοκιμάσουμε μια άλλη κοντινή θέση. Επαναλαμβάνουμε αυτή τη διαδικασία μέχρι τελικά να εντοπίσουμε μια θέση πάνω στο σύνορο του fractal.Μετά, μια επανάληψη του μετασχηματισμού τετραγώνισης παράγει το σχήμα του fractal.Για τους απλούς μετασχηματισμούς στο μιγαδικό επίπεδο ,μια ταχύτερη μέθοδος εντοπισμού θέσεων στην καμπύλη fractal είναι να χρησιμοποιήσουμε την αντίστροφη της συνάρτησης μετασχηματισμού. Τότε ,ένα αρχικό σημείο που επιλέγεται στο εσωτερικό ή το εξωτερικό της καμπύλης ,θα συγκλίνει σε μια θέση στην καμπύλη fractal.
Μια συνάρτηση που είναι πλούσια σε fractal είναι ο μετασχηματισμός τετραγώνισης

$$z' = f(z)=\lambda*z*(1-z)$$

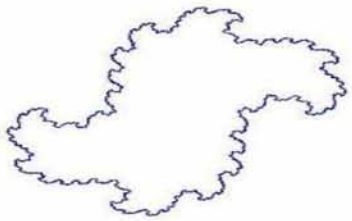
με λ μια μιγαδική σταθερά
Για αυτή την συνάρτηση μπορούμε να χρησιμοποιήσουμε την αντίστροφη μέθοδο για να εντοπίσουμε την καμπύλη fractal.Ο αντίστροφος μετασχηματισμός είναι η δευτεροβάθμια εξίσωση $z = f^{-1}(z') = \frac{1}{2}(1 \pm \sqrt{1-(4z')/\lambda})$
Χρησιμοποιώντας πράξεις μιγαδικών λύνουμε αυτή την εξίσωση για το πραγματικό και το φανταστικό μέρος του z ως

$$x = Re(z) = \frac{1}{2}\left(1 \pm \frac{\sqrt{|διακρ| + Re(διακρ)}}{2}\right)$$

$$y = Im(z) = \pm \frac{1}{2}\left(\frac{\sqrt{|διακρ| - Re(διακρ)}}{2}\right)$$

όπου διακρ = 1-(4z')/λ. Μπορούν αρχικά να υπολογιστούν μερικές τιμές για τα x και y και

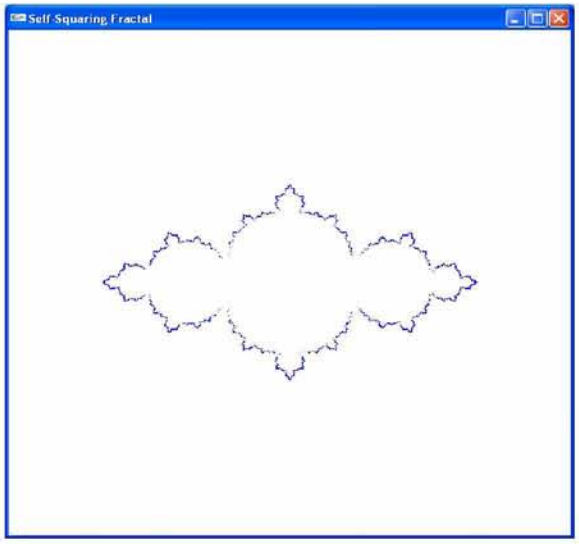
κατόπιν να αγνοηθούν πριν αρχίσουμε να σχεδιάσουμε την καμπύλη fractal.Ακόμη ,αφού αυτή η συνάρτηση δίνει δύο πιθανές μετασχηματισμένες θέσεις (x,y) μπορούμε να επιλέξουμε τυχαία είτε το θετικό είτε το αρνητικό πρόσημο σε κάθε βήμα της επανάληψης εφόσον Im(διακρ) ≥ 0. Όταν Im(διακρ) < 0 οι δύο πιθανές θέσεις βρίσκονται στο δεύτερο και στο τέταρτο τεταρτημόριο. Σε αυτή την περίπτωση , τα x και y πρέπει να έχουν αντίθετα πρόσημα.



Εικόνα 14

Μια καμπύλη fractal που παράχθηκε με την αντίστροφη της συνάρτησης f(z)=λ*z*(1-z) από την διαδικασία selfqTransf χρησιμοποιώντας λ=3 (δεξιά) και λ = 2+i (αριστερά).

Στιγμιότυπο εκτέλεσης του προγράμματος υπολογισμού του fractal νιφάδα:



Εικόνα 15

Μετρήσεις

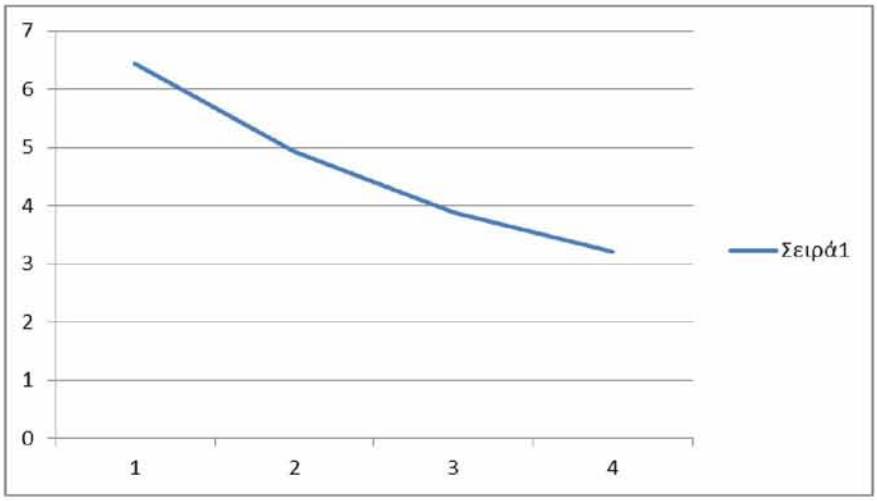
- **Αυτοτετραγωνιζόμενο Fractal (νιφάδα)**

Υπολογισμός 100000000 σημείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	6,440 sec	-	-
2	4,931 sec	1,306	0,653
3	3,887 sec	1,658	0,552
4	3,208 sec	2,007	0,501

Πίνακας 8

Διάγραμμα



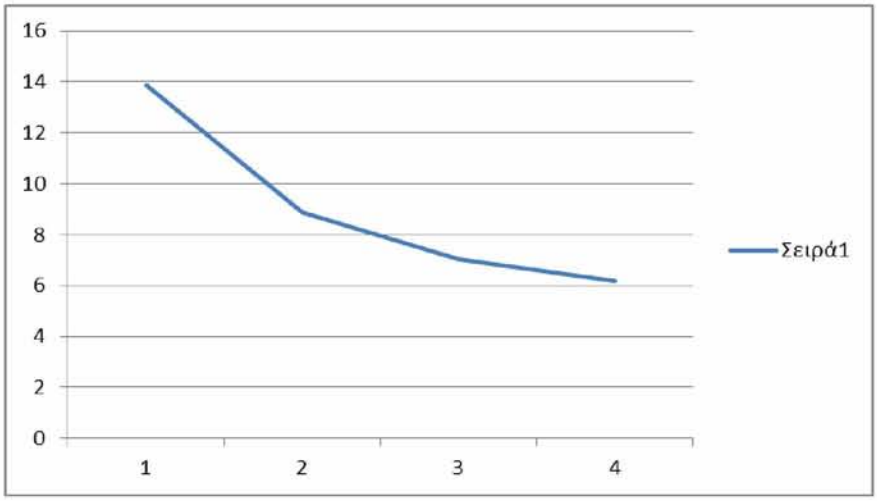
Διάγραμμα 8

Υπολογισμός 200000000 σημείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	13,871 sec	-	-
2	8,854 sec	1,566	0,783
3	7,035 sec	1,971	0,657
4	6,177 sec	2,245	0,561

Πίνακας 9

Διάγραμμα



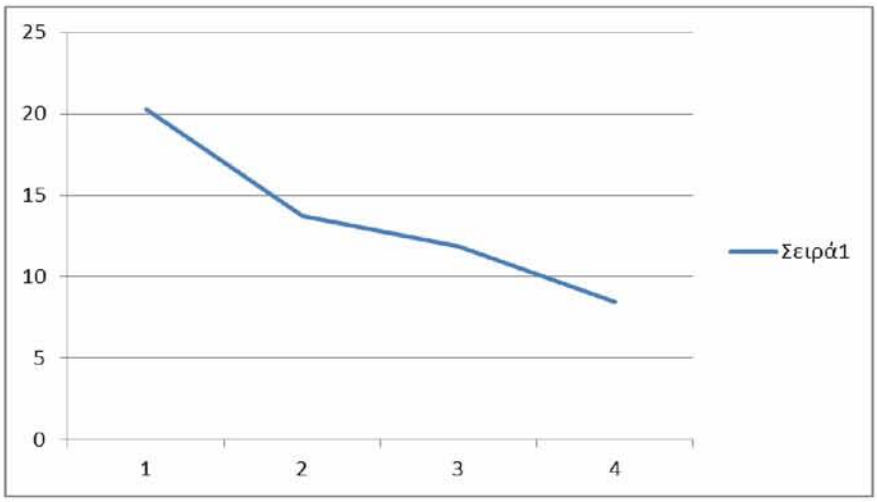
Διάγραμμα 9

Υπολογισμός 300000000 σημείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	20,294 sec	-	-
2	13,761 sec	1,474	0,737
3	11,857 sec	1,711	0,570
4	8,484 sec	2,392	0,598

Πίνακας 10

Διάγραμμα



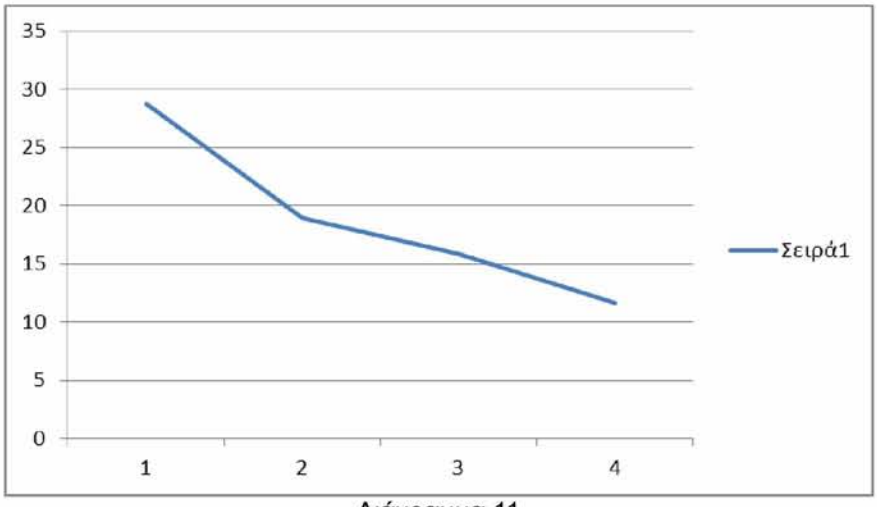
Διάγραμμα 10

Υπολογισμός 400000000 σημείων

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	28,768	-	-
2	18,918	1,520	0,760
3	15,863	1,813	0,604
4	11,658	2,467	0,616

Πίνακας 11

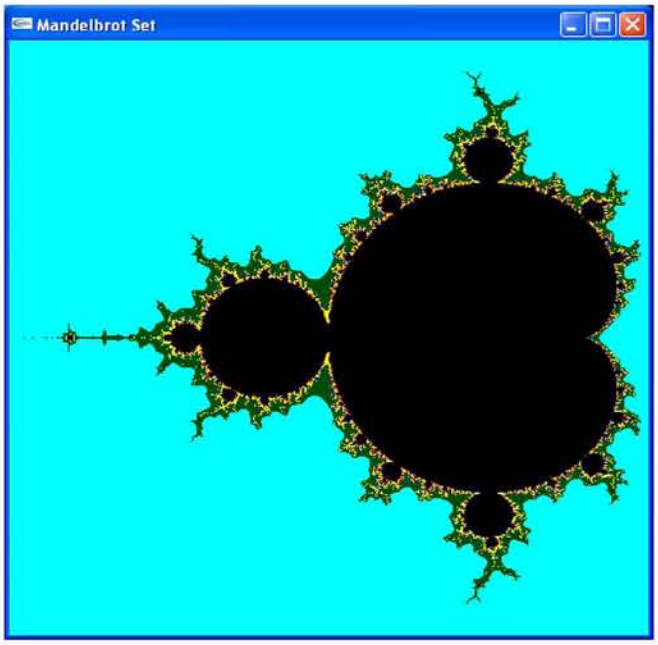
Διάγραμμα



Διάγραμμα 11

Mandelbrot set

Οι συναρτήσεις τετραγώνισης ήταν δύσκολο να αναλυθούν χωρίς τη βοήθεια ενός υπολογιστή. Χρησιμοποιώντας πιο εξελιγμένες τεχνικές γραφικών υπολογιστή, ο Benoit Mandelbrot μελέτησε αυτή τη λειτουργία και βρήκε ένα σύνολο σημείων που είναι γνωστό ως σύνολο Mandelbrot.



Εικόνα 16

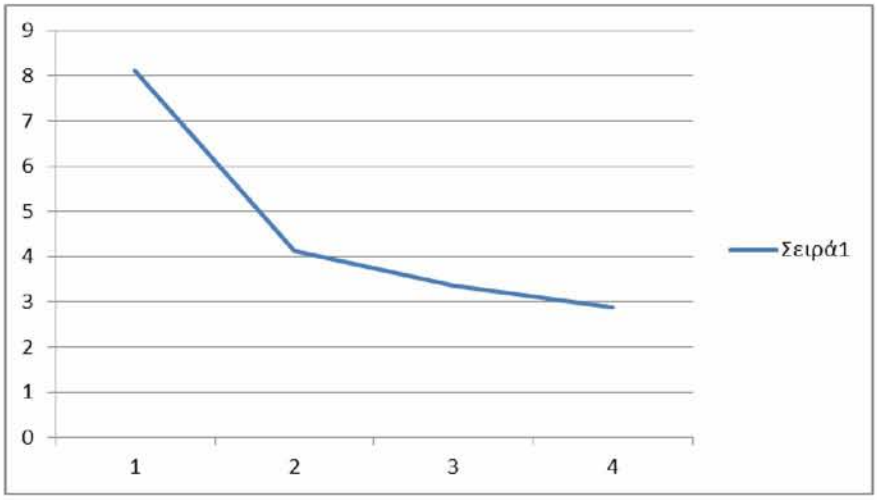
Μετρήσεις

2^{32} υποδιαιρέσεις του x και του y και μέγιστες επαναλήψεις.

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	8,125 sec	-	-
2	4,115 sec	1,957	0,987
3	3,348 sec	2,426	0,808
4	2,872 sec	2,829	0,707

Πίνακας 12

Διάγραμμα



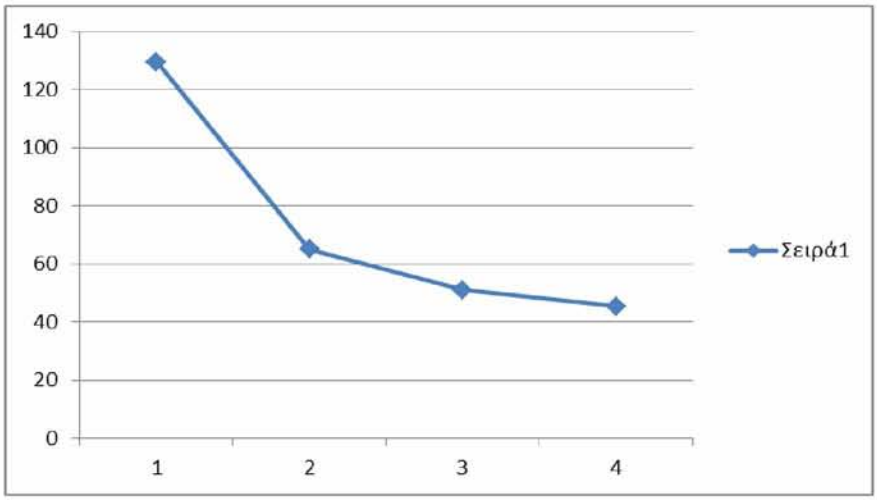
Διάγραμμα 12

2^{36} υποδιαιρέσεις του x και του y και μέγιστες επαναλήψεις.

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	129,511 sec	-	-
2	64,988 sec	1,992	0,996
3	50,944 sec	2,542	0,847
4	45,364 sec	2,854	0,713

Πίνακας 13

Διάγραμμα



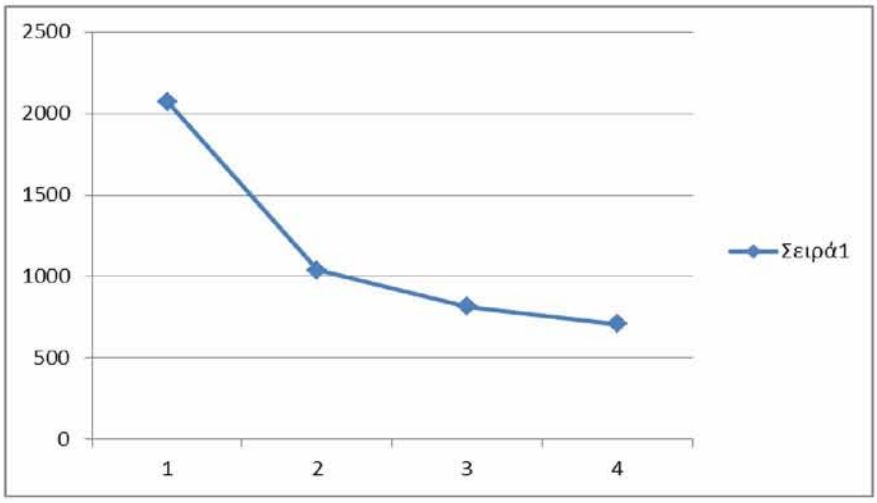
Διάγραμμα 13

2^{40} υποδιαιρέσεις του x και του y και μέγιστες επαναλήψεις.

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	2071,649	-	-
2	1039,573	1,992	0,996
3	814,970	2,451	0,847
4	709,195	2,921	0,730

Πίνακας 14

Διάγραμμα



Διάγραμμα 14

3.5 Αλγόριθμος Floodfill

Ο αλγόριθμος γεμίσματος Floodfill, που ονομάζεται επίσης seed fill, είναι ένας αλγόριθμος που καθορίζει την περιοχή που συνδέεται με ένα δεδομένο κόμβο σε ένα πολύ-διάστατο

πίνακα. Χρησιμοποιείται στα εργαλεία γεμίσματος των προγραμμάτων ζωγραφικής για να καθοριστεί ποια μέρη ενός bitmap θα γεμίσουν με χρώμα, και σε παιχνίδια όπως το Go και ο Ναρκαλιευτής για να καθορίσει ποια κομμάτια θα καθαρίσουν. Όταν εφαρμόζεται σε μια εικόνα για να γεμίσει μια συγκεκριμένη οριοθετημένη περιοχή με χρώμα, ο αλγόριθμος είναι επίσης γνωστός ως boundary fill.

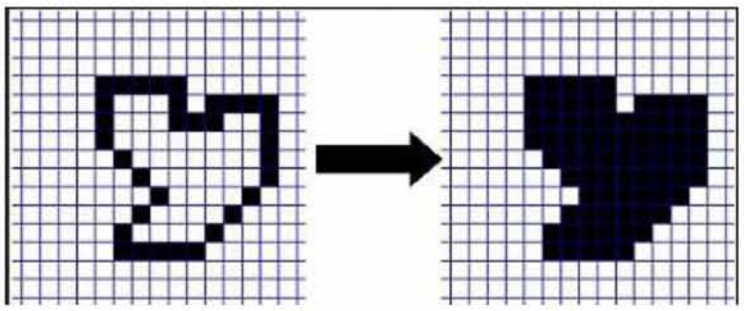
Βασισμένη σε στοίβα αναδρομική υλοποίηση – Ψευδοκώδικας (4-τρόπων)

```
Flood-fill (node, target-color, replacement-color):

1. If the color of node is not equal to target-color, return.
2. Set the color of node to replacement-color.
3. Perform Flood-fill (one step to the west of node, target-color, replacement-color).
   Perform Flood-fill (one step to the east of node, target-color, replacement-color).
   Perform Flood-fill (one step to the north of node, target-color, replacement-color).
   Perform Flood-fill (one step to the south of node, target-color, replacement-color).
4. Return.
```

Κώδικας 8

Παράδειγμα



Εικόνα 17

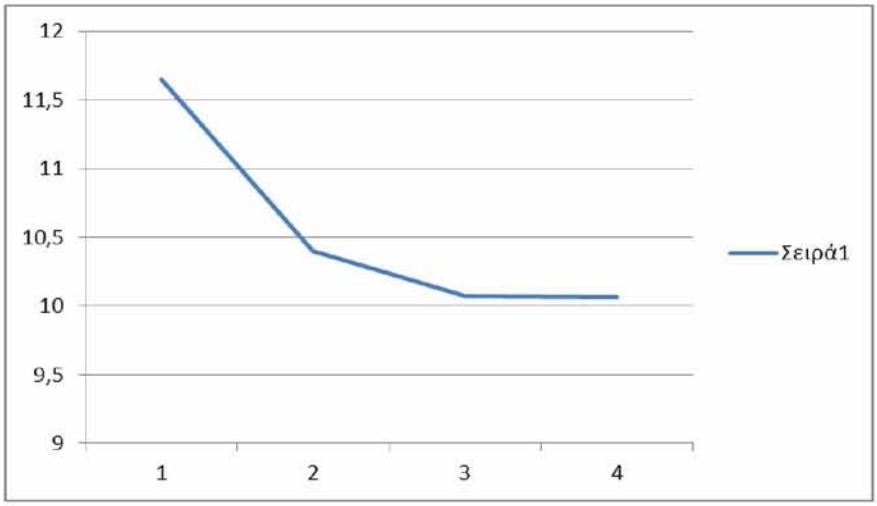
Μετρήσεις

pixel στον άξονα X 100000 - pixel στον άξονα Y 160

Αριθμός επεξεργαστών	Χρόνος ολοκλήρωσης	Χρονοβελτίωση	Απόδοση
1	11,650 sec	-	-
2	10,397 sec	1,120	0,560
3	10,075 sec	1,156	0,385
4	10,063 sec	1,157	0,289

Πίνακας 15

Διάγραμμα



Διάγραμμα 15

4

Επίλογος

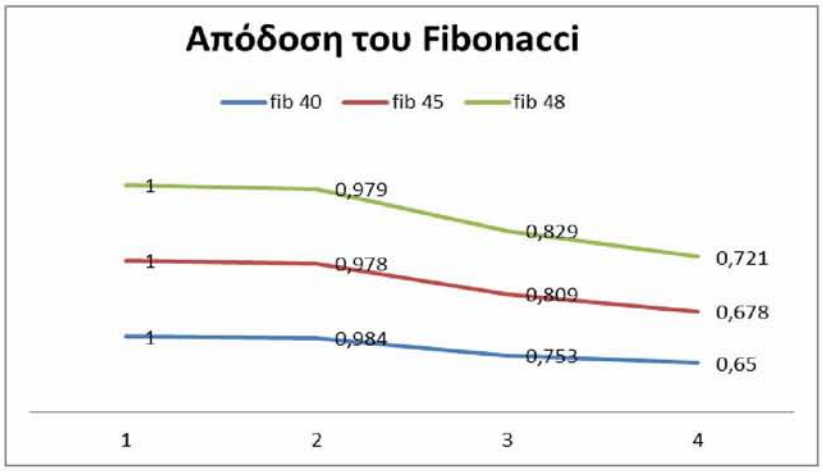
Σύνοψη και συμπεράσματα

Το ερώτημα που προφανώς ανακύπτει από όλη αυτή την μελέτη είναι γιατί να χρησιμοποιηθεί το Cilk και όχι κάποιο άλλο περιβάλλον παράλληλης εκτέλεσης. Η απάντηση που μπορεί να δοθεί είναι ότι η Cilk σαν πολυνηματική γλώσσα προγραμματισμού είναι εξαιρετικά υψηλού επιπέδου κάτι που κάνει την κατανόηση και την χρήση της πολύ εύκολη ακόμη και από αρχάριους προγραμματιστές. Ακόμη το runtime σύστημα του Cilk αναλαμβάνει να κάνει όλη την κατανομή φόρτου στους επεξεργαστές εφαρμόζοντας την τεχνική του work–stealing αφήνοντας στον προγραμματιστή μόνο τον παραλληλισμό. Επίσης προσφέρει πολλά υψηλού επιπέδου και αρκετά εύχρηστα εργαλεία συγχρονισμού. Η δε παραλληλοποίηση του βρόχου for (χρησιμοποιώντας την παράλληλη εκδοχή του cilk_for) είναι εξαιρετικά χρήσιμη, απλή και βολική καθώς μπορεί να εισάγει παραλληλοποίηση σε κάθε επανάληψη του εκάστοτε αλγορίθμου.

Αξιοσημείωτο είναι ότι το Cilk εμφανίζει πολύ καλή απόδοση με την αναδρομικές συναρτήσεις όπου συνήθως επιτυγχάνεται αξιόλογη χρονοβελτίωση.

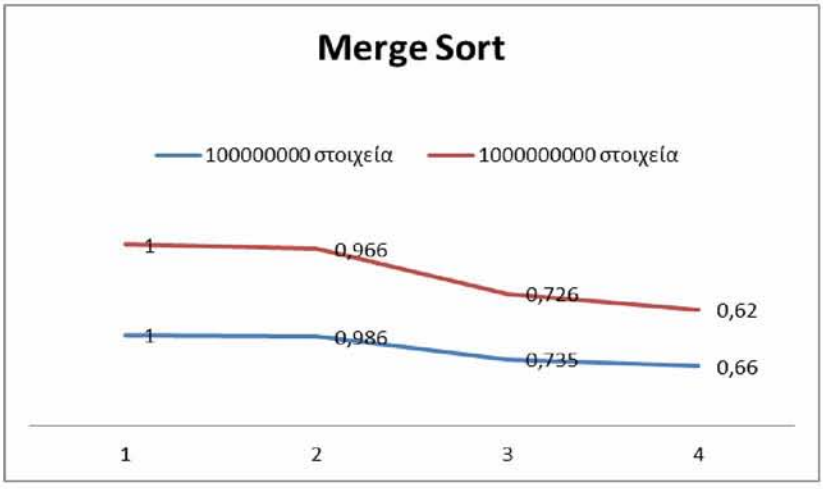
Στο παρακάτω διάγραμμα παρουσιάζεται η απόδοση του Fibonacci για διαφορετικές εισόδους. Αξίζει να παρατηρηθεί ότι μεγαλώνοντας τον αριθμό υπολογισμού έχουμε μια μικρή αύξηση της απόδοσης του αλγορίθμου κυρίως στους 3 και 4 επεξεργαστές. Αυτό πιθανώς να

οφείλεται στο ότι ο χρονοδρομολογητής καταφέρνει να κρατά τους επεξεργαστές απασχολημένους. Όπως φαίνεται και στα διαγράμματα που παρουσιάστηκαν στο προηγούμενο κεφάλαιο η χρονοβελτίωση από τον ένα στους δυο επεξεργαστές είναι σχεδόν γραμμική ενώ φτάνει κοντά στο τρία για τέσσερις επεξεργαστές. Όπως και στους υπόλοιπους αλγορίθμους η μη γραμμικότητα στους τρεις και τέσσερις επεξεργαστές εικάζεται ότι οφείλεται στο hyper threading του επεξεργαστή.



Διάγραμμα 16

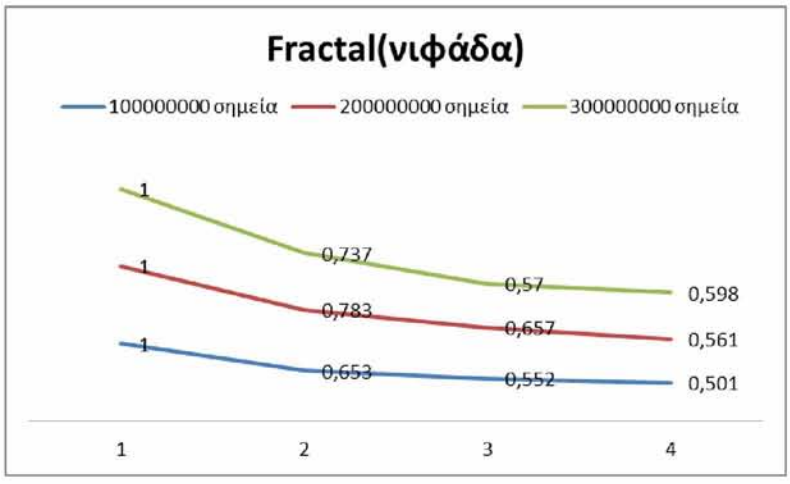
Ακολουθεί η ανάλυση αποδοσης του merge sort που για τα δεκαπλάσια στοιχεία κραταει σχεδόν την ιδια αποδοση. Ομοίως η χρονοβελτίωση από τον ένα στους δυο επεξεργαστές είναι σχεδόν γραμμική ενώ φτάνει στο 2,6 για τέσσερις επεξεργαστές.



Διάγραμμα 17

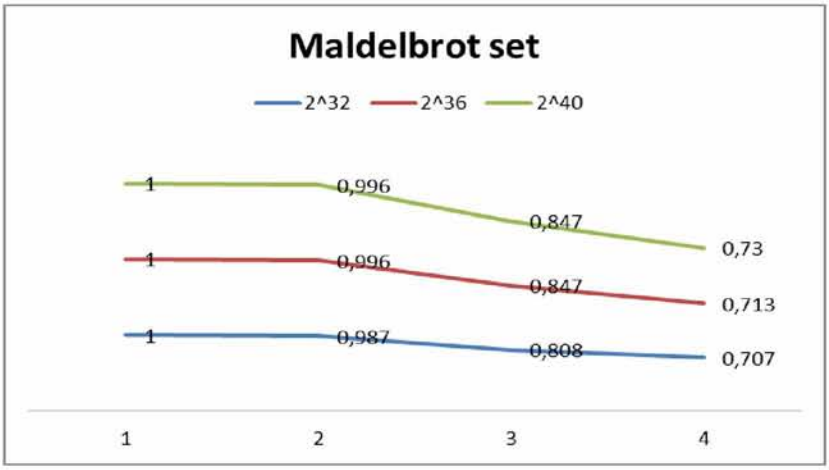
Στά ακόλουθά διαγράμματα παρουσιάζεται η αποδοση του αλγορίθμων αυτοτετραγωνιζόμενων fractal και συγκεκριμένα του σχεδιασμού της νιφάδας και του σχεδιασμού του Maldelbrot set.Πρεπει να σημειωθει ότι ο αλγόριθμοι ζωγραφίζουν μέσω της opengl και ως σειριακοί χρησιμοποιήθηκαν οι κώδικες του βιβλίου Γραφικα Υπολογιστών με Opengl των Hearn Baker ($3^{η}$ έκδοση).Έγιναν αρκετές αλλαγες για να μην επιβαρύνει η χρηση της opengl την παραλληλοποιηση αφού όπως είναι λογικο η opengl δεν επιτρεπει σε πάνω από ένα νηματα να ζωγραφισουν ταυτοχρονα.Ακομη ο αλγόριθμος της νιφάδας χρησιμοποιεί την rand() σε κάποιο σημείο των υπολογισμών του η οποια δεν επιτρέπει παραλληλη χρηση.Αντικατασταθηκε από την rand_r(thread_id) που επιτρεπει την χρησιμοποιήση της από πολλαπλα νήματα.Η παραλληλοποιήση έγινε ως προς τον αριθμό των σημειων που χρησιμοποιούνται για τον σχεδιασμο του fractal νιφάδας και ως προς τον αριθμό των σημειων ως προς τον αξονα του x(έγινε κάθετος τεμαχισμός).Αναλογα με τον αριθμό των νηματων που δινονται ως είσοδος διαιρειται το συνολο των σημειων σε κομματια και κάθε νημα αναλαμβανει ένα κομματι.

Όπως φαίνεται και από το πρώτο διάγραμμα ο αλγόριθμος νιφάδας αυξανει την αποδοση του όσο αυξανεται ο αριθμός των σημείων το οποίο πιθανώς να οφείλεται στο ότι ο χρονοδρομολογητής καταφέρνει να κρατά τους επεξεργαστές απασχολημένους. Όπως φαίνεται και στα διαγράμματα που παρουσιάστηκαν στο προηγούμενο κεφάλαιο η χρονοβελτίωση από τον ένα στους δυο επεξεργαστές είναι σχεδόν γραμμική ενώ φτάνει κοντά στο τρία για τέσσερις επεξεργαστές.



Διάγραμμα 18

Ομοιώς και στο δεύτερο διάγραμμα για το Madelbrot set η αποδοση αυξανει με την αυξηση των υποδιαιρέσεων και του αριθμού των επαναλήψεων. Η χρονοβελτίωση από τον ένα στους δυο επεξεργαστές είναι γραμμική ενώ φτάνει κοντά στο τρία για τέσσερις επεξεργαστές.



Διάγραμμα 19

Τέλος, ο αλγόριθμος γεμίσματος Floodfill όπως προκύπτει και από το διάγραμμα του προηγούμενου κεφαλαίου δεν παρουσιάζει σχεδόν καθόλου χρονοβελτίωση. Αυτό οφείλεται στο γεγονός ότι ο συγκεκριμένος αλγόριθμος δεν έχει καθόλου υπολογισμούς και ο χρονοδρομολογητής δεν μπορεί να απασχολήσει τους επεξεργαστές. Επίσης λόγω της ταχύτητας του αλγορίθμου δεν κατέστη δυνατόν να παρθούν μεγάλες μετρήσεις.

5

Βιβλιογραφία

http://pdplab.it.uom.gr/teaching/llnl-gr/Introduction%20to%20Parallel%20Computing.htm#Whatis

http://en.wikipedia.org/wiki/Parallel_computing

http://inf-server.inf.uth.gr/courses/CE404/lectures/lec.html

http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

http://supertech.csail.mit.edu/cilk/lecture-1.ppt

http://supertech.csail.mit.edu/cilk/lecture-2.ppt

http://supertech.csail.mit.edu/cilk/lecture-3.ppt

http://www.cslab.ntua.gr/courses/pps/files/fall2011/pps-cilk.pdf

http://software.intel.com/sites/default/files/m/8/0/1/5/4/28680-Cilk_User_Guide_-ENG.pdf

http://en.wikipedia.org/wiki/Fibonacci_number

http://www.cs.northwestern.edu/academics/courses/110/html/fib_rec.html

http://en.wikipedia.org/wiki/Merge_sort

http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm

Computer graphics with opengl – Hearn Baker 3rd edition

http://en.wikipedia.org/wiki/Flood_fill

http://lodev.org/cgtutor/floodfill.html