ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη Αλγορίθμων Εκτέλεσης και Χρονοδρομολόγησης Παράλληλων Εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευστράτιος Π. Παλαιολόγος

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μελέτη Αλγορίθμων Εκτέλεσης και Χρονοδρομολόγησης Παράλληλων Εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευστράτιος Π. Παλαιολόγος

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Ιανουαρίου 2014.

Αθήνα, Ιανουάριος 2014

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Δημήτριος Σούντρης Επίκουρος Καθηγητής Ε.Μ.Π.	Δημήτριος Τσουμάκος Επίκουρος Καθηγητής Ιονίου Πανεπιστημίου

(Υπογραφή)

.....

ΠΑΛΑΙΟΛΟΓΟΣ ΕΥΣΤΡΑΤΙΟΣ

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Μια διαδεδομένη τεχνική για την αξιοποίηση πολυεπεξεργαστικών συστημάτων είναι η δόμηση μιας εφαρμογής σε ανεξάρτητες εργασίες που μπορούν να εκτελούνται ταυτόχρονα, γνωστή ως multithreading. Τα περισσότερα εργαλεία παράλληλου προγραμματισμού φροντίζουν αυτές οι εργασίες να κατανέμονται και να εκτελούνται στους επεξεργαστές αυτόματα, χωρίς την παρέμβαση του προγραμματιστή. Αυτό κάνει τον παράλληλο προγραμματισμό αφενός πιο εύκολο, καθώς ο προγραμματιστής ασχολείται μόνο με τον τρόπο που θα δομήσει την εφαρμογή του σε ανεξάρτητες εργασίες, και αφετέρου πιο αποδοτικό, αφού οι εργασίες δρομολογούνται από ένα σύστημα που εγγυημένα δουλεύει σωστά και γρήγορα.

Στόχος αυτής της εργασίας είναι η μελέτη του τρόπου που μπορούμε να διαχειριστούμε παράλληλες εργασίες -πώς να τις δημιουργήσουμε, να τις τερματίσουμε και να τις συγχρονίσουμε μεταξύ τους- καθώς και το πώς μπορούν να δρομολογηθούν αυτές σε ένα πλήθος επεξεργαστών. Η μελέτη γίνεται σε εργαλεία παράλληλου προγραμματισμού που παρέχουν τέτοιες δυνατότητες, κυρίως τα Posix Threads και η Cilk. Κατόπιν, αναλύουμε δυο κύριες τεχνικές χρονοδρομολόγησης παράλληλων εργασιών, το work sharing και το work stealing. Μελετούμε αλγορίθμους που βασίζονται σε αυτές τις φιλοσοφίες και την επίδοση που έχουν βάσει θεωρητικών μοντέλων.

Τέλος, για εκπαιδευτικούς λόγους, εμπνεόμενοι από τα εργαλεία που εξετάστηκαν, υλοποιούμε σε γλώσσα C μια βιβλιοθήκη διαχείρισης tasks και ένα δρομολογητή. Ο δρομολογητής εκτελεί απλοποιημένες εκδοχές αλγορίθμων work sharing και work stealing.

Λέξεις Κλειδιά

Cilk, multitasking, work sharing, work stealing, αλγόριθμοι δρομολόγησης, παράλληλος προγραμματισμός, RTS

Abstract

A common technique for exploiting SMP systems is the division of an application into independent tasks that can be performed simultaneously, known as multithreading. Most parallel programming tools ensure these tasks are distributed and executed on processors automatically, without intervention by the programmer. This makes parallel programming easier, as the programmer only deals with how to structure the application into independent tasks, and also more efficient, since the work initiated by a system is guaranteed to work correctly and quickly.

The aim of this work is the study of how we can manage parallel work - how to create, terminate and synchronize them - and how they can be scheduled to a set of processors. The study is using parallel programming tools that provide such capabilities, particularly the Posix Threads and Cilk. Then, we analyze two main techniques of scheduling parallel tasks, work sharing and work stealing. We study algorithms based on these philosophies and their performance based on theoretical models.

Finally, for educational purposes, inspired from the examined tools, we implement a library management tasks and a scheduler using C language. The scheduler performs simplified versions of work sharing and work stealing algorithms.

Keywords

Cilk, multitasking, work sharing, work stealing, scheduling algorithms, parallel programming, RTS, pthreads

ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΙΝΑΚΑΣ ΣΧΗΜΑΤΩΝ

ΚΕΦΑΛΑΙΟ 1: ΕΙΣΑΓΩΓΗ

1.1) Ιστορική Αναδρομή

1.1.1) Ο Νόμος του Moore

Ο παράλληλος προγραμματισμός αναφέρεται στον κλάδο της Επιστήμης των Υπολογιστών που ασχολείται με την αρχιτεκτονική συστημάτων σε επίπεδο hardware, προγραμματισμό εργασιών σε επίπεδο software και στη μελέτη της απόδοσης τους, για συστήματα που μπορούν να εκτελέσουν ταυτόχρονα εφαρμογές.

Η ιδέα του παράλληλου προγραμματισμού δεν είναι πρόσφατη, αλλά ήδη από τις αρχές της δεκαετίας του '60, με την παρουσίαση του 704 από την IBM με αρχιτέκτονα τον Gene Amdahl, τίθενται οι βάσεις για τη μελέτη συστημάτων παράλληλης επεξεργασίας, οπότε και παρουσιάζονται εργασίες και συζητήσεις πάνω σε αυτόν τον τομέα [1]. Στα επόμενα χρόνια, γίνεται λόγος για την απόδοση και τη βέλτιστη αξιοποίηση των παράλληλων συστημάτων με κατάληξη στη διατύπωση των νόμων του Amdahl [2] και του Gustafson [3], βάσει των οποίων η επιτάχυνση ενός προγράμματος δεν είναι ακριβώς ανάλογη με το πλήθος των πυρήνων -όπως θα εύχονταν όλοι- αλλά εξαρτάται και από τη φύση του προγράμματος, πόσο αυτό παραλληλοποιείται. Για τις επόμενες δεκαετίες η έννοια του παράλληλου προγραμματισμού συνδέθηκε στενά με την έννοια του supercomputing. Οι υπερυπολογιστές είναι υπολογιστές κατασκευασμένοι με σκοπό να έχουν μεγάλη υπολογιστική ταχύτητα και ισχύ. Χρησιμοποιούνται σε διάφορες επιστημονικές εφαρμογές, όπως στη μελέτη του διαστήματος και σε ιατρικές εφαρμογές. Αρχικά αποτελούνταν από λίγες επεξεργαστικές μονάδες, αλλά από τη δεκαετία του '90 και μετά άρχισαν να εμφανίζονται μηχανήματα με χιλιάδες πυρήνες. Μέχρι και πριν είκοσι χρόνια, η εφαρμογή των παράλληλων συστημάτων γινόταν μόνο για μη εμπορικές εφαρμογές και σε πολύ ειδικούς τομείς. Πλέον, όμως, η ταυτόχρονη εκτέλεση εργασιών δεν αφορά μόνο τέτοιους ειδικούς τομείς απαιτητικών εφαρμογών, αλλά κάθε εφαρμογή που τρέχει σε πολυπύρηνο σύστημα. Εύλογα μπορεί κάποιος να αναρωτηθεί γιατί μόλις τα τελευταία είκοσι χρόνια η επιστημονική κοινότητα ασχολείται πιο θερμά με έναν τομέα, ο οποίος είχε εμφανιστεί πριν εξήντα χρόνια, και γιατί άργησε τόσο πολύ να αποκτήσει εμπορική χρήση. Η απάντηση είναι απλή: η επίδοση.

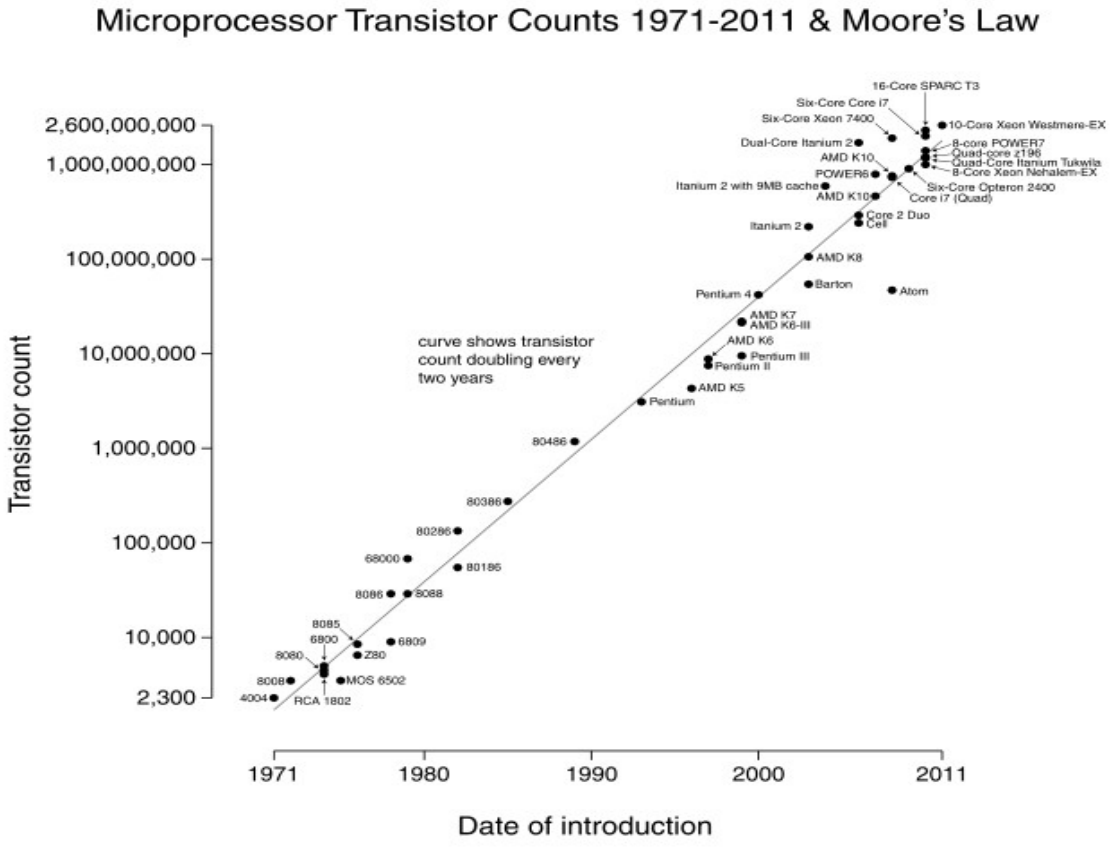
Ο κόσμος των μικροεπεξεργαστών, από την αρχή της δημιουργίας τους, διέπεται από ένα νόμο, το νόμο του Moore. Ο Gordon Moore, συνιδρυτής της εταιρίας κατασκευαστών μικροεπεξεργαστών Intel, έκανε το 1965 την πρόβλεψη ότι η πυκνότητα ενός μικροεπεξεργαστή σε τρανζίστορ θα διπλασιάζεται κάθε 18 με 24 μήνες (Σχήμα 1). Η πρόβλεψη αυτή αποδείχθηκε έγκυρη, καθώς για σχεδόν μισό αιώνα η κατασκευή μικροεπεξεργαστών ακολουθούσε αυτό το νόμο. Μόνο τα τελευταία δυο-τρία χρόνια

παρατηρείται μείωση στον εκθετικό ρυθμό αύξησης, γεγονός που οφείλεται στο ότι η τεχνολογία πλησιάζει στα όρια του υλικού.

Μαζί με το πλήθος των τρανζίστορ αυξανόταν μέχρι και το 2004 και η συχνότητα των μικροεπεξεργαστών με παρόμοιο ρυθμό. Και αυτό το γεγονός ήταν που οδηγούσε στη συνεχόμενη αύξηση της αποδοτικότητας των υπολογιστών. Ο χρόνος που απαιτείται για να τρέξει μια εφαρμογή δίνεται από τον τύπο:

$$Runtime = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}$$

Έτσι, αυξανόμενη η συχνότητα του ρολογιού, μείωνε τον τελευταίο όρο της εξίσωσης, μειώνοντας το συνολικό χρόνο που χρειαζόταν να εκτελεστεί μια εφαρμογή. Γι' αυτό το λόγο, για κάποιες δεκαετίες, επίδοση και συχνότητα του ρολογιού ήταν έννοιες συνυφασμένες. Έφτασε, όμως, ένα σημείο όπου η περαιτέρω αύξηση της συχνότητας ήταν αδύνατη. Κι αυτό γιατί μαζί με αυτήν αυξάνεται η ισχύς που καταναλώνει το ολοκληρωμένο κύκλωμα και κατ' επέκταση η θερμοκρασία του. Αυτό είχε ως αποτέλεσμα την προβληματική λειτουργία του επεξεργαστή.



Σχήμα 1: Γραφική Παράσταση του Νόμου του Moore

Παρ' όλα αυτά ο νόμος του Moore συνέχισε να ισχύει. Τα κυκλώματα που φτιάχνονταν είχαν συνεχώς περισσότερα τρανζίστορ, αλλά η συχνότητα στην οποία λειτουργούσαν δεν μπορούσε να αυξηθεί. Οδηγηθήκαμε, λοιπόν, σε ολοκληρωμένα κυκλώματα, τα οποία μπορούσαν να φιλοξενήσουν στο ίδιο κομμάτι πυριτίου δεκάδες και εκατοντάδες επεξεργαστικές μονάδες, οι οποίες, όμως, είχαν συχνότητα ρολογιού λίγο μικρότερη από τη μέγιστη που επιτυγχανόταν αν είχαμε έναν επεξεργαστή στο ολοκληρωμένο κύκλωμα. Το πρόβλημα ήταν να μπορέσουν να αξιοποιηθούν όλες αυτές οι μονάδες ώστε να βελτιωθεί η επίδοση των συστημάτων. Η λύση δόθηκε από τον παραλληλισμό.

1.1.2 Επίπεδα Παραλληλισμού

Ο παραλληλισμός ανέκαθεν ήταν ο παράγοντας που έδινε λύση στην εκμετάλλευση των τρανζίστορ του μικροεπεξεργαστή, που συνεχώς αυξάνονταν. Αρχικά, ο παραλληλισμός ήταν bit-level (σε επίπεδο bit). Μέχρι το 1986, η αξιοποίηση γινόταν με την αύξηση του μήκους λέξης, του μικρότερου bit unit που διαχειρίζεται ο επεξεργαστής. Με αυτόν τον τρόπο μειωνόταν ο αριθμός των εντολών για να εκτελεστεί μια πράξη, αυξάνοντας την απόδοση. Έτσι, από τους αρχικά 4-bit επεξεργαστές οδηγηθήκαμε στους 8-bit, έπειτα στους 16-bit, για να φτάσουμε στους 32-bit και τα τελευταία χρόνια να αποκτούν όλο και μεγαλύτερο χώρο στην αγορά οι 64-bit επεξεργαστές.

Το επόμενο επίπεδο παραλληλίας είναι το instruction level parallelism (ILP). Μέχρι τα μέσα της δεκαετίας του '80 οι εντολές σε έναν μικροεπεξεργαστή εκτελούνταν σειριακά. Από τότε, όμως, και για περίπου μια δεκαετία, αναπτύχθηκαν τεχνικές έτσι ώστε οι εντολές να μπορούν να τρέχουν παράλληλα και με διαφορετική σειρά, χωρίς αυτό να αλλοιώνει το τελικό αποτέλεσμα. Με τη χρήση της τεχνικής του pipeline [4], ο επεξεργαστής διασπά κάθε εντολή σε στάδια, έτσι ώστε να μπορούν τελικά να εκτελούνται ταυτόχρονα τόσες εντολές όσο είναι το πλήθος των σταδίων του pipeline. Δεδομένου ότι κάθε στάδιο διαρκεί έναν κύκλο ρολογιού, αυτή η τεχνική οδήγησε στο να μπορεί να εκτελείται μια εντολή ανά κύκλο, ανεξάρτητα από το μέγεθός της, πόσους δηλαδή κύκλους ρολογιού θα χρειαζόταν αν εκτελούνταν σειριακά. Κατόπιν αυτή η τεχνική αξιοποιήθηκε ακόμη περισσότερο από υπερβαθμωτούς επεξεργαστές, οι οποίοι επιτρέπουν να υπάρχουν στο ίδιο στάδιο του pipeline παραπάνω από μια εντολές, χρησιμοποιώντας ταυτόχρονα μεθόδους για out-of-order εκτέλεση εντολών.

Όταν άρχισε να εξαντλείται και αυτό το επίπεδο παραλληλίας στο βαθμό που βελτίωνε την υπολογιστική επίδοση, ήρθε στο φως η παραλληλία σε επίπεδο thread (multithreading parallelism), όπου thread (νήμα) είναι μια ακολουθία εντολών. Αναπτύχθηκαν τεχνικές με τις οποίες ο επεξεργαστής μπορούσε να τρέξει ακολουθίες εντολών παράλληλα, οι οποίες να εκτελούν υπολογισμούς είτε στα ίδια είτε σε διαφορετικά δεδομένα. Παρακάτω σε αυτήν την εργασία θα μας απασχολήσει ιδιαίτερα αυτό το επίπεδο παραλληλισμού, πάνω στο οποίο γίνεται μεγάλη έρευνα σήμερα από την επιστημονική κοινότητα.

Τέλος, στις μέρες μας, η παραλληλία έχει φτάσει σε επίπεδο πυρήνων. Δηλαδή, γίνονται έρευνες ώστε να αξιοποιηθούν με το βέλτιστο τρόπο πολλές επεξεργαστικές μονάδες οι οποίες βρίσκονται ενσωματωμένες στο ίδιο ολοκληρωμένο κύκλωμα. Το ενδιαφέρον -και χιουμοριστικό- ερώτημα που τίθεται σε αυτήν την περίπτωση είναι αν θα σέρνουν ένα κάρο (μια εφαρμογή) δύο άλογα (δυο επεξεργαστές υψηλής συχνότητας) ή χίλιες κότες (χαμηλότερης επίδοσης επεξεργαστές).

1.2 Αρχιτεκτονικές Συστημάτων Παράλληλου Προγραμματισμού

Πριν παρουσιάσουμε τρόπους και μοντέλα παράλληλου προγραμματισμού, είναι ενδιαφέρον να κάνουμε μια σύντομη παρουσίαση στις αρχιτεκτονικές συστημάτων που υποστηρίζουν παράλληλο προγραμματισμό. Άλλωστε, για την καλύτερη επίδοση των εφαρμογών, οφείλουμε να γνωρίζουμε τα χαρακτηριστικά του συστήματος στο οποίο θα εκτελεστούν και με ποιον τρόπο μπορεί αυτό να αξιοποιηθεί στο μέγιστο βαθμό. Αρχικά γίνεται μια ταξινόμηση των συστημάτων κι έπειτα περιγράφονται συνοπτικά οι τρεις επικρατέστερες αρχιτεκτονικές.

➔ *Ταξινόμηση Flynn*

Μια μεγάλη ταξινόμηση στις αρχιτεκτονικές συστημάτων παράλληλης επεξεργασίας παρουσιάστηκε από τον Michael J. Flynn το 1966 [5]. Η ταξονομία του Flynn, όπως έχει γίνει γνωστή, βασίζεται στο πλήθος των υποστηριζόμενων παράλληλων εντολών και ροών δεδομένων στην αρχιτεκτονική. Πιο αναλυτικά, υπάρχουν τέσσερις κατηγοριοποιήσεις:

(1) SISD (Single Instruction Single Data) : πρόκειται για σειριακό επεξεργαστή, ο οποίος δεν έχει κανενός είδους παραλληλισμού ούτε ως προς τις εντολές ούτε ως προς τα δεδομένα.

(2) SIMD (Single Instruction Multiple Data) : αρχιτεκτονική που επιτρέπει την ίδια εντολή να εκτελεστεί πάνω σε πολλά data sets ταυτόχρονα. Χρησιμοποιείται αρκετά σε GPUs.

(3) MISD (Multiple Instruction Single Data) : αρχιτεκτονική όπου πολλές εντολές μπορούν παράλληλα να χρησιμοποιήσουν το ίδιο set δεδομένων. Χρησιμοποιείται πολύ σπάνια, κυρίως σε συστήματα ανοχής σφαλμάτων.

(4) MIMD (Multiple Instruction Multiple Data) : πρόκειται για πολλούς επεξεργαστές που εκτελούν διαφορετικές εντολές στο δικό τους set δεδομένων παράλληλα. Αρχιτεκτονική που χρησιμοποιείται ως επί των πλείστων στα συστήματα

παράλληλης επεξεργασίας με εξαιρετικές αποδόσεις. Δεν είναι τυχαίο που από το 2007 και μετά οι καλύτεροι υπερυπολογιστές είναι σχεδιασμένοι με τέτοιων ειδών αρχιτεκτονικές.

Το 1990, βασισμένος στην ταξονομία του Flynn, ο Ralph Duncan, πρότεινε μια επέκτασή της [6] ώστε να περιλαμβάνει pipelined vector processors, αλλά και κάποιες επιπλέον αρχιτεκτονικές που λειτουργούσαν παράλληλα, αλλά δεν κατηγοριοποιούνταν σύμφωνα με το Flynn. Έτσι, η νέα κατηγοριοποίηση του Duncan περιλαμβάνει τρεις μεγάλες κλάσεις (Σύγχρονες, MIMD και MIMD paradigm), οι οποίες υποδιαιρούνται σε μικρότερες.

Όταν αναφερόμαστε σε συστήματα πολλών επεξεργαστών, ανάλογα με την τοπολογία επεξεργαστών-μνήμης, πώς συνδέονται οι επεξεργαστές με τη μνήμη (ή τις μνήμες) που βρίσκονται αποθηκευμένα τα δεδομένα, διακρίνουμε τριών ειδών αρχιτεκτονικές: κοινής μνήμης, κατανεμημένης μνήμης και υβριδικές. Αυτές αναλύουμε παρακάτω.

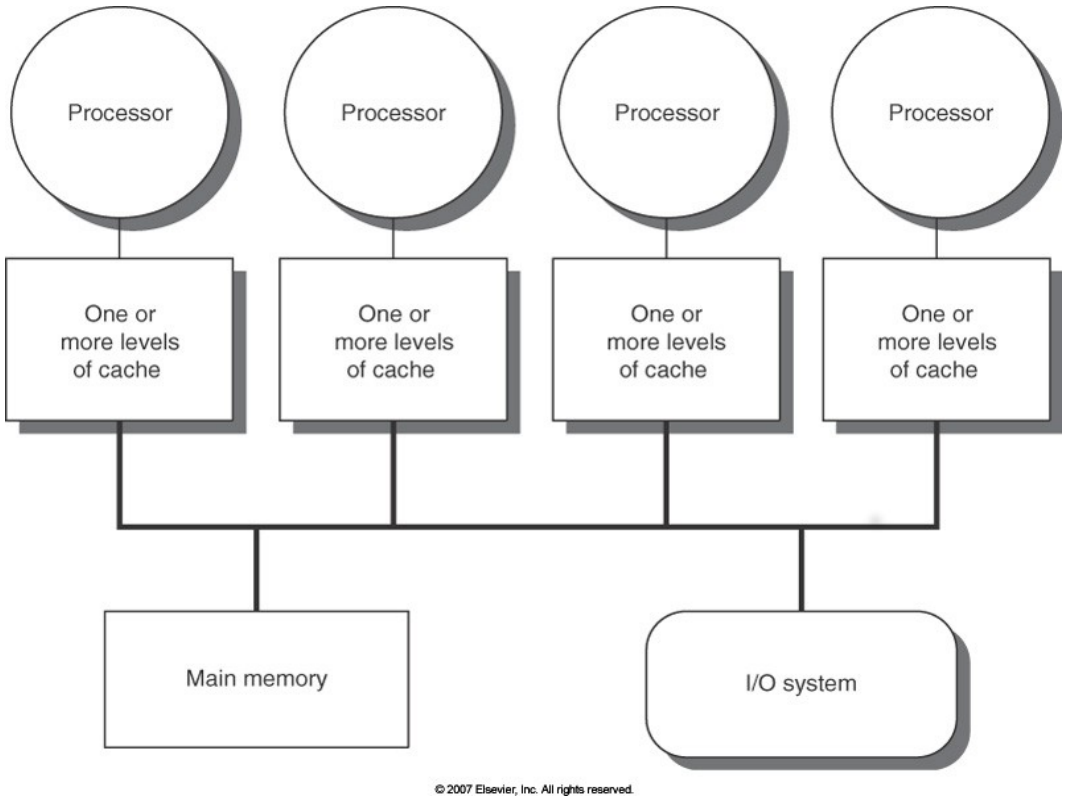
➔ *Αρχιτεκτονική Κοινής Μνήμης (Shared Memory)*

Ένα σύστημα πολλών επεξεργαστών έχει αρχιτεκτονική κοινής μνήμης όταν όλοι οι επεξεργαστές μοιράζονται την ίδια φυσική μνήμη. Καθένας τους έχει δική του ιεραρχία τοπικών μνημών (cache memories) και συνδέονται σε ένα κοινό διάδρομο (bus), μέσω του οποίου επικοινωνούν με την κοινή μνήμη (Σχήμα 2).

Πιο διαδεδομένα είναι τα συστήματα που επιτρέπουν συμμετρική πρόσβαση στην κοινή μνήμη για κάθε επεξεργαστή, οι οποίοι είναι πανομοιότυποι, γνωστά ως SMPs (Symmetric MultiProcessor). Τα SMPs κυριαρχούν στον τομέα των servers και σιγά σιγά γίνονται όλο και πιο διαδεδομένα στα desktops. Ανάλογα με το χρόνο προσπέλασης της μνήμης, τα συστήματα αυτά διακρίνονται σε UMA και NUMA. Στον UMA (Uniform Memory Access) σχεδιασμό, κάθε επεξεργαστής μπορεί να προσπελάσει οποιαδήποτε διεύθυνση μνήμης στον ίδιο χρόνο, σε αντίθεση με το NUMA (Non- Uniform Memory Access), όπου ο χρόνος προσπέλασης μιας θέσης μνήμης δεν είναι σταθερός, αλλά εξαρτάται από την απόστασή της από τον επεξεργαστή.

Τα συστήματα κοινής μνήμης διευκολύνουν τον παράλληλο προγραμματισμό, λόγω της δυνατότητάς τους κάθε επεξεργαστής να γράφει και να διαβάζει από οποιαδήποτε θέση μνήμης. Έτσι, μπορούν να ενημερώνονται όλοι για κάθε αλλαγή στα δεδομένα αυτόματα μέσω της μνήμης. Αυτό φυσικά απαιτεί από τον προγραμματιστή να λαμβάνει υπόψιν του τον τυχόν συναγωνισμό των επεξεργαστών πάνω στα ίδια δεδομένα, θέτοντας κάποια προτεραιότητα στην πρόσβαση, ώστε να εξασφαλιστεί η ορθή λειτουργία. Επιπλέον, λόγω της αύξησης της χωρητικότητας των κρυφών μνημών και της μείωσης του μεγέθους των επεξεργαστών, αυτά τα συστήματα είναι εξαιρετικά αποδοτικά. Το αρνητικό τους στοιχείο είναι ότι δεν μπορούν να υποστηρίξουν αποδοτικά περισσότερους από 32 επεξεργαστές,

εξαιτίας του κοινού διαδρόμου σύνδεσης με την κοινή μνήμη. Αυτό συνεπάγεται ότι δεν μπορεί να κλιμακωθεί.



Σχήμα 2: Αρχιτεκτονική Κοινής Μνήμης

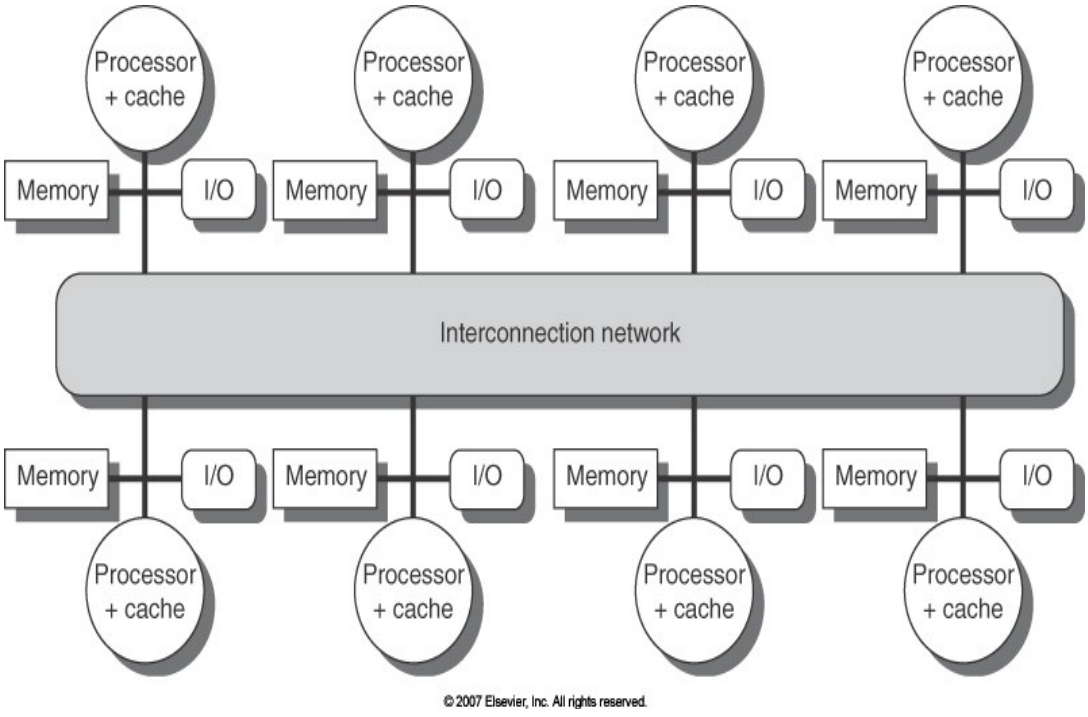
➔ *Αρχιτεκτονική Κατανεμημένης Μνήμης (Distributed Memory)*

Ένα διαφορετικό είδος σχεδιασμού είναι εκείνο της κατανεμημένης μνήμης. Το σύστημα αποτελείται από επεξεργαστές, καθένας από τους οποίους έχει τη δική του κρυφή μνήμη και ιδιωτική κύρια μνήμη. Οι κόμβοι "επεξεργαστής-μνήμη" συνδέονται μεταξύ τους μέσω ενός δικτύου διασύνδεσης (Σχήμα 3).

Κάθε επεξεργαστής μπορεί να εκτελέσει υπολογισμούς μόνο με τα δεδομένα της μνήμης του και σε περίπτωση που χρειαστεί μη τοπικά δεδομένα, πρέπει να τα ζητήσει από τους άλλους επεξεργαστές μέσω αποστολής μηνυμάτων. Με αυτόν τον τρόπο, ναι μεν αποκλείονται περιπτώσεις συναγωνισμού των επεξεργαστών για την προτεραιότητα εγγραφής και διαβάσματος στα δεδομένα, όπως συνέβαινε στις αρχιτεκτονικές κοινής μνήμης, αλλά το πρόβλημα μετατίθεται στην κατανομή των δεδομένων στις ιδιωτικές μνήμες των επεξεργαστών και στην επικοινωνία τους μέσω του δικτύου διασύνδεσης. Αυτό

είναι που κάνει τον προγραμματισμό τέτοιων συστημάτων ιδιαίτερα απαιτητικό και δύσκολο.

Τα κατανεμημένης μνήμης συστήματα έχουν το βασικό πλεονέκτημα ότι μπορούν να κλιμακωθούν ώστε στο δίκτυο να συνδέονται χιλιάδες υπολογιστές, σχηματίζοντας computer clusters. Κλειδί για την επίδοση αποτελεί η τοπολογία του δικτύου. Πιο συνηθισμένη τεχνολογία διασύνδεσης είναι το Gigabit Ethernet, αλλά υπάρχουν και πιο εξελιγμένες διασυνδέσεις, όπως το Myrinet [7] και το Infiniband [8]. Όσον αφορά υπερυπολογιστές, το δίκτυο διασύνδεσης κατασκευάζεται κατάλληλο για τις απαιτήσεις των εφαρμογών, τις οποίες σχεδιάζεται να εκτελέσει.



Σχήμα 3: Αρχιτεκτονική Κατανεμημένης Μνήμης

➔ *Υβριδικές Αρχιτεκτονικές (Hybrid Architectures)*

Οι σύγχρονοι υπολογιστές τελικά δε χρησιμοποιούν καμία από τις δύο αυτές αρχιτεκτονικές, αλλά ένα κράμα τους. Τα συστήματα που επικρατούν αποτελούνται από πολλά SMPs, τα οποία είναι διασυνδεδεμένα μέσω κάποιου δικτύου. Έτσι, κάθε κόμβος αποτελείται από ένα πολυεπεξεργαστικό σύστημα βασισμένο σε αρχιτεκτονική κοινής μνήμης και όλοι οι κόμβοι επικοινωνούν μεταξύ τους μέσω δικτύου διασύνδεσης βασισμένο σε αρχιτεκτονική κατανεμημένης μνήμης.

Επίσης, ως υβριδική προτείνεται και η αρχιτεκτονική κατανεμημένης-κοινής μνήμης, όπου οι ιδιωτικές μνήμες κάθε επεξεργαστή μπορούν να διευθυνσιοδοτηθούν ώστε να αντιλαμβάνονται από τους επεξεργαστές ως ένας ενιαίος χώρος διευθύνσεων. Αυτή η πρακτική είναι μια εξέλιξη των συστημάτων κατανεμημένης μνήμης και έχει το πλεονέκτημα ότι προσφέρει αυτόματα το μηχανισμό επικοινωνίας των επεξεργαστών, που σε άλλες περιπτώσεις καλείται να λύσει ο προγραμματιστής.

1.3 Μοντέλα και Εργαλεία Παράλληλου Προγραμματισμού

Σε αντιστοιχία με τις αρχιτεκτονικές, υπάρχουν και δυο κύρια μοντέλα παράλληλου προγραμματισμού: του κοινού χώρου διευθύνσεων και της ανταλλαγής μηνυμάτων. Αυτά τα μοντέλα είναι βασισμένα και προσαρμόζονται πάνω στις αρχιτεκτονικές που περιγράψαμε προηγουμένως, αλλά δε σημαίνει ότι εφαρμόζονται κατ' ανάγκη σε αυτές. Απλά αποτελούν μέθοδοι ώστε ο προγραμματιστής να παραλληλοποιεί εφαρμογές και μπορούν -άλλοτε με περισσότερη και άλλοτε με λιγότερη δυσκολία- να προσαρμοστούν σε οποιαδήποτε αρχιτεκτονική.

Το μοντέλο *κοινού χώρου διευθύνσεων (shared memory model)* αποτελεί ένα γενικό προγραμματιστικό μοντέλο, όπου θεωρεί ότι όλα τα δεδομένα είναι προσπελάσιμα από κάθε επεξεργαστή. Αντιστοιχεί στην αρχιτεκτονική κοινκής μνήμης. Αυτό το μοντέλο παρέχει ευκολία στον προγραμματιστή, καθώς κάθε αλλαγή σε δεδομένα γίνεται άμεσα ορατή από κάθε επεξεργαστή, αφού πραγματοποιείται στην κοινή μνήμη μέσω συναρτήσεων φόρτωσης και αποθήκευσης. Αυτό απαιτεί βέβαια να υπάρχουν μέθοδοι συγχρονισμού, με τη χρήση κλειδωμάτων, σημαφόρων ή άλλων μεθόδων. Στην πραγματικότητα, η διαδικασία του συγχρονισμού των επεξεργαστών ώστε να μην αλλοιωθεί το τελικό αποτέλεσμα από την παράλληλη εκτέλεση είναι το βασικό και δύσκολο πρόβλημα που αντιμετωπίζει ο προγραμματιστής σε αυτό το μοντέλο. Η αποδοτικότητα αυτού του μοντέλου είναι συνήθως πολύ υψηλή.

Το μοντέλο *ανταλλαγής μηνυμάτων (distributed memory model)* είναι το αντίστοιχο της αρχιτεκτονικής κατανεμημένης μνήμης. Κάθε επεξεργαστής χειρίζεται ανεξάρτητα από τους άλλους το δικό του set δεδομένων και η μεταξύ τους επικοινωνία γίνεται μέσω μηνυμάτων που ανταλλάσσουν. Αυτό το μοντέλο παρέχει μια αρκετά απαιτητική δουλειά στον προγραμματιστή και η υλοποίησή του δεν είναι εύκολη υπόθεση.

Τέλος, υπάρχει και το υβριδικό μοντέλο προγραμματισμού, που αποτελεί ένα κράμα των παραπάνω μοντέλων, κατ' αντιστοιχία με τις υβριδικές αρχιτεκτονικές.

Πάνω σε αυτά τα μοντέλα είναι βασισμένα και τα εργαλεία που έχουν αναπτυχθεί για την παραλληλοποίηση προγραμμάτων. Παρακάτω γίνεται μια συνοπτική παρουσίαση στα σημαντικότερα από αυτά.

➔ *OpenMP*

Η πρώτη έκδοση OpenMP 1.0 [9] παρουσιάστηκε το 1997 για τη γλώσσα Fortan και ένα χρόνο αργότερα για τη C/C++ και από τότε συνεχώς ανανεώνεται. Αποτελείται από οδηγίες προς το μεταγλωττιστή, βιβλιοθήκη με ρουτίνες και μεταβλητές που επηρεάζουν τη συμπεριφορά του προγράμματος στο χρόνο εκτέλεσης. Είναι ένα εργαλείο που παρέχει ποικιλία στο προγραμματιστικό στυλ, προσφέροντας ταυτόχρονα απλότητα, που οφείλεται στο ότι ο κώδικας του σειριακού προγράμματος δεν χρειάζεται δομικές αλλαγές, αλλά την προσθήκη μερικών επιπλέον key words για την παραλληλοποίηση. Η ευκολία του έγκειται στο γεγονός ότι πολλά ζητήματα (επικοινωνία των thread, διαμοιρασμός δεδομένων) λύνονται αυτόματα μέσω του μεταγλωττιστή. Από την άλλη, η επίδοσή του εξαρτάται ιδιαίτερα από τις προδιαγραφές του συστήματος, καθώς η κλιμάκωσή του ρυθμίζεται από τη διαθέσιμη μνήμη της αρχιτεκτονικής, αλλά και από το είδος της (κοινής ή κατανεμημένης μνήμης).

Ο τρόπος που παραλληλοποιεί εργασίες το OpenMP είναι πολυνηματικός, δημιουργώντας ένα master thread (κύριο νήμα), το οποίο με τη σειρά του φτιάχνει αντίγραφα του εαυτού του (νήματα δούλοι) στα οποία διαμοιράζεται μια εργασία προς εκτέλεση. Τα threads δρομολογούνται σε διαφορετικούς επεξεργαστές μέσω του runtime συστήματος (RTS), με αποτέλεσμα να τρέχουν ταυτόχρονα. Πλέον το OpenMP δίνει τη δυνατότητα ορισμού tasks στον προγραμματιστή, ανεξαρτήτων δηλαδή μονάδων εργασίας, των οποίων η δρομολόγηση είναι ευθύνη του προγραμματιστή. Υπάρχουν διάφορες προτεινόμενες τεχνικές για τη δρομολόγηση [10], αλλά εν γένει με αυτήν την τεχνική βελτιώνεται η επίδοση των συστημάτων, καθώς η δρομολόγηση είναι δυναμική και καθορίζεται από τον προγραμματιστή για τις ανάγκες της κάθε εφαρμογής. Η ύπαρξη αυτού του νέου χαρακτηριστικού του OpenMP είναι ιδιαίτερα σημαντική για εμάς, διότι δείχνει ότι το focus στρέφεται πλέον σε tasks που δρομολογούνται από τον ίδιο τον προγραμματιστή δυναμικά, ό,τι δηλαδή πραγματεύεται και η συγκεκριμένη εργασία.

➔ *MPI*

Το MPI (Message Passing Interface) [11] είναι ένα πρότυπο ανταλλαγής μηνυμάτων συστημάτων κατανεμημένης μνήμης. Δεν είναι γλώσσα, ούτε έχει συγκεκριμένη υλοποίηση, αλλά οι λειτουργίες που παρέχει εκφράζονται ως συναρτήσεις, μέθοδοι ή ρουτίνες για την εκάστοτε γλώσσα στην οποία υλοποιείται. Είναι δομημένο σε στρώματα, με το υψηλότερο να είναι η αλληλεπίδραση με τον προγραμματιστή και το χαμηλότερο η επικοινωνία με το δίκτυο διασύνδεσης.

Ο στόχος του σχεδιασμού του MPI είναι η υψηλή απόδοση, η δυνατότητα κλιμάκωσης σε δίκτυο πολλών επεξεργαστών και η φορητότητα. Γι' αυτό το λόγο έχει αναχθεί στο βασικό τρόπο προγραμματισμού υπερυπολογιστικών συστημάτων και επικοινωνίας διεργασιών σε πολυπύρηνα συστήματα ή clusters.

Η φιλοσοφία του σχεδιασμού είναι ο χωρισμός του προγράμματος σε διεργασίες, οι οποίες εκτελούν το ίδιο πρόγραμμα. Για βέλτιστη απόδοση κάθε διεργασία πρέπει να αντιστοιχίζεται σε ένα δικό της επεξεργαστή. Κάθε διεργασία αναλαμβάνει να εκτελέσει τις εντολές σε ένα υποσύνολο δεδομένων και έπειτα να επιστρέψει τα αποτελέσματά της.

Ο προγραμματιστής είναι επιφορτισμένος με τον βέλτιστο διαμοιρασμό των δεδομένων και την επικοινωνία μεταξύ των διεργασιών μέσω μηνυμάτων, με στόχο την μεγιστοποίηση της παραλληλίας και την ελαχιστοποίηση των δεδομένων επικοινωνίας. Παρόλο που το MPI παρέχει χρήσιμες ρουτίνες, η δουλειά του προγραμματιστή είναι αρκετά δύσκολη και απαιτητική. Έχει, όμως, ως αποτέλεσμα τη βέλτιστη αξιοποίηση του συστήματος.

➔ *CUDA*

Η CUDA (Compute Unified Device Architecture) [12] είναι μια πλατφόρμα παράλληλου προγραμματισμού που δημιουργήθηκε από την NVIDIA, η οποία βασίζεται σε GPUs (Graphics Processing Unit). Η βασική διαφορά μιας CPU και μιας GPU είναι ότι η δεύτερη έχει στόχο να εκτελεί μόνο αριθμητικές πράξεις, χωρίς να έχει την ποικιλία εντολών που παρέχει μια CPU. Γι' αυτό και στις GPUs ανατίθενται δύσκολα υπολογιστικά φορτία, αλλά δεν έχουν δυνατότητα να κάνουν επιλογές ή πιο πολύπλοκες διαδικασίες.

Η CUDA χρησιμοποιείται ως επέκταση της C, εισάγοντας επιπλέον τύπους και προσδιοριστές και βασίζεται στη συνεργασία της CPU με τη GPU, καθώς η πρώτη αναθέτει υπολογιστικό φορτία στη δεύτερη, η οποία μπορεί να το εκτελέσει πολύ γρηγορότερα. Έτσι, ο προγραμματιστής επιλέγει ποια κομμάτια κώδικα θα σταλούν στη GPU προς εκτέλεση. Η ιδέα είναι η παραλληλοποίηση δεδομένων, δηλαδή, αντίστοιχα με το MPI, η εκτέλεση ίδιων εντολών πάνω σε διαφορετικά δεδομένα ταυτόχρονα. Για να γίνει αυτό δημιουργούνται χιλιάδες threads, τα οποία ομαδοποιούνται (wraps, blocks, grids) και δρομολογούνται από τους επεξεργαστές ροών που διαθέτει η GPU με μηδενικό κόστος. Επιπλέον, στην αύξηση της απόδοσης βοηθάει η δυνατότητα χρησιμοποίησης διαφορετικών επιπέδων μνήμης από τον προγραμματιστή (καθολική, μνήμη σταθερών, texture, μοιραζόμενη).

Γενικά, η CUDA, είναι ένα εργαλείο ιδιαίτερα χρήσιμο, αρκετά απλό στην κατανόηση και στη χρησιμοποίησή του. Παρόλ' αυτά, για τη μεγιστοποίηση της απόδοσης χρειάζεται η αλγοριθμική επεξεργασία των προγραμμάτων για τη βέλτιστη εκμετάλλευση της μνήμης.

➔ *PGAS*

Τα PGAS (Partitioned Global Address Space) [13] είναι γλώσσες παράλληλου προγραμματισμού. Στόχος είναι να εκμεταλλευτούν τα πλεονεκτήματα των δυο βασικών μοντέλων: της κοινής και της κατανεμημένης μνήμης. Συνδυάζουν, δηλαδή την προγραμματιστική ευκολία του κοινού χώρου διευθύνσεων και την υψηλή επίδοση με τη δυνατότητα κλιμάκωσης του μοντέλου ανταλλαγής μηνυμάτων. Χρησιμοποιούν τον κοινό χώρο διευθύνσεων, κατακερματίζοντάς και μοιράζοντάς τον στην κάθε διεργασία, ώστε να

είναι ιδιωτικός. Με αυτόν τον τρόπο αξιοποιείται η χωρική τοπικότητα των μεταβλητών χωρίς να αναιρείται η έννοια των κοινών μεταβλητών που είναι ορατές από κάθε διαδικασία. Για την αξιοποίηση της απόδοσής τους απαιτείται ισχυρό run-time σύστημα.

Ως τώρα παρουσιάστηκαν συνοπτικά ορισμένα από τα πιο βασικά εργαλεία που χρησιμοποιούνται σήμερα και επιτρέπουν παράλληλο προγραμματισμό. Σκόπιμα δεν έγινε αναφορά σε δυο επίσης σπουδαία εργαλεία, τη γλώσσα Cilk και τα posix threads, τα οποία θα αναπτυχθούν εκτενώς σε αυτήν την εργασία, μιας και η μελέτη μας βασίστηκε πάνω στη φιλοσοφία τους.

1.4 Εισαγωγή στη γλώσσα Cilk

Η Cilk [14] είναι μια γλώσσα παράλληλου πολυνηματικού προγραμματισμού βασισμένη στην ANSI C. Επεκτείνει τη C, χρησιμοποιώντας επιπλέον λέξεις κλειδιά. Η φιλοσοφία της είναι ότι ο προγραμματιστής ασχολείται με τη δόμηση της εφαρμογής για να εκμεταλλευτεί εργασίες που μπορούν να τρέξουν παράλληλα, αξιοποιώντας την τοπικότητα των δεδομένων, ενώ το runtime σύστημα, που υλοποιεί η ίδια η γλώσσα, αναλαμβάνει τη χρονοδρομολόγηση των εργασιών ώστε να αξιοποιηθεί αποδοτικά το εκάστοτε σύστημα. Επιπλέον, χαρακτηρίζεται ως αλγοριθμική γλώσσα, αφού το runtime σύστημά της εγγυάται σταθερή απόδοση. Όσον αφορά τη μνήμη, χρησιμοποιείται κυρίως σε συστήματα αρχιτεκτονικής κοινής μνήμης, όπως θα δούμε παρακάτω.

Η Cilk αναπτύσσεται από το 1994 στο Εργαστήριο Επιστήμης Υπολογιστών στο MIT. Η αρχική υλοποίησή της βασίστηκε στο πακέτο PCM/Threaded-C, το οποίο δημιουργούσε threads και τα χρονοδρομολογούσε μέσω ενός scheduler. Αργότερα, προστέθηκαν όλα τα χαρακτηριστικά της C, καθώς και κοινή μνήμη στο σύστημα, με αποτέλεσμα η Cilk να μπορεί να χρησιμοποιηθεί σε ευρύτερο πεδίο εφαρμογών. Η μεγάλη τομή έγινε όταν στην έκδοση Cilk-4 άλλαξε εκ βάθρων ο μεταγλωττιστής και το runtime σύστημα, ώστε οι αποφάσεις για τη χρονοδρομολόγηση των εργασιών να λαμβάνονται απευθείας από το μεταγλωττιστή. Αυτή η τομή αύξησε κατακόρυφα την επίδοση της γλώσσας κάνοντας τη δημιουργία ενός παράλληλου thread να έχει κόστος μόλις τρεις φορές το κόστος που έχει η δημιουργία μιας τυπικής διαδικασίας στη C. Στόχος των τελευταίων εκδόσεων ήταν η Cilk να γίνει ένα εργαλείο που να χρησιμοποιείται ευρέως σε εφαρμογές και από προγραμματιστές που δεν ειδικεύονται αναγκαστικά στον τομέα του παράλληλου προγραμματισμού. Πλέον η τεχνολογία της Cilk έχει αγοραστεί από την Intel, η οποία προσθέτοντας κάποιες δομές παραλληλισμού δεδομένων, την έχει κάνει εμπορικά διαθέσιμη.

1.4.1 Statements της Cilk

Η Cilk έχει ως βασικό στόχο την απλότητα, γεγονός που υποδηλώνει και το όνομά της, που προέρχεται από τον παραλληλισμό με τη λέξη "silk" (μετάξι). Αντίθετα με τα περισσότερα εργαλεία που έχουμε ως τώρα αναφέρει, η Cilk αναλαμβάνει από μόνη της τον τρόπο που θα τρέξουν οι παράλληλες εργασίες, πώς θα επικοινωνούν μεταξύ τους, πώς θα χρονοδρομολογηθούν. Όλα αυτά τα αναλαμβάνει το runtime σύστημά της. Η δουλειά που καλείται να κάνει κάποιος που γράφει μια εφαρμογή σε Cilk είναι κατά βάση αλγοριθμική, δηλαδή να βρει ποια κομμάτια του κώδικα μπορούν να τρέξουν ανεξάρτητα μεταξύ τους παράλληλα.

Η δομή της Cilk στηρίζεται στην απλότητα και την ευκολία. Αποτελείται από τη γλώσσα C με την προσθήκη μόλις τριών λέξεων-κλειδιά που υποδεικνύουν τον παραλληλισμό και το συγχρονισμό. Αυτές οι λέξεις-κλειδιά είναι: *cilk*, *spawn* και *sync*. Όταν ένα πρόγραμμα σε Cilk τρέχει σε έναν επεξεργαστή έχει την ίδια σημασιολογία με ένα το αντίστοιχο σε C. Αυτό προκύπτει με την αφαίρεση των επιπλέον λέξεων-κλειδιά της Cilk και το πρόγραμμα που προκύπτει ονομάζεται C elision.

Μια συνάρτηση προσδιορίζεται ως διεργασία Cilk προσθέτοντας μπροστά της τη λέξη *cilk*. Έχει λίστα μεταβλητών και κυρίως σώμα, δηλαδή ακριβώς ίδια δομή με μια συνάρτηση C, Μια διεργασία Cilk μπορεί να δημιουργήσει ανεξάρτητες υποδιεργασίες, με τη δυνατότητα να τρέχουν παράλληλα και να συγχρονίζονται όταν επιστρέφουν. Η κυρίως εργασία γίνεται σειριακά. Η παραλληλοποίηση κρύβεται στη λέξη *spawn*, η οποία εισάγεται πριν από την κλήση μιας συνάρτησης. Όταν καλείται μια συνάρτηση δημιουργείται ένα παιδί και ο έλεγχος μεταφέρεται στην εκτέλεση του παιδιού, όπως και στη C. Η σημασιολογία του *spawn* διαφέρει από εκείνη της κλήσης μιας συνάρτησης σε C μόνο στο γεγονός ότι ο πατέρας έχει τη δυνατότητα να συνεχίσει να εκτελείται ταυτόχρονα με το παιδί που δημιούργησε, αντί να περιμένει να τερματίσει εκείνο. Με αυτόν τον τρόπο ο πατέρας μπορεί να δημιουργεί συνεχώς παιδιά, μοιράζοντας την εργασία, δημιουργώντας υψηλό βαθμό παραλληλίας.

Για να εξασφαλιστεί η ορθή εκτέλεση του προγράμματος, μια διεργασία Cilk δεν μπορεί να χρησιμοποιήσει με ασφάλεια τις τιμές που επιστρέφουν τα παιδιά, μέχρι να εκτελεστεί η εντολή *sync*. Η λέξη *sync* τοποθετεί ένα "φράγμα" στον πατέρα, ο οποίος "παγώνει" μέχρι να τερματίσουν όλα τα παιδιά της. Το φράγμα αυτό είναι τοπικό με την έννοια ότι ο πατέρας περιμένει μόνο τα παιδιά του να ολοκληρώσουν την εκτέλεσή τους κι όχι όλες τις διεργασίες που εκτελούνται παράλληλα. Όταν επιστρέψουν όλα τα παιδιά, ο πατέρας μπορεί να συνεχίσει την εκτέλεσή του από εκείνο το σημείο. Ο λόγος που υπάρχει ο συγχρονισμός είναι γιατί κάποιες μεταβλητές δεν μπορούν αν χρησιμοποιηθούν αν δεν έχουν πρώτα υπολογιστεί, διασφαλίζοντας ότι η παραλληλοποίηση δεν θα επηρεάσει την ορθή λειτουργία του προγράμματος.

Αυτά είναι τα τρία βασικά και απλά συστατικά τα οποία παρέχει η Cilk για παραλληλία. Είναι εμφανές ότι αυτή η απλότητα οδηγεί σε ευκολία προγραμματισμού,

αφού ο χρήστης δεν ασχολείται με τίποτα άλλο, παρά με τον τρόπο που θα κατακερματίσει το πρόγραμμα σε ανεξάρτητα κομμάτια που μπορούν εκτελεστούν ταυτόχρονα. Πέραν αυτών, η Cilk παρέχει και κάποια πιο εξελιγμένα χαρακτηριστικά, που βοηθούν σε ορισμένες περιπτώσεις στην αύξηση της επίδοσης. Είναι οι λέξεις-κλειδιά: *inlet* και *abort.*

Το *inlet* είναι ένας τρόπος επιστροφής του αποτελέσματος του παιδιού στον πατέρα με έναν πιο περίπλοκο τρόπο απ' ότι με την αποθήκευση της τιμής σε μια μεταβλητή του πατέρα. Πρόκειται για μια συνάρτηση C φωλιασμένη μέσα σε μια διεργασία Cilk. Ενώ το *spawn* είναι ξεχωριστή εντολή, το *inlet* επιτρέπει τη δημιουργία συνάρτησης που παίρνει ως όρισμα ένα *spawn.* Το παιδί δημιουργείται και εκτελείται κανονικά και όταν επιστρέφει καλείται η συνάρτηση που έχει οριστεί στο *inlet*. Η συνάρτηση χειρίζεται το αποτέλεσμα όπως επιθυμεί, χωρίς όμως να μπορεί να χρησιμοποιήσει *spawn* ή *sync*, μιας και δεν είναι διεργασία Cilk. Είναι συνάρτηση της C και εκτελείται ως ξεχωριστό thread. Επειδή υπάρχει περίπτωση οι ίδιες μεταβλητές μιας διεργασίας να χρησιμοποιούνται ταυτόχρονα από ένα inlet (ή περισσότερα) και από τη διεργασία την ίδια, οι σχεδιαστές της Cilk έχουν μεριμνήσει και εγγυώνται ότι στην εκτέλεση όλα τα threads μιας διεργασίας εκτελούνται ατομικά με σεβασμό το ένα στο άλλο. Αυτό βέβαια δε σημαίνει το ίδιο και για threads διαφορετικών διεργασιών.

Ορισμένες φορές, μια διεργασία δημιουργεί παιδιά, των οποίων η εργασία τελικά δε χρειάζεται. Η Cilk επιτρέπει στον χρήστη να διακόψει την εργασία τους με τη χρήση του *abort.* Αυτό ισχύει μόνο για τα υπάρχοντα παιδιά κι όχι γι' αυτά που δεν έχουν δημιουργηθεί ακόμα. Αυτή η τεχνική είναι ιδιαίτερα χρήσιμη σε αλγορίθμους αναζήτησης που ψάχνουν ένα χώρο λύσεων παράλληλα. Μόλις βρουν την επιθυμητή λύση μπορούν να ακυρώσουν όλη την υπόλοιπη εργασία που έχει ανατεθεί σε άλλα threads. Μια συνηθισμένη τακτική είναι το *abort* να χρησιμοποιείται μέσα στον κώδικα του *inlet,* που χειρίζεται το αποτέλεσμα της επιστροφής του παιδιού. Μόλις εκτελεστεί το *abort*, όλα τα παιδιά του πατέρα τερματίζονται, όχι όμως αυτόματα. Η Cilk δεν εγγυάται για τις τιμές που θα επιστραφούν και ο χρήστης είναι υπεύθυνος να τις διαχειριστεί όπως επιθυμεί.

Μια κλασική εφαρμογή της χρήσης της Cilk είναι ο αναδρομικός αλγόριθμος υπολογισμός αριθμών fibonacci. Η συνάρτηση είναι η ακόλουθη:

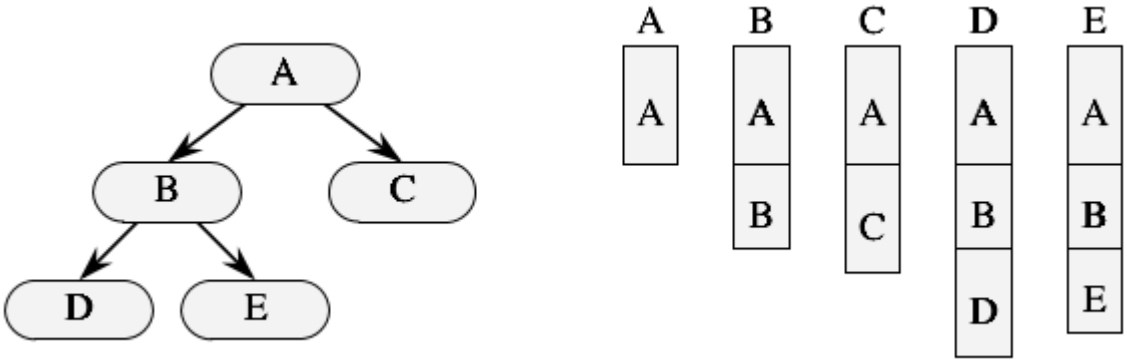
```
cilk int fib (int n) {
  if(n<2) return n;
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);

    sync;
    return x+y; }
    }
```

Παρατηρούμε ότι η συνάρτηση είναι διεργασία cilk, αφού δηλώνεται ως τέτοια. Όταν την καλούμε αναδρομικά, χρησιμοποιούμε το *spawn* ώστε η διεργασία cilk που δημιουργείται να μπορεί να εκτελεστεί παράλληλα με την αρχική, αλλά και με όλες τις υπόλοιπες. Τέλος, για να εξασφαλίσουμε την ορθότητα του προγράμματος φροντίζουμε μέσω του *sync* η διεργασία να επιστρέψει μόνο αφού έχουν τερματίσει και τα δυο της παιδιά.

1.4.2 Διαχείριση μνήμης στη Cilk

Η Cilk, όπως και η γλώσσα C, της οποίας είναι επέκταση παρέχει δυο ειδών μνήμες: μνήμη στοίβας (*stack)* και σωρού *(heap)* [15]. Στη στοίβα αποθηκεύονται όλες οι τοπικές μεταβλητές και χρησιμοποιείται για να περάσουν τα ορίσματα στις συναρτήσεις μαζί με τη διεύθυνση της εντολής στην οποία θα επιστρέψει ο έλεγχος μετά την ολοκλήρωση κάθε συνάρτησης. Η στοίβα μεγαλώνει προς τα κάτω όταν είναι να προστεθεί ένα καινούργιο πλαίσιο, λόγω κλήσης μιας νέας συνάρτησης. Ο σωρός είναι η μνήμη που δεσμεύεται δυναμικά κατά τη διάρκεια του προγράμματος από το runtime σύστημα. Μνήμη στο σωρό δεσμεύεται προς τα πάνω με χρήση συναρτήσεων *malloc()* και *calloc()* και αποδεσμεύεται χρησιμοποιώντας τη *free()*.

Η υλοποίηση του σωρού της Cilk είναι ακριβώς ίδια με εκείνη στη C. Όσον αφορά, όμως, τη στοίβα υπάρχουν ορισμένες διαφοροποιήσεις. Αυτό γίνεται γιατί η διατήρηση μιας γραμμικής στοίβας όπως στη C, είναι όχι μόνο μη αποδοτική, αλλά πολλές φορές και προβληματική [16]. Η Cilk χρησιμοποιεί μια στοίβα-κάκτο (cactus stack) [14], η δομή της οποίας φαίνεται στο Σχήμα 4. Αυτή η δενδρικής δομής στοίβα, που αναφέρεται και ως "spaghetti stack", αρχικά χρησιμοποιήθηκε σε περιβάλλοντα σειριακής εκτέλεσης, που υποστήριζαν μηχανισμούς ελέγχου μη συμβατούς με μια γραμμική στοίβα.



Σχήμα 4: Στοίβα-Κάκτος της Cilk

Από την οπτική γωνία μιας διεργασίας Cilk, η στοίβα-κάκτος συμπεριφέρεται όπως μια συνηθισμένη στοίβα. Η δέσμευση και ελευθέρωση μνήμης γίνεται με διαδικασίες push και pop από τη στοίβα και κάθε διεργασία αντιμετωπίζει το πλαίσιο της ως συνέχεια του

πατέρα της. Ως εδώ η δομή είναι η ίδια. Αλλάζει όταν πολλές διεργασίες εκτελούνται παράλληλα, οπότε καθεμία βλέπει τη στοίβα που αντιστοιχεί στην ιστορία της. Με αυτόν τον τρόπο προκύπτει η στοίβα-κάκτος.

Επιπλέον, αν κατατάσσαμε τη Cilk σε μοντέλα προγραμματισμού, θα λέγαμε ότι είναι κοινής μνήμης. Η Cilk επιτρέπει πολλές διεργασίες που τρέχουν παράλληλα να έχουν πρόσβαση σε καθολικές μεταβλητές, αλλά και το πέρασμα δεικτών ως ορίσματα σε συναρτήσεις. Μπορεί, παρόλα αυτά, η ενημέρωση των κοινών θέσεων μνήμης να οδηγήσει σε μη ντετερμινιστικές ανωμαλίες. Πολλές φορές αυτά τα προβλήματα προκύπτουν ως προγραμματιστικά λάθη, τα οποία αποφεύγονται με το να μην επιτρέπεται σε ένα thread να τροποποιήσει μια μεταβλητή την οποία μπορεί να διαβάσει ή να γράψει ένα άλλο thread που εκτελείται παράλληλα. Αν αυτός ο κανόνας τηρηθεί, το runtime σύστημα της Cilk εγγυάται ότι ανεξάρτητα από τη χρονοδρομολόγηση των threads, το πρόγραμμα θα εκτελεστεί ντετερμινιστικά. Ο μη ντετερμινισμός μπορεί να χρησιμοποιηθεί σε ορισμένες εφαρμογές για την επίτευξη μεγαλύτερου βαθμού παραλληλισμού, αλλά σε τέτοιες περιπτώσεις πρέπει να ληφθεί υπόψη το μοντέλο συνέπειας της μνήμης της μηχανής στην οποία εκτελείται η εφαρμογή.

Αυτή ήταν μια γενική εισαγωγή για τη Cilk, η οποία θα μας απασχολήσει αρκετά σε αυτήν την εργασία, λόγω του τρόπου λειτουργίας του χρονοδρομολογητή της και του ισχυρού θεωρητικού μοντέλου πάνω στο οποίο είναι βασισμένη η λειτουργία του.

ΚΕΦΑΛΑΙΟ 2: MULTITHREADING-ΠΟΛΥΝΗΜΑΤΙΣΜΟΣ

2.1 Διεργασίες

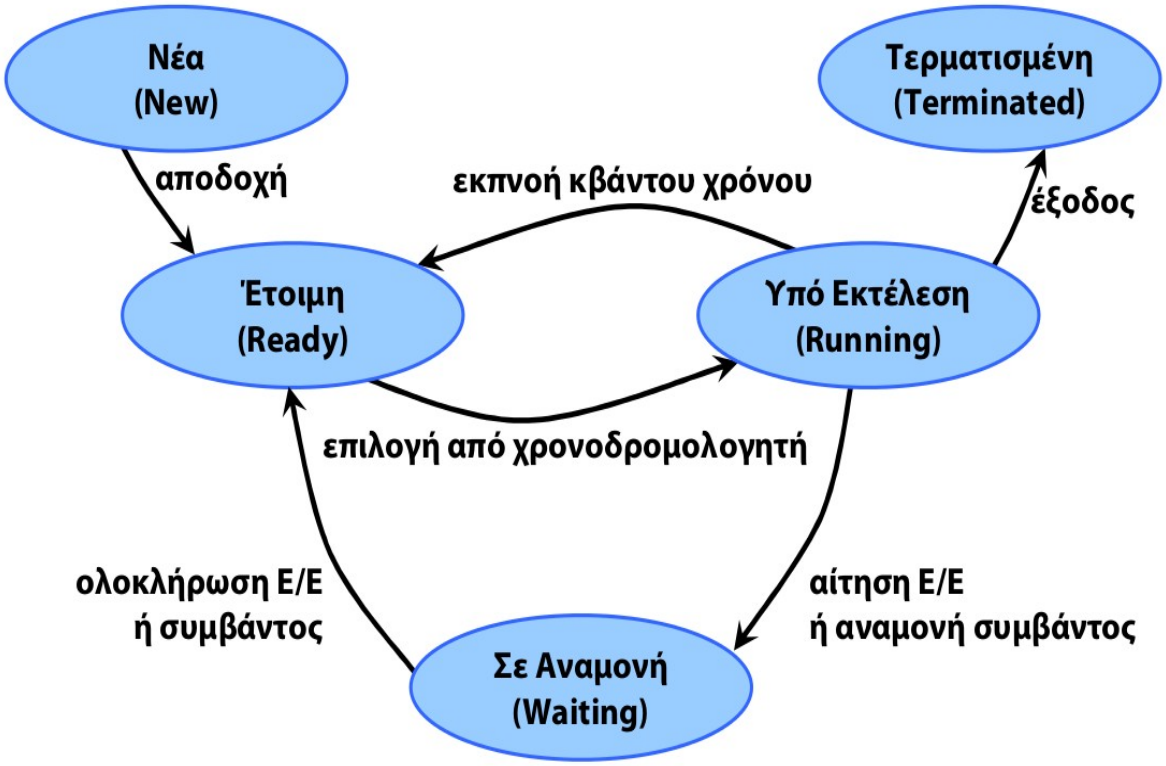
Μια διεργασία είναι ένα στιγμιότυπο ενός προγράμματος που εκτελείται. Ενώ το πρόγραμμα είναι απλώς μια ακολουθία εντολών, η διεργασία είναι η πραγματική εκτέλεση αυτών των εντολών. Ένα πρόγραμμα μπορεί να απαρτίζεται από πολλές διεργασίες, η καθεμία από τις οποίες έχει δική της ξεχωριστή κατάσταση και ξεχωριστή ταυτότητα. Κάθε διεργασία απαρτίζεται από τον εκτελέσιμο κώδικα του προγράμματος, από ιδιωτικό, απομονωμένο χώρο μνήμης (κάτι που επιτυγχάνεται με εικονική μνήμη), από πόρους του συστήματος, όπως ανοιχτά αρχεία και διαδικτυακές συνδέσεις και από την κατάσταση του επεξεργαστή, όπως τις τιμές των καταχωρητών ή του μετρητή προγράμματος.

Σημαντικό στοιχείο των διεργασιών είναι η ανεξαρτησία τους. Καθεμία εκτελείται απομονωμένη από τις άλλες, με το λειτουργικό σύστημα να φροντίζει να έχουν τη ψευδαίσθηση ότι είναι οι μόνες που εκτελούνται στη μηχανή, ακόμη κι όταν εκτελούνται ταυτόχρονα με άλλες. Η ταυτόχρονη εκτέλεσή τους, δε σημαίνει ότι αναγκαστικά εκτελούνται παράλληλα, αλλά ότι το λειτουργικό σύστημα φροντίζει να κατανέμει σε αυτές υπολογιστικό χρόνο. Έτσι, οι διεργασίες εναλλάσσονται συνεχώς σε έναν επεξεργαστή, εκτελώντας κάθε φορά ένα μέρος των εντολών τους, μέχρι να ολοκληρωθούν. Επειδή αυτή η εναλλαγή είναι πολύ γρήγορη για τον άνθρωπο, δημιουργείται η ψευδαίσθηση ότι οι διεργασίες εκτελούνται ταυτόχρονα. Ο τρόπος που καταμερίζεται ο χρόνος σε κάθε διεργασία καθορίζεται από τον αλγόριθμο χρονοδρομολόγησης, που υλοποιείται από το χρονοδρομολογητή.

Δεδομένου ότι οι επεξεργαστές λειτουργούν σε πολύ υψηλές συχνότητες και είναι πολύ πιο γρήγοροι από τις μνήμες, βασικός στόχος είναι να εκμεταλλευτούμε την ταχύτητά τους. Αυτό γίνεται με τη χρονοδρομολόγηση των διεργασιών, ώστε κατά τη διάρκεια που μια διεργασία περιμένει κάποια δεδομένα από τη μνήμη ή να επικοινωνήσει με μια συσκευή εισόδου-εξόδου, ο επεξεργαστής μπορεί να εκτελέσει μια άλλη. Ο χρονοδρομολογητής είναι ένα πρόγραμμα (ή και μια συνάρτηση), που επιλέγει ποια διεργασία θα εκτελείται στον επεξεργαστή και αναλαμβάνει το context switch μεταξύ των διεργασιών. Context switch είναι η διαδικασία αποθήκευσης και επανάκλησης μιας κατάστασης της CPU, ώστε η εκτέλεση να συνεχιστεί από εκεί που είχε σταματήσει παλαιότερα. Παρακάτω εξετάζουμε τον τρόπο λειτουργίας ενός χρονοδρομολογητή.

2.2 Αλγόριθμοι χρονοδρομολόγησης διεργασιών σε έναν επεξεργαστή

Στόχος του χρονοδρομολογητή είναι η αξιοποίηση της κεντρικής μονάδας επεξεργασίας στο μεγαλύτερο δυνατό βαθμό [17]. Πολλές φορές προκύπτουν καθυστερήσεις σε εκτέλεση εντολών όταν η διεργασία αναμένει κάποιους πόρους ή θέλει να επικοινωνήσει με μια μονάδα εισόδου-εξόδου, που αποκρίνονται πολύ αργά σε σχέση με την ταχύτητα του επεξεργαστή. Αυτός ο χαμένος χρόνος καλύπτεται από το χρονοδρομολογητή με την αλλαγή της υπό εκτέλεσης διεργασίας. Κάθε διεργασία διακρίνεται ανάλογα με την κατάσταση της (νέα, έτοιμη, υπό εκτέλεση, σε αναμονή, τερματισμένη). Ο υπολογιστικός χρόνος κατανέμεται σε αυτές που είναι έτοιμες προς εκτέλεση, οι οποίες μπαίνουν στην ουρά του χρονοδρομολογητή. Στο παρακάτω σχήμα φαίνεται πώς ο χρονοδρομολογητής χειρίζεται τις διεργασίες.



Σχήμα 5: Κύκλος ζωής διεργασίας

Ο τρόπος με τον οποίο επιλέγονται οι διεργασίες που θα εκτελεστούν καθορίζεται από τον αλγόριθμο χρονοδρομολόγησης. Στόχοι του αλγορίθμου είναι η καλύτερη απόδοση του υπολογιστικού συστήματος και η μέγιστη ταχύτητα επιλογής από τον ίδιο τον αλγόριθμο. Ας μην ξεχνάμε ότι και ο χρονοδρομολογητής είναι ένα πρόγραμμα το οποίο απασχολεί τον

επεξεργαστή και επιθυμούμε αυτός ο χρόνος απασχόλησης να είναι όσο δυνατόν πιο μικρός.

Ένας αλγόριθμος επιλέγεται με βάση κάποια κριτήρια. Αυτά είναι:

1. Ο βαθμός χρησιμοποίησής της CPU. Είναι ο λόγος του χρόνου που απασχολείται η CPU προς το συνολικό χρόνο που λειτουργεί.
2. Η ρυθμαπόδοση, το πλήθος δηλαδή των διεργασιών που ολοκληρώνονται στη μονάδα του χρόνου.
3. Ο χρόνος ανακύκλωσης. Είναι ο χρόνος για την ολοκλήρωση μιας διεργασίας από τη στιγμή που μεταβαίνει στην κατάσταση έτοιμη μέχρι να μεταβεί στην κατάσταση τερματισμένη.
4. Ο χρόνος αναμονής. Είναι ο συνολικός χρόνος που η διεργασία βρίσκεται στην ουρά του χρονοδρομολογητή αναμένοντας την εκτέλεσή της.
5. Ο χρόνος απόκρισης. Είναι ο χρόνος από τη στιγμή που μια διεργασία μεταβαίνει στην κατάσταση έτοιμη μέχρι την πρώτη φορά που θα επιλεχθεί από το χρονοδρομολογητή προς εκτέλεση.

Ιδανικό σενάριο είναι να μεγιστοποιείται ο ρυθμός χρησιμοποίησης της CPU και η ρυθμαπόδοση και να ελαχιστοποιούνται οι χρόνοι ανακύκλωσης, αναμονής και απόκρισης, αλλά στην πράξη απαιτούνται συμβιβασμοί. Οι συμβιβασμοί αυτοί γίνονται κυρίως με κριτήριο το είδος της εφαρμογής και τι απαιτήσεις έχει. Για παράδειγμα, μια εφαρμογή live streaming είναι απαραίτητο να έχει μικρό χρόνο απόκρισης, ενώ σε ένα server βασικός στόχος είναι η υψηλή ρυθμαπόδοση.

Οι αλγόριθμοι χρονοδρομολόγησης διακρίνονται σε δυο μεγάλες κατηγορίες: στους συνεργατικούς (cooperative) και στους διακοπτούς (preemptive). Οι συνεργατικοί αλγόριθμοι αφήνουν τη διεργασία να εκτελείται μέχρι να ολοκληρωθεί ή μέχρι να βρεθεί σε κατάσταση αναμονής, ενώ οι διακοπτοί δίνουν ένα κομμάτι χρόνου στις διεργασίες μέχρι να τις διακόψουν ανεξάρτητα από το στάδιο εκτέλεσης που βρίσκονται. Ας εξετάσουμε τους πιο διαδεδομένους αλγορίθμους.

➔ *FCFS (First Come First Served)*

Πρόκειται για ένα συνεργατικό αλγόριθμο, ο οποίος εξυπηρετεί τις διεργασίες με σειρά άφιξης. Η υλοποίησή του είναι πολύ απλή, αφού η λίστα έτοιμων διεργασιών λειτουργεί ως μια σειρά αναμονής. Κάθε διεργασία που δημιουργείται ή που ήταν σε αναμονή και το αίτημά της ικανοποιήθηκε μεταβαίνει στο τέλος της ουράς και περιμένει τη σειρά της μέχρι να εκτελεστεί. Αυτός ο αλγόριθμος έχει το πλεονέκτημα της απλότητας, οπότε καταναλώνει ελάχιστο χρόνο στην εκτέλεσή του. Από την άλλη, η επίδοσή του δεν είναι υψηλή, καθώς οι χρόνοι απόκρισης και ανακύκλωσης είναι αρκετά μεγάλοι στην

περίπτωση που υπάρχουν πολλές διεργασίες που θέλουν να εξυπηρετηθούν. Δημιουργείται ακόμη το φαινόμενο της φάλαγγας, όταν μικρές σε διάρκεια διεργασίες περιμένουν να εξυπηρετηθούν πίσω από μεγάλες διεργασίες. Γενικά, η απόδοσή του δεν είναι σταθερή αλλά εξαρτάται από τη σειρά άφιξης των διεργασιών.

➔ *SJF (Shortest Job First)*

Ο αλγόριθμος εξυπηρετεί τις διεργασίες δίνοντας προτεραιότητα σε αυτές με τη μικρότερη διάρκεια. Κάθε διεργασία εισάγεται στην ουρά στο κατάλληλο σημείο ανάλογα με τη διάρκειά της. Με αυτόν τον τρόπο, ελαχιστοποιείται ο χρόνος αναμονής των διεργασιών. Μάλιστα, κανείς άλλος αλγόριθμος δεν μπορεί να επιτύχει καλύτερο χρόνο αναμονής. Το πρόβλημα του αλγορίθμου είναι ο τρόπος υλοποίησής του, καθώς η χρονική διάρκεια μιας διεργασίας δεν είναι γνωστή εκ των προτερών. Μπορεί μόνο να γίνει κάποια εκτίμηση με βάση στατιστικά στοιχεία από προηγούμενες διεργασίες, κάτι το οποίο, όμως, έχει υπολογιστικό κόστος για τον αλγόριθμο. Βασικό πρόβλημα του αλγορίθμου είναι η λιμοκτονία: το φαινόμενο μια μεγάλη σε διάρκεια διεργασία να αναμένει πολύ χρόνο μέχρι να εκτελεστεί διότι προηγούνται συνεχώς διεργασίες μικρότερης διάρκειας.

Ο συγκεκριμένος αλγόριθμος έχει δυο υλοποιήσεις: ως συνεργατικός και ως διακοπτός. Η διαφορά τους είναι ότι ο διακοπτός θα διακόψει τη διεργασία που εκτελείται όταν έρθει κάποια άλλη που έχει μικρότερο χρόνο εκτέλεσης και θα την τοποθετήσει πίσω στην ουρά των έτοιμων διεργασιών, ενώ ο συνεργατικός θα τοποθετήσει τη διεργασία στην κορυφή της ουράς αναμονής και θα περιμένει την ολοκλήρωση της ήδη υπό εκτέλεσης διεργασίας.

➔ *HRF (Highest Response Ratio First)*

Οι διεργασίες δρομολογούνται εδώ με βάση το χρόνο απόκρισής τους. Η προτεραιότητα καθορίζεται από το λόγο απόκρισης που είναι ο λόγος του χρόνου απόκρισης προς τη διάρκεια εκτέλεσης της διεργασίας. Δηλαδή:

$$\textit{Λόγος Απόκρισης} = \frac{\textit{Χρόνος Απόκρισης}}{\textit{Χρόνος Εκτέλεσης}} = \frac{\textit{Χρόνος Εκτέλεσης} + \textit{Χρόνος Αναμονής}}{\textit{Χρόνος Εκτέλεσης}}$$
$$= 1 + \frac{\text{Χρόνος Αναμονής}}{\text{Χρόνος Εκτέλεσης}}$$

Ο τύπος δείχνει ότι η προτεραιότητα των εντολών αυξάνεται όσο αυτές βρίσκονται στην ουρά. Αυτή η μέθοδος είναι γνωστή ως μέθοδος γήρανσης, αφού η προτεραιότητα μιας διεργασίας αυξάνει όσο περισσότερο βρίσκεται στην ουρά αναμονής. Είναι ο τρόπος με τον οποίο λύνεται το πρόβλημα λιμοκτονίας που αντιμετωπίζουν διεργασίες σε αλγορίθμους

στατικής προτεραιότητας, όπως ήταν ο SJF, όπου οι μεγάλες διεργασίες μπορεί να είχαν υψηλούς χρόνους αναμονής. Και αυτός ο αλγόριθμος καταναλώνει κάποιο υπολογιστικό χρόνο, αλλά σαφώς μικρότερο σε σχέση με τον προηγούμενο.

Και οι τρεις αλγόριθμοι που έχουμε περιγράψει ως τώρα ανήκουν στην ευρύτερη κατηγορία των αλγορίθμων προτεραιότητας, όπου σε κάθε διεργασία αντιστοιχίζεται μια τιμή, που δηλώνει την προτεραιότητα με την οποία θα εκτελεστεί στον επεξεργαστή. Οι δυο πρώτοι προσφέρουν στατική προτεραιότητα, ενώ ο τελευταίος δυναμική προτεραιότητα. Υπάρχουν και άλλοι αλγόριθμοι τέτοιου τύπου, όπως επίσης και αλγόριθμοι όπου ο χρήστης μπορεί να επιλέξει την προτεραιότητα που θα έχει κάθε διεργασία, ανάλογα με τις ανάγκες του.

➔ *Round Robin Scheduling*

Είναι ένας αλγόριθμος διακοπτός, ο οποίος δίνει σε κάθε διεργασία ένα κβάντο υπολογιστικού χρόνου. Όταν εκπνεύσει ο χρόνος που της έδωσε την επαναφέρει στο τέλος της ουράς ανεξάρτητα από το σημείο εκτέλεσης στο οποίο βρίσκεται. Η ουρά που διαθέτει είναι τύπου FIFO (First In First Out), οπότε η αρχαιότερη διεργασία που βρίσκεται στη σειρά είναι η επόμενη προς εκτέλεση. Βασική παράμετρος για την απόδοση αυτού του αλγορίθμου είναι ο χρόνος που δίνεται σε κάθε διεργασία για να εκτελεστεί. Αν αυτός ο χρόνος είναι πολύ μεγάλος, ο αλγόριθμος εκφυλίζεται στον FCFS, ενώ αν είναι πολύ μικρός, ο επεξεργαστής χρησιμοποιείται πολύ συχνά από τον ίδιο το χρονοδρομολογητή και πολύς υπολογιστικός χρόνος καταναλώνεται στα context switches. Συνήθως το κβάντο χρόνου είναι 10ms, χωρίς όμως αυτό να είναι παγιωμένο. Επειδή έχει σχέση και με το σύστημα στο οποίο εκτελείται ένας κανόνας υπολογισμού του που ακολουθείται είναι να είναι μεγαλύτερο σε διάρκεια από το 80% των χρόνων εκτέλεσης των διεργασιών. Βασικό πλεονέκτημα του αλγορίθμου είναι ότι επιτρέπει στις μικρές διεργασίες να ολοκληρώνονται σχετικά γρήγορα, διατηρώντας παράλληλα καλό χρόνο απόκρισης για όλες τις διεργασίες.

➔ *Multi Queue Scheduling*

Αποτελεί συνδυασμό των παραπάνω αλγορίθμων. Η ουρά με τις έτοιμες διεργασίες του χρονοδρομολογητή χωρίζεται σε πολλαπλές ουρές και καθεμία έχει το δικό της αλγόριθμο χρονοδρομολόγησης. Έτσι, οι διεργασίες χωρίζονται σε κλάσεις ανάλογα με τις ιδιότητες και τις ανάγκες τους, ώστε να χρονοδρομολογηθούν με τον αλγόριθμο που ταιριάζει καλύτερα σε αυτές. Έπειτα, οι ουρές χρονοδρομολογούνται με τη σειρά τους, ώστε κάθε φορά να διαλέγεται μια διεργασία προς εκτέλεση. Μια τροποποίηση σε αυτόν τον υβριδικό τρόπο χρονοδρομολόγησης είναι οι ουρές που επιτρέπουν ανατροφοδότηση. Αυτή η υλοποίηση επιτρέπει τη μετακίνηση διεργασιών ανάμεσα στις ουρές. Κριτήριο απόδοσης

είναι το πλήθος των ουρών, ο αλγόριθμος χρονοδρομολόγησης που καθεμία υλοποιεί και ο τρόπος που επιλέγεται η ουρά που θα καταλήξει μια νέα διεργασία.

2.3 Multithreading

Μολονότι οι διεργασίες προσφέρουν καθαρότητα και απλότητα στην εκτέλεση με την ανεξαρτησία τους, είναι αρκετά μεγάλες και δύσχρηστες. Αυτό έχει ως συνέπεια η εναλλαγή τους μέσω του χρονοδρομολογητή να απαιτεί αρκετό χρόνο. Επιπλέον, η μεταξύ τους επικοινωνία είναι αρκετά περίπλοκη και αργή. Χρειάστηκε, λοιπόν, η υλοποίηση μικρότερων μονάδων που να μπορούν να επικοινωνούν μεταξύ τους με χαμηλότερο κόστος και εν γένει να χειρίζονται πιο εύκολα. Έτσι προέκυψαν τα threads (νήματα).

Τα threads είναι χωριστές ροές εκτέλεσης μέσα στην ίδια διεργασία. Είναι η μικρότερη ακολουθία εντολών που μπορεί να διαχειριστεί και να δρομολογήσει ένα λειτουργικό σύστημα. Threads που βρίσκονται μέσα στην ίδια διεργασία μοιράζονται τους πόρους του λειτουργικού συστήματος, έχουν κοινή μνήμη, αρχεία και δικαιώματα, με αποτέλεσμα να έχουν άμεση πρόσβαση σε κοινά δεδομένα, αλλά μπορούν να εκτελούνται ανεξάρτητα το ένα από το άλλο. Τα threads έχουν πολλά πλεονεκτήματα σε σχέση με τις διεργασίες. Έχουν πολύ μικρότερο κόστος δημιουργίας και καταστροφής. Για να δημιουργηθεί μια διεργασία πρέπει να δεσμευτεί νέος χώρος στη μνήμη για τον κώδικα των εντολών που θα εκτελέσει και για τις μεταβλητές που θα χρησιμοποιήσει και αυτή η διαδικασία είναι αρκετά δαπανηρή. Αντίθετα, ένα thread χρησιμοποιεί τους ήδη υπάρχοντες πόρους που του παρέχει η διεργασία. Επιπλέον, η επικοινωνία μεταξύ των thread είναι αρκετά εύκολη, μιας και μοιράζονται την ίδια μνήμη. Για να επικοινωνήσουν διεργασίες μεταξύ τους επιβάλλεται η ανταλλαγή μηνυμάτων, αφού εξ ορισμού είναι ανεξάρτητες και απομονωμένες, κάτι που δεν είναι ιδιαίτερα εύκολο.

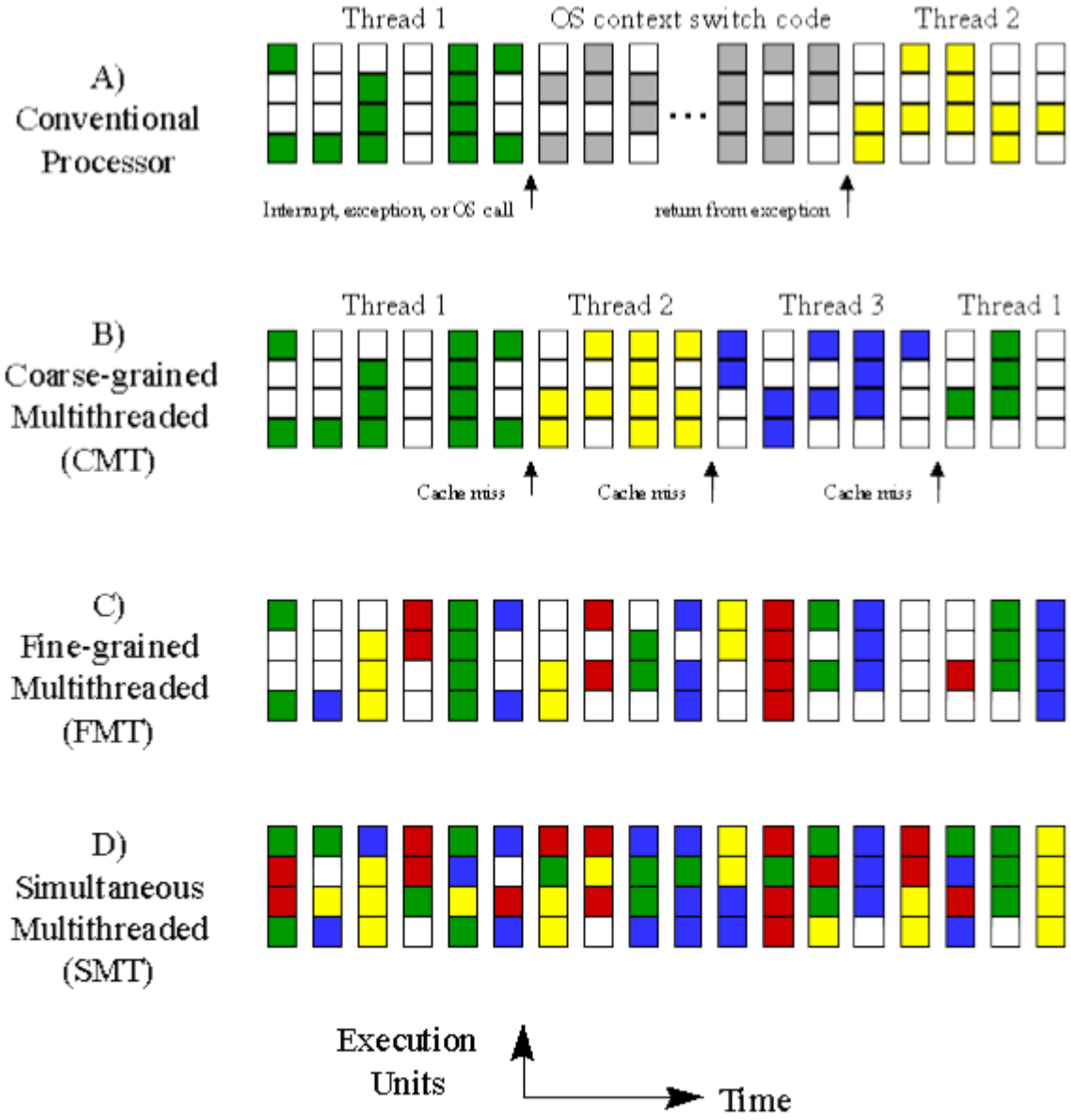
Η ιδέα του multithreading είναι η εκτέλεση πολλών threads σε έναν ή περισσότερους επεξεργαστές. Αυτό το είδους παραλληλισμού είναι το Thread Level Parallelism (TLP) στο οποίο αναφερθήκαμε στην εισαγωγή και αφορά την παράλληλη εκτέλεση threads σε επεξεργαστές. Όταν πρόκειται για έναν επεξεργαστή, η δρομολόγηση είναι χρονική και ακολουθεί κάποιον αλγόριθμο από αυτούς που περιγράψαμε για τις διεργασίες. Η αύξηση της επίδοσης σε σχέση με τις διεργασίες οφείλεται στο context switch, το οποίο είναι πολύ πιο γρήγορο για νήματα που ανήκουν στην ίδια διεργασία, αφού μοιράζονται όλα την ίδια μνήμη. Η διαδικασία στο context switch για τα thread είναι η αλλαγή της κατάστασης της στοίβας, των καταχωρητών και των δεδομένων που προσδιορίζει κάθε thread ξεχωριστά, σε αντίθεση με το context switch των διεργασιών που απαιτείται η αλλαγή των δεδομένων στην εικονική μνήμη. Όταν πρόκειται για πολλούς επεξεργαστές, η δρομολόγηση πέρα από χρονική είναι και χωρική, καθώς τα threads κατανέμονται βάσει κάποιου αλγορίθμου στους διαθέσιμους επεξεργαστές.

Αρχικά θα μελετήσουμε την τεχνική του multithreading σε έναν επεξεργαστή και πώς αυτό επιτυγχάνεται μέσω του hardare. Υπάρχουν δυο βασικές προσεγγίσεις για

multithreading: fine-grained και coarse-grained multithreading [18]. Η πρώτη προσέγγιση αφορά την αλλαγή threads που εκτελούνται στον επεξεργαστή ανά εντολή. Αγνοώντας τα threads που βρίσκονται σε αναμονή, ο επεξεργαστής τα εναλλάσσει με μία κυκλική μέθοδο σε κάθε κύκλο εντολής, γι' αυτό και συχνά στη βιβλιογραφία αυτή η μέθοδος αναφέρεται και ως interleaved (διαπλεκόμενο) multithreading. Είναι σημαντικό να δούμε ότι η εναλλαγή των thread δεν απαιτεί να επέμβει το λειτουργικό σύστημα, αλλά μπορεί να το κάνει και το ίδιο το harware. Κύριος στόχος αυτής της τακτικής είναι να απαλείψει όλες τις εξαρτήσεις δεδομένων που μπορεί να προκύψουν στο pipeline του επεξεργαστή. Η δυσκολία στην υλοποίηση είναι ότι η εναλλαγή των νημάτων πρέπει να γίνεται σε μηδενικό χρόνο. Έχει ως πλεονέκτημα ότι ο επεξεργαστής εκτελεί συνεχώς εντολές, παρακάμπτοντας τις πιθανές καθυστερήσεις, αφού όταν ένα thread είναι σταματημένο, εκτελούνται άλλα. Το βασικό του μειονέκτημα είναι ότι η συνεχής εναλλαγή μεταξύ των threads καθυστερεί ανεξάρτητα threads που μπορούν να εκτελεστούν χωρίς καθυστερήσεις.

Σε αντίθεση με το fine-grained multithreading, το coarse-grained είναι πιο απλή μέθοδος, η οποία επιτρέπει ένα thread να εκτελείται στον επεξεργαστή μέχρι να εκτελεστεί μια εντολή που θα προκαλέσει μεγάλη καθυστέρηση, όπως ένα διάβασμα από τη μνήμη. Αυτό σημαίνει ότι η εναλλαγή των thread δεν είναι τόσο συχνή, οπότε η ανάγκη μηδενικού χρόνου της αλλαγής δεν είναι τόσο επιτακτική και ο σχεδιασμός του hardware μπορεί να γίνει πιο απλός. Επίσης, οι εκτελέσεις ανεξάρτητων νημάτων είναι λιγότερο πιθανό να επιβραδυνθούν. Εντούτοις, αυτή η τακτική έχει ένα σημαντικό μειονέκτημα: δεν αντιμετωπίζει τις μικρές καθυστερήσεις των εντολών, όπως την εκτέλεση μιας διαίρεσης. Έτσι, σημαντικός παράγοντας είναι το όριο καθυστέρησης ενός thread ώστε να αποφασιστεί αν θα αλλαχθεί ή όχι. Το coarse-grained multithreading έχει καλύτερη απόδοση όταν οι καθυστερήσεις είναι μεγάλες, και μάλιστα μεγαλύτερες από το χρόνο επαναπλήρωσης του pipeline. Αν παρατηρήσουμε και τις δυο τεχνικές θα δούμε ότι έχουν πολλά κοινά με τις δυο βασικές κατηγοριοποιήσεις των αλγορίθμων χρονοδρομολόγησης: το fine-grained είναι αντίστοιχο με τους διακοπτούς αλγορίθμους, ενώ το coarse-grained με τους συνεργατικούς.

Πέραν αυτών, το hardware επιτρέπει και multithreading με την ταυτόχρονη εκτέλεση των threads. Υπάρχει μια τεχνική η οποία ονομάζεται SMT (Simultaneous Multithreading) και εφαρμόζεται σε υπερβαθμωτούς επεξεργαστές. Οι υπερβαθμωτοί επεξεργαστές επιτρέπουν την ταυτόχρονη εκκίνηση πολλών εντολών σε ένα κύκλο ρολογιού και η τεχνική SMT επιτρέπει να εκκινήσουν εντολές από διαφορετικά threads. Με αυτόν τον τρόπο μπορούν να γίνουν εκμεταλλεύσιμες όλες οι λειτουργικές μονάδες των σύγχρονων επεξεργαστών που μπορούν να λειτουργούν παράλληλα. Όπως και στις προηγούμενες τεχνικές, έτσι κι εδώ η καθυστέρηση ενός thread αντιμετωπίζεται με τη δρομολόγηση άλλων στη θέση του, όσο εκείνο περιμένει την ικανοποίηση του αιτήματός του. Παράδειγμα SMT είναι η τεχνολογία HyperThreading της Intel, η οποία αυξάνει την απόδοση των υπερβαθμωτών επεξεργαστών της κατά 30% [19].



Ένα παράδειγμα των τεσσάρων διαφορετικών τεχνικών για multithreading που περιγράψαμε δίνονται στο παρακάτω σχήμα (Σχήμα 6).

Σχήμα 6: Multithreading σε έναν επεξεργαστή με διαφορετικά επίπεδα παραλληλισμού

Ο επεξεργαστής είναι υπερβαθμωτός και μπορεί να εκκινήσει τέσσερις εντολές σε κάθε κύκλο ρολογιού. Αυτό δείχνουν τα κατακόρυφα τετράγωνα. Στην οριζόντια διεύθυνση δείχνονται οι διαφορετικοί κύκλοι ρολογιού του επεξεργαστή. Οι εντολές κάθε thread έχουν δικό τους χρώμα (πράσινο, μπλε, κίτρινο, κόκκινο), τα λευκά τετραγωνάκια δείχνουν ότι η συγκεκριμένη υποδοχή του επεξεργαστή είναι ανενεργή και τα γκρι είναι όταν στον επεξεργαστή εκτελείται κώδικας του λειτουργικού συστήματος. Στην πρώτη περίπτωση, όπου το multithreading γίνεται σε επίπεδο χρονοδρομολόγησης καταμερισμού χρόνου

(Σχήμα 6A), βλέπουμε ότι στην πρώτη μεγάλη καθυστέρηση του thread1, αναλαμβάνει ο χρονοδρομολογητής για να κάνει context switch, δεσμεύοντας έτσι χρόνο από τον επεξεργαστή που θα μπορούσε να είχε χρησιμοποιηθεί για τον κώδικα του χρήστη. Στην περίπτωση του coarse-grained multithreading (Σχήμα 6B), οι αλλαγές των threads γίνονται πάλι όταν υπάρξει καθυστέρηση σε ένα thread, μα εδώ δεν εμπλέκεται ο χρονοδρομολογητής, αλλά το ίδιο το hardware αναλαμβάνει να εκκινήσει ένα καινούργιο thread στη θέση του προηγούμενου. Αντίστοιχα, στο fine-grained multithreading (Σχήμα 6C), ο επεξεργαστής αλλάζει σε κάθε κύκλο ρολογιού τα threads εκ περιτροπής. Τέλος, το ταυτόχρονο multithreading (Σχήμα 6D) είναι αυτό που σε κάθε κύκλο ρολογιού μπορούμε να έχουμε ταυτόχρονη εκτέλεση των εντολών διαφορετικού thread. Αν θέλαμε να μετρήσουμε την απόδοση με βάση τα σχήματα, θα χρειαζόταν να δούμε σε ποια από όλες τις τεχνικές, ο επεξεργαστής έχει τα λιγότερα άσπρα τετραγωνάκια, δηλαδή δουλεύει περισσότερη ώρα. Είναι εμφανές ότι όταν έχω παραλληλισμό των threads, η επίδοση είναι η υψηλότερη που μπορούμε να πετύχουμε και ο επεξεργαστής αξιοποιείται με τον καλύτερο δυνατό τρόπο.

Γενικά, η τεχνική του multithreading είναι αρκετά διαδεδομένη και προσφέρει σχετική απλότητα στον κώδικα μαζί με αποδοτικότητα στην επίδοση. Παρακάτω εκθέτουμε μερικές εφαρμογές στις οποίες φαίνεται η χρησιμότητα στην παράλληλη εκτέλεση των threads. Αρχικά, προγράμματα που εκτελούν πολλές εντολές εισόδου-εξόδου, οι οποίες καθυστερούν πολύ, διότι η απόκριση των μνημών ή των συσκευών εισόδου-εξόδου είναι πολύ πιο αργή σε σχέση με τη συχνότητα ρολογιού του επεξεργαστή. Η χρησιμότητα των threads σε αυτές τις περιπτώσεις βρίσκεται στο ότι για κάθε τέτοια εντολή μπορεί να δημιουργηθεί ένα ξεχωριστό thread, το οποίο θα εκτελείται αυτόνομα.

Ένας άλλος τομέας εφαρμογών είναι αυτές που έχουν ρόλο την διεπαφή χρήστη-μηχανής. Με τη χρησιμοποίηση των threads, η μηχανή μπορεί να ανταποκρίνεται στις εντολές του χρήστη, ακόμα κι αν εκτελεί μεγάλες και απαιτητικές εφαρμογές. Αυτό γίνεται πολύ ξεκάθαρο στην περίπτωση των servers, οι οποίοι πρέπει να εξυπηρετούν ταυτόχρονα πολλούς πελάτες. Τέλος, η επίδοση αυξάνεται σημαντικά εκτελώντας ένα thread ανά επεξεργαστή, όταν χρησιμοποιούμε μια βιβλιοθήκη thread που υποστηρίζει πολλούς επεξεργαστές. Με άλλα λόγια, μπορούμε να διαμοιράζουμε τα threads που δημιουργούνται σε πολλούς επεξεργαστές και αυτά να εκτελούνται ταυτόχρονα. Αυτή είναι η εφαρμογή που ενδιαφέρει ιδιαίτερα εμάς σε αυτήν την εργασία, αφού ασχολούμαστε με συστοιχίες πολλών επεξεργαστών και πολυπύρηνα συστήματα.

Αν θέλαμε να εντοπίσουμε το βασικό πρόβλημα που εμφανίζει το multithreading, αυτό είναι ο μη-ντετερμινισμός, το γεγονός δηλαδή ότι δεν μπορούμε να ξέρουμε το αποτέλεσμα που θα πάρουμε από την εκτέλεση ενός προγράμματος, του οποίου ο κώδικας είναι διαμοιρασμένος σε threads που εκτελούνται παράλληλα . Αυτό οφείλεται στο συναγωνισμό των threads πάνω στις ίδιες μεταβλητές. Ας μην ξεχνάμε ότι μοιράζονται την ίδια μνήμη. Η λύση είναι σαφώς αυτή που προτάθηκε και στα συστήματα αρχιτεκτονικής κοινής μνήμης: ο καθορισμός προτεραιοτήτων μεταξύ των threads ως προς το συναγωνισμό δεδομένων και

το μπλοκάρισμα μιας μεταβλητής όταν αυτή χρησιμοποιείται από κάποιο thread. Η λύση αυτή είναι σχετικά απλή και οδηγεί στην ορθότητα της εκτέλεσης των προγραμμάτων. Παρόλ' αυτά, υπάρχουν εργασίες που αποτιμούν τη χρησιμότητα των threads, κρίνοντας ότι ο προγραμματισμός με threads δεν είναι ιδιαίτερα αποδοτικός και αφορά κυρίως εξοικειωμένους προγραμματιστές κι όχι ένα ευρύ φάσμα ανθρώπων. Αντί των threads, προτείνονται άλλοι τρόποι επίτευξης παράλληλου προγραμματισμού, όπως coordination languages, που βασίζονται σε ντετερμινιστικά μοντέλα [20].

2.4 Posix Threads

Αφού έχουμε μελετήσει την έννοια των threads και του multithreading σε έναν επεξεργαστή, το τελευταίο εργαλείο παράλληλου προγραμματισμού, με το οποίο θα ασχοληθούμε και επιτρέπει multithreading σε πολλούς επεξεργαστές, είναι η βιβλιοθήκη Posix Threads, γνωστή ως pthread, η οποία ορίζει ένα API (Application Programming Interface) για δημιουργία και χειρισμό threads [17]. Παρέχει τύπους, συναρτήσεις και σταθερές γραμμένες σε C για διαχείριση threads και υλοποιείται μέσω της βιβλιοθήκης pthread.h. Υλοποιήσεις των pthreads υπάρχουν για όλα τα Unix-based λειτουργικά συστήματα, αλλά και για Windows και Mac OS.

Ένα pthread αναπαρίσταται από τον πρωτογενή τύπο *pthread_t,* ο οποίος είναι ένας χειριστής για το pthread. Υπάρχουν τρεις βασικές συναρτήσεις με τις οποίες μπορούμε να χειριστούμε τα pthreads. Και οι τρεις επιστρέφουν έναν ακέραιο, ο οποίος ανάλογα με την τιμή του, δηλώνει αν έχει εκτελεστεί όπως αναμένεται η συνάρτηση ή αν προέκυψε κάποιο πρόβλημα, όπως ότι το σύστημα δεν είχε αρκετή μνήμη για να δημιουργήσει το pthread.

- *int pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes, void *(*thread_function) (void *), void *arguments)*

Η συνάρτηση αυτή δημιουργεί ένα pthread έτοιμο προς εκτέλεση. Έχει τέσσερα ορίσματα: το πρώτο είναι ο χειριστής του pthread, το οποίο έχει μια ταυτότητα. Το δεύτερο όρισμα αναφέρεται σε κάποιες ιδιότητες που μπορεί να έχει το pthread, τις οποίες θα δούμε παρακάτω πιο αναλυτικά. Το τρίτο όρισμα είναι η συνάρτηση που θα αναλάβει να εκτελέσει το pthread και το τέταρτο τα ορίσματα της συνάρτησης. Βλέπουμε ότι η συνάρτηση που μπορεί να λάβει ως όρισμα ένα pthread είναι της μορφής *void *func(void *)*. Αυτό είναι σημαντικό για τον προγραμματιστή, καθώς δεσμεύεται στη μορφή της συνάρτησης που πρέπει να δημιουργήσει. Με την ορθή εκτέλεση της *pthread_create()* δημιουργείται το pthread έτοιμο προς εκτέλεση και χρονοδρομολογείται από το σύστημα σε κάποιον επεξεργαστή. Το pthread τερματίζει και αποδεσμεύεται η μνήμη που κατέχει μόλις επιστρέψει η συνάρτηση την οποία εκτελεί ή μόλις κληθεί η *pthread_exit()*.

- *int pthread_exit (void *status)*

Με αυτή τη συνάρτηση τερματίζει ένα pthread. Η τιμή *status* που λαμβάνει ως όρισμα είναι η τιμή που επιστρέφει το pthread σε αυτό που το δημιούργησε. Ουσιαστικά, είναι η τιμή επιστροφής της συνάρτησης *func* που είχε αναλάβει να εκτελέσει με τη δημιουργία του (αν φυσικά υπάρχει). Ιδιαίτερη προσοχή είναι ότι ο τύπος επιστροφής είναι *void **, γεγονός που επιτρέπει η τιμή επιστροφής να έχει τελικά οποιοδήποτε τύπο επιλέξει ο χρήστης σε κάθε εφαρμογή (π.χ. int, char, float, κλπ).

- *int pthread_join (pthread_t thread, void **status_ptr)*

Η τρίτη συνάρτηση που βλέπουμε είναι αυτή που προσφέρει συγχρονισμό μεταξύ των pthreads. Όταν καλείται από ένα pthread-πατέρα, το σταματάει μέχρι να τερματίσει το pthread-παιδί που λαμβάνει ως πρώτο όρισμα. Το δεύτερο όρισμα είναι η τιμή επιστροφής του pthread-παιδιού, είναι δηλαδή η τιμή που επιστρέφει το pthread-παιδί, μέσω της *pthread_exit()*. Μόλις το pthread-παιδί τερματίσει, το pthread-πατέρας λαμβάνει την τιμή επιστροφής στο δεύτερο όρισμά και συνεχίζει κανονικά την εκτέλεσή του από αυτό το σημείο.

Αυτές οι συναρτήσεις προσφέρουν τις βασικές λειτουργίες που χρειάζεται να έχει ένα thread. Όπως είδαμε, μέσω του δεύτερου ορίσματος της *pthread_create(),* το όρισμα *attributes,* καθορίζονται οι ιδιότητες που έχει το pthread. Ο τύπος του είναι *pthread_attr_t,* που είναι πρωτογενής τύπος για τα pthreads, και για το χειρισμό του χρησιμοποιούνται συναρτήσεις που παρέχει η βιβλιοθήκη. Οι ιδιότητες ενός pthread που έχει την ευχέρεια να χειριστεί ο προγραμματιστής είναι

(a) η εμβέλεια: καθορίζει αν το pthread είναι δεσμευμένο ή όχι. Τα δεσμευμένα pthreads συναγωνίζονται με όλα τα pthreads μιας διεργασίας, ανήκουν σε αυτήν και δρομολογούνται από το λειτουργικό σύστημα. Ως πλεονέκτημα έχουν υψηλή επίδοση και ντετερμινιστικά αποτελέσματα, αλλά για τη δημιουργία τους απαιτείται η δέσμευση πόρων αντίστοιχα με τις διεργασίες. Τα μη δεσμευμένα pthreads δρομολογούνται από το pthread runtime σύστημα, το οποίο πλέκει τα pthreads σε ομάδες που δρομολογούνται μέσω ενός μικρότερου αριθμού "οχημάτων" από το λειτουργικό. Αυτό έχει ως αποτέλεσμα πιο γρήγορη χρονοδρομολόγηση.

(b) η detached state: διαχωρίζει τα pthreads σε JOINABLE και DETACHED. Στα JOINABLE pthreads το σύστημα διατηρεί το id τους και το status τερματισμού τους, σε περίπτωση που κάποιο άλλο pthread κάνει *pthread_join()* σε αυτά μετά τον τερματισμό τους. Αντίθετα, στα DETACHED pthreads μετά τον τερματισμό τους το σύστημα δε διατηρεί τίποτα και μπορεί να δώσει το id τους σε pthreads που θα

δημιουργηθούν μετά. Με τον τερματισμό τους ελευθερώνουν πόρους μνήμης, αλλά δεν μπορούν να γίνουν join.

(c) η στοίβα: ο προγραμματιστής μπορεί να ρυθμίσει το μέγεθος της στοίβας, αλλά και το σημείο της μνήμης που θα δεσμευτεί η στοίβα.

(d) η πολιτική χρονοδρομολόγησης: πέραν της εμβέλειας, ο προγραμματιστής μπορεί να επιλέξει τον αλγόριθμο χρονοδρομολόγησης των pthreads. Οι αλγόριθμοι που δίνονται είναι ο FIFO, ο RoundRobin και ο SCHED_OTHER. Για τους δυο πρώτους έχουμε συζητήσει στη χρονοδρομολόγηση εργασιών. Ο τρίτος είναι μια πολιτική που αφήνει τη χρονοδρομολόγηση στο ίδιο το σύστημα, στο οποίο εκτελείται η εφαρμογή. Τα περισσότερα συστήματα εκτελούν κάποιον διακοπτό αλγόριθμο που δίνει ένα κβάντο χρόνου στα pthreads, στα οποία δίνεται και κάποια προτεραιότητα. Πέραν των αλγορίθμων, η βιβλιοθήκη των pthreads δίνει τη δυνατότητα ο χρήστης να δώσει κάποια προτεραιότητα στα pthreads ως προς τη χρονοδρομολόγησή τους, όπως επίσης και τη δυνατότητα της κληρονομικότητας. Ένα pthread μπορεί να κληρονομήσει από το δημιουργό του τα χαρακτηριστικά χρονοδρομολόγησής του.

Όλες αυτές οι ιδιότητες μπορούν να αξιοποιηθούν από το χρήστη μέσω συναρτήσεων που παρέχει η βιβλιοθήκη και αναλύονται σε διάφορες μελέτες [22]. Χρήσιμο σε αυτήν την εργασία θα ήταν να δοθούν οι δυο βασικές συναρτήσεις των attirbutes:

- *int pthread_attr_init(pthread_attr_t *tattr)*

Δημιουργία ενός attribute με τα default χαρακτηριστικά, που αντιπροσωπεύει pthread μη δεσμευμένης εμβέλειας, JOINABLE, 1 megabyte μέγεθος στοίβας που δεσμεύεται σε χώρο που του δίνει το ίδιο το σύστημα, με πολιτική χρονοδρομολόγησης SCHED_OTHER, κληρονομώντας την πολιτική του δημιουργού του.

- *int pthread_attr_destroy(pthread_attr_t *tattr);*

Η συνάρτηση καταστρέφει ένα attribute, αποδεσμεύοντας τη μνήμη που κατέχει.

Όπως είπαμε προηγουμένως μια βασική παράμετρος που οφείλουμε να προσέχουμε όταν προγραμματίζουμε σε επίπεδο threads είναι ο συναγωνισμός τους για τις ίδιες μεταβλητές. Ας μην ξεχνάμε ότι όλα τα threads μοιράζονται κοινή μνήμη. Χρειάζονται, επομένως, κάποια εργαλεία για τον καθορισμό προτεραιοτήτων ανάμεσα στα threads. Γι' αυτό το λόγο, το API των pthreads προσφέρει δυο ειδών μεταβλητές: τα mutex και τα condition.

Οι μεταβλητές mutex παίρνουν το όνομά τους από τις αρχές των λέξεων mutual exclusion, που σημαίνει αμοιβαίος αποκλεισμός. Είναι από τα βασικά στοιχεία που επιτρέπουν συγχρονισμό μεταξύ των pthreads και προστασία στις μοιραζόμενες μεταβλητές όταν προκύπτουν πολλαπλές εγγραφές. Λειτουργούν ως κλείδωμα, προστατεύοντας την πρόσβαση σε κοινά δεδομένα. Κάθε φορά μόνο ένα pthread μπορεί να κατέχει (δηλαδή να κλειδώνει) ένα mutex, έτσι ώστε όταν πολλά pthreads κάνουν απόπειρα να κλειδώσουν ένα mutex, μόνο ένα να επιτυγχάνει. Όλα τα υπόλοιπα θα παραμείνουν σταματημένα μέχρι να ξεκλειδωθεί το mutex από το pthread-κάτοχο, οπότε θα το κλειδώσει κάποιο άλλο και θα εκτελεστεί. Έτσι, τα pthreads παίρνουν σειρά για να έχουν πρόσβαση στα κοινά δεδομένα.

Η τυπική διαδικασία για τη χρησιμοποίηση ενός mutex είναι η ακόλουθη:

- Δημιουργία και αρχικοποίηση ενός mutex
- Ορισμένα pthreads προσπαθούν να κλειδώσουν το mute
- Μόνο ένα επιτυγχάνει και τα υπόλοιπα αδρανοποιούνται
- Το pthread-κάτοχος εκτελεί μια σειρά από εντολές
- Το pthread-κάτοχος ξεκλειδώνει το mutex
- Ένα άλλο pthread από αυτά που είχαν αδρανοποιηθεί κλειδώνει το mutex και επαναλαμβάνεται η παραπάνω διαδικασία
- Όταν όλα τα pthreads ολοκληρώσουν τις εργασίες τους και το mutex δε χρειάζεται πλέον, καταστρέφεται.

Ο πρωτογενής τύπος μιας μεταβλητής mutex είναι *pthread_mutex_t* και για να χρησιμοποιηθεί πρέπει πρώτα να αρχικοποιηθεί. Οι βασικές ρουτίνες που παρέχονται για τη διαχείριση των mutex είναι οι ακόλουθες. Αντίστοιχα με τις συναρτήσεις για το χειρισμό των pthreads, η τιμή επιστροφής τους είναι ένας ακέραιος που δείχνει αν εκτελέστηκε η συνάρτηση με ορθό τρόπο ή αν προέκυψε κάποιο μη αναμενόμενο σφάλμα.

- *int pthread_mutex_init (pthread_mutex_t *mut, const pthread_mutexattr_t *attr)*

Δημιουργία και αρχικοποίηση μιας μεταβλητής *mut,* τύπου pthread_mutex_t. Το δεύτερο όρισμα αφορά κάποιες ιδιότητες που μπορεί να έχει ένα mutex (protocol, priocieling, process-shared), και αν τεθεί στην τιμή NULL, το mutex αποκτά τις default ιδιότητες που παρέχει η βιβλιοθήκη.

- *int pthread_mutex_lock (pthread_mutex_t *mut)*

Το pthread που εκτελεί αυτή τη συνάρτηση κλειδώνει το mutex *mut.* Αν το συγκεκριμένο mutex έχει ήδη κλειδωθεί, τότε το pthread που κάλεσε τη συνάρτηση μπαίνει σε κατάσταση αναμονής μέχρι να ελευθερωθεί το mutex. Όταν εκείνο ελευθερωθεί, τότε μπορεί να το κλειδώσει και να συνεχίσει κανονικά την εκτέλεσή του.

- *int pthread_mutex_unlock (pthread_mutex_t *mut)*

Το pthread απελευθερώνει το mutex *mut,* που είχε κλειδώσει νωρίτερα. Αυτή η συνάρτηση πρέπει να καλείται οπωσδήποτε όταν το τρέχων pthread έχει ολοκληρώσει τις εργασίες του πάνω στις κοινές μεταβλητές, γιατί μπορεί άλλα pthreads να περιμένουν να κλειδώσουν το mutex ώστε να εκτελέσουν τις εργασίες τους.

- *int pthread_mutex_destroy (pthread_mutex_t *mut)*

Καταστροφή της μεταβλητής *mut* και αποδέσμευση των πόρων του συστήματος που χρησιμοποιεί. Αυτή η συνάρτηση καλείται όταν κανένα pthread δε χρειάζεται πλέον να ενεργήσει στις κοινές μεταβλητές, οπότε παύει η χρησιμότητα του mutex.

Το άλλο είδος μεταβλητών που αξιοποιούνται για συγχρονισμό είναι οι condition variables. Έχουν πρωτογενή τύπο *pthread_cond_t* και χειρίζονται με αντίστοιχο τρόπο με τα mutex. Η διαφορά τους βρίσκεται στο γεγονός ότι ενώ τα mutex εξασφαλίζουν συγχρονισμό ελέγχοντας την πρόσβαση των pthreads στα δεδομένα, τα condition ελέγχουν το συγχρονισμό μέσω της τιμές ενός δεδομένου. Υπάρχουν φορές που ένα pthread περιμένει να εκπληρωθεί μια συνθήκη για να εκτελέσει μια σειρά από ενέργειες. Χωρίς τις condition μεταβλητές θα έπρεπε να ελέγχει συνέχεια αν πληρείται αυτή η συνθήκη (π.χ. σε ένα while loop), καταναλώνοντας άσκοπα υπολογιστικό χρόνο. Η χρησιμότητα των condition έγκειται στο γεγονός ότι καταργεί αυτό το συνεχή έλεγχο, αλλά βάζει σε αναμονή το pthread και το ενημερώνει μόλις ενεργοποιηθεί η συνθήκη του.

Μια condition μεταβλητή λειτουργεί πάντα σε συνδυασμό με τις mutex μεταβλητές. Η τυπική διαδικασία για τη χρησιμοποίηση ενός condition είναι η ακόλουθη:

- Δημιουργία και αρχικοποίηση ενός mutex
- Δημιουργία και αρχικοποίηση ενός condition
- Ένα pthread κλειδώνει το mutex και εκτελεί τις εργασίες του μέχρι να χρειαστεί να εκπληρωθεί κάποια συνθήκη (πχ. η condition μεταβλητή να λάβει μια συγκεκριμένη τιμή)
- Το pthread περιμένει να ικανοποιηθεί η συνθήκη, οπότε αδρανοποιείται και ταυτόχρονα απελευθερώνεται το mutex
- Ένα άλλο pthread δεσμεύει το mutex και εκτελεί τις εργασίες του, αλλάζοντας την τιμή της κοινής μεταβλητής
- Ελέγχει αν η τιμή αυτή ικανοποιεί τη συνθήκη που είχε θέσει το pthread που περιμένει και αν ναι, του στέλνει σήμα και απελευθερώνει το mutex
- Το αρχικό pthread λαμβάνει το σήμα και επανέρχεται σε κατάσταση εκτέλεσης, συνεχίζοντας τις εργασίες του, έχοντας αυτόματα επανακτήσει το mutex

Οι βασικές ρουτίνες που παρέχονται για αρχικοποίηση και χειρισμό των condition μεταβλητών δίνονται παρακάτω:

- *int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)*

Δημιουργία και αρχικοποίηση μιας μεταβλητής *cond*, τύπου pthread_cond_t. Το δεύτερο όρισμα αφορά ιδιότητες αντίστοιχες με αυτές των mutex και όταν παίρνει την τιμή NULL, η μεταβλητή *cond* αποκα τις default ιδιότητες.

- *int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)*

Το pthread που την καλεί μπαίνει σε κατάσταση αναμονής μέχρι να ειδοποιηθεί ότι ικανοποιήθηκε η συνθήκη του. Προϋποθέτει ότι έχει ήδη δεσμεύσει το mutex που λαμβάνει ως δεύτερο όρισμα η συνάρτηση. Μόλις κληθεί η συνάρτηση και εκτελεστεί ορθά, το mutex ελευθερώνεται έτσι ώστε να μπορεί να δεσμευτεί από οποιοδήποτε άλλο thread. Όταν ικανοποιηθεί η συνθήκη, το pthread θα ξυπνήσει, θα επανακτήσει αυτόματα το mutex και θα συνεχίσει την εκτέλεσή του από εκείνο το σημείο.

- *int pthread_cond_signal(pthread_cond_t *cond)*

Καλείται από ένα pthread ώστε να ξυπνήσει ένα άλλο το οποίο περίμενε να ικανοποιηθεί η *cond*. Για να κληθεί, πρέπει το pthread να έχει δεσμεύσει πρώτα το αντίστοιχο mutex. Αφού κληθεί, το συγκεκριμένο pthread πρέπει να ελευθερώσει το mutex, καλώντας την *pthread_mutex_unlock(),* ώστε να ολοκληρωθεί η *pthread_cond_wait()* που είχε καλεστεί από το άλλο pthread.

- *int pthread_cond_broadcast(pthread_cond_t *cond)*

Είναι αντίστοιχη με την *pthead_cond_signal(),* αλλά καλείται όταν περισσότερα από ένα pthead περιμένουν να ικανοποιηθεί η συγκεκριμένη συνθήκη. Με αυτή τη ρουτίνα το σήμα αποστέλλεται σε όλα τα pthreads.

- *int pthread_cond_destroy(pthread_cond_t *cond)*

Καταστροφή της μεταβλητής *cond* και αποδέσμευση των πόρων του συστήματος που χρησιμοποιεί. Καλείται όταν πλέον η μεταβλητή δε χρειάζεται από κανένα pthread.

Αυτή ήταν μια περιεκτική ανάλυση των βασικών δυνατοτήτων που παρέχει το API των pthreads. Υπάρχουν κι άλλα χαρακτηριστικά, όπως η αποστολή σημάτων και τα barriers,

των οποίων, όμως, η ανάλυση δεν αφορά την παρούσα εργασία, αλλά μπορούν να βρεθούν σε σχετική βιβλιογραφία [21].

2.5 RTS vs OS

Αυτή η ενότητα αναφέρεται στην "αντιπαλότητα" του RTS (runtime system) με το OS (operating system). Αυτό που πραγματεύεται είναι η σύγκριση της χρονοδρομολόγησης εργασιών μέσω των δυο αυτών συστημάτων. Το λειτουργικό σύστημα (OS) είναι το λογισμικό υπεύθυνο για τη διαχείριση, το συντονισμό και την κατανομή πόρων των διεργασιών, ενώ το RTS είναι ένα σύνολο εντολών που υλοποιεί μια γλώσσα προγραμματισμού ή ένα προγραμματιστικό περιβάλλον και καθορίζει τη "συμπεριφορά" της στον πυρήνα.

Για απλότητα, θα ορίσουμε δυο χώρους όπου μπορεί να γίνει η δρομολόση των εργασιών: το χώρο χρήστη, όπου αναλαμβάνει το RTS και το χώρο πυρήνα όπου υπεύθυνο είναι το OS. Στο χώρο πυρήνα μπορούν να δημιουργούνται και να δρομολογούνται threads όπως έχουν ως τώρα περιγραφεί, ενώ στο χώρο χρήστη δημιουργούνται tasks, τα οποία μπορούν να δρομολογηθούν, αλλά όχι από το λειτουργικό σύστημα. Αντίθετα, η δρομολόγηση στο χώρο χρήστη πρέπει να γίνει από το RTS, δηλαδή με πιο απλά λόγια, από κώδικα που πρέπει να γράψει ο ίδιος ο χρήστης.

Ένα task είναι παρόμοιο με ένα thread. Αντιπροσωπεύει μια σειρά εντολών που μπορούν να εκτελεστούν ανεξάρτητα από άλλα tasks μέχρι κάποιο σημείο. Ένα πρόγραμμα μπορεί να δομηθεί σε ανεξάρτητα tasks, τα οποία θα εκτελούνται παράλληλα και θα συγχρονίζονται μεταξύ τους όποτε χρειάζεται. Αντίστοιχα, λοιπόν, με την τεχνική του multithreading προκύπτει η τεχνική του mulitasking. Το RTS είναι υπεύθυνο για τη δημιουργία, τη δρομολόγηση, την καταστροφή τους και για το χειρισμό της μνήμης τους. Κάνει δηλαδή την ίδια ακριβώς δουλειά που κάνει το OS για τα threads. Αντίθετα, όμως τα threads, ο πυρήνας δε γνωρίζει τίποτα για τα tasks, αλλά τα βλέπει ως ενσωματωμένα κομμάτια σε ένα thread. Και η ερώτηση που προκύπτει είναι: γιατί να μην αντιστοιχίσω κάθε task με ένα thread και όλη η διαχείριση των tasks να γίνεται στο χώρο του πυρήνα, δηλαδή από το λειτουργικό σύστημα; Άλλωστε, αυτή είναι μια από τις κύριες εργασίες του. Γιατί να χρειαστεί να εμπλακεί ο προγραμματιστής με τη διαχείριση των tasks, ενώ προσφέρεται έτοιμη και αυτόματα η διαχείριση των threads; Αν θέλαμε να δώσουμε την απάντηση με μια λέξη είναι η εξής: η επίδοση.

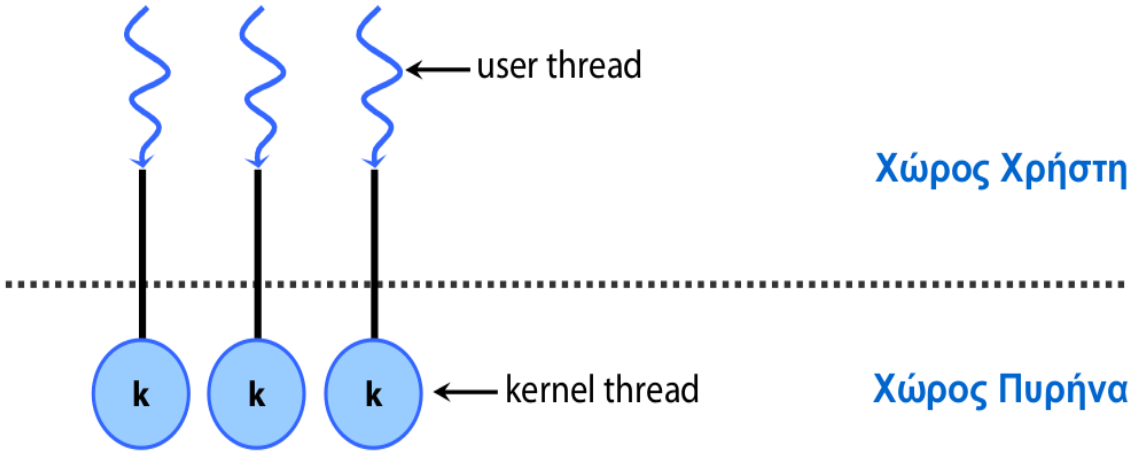
Έχουμε ήδη δει στην περίπτωση του multithreading, πώς το hardware μπορεί να υποστηρίξει την ταυτόχρονη εκτέλεση εντολών διαφορετικών thread στον ίδιο επεξεργαστή (Σχήμα 6). Είδαμε ότι η εναλλαγή τους γινόταν μέσα από το hardware, χωρίς να εμπλακεί το OS, κερδίζοντας έτσι πολύτιμο υπολογιστικό χρόνο. Ακόμη, στα pthreads όταν αναφερθήκαμε στην εμβέλεια, είδαμε ότι η default πρακτική της βιβλιοθήκης είναι τα pthreads να ομαδοποιούνται από το pthread RTS κι έπειτα αυτές οι ομάδες να

δρομολογούνται μέσω του OS. Αυτές είναι μερικές ενδείξεις ότι η δρομολόγηση μέσω του RTS μπορεί να γίνει πιο αποδοτική από εκείνη του OS.

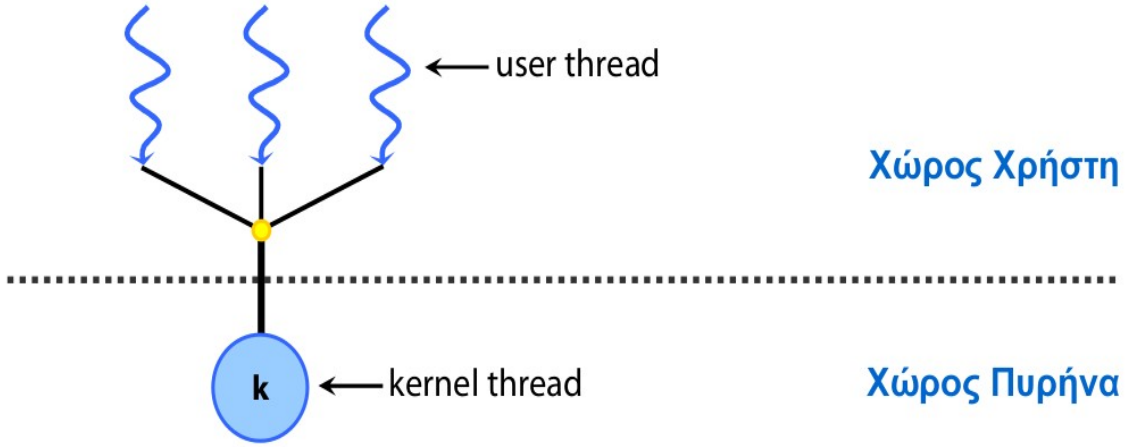
Θα θεωρήσουμε με ασφάλεια ότι ο αριθμός των παράλληλων tasks είναι μεγαλύτερος από των επεξεργαστών που χρησιμοποιεί το σύστημα (αλλιώς η διαφορά στην επίδοση είναι αμελητέα). Θα μπορούσαμε να τα δρομολογήσουμε αναθέτοντας καθένα από αυτά σε ένα thread, το οποίο διαχειρίζεται το OS. Αυτή όμως, η τακτική υστερεί σε σχέση με τη δρομολόγηση μέσω του RTS στο ότι το OS δεν μπορεί να κάνει καμία παραδοχή ως προς τη φύση των tasks. Το OS υλοποιεί ένα συγκεκριμένο αλγόριθμο δρομολόγησης για όλα τα threads, αλλά το RTS μπορεί ανάλογα με τις ιδιότητες του κάθε task να επιλέξει τον τρόπο και τις συνθήκες δρομολόγησής του. Από αυτό το γεγονός έχουμε δύο κέρδη: αφενός την αποδοτικότητα, μιας και ο τρόπος δρομολόγησης αξιοποιεί τις ιδιότητες που έχει το κάθε task ξεχωριστά, και αφετέρου τη φορητότητα, καθώς το RTS μπορεί να προσαρμόζεται σε συστήματα με διαφορετικό πλήθος επεξεργαστών κι έτσι να εκμεταλλεύεται όλους τους διαθέσιμους πόρους που του παρέχονται. Επίσης, η χρονοδρομολόγηση μέσω του RTS μπορεί να αξιοποιήσει αποδοτικότερα μη ομογενή συστήματα, όπου κάθε συστοιχία επεξεργαστών έχει διαφορετική αρχιτεκτονική.

Συνήθως το OS χρησιμοποιεί διακοπτούς αλγορίθμους (πχ. Round Robin), ενώ το RTS πολλές φορές χρησιμοποιεί συνεργατικούς. Στους μεν διακοπτούς το OS αναλαμβάνει να αποθηκεύσει όλη την αρχιτεκτονική κατάσταση του task όταν προκύπτει ένα context switch, ενώ στους συνεργατικούς το RTS χρειάζεται μόνο την πληροφορία για το σημείο που είχε σταματήσει το task όταν έγινε το context switch. Με αυτόν τον τρόπο, το context switch γίνεται πολύ πιο γρήγορο στο RTS απ' ότι στο OS.

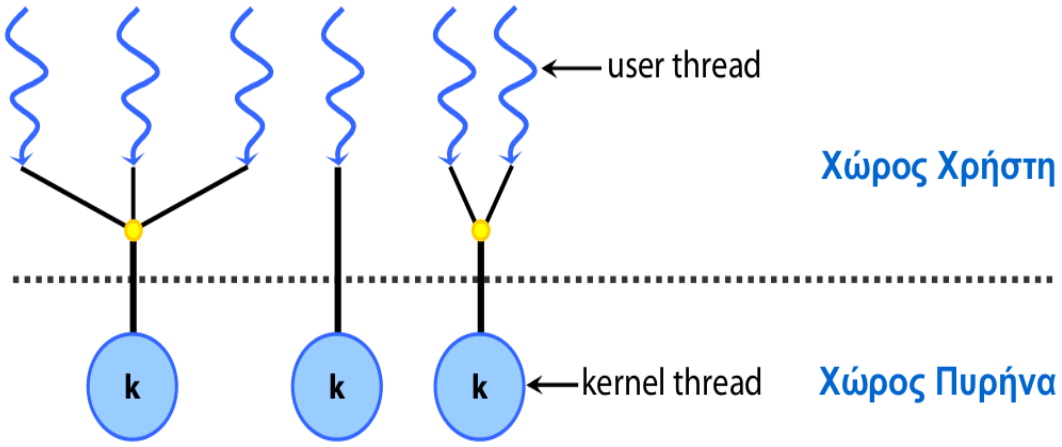
Με βάση αυτά, υπάρχουν τρεις τεχνικές ώστε να κάνουμε multitasking, οι οποίες παρουσιάζονται στα παρακάτω σχήματα (πηγή: διαφάνειες μαθήματος “Λειτουργικά Συστήματα”). Αρχικά η αντιστοίχιση κάθε task που δημιουργούμε (στο σχήμα αναφέρεται ως user thread) με ένα thread πυρήνα και η δρομολόγηση του μέσω του OS (Σχήμα 7Α). Αυτή η τακτική είπαμε ότι είναι απλή και εύκολη στην υλοποίηση, αλλά υστερεί σε απόδοση. Η επόμενη μέθοδος είναι η δημιουργία tasks στο επίπεδο χρήστη, των οποίων η χρονοδρομολόγηση καθορίζεται από το RTS (Σχήμα 7Β). Ο πυρήνας δεν τα βλέπει ως ξεχωριστές εργασίες, αλλά τα αντιμετωπίζει όλα ως ένα thread, και τα δρομολογεί όλα μαζί. Αυτή η τακτική μας ενδιαφέρει, αλλά έχει ως πρόβλημα ότι υπάρχει μόνο ένα thread στο χώρο πυρήνα, οπότε μπορεί να χρησιμοποιηθεί για έναν επεξεργαστή. Γι' αυτό, σε πολυεπεξεργαστικά συστήματα χρησιμοποιούμε την τεχνική στο Σχήμα 7Γ, όπου δημιουργούνται τα tasks στο χώρο χρήστη κι έπειτα διαμοιράζονται σε threads στο χώρο πυρήνα, τα οποία μπορούν να εκτελεστούν στους διάφορους επεξεργαστές του συστήματος (multithreading).



Σχήμα 7Α: 1:1 Threads σε επίπεδο πυρήνα



Σχήμα 7Β: N:1 Tasks σε επίπεδο χρήστη



Σχήμα 7Γ: N:M Συνδυασμός Tasks και Threads

2.6 Η βιβλιοθήκη task.h

Βασικό κομμάτι της εργασίας μας ήταν η δημιουργία του δικού μας RTS (για καθαρά εκπαιδευτικούς λόγους) ώστε να μπορούμε να διαχειριζόμαστε ανεξάρτητες εργασίες, οι οποίες θα μπορούν να εκτελούνται και παράλληλα. Είδαμε ότι το RTS δουλεύει στο χώρο χρήστη και μπορεί να είναι πολύ πιο αποδοτικό από το OS και γι' αυτό επιλέξαμε να υλοποιήσουμε τη δική μας εκδοχή RTS και να πειραματιστούμε με αυτή. Η δημιουργία του είναι εμπνευσμένη από τη βιβλιοθήκη pthread.h και από τον τρόπο που εκείνη διαχειρίζεται τα pthreads (ενότητα 2.3), όπως θα γίνει φανερό και στη συνέχεια.

Στόχος της υλοποίησης του RTS είναι να επιτρέπει στον προγραμματιστή να διαχειρίζεται και να δρομολογεί ανεξάρτητες εργασίες, οι οποίες θα μπορούν να εκτελούνται παράλληλα σε ένα πλήθος επεξεργαστών. Το RTS μας θα παρέχει στον προγραμματιστή κατάλληλες δομές, με βασική τη δομή task_t που θα είναι η κάθε ανεξάρτητη εργασία. Επίσης, θα παρέχει συναρτήσεις μέσω των οποίων θα μπορεί να διαχειρίζεται τις ανεξάρτητες εργασίες του και τέλος ένα δρομολογητή, ο οποίος θα αναλαμβάνει να διαμοιράσει τις εργασίες στους υπάρχοντες επεξεργαστές. Σε αυτήν την ενότητα θα μελετήσουμε τα δυο πρώτα στοιχεία του RTS μας: τις δομές δεδομένων (βιβλιοθήκη *struct.h*) και τις συναρτήσεις που διαχειρίζονται αυτές τις δομές (βιβλιοθήκη *task.h*). Ο τρόπος λειτουργίας του δρομολογητή όπως και οι αλγόριθμοι που θα εκτελεί θα μας απασχολήσουν στα δυο επόμενα κεφάλαια. Ας δούμε αναλυτικά τις δομές και τις συναρτήσεις:

➔ *Η δομή task_t*

Η βασικότερη δομή είναι το task, το οποίο στη βιβλιοθήκη μας ονομάζεται *task_t*. Αποτελείται από τα ακόλουθα πεδία, τα οποία αναλύουμε στη συνέχεια.

```
typedef struct _task {
    int id;
    int state;
    void *(*func)(void *);
    void *arg;
    void *retval;
    int wait_for_id;
    void *wait_for_value;
    int ret_sched;
    ucontext_t uc;
}task_t;
```

Το task αποτελείται από ένα *id,* μοναδικό για καθένα ώστε να διαχωρίζεται από όλα τα άλλα. Το πεδίο *state* της δομής αναφέρεται στην κατάσταση του task και μπορεί να είναι RUNNABLE, WAITING ή FINALIZED. Τα πεδιά *func* και *arg* αναφέρονται στη συνάρτηση και τα ορίσματά της τα οποία έχει αναλάβει το task να εκτελέσει κατ' αντιστοιχία με τον τρόπο που ένα pthread αναλαμβάνει να εκτελέσει μια συνάρτηση. Υπάρχει ακόμα ένα πεδίο *retval* στο οποίο θα αποθηκεύεται η μεταβλητή (όποιον τύπο κι αν αυτή έχει) που επιστρέφει το ίδιο το task όταν τερματίζει και ένα πεδίο *wait_for_value* το οποίο αφορά τη μεταβλητή που μπορεί να περιμένει ένα task-πατέρας από το task-παιδί του. Επίσης, το πεδίο *wait_for_id* αναφέρεται στο id του task-παιδιού που περιμένει να τερματίσει ώστε το συγκεκριμένο task να συνεχίσει τη δουλειά του. Ένα πεδίο που χρειάστηκε ακόμη ήταν το *ret_sched* που είναι ένας ακέραιος που δείχνει από ποιον επεξεργαστή δημιουργήθηκε αυτό το task. Το πεδίο αυτό χρειάζεται έτσι ώστε όταν ένα task επιστρέψει από την κατάσταση αναμονής σε κατάσταση έτοιμο για εκτέλεση, να γνωρίζουμε σε ποιον επεξεργαστή θα συνεχίσει την εκτέλεσή του.

Το τελευταίο πεδίο *uc* είναι από τα πιο σημαντικά πεδία που περιέχει το task. Είναι αυτό που μας επιτρέπει να δρομολογούμε τα tasks σε χώρο χρήστη χωρίς να εμπλέκεται ο πυρήνας. Ο τύπος του *uc* είναι *ucontext_t,* μια δομή της C που περιγράφεται στη βιβλιοθήκη *ucontext.h*. Αυτή η βιβλιοθήκη παρέχει έτοιμες συναρτήσεις, οι οποίες μπορούν να αποθηκεύουν και να ανακαλούν το context ενός task. Ως context του task ορίζουμε την ελάχιστη πληροφορία δεδομένων που πρέπει να αποθηκεύσουμε στη μνήμη ώστε το task να μπορεί να διακοπεί για κάποιο χρονικό διάστημα και έπειτα η εκτέλεσή του να μπορεί να συνεχίσει από το σημείο διακοπής. Ακολουθεί μια σύντομη περιγραφή των στοιχείων της ucontext.h [23].

➔ *Η βιβλιοθήκη ucontext.h*

- Η βασική δομή της βιβλιοθήκη είναι το ucontext_t που έχει τα εξής πεδία:

```
typedef struct ucontext {
    unsigned long int uc_flags;
    struct ucontext *uc_link;
    stack_t uc_stack;
    mcontext_t uc_mcontext;
    _sigset_t uc_sigmask;
    struct _fpstate _fpregs_mem;
}ucontext_t
```

Ιδιαίτερο ενδιαφέρον για εμάς έχουν τα πεδιά uc_mcontext που είναι η πραγματική

κατάσταση του task και το πεδίο uc_link που δείχνει το “διάδοχο” στον οποίο επιστρέφει το τρέχων context όταν τελειώσει η εργασία του.

- int getcontext(ucontext_t *ucp);

Με αυτή τη συνάρτηση παίρνουμε το ucontext του τρέχοντος task και το αποθηκεύουμε στη διεύθυνση του ucp. Επιστρέφει 0 αν εκτελεστεί ορθά αλλιώς -1 σε περίπτωση σφάλματος.

- int setcontext(ucontext_t *ucp);

Εδώ θέτουμε το ucontext του τρέχοντος task ίσο με το ucontext του ucp που δέχεται ως όρισμα η setcontext. Το όρισμα πρέπει να έχει ήδη δημιουργηθεί (είτε από τη getcontext είτε μέσω της sigaction).

- void makecontext(ucontext_t *ucp, void *func(), int argc,..);

Τροποποιούμε το περιεχόμενο του ucp. Για να εκτελεστεί η makecontext πρέπει πρώτα να έχουμε δεσμεύσει μια νέα στοίβα για τη διεύθυνση uc_stack και να έχει οριστεί ένας διάδοχος στο uc_link. Όταν ενεργοποιηθεί το ucontext (από τη setcontext ή από τη swapcontext) τότε εκτελείται η συνάρτηση func, που έχει πλήθος ορισμάτων ίσο με argc. Μόλις η func τερματίσει και επιστρέψει, ενεργοποιείται ο διάδοχος του ucp (αυτός που δείχνει το ucp_link). Αν δεν υπάρχει διάδοχος (το πεδίο uc_link = NULL) τότε τερματίζεται το task.

- int swapcontext(ucontext_t *oucp, ucontext_t *ucp);

Αποθηκεύει το context του τρέχοντος task στο oucp και ενεργοποιεί το ucp. Σε περίπτωση αποτυχίας όλες οι συναρτήσεις επιστρέφουν -1.

Αυτές είναι και οι συναρτήσεις που θα χρησιμοποιούμε για να αποθηκεύουμε την τρέχουσα κατάσταση ενός task και να μπορούμε να δίνουμε τον έλεγχο στο δρομολογητή, ο οποίος θα διαλέγει -πιθανώς- κάποιο άλλο task για να εκτελεστεί.

➔ *Δομές αποθήκευσης των tasks*

Για να διαχειριζόμαστε τα tasks χρειαζόμαστε κάποιες δομές αποθήκευσης. Χρειαζόμαστε μια δομή για να αποθηκεύουμε τα tasks που είναι έτοιμα προς εκτέλεση, μια για αυτά που βρίσκονται σε κατάσταση αναμονής, αλλά και άλλη μια για αυτά που έχουν τερματίσει. Τα προς εκτέλεση tasks θα αποθηκεύονται σε μια δομή με όνομα deque_t. Κάθε

επεξεργαστής θα έχει μια δική του τέτοια δομή, την οποία θα αντιμετωπίζει ως στοίβα. Δηλαδή θα κάνει push και pop tasks από την ουρά της. Στο κεφάλαιο 4 θα μιλήσουμε πιο αναλυτικά για τη δομή deque, μιας και παίζει σημαντικό ρόλο στον τρόπο με τον οποίο τα tasks μεταφέρονται από κάποιον επεξεργαστή σε έναν άλλο μέσω του δρομολογητή. Η deque_t αντιμετωπίζεται ως ένας πίνακας από tasks, όπως φαίνεται και παρακάτω:

```
typedef struct _task_deque {
    task_t **head;
    int used_space;
    int size;
}deque_t
```

Το μέγεθος κάθε φορά της deque είναι πεπερασμένο, αλλά φροντίζουμε κάθε φορά στις εφαρμογές μας να έχει το επιθυμητό μέγεθος ώστε να μην αντιμετωπίσουμε πρόβλημα χώρου. Ο λόγος που επιλέχθηκε μια δομή σταθερού μεγέθους κι όχι μια δυναμική, όπως θα μπορούσε να γίνει με μία συνδεδεμένη λίστα είναι γιατί είναι πιο αποδοτικό ο χώρος μνήμης να έχει δεσμευτεί εξαρχής. Η δυναμική δέσμευσή του θα ήταν μια διαδικασία που θα κόστιζε περισσότερο επειδή κάτι τέτοιο θα συνέβαινε αρκετά συχνά κατά τη διάρκεια της εκτέλεσης.

Η δομή που θα αποθηκεύονται τα tasks που βρίσκονται σε αναμονή αλλά και αυτά που έχουν τερματίσει την εκτέλεσή τους είναι ίδιας μορφής και πρόκειται για μια διπλά συνδεδεμένη λίστα. Στην υλοποίησή μας η δομή έχει το όνομα list_t.

```
typedef struct _task_list {
    struct dllist *head, *tail;
}list_t

struct dllist {
  task_t *wait_task;
  struct dllist *next, *prev;
};
```

Τη λίστα των εν αναμονή tasks την ονομάζουμε wait και εκείνη των τερματισμένων την ονομάζουμε ready. Ο λόγος που χρειάζεται η πρώτη είναι προφανής, μιας και όταν ένα task βρίσκεται σε αναμονή αυτό συμβαίνει διότι περιμένει το αποτέλεσμα από κάποιο άλλο task. Μόλις αυτό τερματίσει και επιστρέψει το αποτέλεσμα, το task που βρίσκεται σε αναμονή ενημερώνεται και μεταβαίνει από τη λίστα wait στη deque του επεξεργαστή του. Η αναγκαιότητα της λίστας ready έγκειται στο γεγονός ότι όταν τερματίζεται ένα task μπορεί να επιστρέφει ένα αποτέλεσμα, το οποίο θα χρειαστεί στο μέλλον κάποιο άλλο. Στην

πραγματικότητα μόνο ο πατέρας του μπορεί να περιμένει το αποτέλεσμα που θα επιστρέψει. Γι' αυτό το λόγο αν ο πατέρας δεν έχει ζητήσει ακόμη το αποτέλεσμα του παιδιού του, το παιδί αποθηκεύεται στη ready λίστα όταν τερματίσει και θα βρίσκεται εκεί μέχρι να αναζητήσει το αποτέλεσμα επιστροφής του ο πατέρας του.

➔ *Συναρτήσεις Διαχείρισης των wait και ready λιστών*

- void *ready_list_check(int id)

Αυτή η συνάρτηση διατρέχει τη ready λίστα ελέγχοντας αν υπάρχει εκεί το task με id ίσο με το όρισμα της συνάρτησης. Αν δεν το βρει, δηλαδή δεν έχει τερματίσει ακόμα αυτό το task επιστρέφει NULL. Αν όμως υπάρχει στη λίστα, τότε αφαιρεί αυτό το task από τη ready και η συνάρτηση επιστρέφει ως αποτέλεσμα την τιμή επιστροφής του task που έψαχνε, δηλαδή το πεδίο retval.

- int wait_list_check(task_t *t)

Η συνάρτηση διατρέχει τη wait λίστα ψάχνοντας να βρει αν υπάρχει κάποιο task που να περιμένει να τερματίσει το t που λαμβάνεται ως όρισμα. Αν δεν υπάρχει επιστρέφει 0 (με άλλα λόγια false). Αν υπάρχει τότε η συνάρτηση επαναφέρει το task από τη wait λίστα στη deque του επεξεργαστή στον οποίο εκτελούνταν, αλλάζοντας την κατάστασή του από WAITING σε READY και δίνοντάς του την τιμή που περίμενε από το task t που έχει τερματίσει. Σε αυτήν την περίπτωση η συνάρτηση επιστρέφει 1 (δηλαδή true).

➔ *Συναρτήσεις Διαχείρισης των tasks*

- void task_init(int n);

Είναι η συνάρτηση που αρχικοποιεί όλες τις απαραίτητες παραμέτρους που χρειάζεται το RTS για να λειτουργήσει σωστά. Πρέπει να καλείται από τον προγραμματιστή στην αρχή του κώδικά του προτού επιχειρήσει να χρησιμοποιήσει tasks ή οποιαδήποτε συνάρτηση διαχείρισής τους. Ως όρισμα δέχεται το πλήθος των επεξεργαστών που διαθέτει το σύστημα και δεν έχει καμία τιμή επιστροφής. Ακολουθεί τον παρακάτω ψευδοκώδικα.

```
void task_init(int n) {

  dqs_init(); //δέσμευση χώρου των deque για κάθε επεξεργαστή
  mutex_init(); //αρχικοποίηση των mutexes για κάθε επεξεργαστή
  list_init(); //δέσμευση χώρου για τις λίστες ready και wait
```

```
  scheduler_init(); //αρχικοποίηση του ucontext του scheduler
  thread_init(); //δημιουργία ενός pthread για κάθε επεξεργαστή
};
```

Αρχικά δεσμεύεται ο χώρος για όλες τις απαραίτητες δομές του RTS που περιλαμβάνει τις λίστες αποθήκευσης των tasks (deques, wait και ready), καθώς και κάποιες παραμέτρους που έχει κάθε επεξεργαστής (όπως το id του και ένα current_task που δείχνει το task που εκτελείται εκείνη την ώρα σε αυτόν). Επίσης, αρχικοποιείται το context του scheduler. Ο scheduler είναι μια συνάρτηση που εκτελείται σε κάθε επεξεργαστή και αναλαμβάνει τη δρομολόγηση των tasks σε αυτούς. Υπάρχουν τόσα scheduler contexts όσα και επεξεργαστές, αλλά η συνάρτηση δρομολόγησης είναι η ίδια για όλους. Η αρχικοποίηση γίνεται μέσω των συναρτήσεων getcontext και makecontext που περιγράφηκαν παραπάνω. Τέλος, για κάθε επεξεργαστή δημιουργείται ένα pthread. Το pthread αναλαμβάνει να εκτελέσει τη συνάρτηση scheduler κι έτσι ξεκινάει και η δρομολόγηση των tasks στους επεξεργαστές.

- task_t task_create(void *(*func)(void *arg), void *arg);

Αυτή η συνάρτηση καλείται όταν θέλουμε να δημιουργηθεί ένα task. Έχει δυο ορίσματα: τη συνάρτηση που θα εκτελέσει το task και τα ορίσματά της . Η τιμή επιστροφής της είναι το task που δημιουργεί. Ακολουθεί τον παρακάτω ψευδοκώδικα.

```
task_t task_create(void *(*func)(void *arg), void *arg) {

  δέσμευση χώρου για μια δομή task_t;
  αρχικοποίηση των πεδίων της δομής;
  αρχικοποίηση του context του task_t;

  push(task, dqs[current_processor]);
  swapcontext(current_context, scheduler_context[current_processor]);

  return task;
}
```

Ιδιαίτερη σημασία έχει πώς καθορίζεται το context του task. Το μέγεθος της stack είναι το default που έχουμε θεωρήσει για όλα τα contexts (και ίσο με 16*1024) ενώ στο πεδίο uc.uc_link, που είναι το context που θα επιστρέψει ο επεξεργαστής μόλις τερματίσει το task, βάζουμε το context του scheduler που τρέχει ο επεξεργαστής που το δημιουργεί. Στο τέλος και πριν επιστρέψει η συνάρτηση επιλέγουμε να μεταφέρουμε τον έλεγχο από το

current_context στο scheduler. Δεδομένου ότι έχει δημιουργηθεί ένα καινούργιο task έτοιμο για εκτέλεση πρέπει να αποφασίσει ο αλγόριθμος δρομολόγησης σε ποιον επεξεργαστή θα τοποθετηθεί και πότε θα εκτελεστεί. Θα δούμε παρακάτω ότι θα επιλέγουμε τα task που δημιουργούνται να εκτελούνται κατευθείαν στον επεξεργαστή, παίρνοντας τη θέση του πατέρα τους. Βέβαια, το πρώτο task που δημιουργείται δεν έχει πατέρα γι' αυτό και η αλλαγή context δεν γίνεται από το context του task-πατέρα στο context του scheduler, αλλά από ένα αρχικό context init_uc, που έχει δημιουργηθεί μέσα στην task_init() και χρησιμεύει για αυτό το σκοπό.

- void *task_wait(task_t *t);

Η συνάρτηση καλείται από ένα task όταν χρειάζεται να χρησιμοποιήσει μια τιμή που υπολογίζει το task t που παίρνει ως όρισμα. Επιστρέφει αυτήν την τιμή. Σε περίπτωση που το t δεν έχει επιστρέψει, τότε το task που καλεί την task_wait μπαίνει σε κατάσταση αναμονής και τοποθετείται στη λίστα wait μέχρι να τερματίσει το t. Αν η τιμή που ζητά είναι διαθέσιμη, τότε η συνάρτηση επιστρέφει αυτήν την τιμή και το task συνεχίζει να εκτελείται από αυτό το σημείο. Η task_wait ακολουθεί τον παρακάτω ψευδοκώδικα.

```
void *task_wait(task_t *t) {

  wait_value = ready_list_check(t->id);

  if(wait_value == NULL) {
    change(current_task->fields);
    append(current_task,wait);
    swapcontext(current_context,scheduler_context[current_processor]);

    return wait_value;
}
```

Αρχικά ελέγχεται αν το task t που περιμένουμε έχει τερματίσει. Αν όχι, τότε ενημερώνονται τα κατάλληλα πεδία του τρέχοντος task (το state και το wait_for_id) κι έπειτα το task προστίθεται στη wait λίστα, μαζί με όλα τα υπόλοιπα που βρίσκονται σε κατάσταση αναμονής. Τέλος, αλλάζει context και ο έλεγχος μεταβαίνει στη συνάρτηση scheduler που τρέχει ο συγκεκριμένος επεξεργαστής. Όταν το task που ανέμενε τερματίσει, τότε αυτό το task θα επανέλθει στη deque του επεξεργαστή και η εκτέλεσή του θα συνεχίσει από αυτό το σημείο της task_wait(), η οποία θα επιστρέψει την τιμή που είχε ζητήσει το task. Αν τώρα, το task t έχει ήδη τερματίσει, τότε η συνάρτηση επιστρέφει την κατάλληλη τιμή.

- void task_exit(void *retval);

Η συνάρτηση καλείται για να τερματίσει ένα task. Το όρισμα που δέχεται είναι η τιμή επιστροφής του task, που μπορεί να είναι και NULL. Δεν επιστρέφει τίποτα. Ακολουθεί τον παρακάτω ψευδοκώδικα.

```
void task_exit(void *retval) {

  change(current_task->fields);
  i = wait_list_check(current_task);
  if(!i) append(current_task,ready);
  swapcontext(current_context,scheduler_context[current_processor]);
  return;
}
```

Αρχικά ενημερώνονται τα κατάλληλα πεδία του task (το state και το retval). Έπειτα, ελέγχεται αν κάποιο άλλο task περιμένει την τιμή επιστροφής του διατρέχοντας τη wait λίστα. Αν γίνεται κάτι τέτοιο, τότε το task που περίμενε παίρνει την τιμή και επιστρέφει στην deque του επεξεργαστή του για να συνεχίσει την εκτέλεσή του. Αν δεν το περιμένει κανένα, τότε το task που κάλεσε την task_exit() αποθηκεύεται προσωρινά στη ready λίστα μέχρι να ζητηθεί από κάποιο άλλο task η τιμή επιστροφής του (αυτό γίνεται με την task_wait() όπως ήδη έχουμε δει). Στο τέλος, πριν επιστρέψει η συνάρτηση αποφασίσαμε να αλλάζουμε context στο scheduler του τρέχοντος επεξεργαστή. Η επιλογή αυτή έγινε για απλοποίηση κάποιων περιπτώσεων όπου θα έπρεπε να ελέγχουμε αν το task που τερματίζει είχε κλαπεί ή όχι από τον επεξεργαστή, όπως θα δούμε παρακάτω.

ΚΕΦΑΛΑΙΟ 3: WORK SHARING

Ως τώρα έχουμε παρουσιάσει τον τρόπο με τον οποίο παραλληλοποιούμε εργασίες. Τις σπάμε σε ανεξάρτητα tasks, τα οποία εκτελούνται παράλληλα στους επεξεργαστές. Από εδώ και πέρα, θα μελετήσουμε τρόπους με τους οποίους δρομολογούνται τα tasks στους επεξεργαστές. Υπάρχουν δυο βασικές προσεγγίσεις: οι work sharing αλγόριθμοι και οι work stealing αλγόριθμοι. Σε αυτό το κεφάλαιο θα μελετήσουμε τους πρώτους ενώ στο κεφάλαιο 4 θα μελετήσουμε τους δεύτερους.

3.1 Ένα μοντέλο για multitasking υπολογισμούς

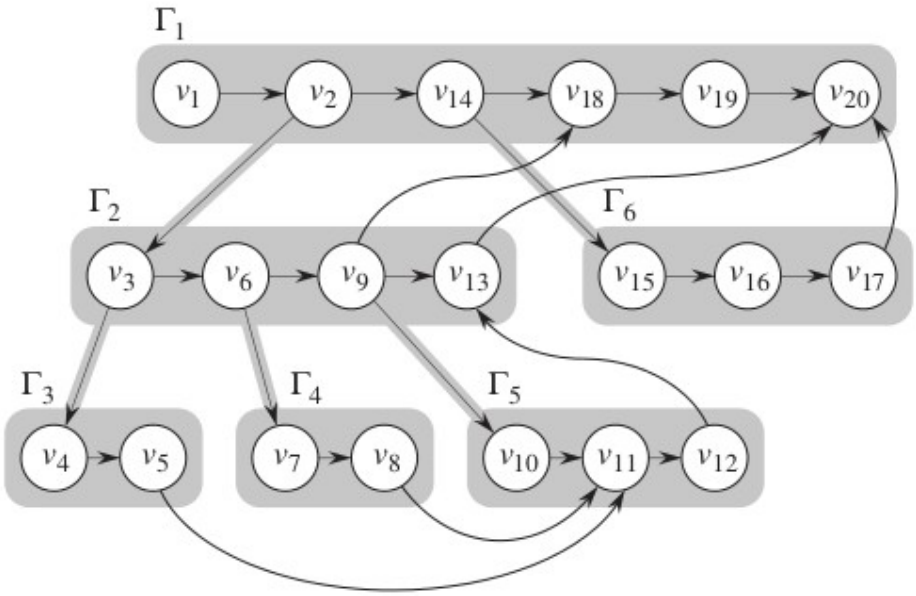
Σε αυτήν την ενότητα θα παρουσιάσουμε ένα μαθηματικό μοντέλο για multitasking υπολογισμούς, με βάση το οποίο θα χτίσουμε ένα θεωρητικό μοντέλο για τον προσδιορισμό της απόδοσης αλγορίθμων δρομολόγησης παράλληλων εργασιών. Στη βιβλιογραφία, αυτή η μελέτη αναφέρεται σε multithreading υπολογισμούς, εννοώντας threads που δρομολογούνται σε χώρο χρήστη [24]. Εμείς προτιμούμε να αναφερόμαστε σε αυτά ως tasks για να μην υπάρχει σύγχυση με τα threads του χώρου πυρήνα, τα οποία διαχειρίζεται το OS.

Ο multitasking υπολογισμός αποτελεί τη ροή των παράλληλων tasks που προκύπτει από το πρόγραμμα όταν αυτό εκτελείται για κάποια είσοδο και εκδηλώνεται με δυναμικό τρόπο κατά τη διάρκεια της εκτέλεσής του. Δεν είναι ισοδύναμος με ένα multitasking πρόγραμμα, αλλά μπορούμε να σκεφτόμαστε ότι είναι το πρόγραμμα μαζί με τα δεδομένα εισόδου.

Ένας βασικός λόγος που προτάθηκε αυτή η μοντελοποίηση είναι γιατί προσφέρει ένα μοντέλο συνέπειας μνήμης, καθώς εργαζόμαστε σε συστήματα κοινής μνήμης. Αρχικά, δημιουργήθηκαν μοντέλα τα οποία ικανοποιούσαν το μοντέλο της σειριακής συνέπειας του Lamport [25]: "Ένα πολυπεξεργαστικό σύστημα είναι σειριακά συνεπές αν το αποτέλεσμα κάθε εκτέλεσης είναι το ίδιο που θα προέκυπτε αν οι εντολές όλων των επεξεργαστών εκτελούνταν με σειριακό τρόπο και οι εντολές καθενός επεξεργαστή εκτελούνται στη σειρά που ορίζει το πρόγραμμα". Στην πραγματικότητα είναι αρκετά δύσκολη και περίπλοκη η υλοποίηση αυτού του μοντέλου, γι' αυτό το λόγο προτάθηκαν άλλα πιο χαλαρά μοντέλα, τα οποία όριζαν τη συνέπεια με βάση τις ενέργειες των επεξεργαστών. Στη Cilk χρησιμοποιείται το μοντέλο της συνέπειας του *dag*, το οποίο είναι πιο χαλαρό από εκείνο του Lamport και είναι πιο εύκολο να υλοποιηθεί σε επίπεδο λογισμικού, όπως παρουσιάζεται και με τον αλγόριθμο συνάφειας μνήμης BACKER για διατήρηση της συνέπειας του *dag* [26].

Η μαθηματική μοντελοποίηση γίνεται με τη χρήση ακυκλικών κατευθυνόμενων

γράφων, στους οποίους θα αναφερόμαστε από δω και στο εξής ως *dag (directed acyclic graph)*. Ένας multitasking υπολογισμός απαρτίζεται από ένα σύνολο από tasks, καθένα από τα οποία είναι μια σειρά από εντολές. Οι εντολές αυτές εκτελούνται σειριακά, η μία μετά την άλλη. Οι κόμβοι του *dag* αντιπροσωπεύουν τις εντολές των tasks (η εντολή i είναι ο κόμβος u_i) και οι ακμές του τις εξαρτήσεις μεταξύ των εντολών γι' αυτό και ονομάζονται *ακμές εξάρτησης*. Στο Σχήμα 8 δίνεται ένα παράδειγμα ενός multitasking υπολογισμού. Τα tasks αναπαριστώνται με τις σκιασμένες περιοχές που παίρνουν ονομασίες Γ_i . Οι εντολές που ανήκουν στο ίδιο task συνδέονται με ακμές, που τις ονομάζουμε *ακμές συνέχειας*, και δηλώνουν τη σειρά με την οποία πρέπει να εκτελεστούν, έτσι ώστε η εντολή στην οποία καταλήγει η ακμή να έπεται εκείνης από την οποία ξεκινάει. Για παράδειγμα το task Γ_6 αποτελείται από τις εντολές $u_{15,}u_{16,}u_{17}$, οι οποίες πρέπει να εκτελεστούν με τη σειρά που έχουν γραφεί.



Σχήμα 8: Ο dag ενός multitasking υπολογισμού

Κατά τη διάρκεια της εκτέλεσης ενός task, αυτό μπορεί να δημιουργήσει (μέσω ενός *spawn* ή μιας *task_create()*) ένα νέο task. Το καινούργιο task θα το αποκαλούμε παιδί, ενώ το task που το δημιούργησε πατέρα. Πατέρας και παιδί είναι πλέον ανεξάρτητα μεταξύ τους, μπορούν να εκτελεστούν παράλληλα και καθένα μπορεί να δημιουργήσει όσα tasks επιθυμεί. Με αυτόν τον τρόπο δημιουργείται μέσα στο *dag* ένα δέντρο από tasks, το οποίο θα ονομάζουμε *spawn tree*. Οι κόμβοι του είναι τα διάφορα tasks, δηλαδή τα σκιασμένα πλαίσια και οι ακμές του, που θα τις ονομάζουμε *spawn ακμές*, στο σχήμα αναπαρίστανται με τις σκιασμένες ακμές. Η αρχή κάθε *spawn ακμής* βρίσκεται στην εντολή του task-πατέρα που δημιούργησε το νέο task και το τέλος της στην αρχική εντολή του task-παιδιού. Δεχόμαστε ως παραδοχή ότι μια εντολή μπορεί να δημιουργήσει μόνο ένα task, δηλαδή από

κάθε κόμβο του *dag* μπορεί να ξεκινάει το πολύ μια *spawn ακμή*. Στο σχήμα το *spawn tree* έχει κόμβους τα tasks $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4, \Gamma_5, \Gamma_6$ με ρίζα το Γ_1 και φύλλα, δηλαδή tasks τα οποία δεν έχουν κανένα παιδί, τα $\Gamma_3, \Gamma_4, \Gamma_5, \Gamma_6$.

Τέλος, για να ολοκληρωθεί ο γράφος, χρειάζονται κάποιες επιπλέον ακμές εξάρτησης, οι οποίες θα δηλώνουν τις εξαρτήσεις που μπορούν να έχουν μεταξύ τους οι εντολές όταν μοιράζονται κάποια δεδομένα, όπως για παράδειγμα η τιμή μιας μεταβλητής που υπολογίζει το παιδί και θέλει να χρησιμοποιήσει κάποια στιγμή ο πατέρας. Αυτές οι ακμές θα ονομάζονται *join ακμές* και αναπαρίστανται με τις καμπυλωτές γραμμές στο *dag*. Όταν κατά την εκτέλεση ενός task φτάσουμε σε μια εντολή στην οποία καταλήγει μια *join ακμή*, τότε χρειάζεται να ελεγχτεί αν έχει εκτελεστεί η εντολή από την οποία ξεκινάει η *join ακμή*. Αν ναι, τότε η εντολή εκτελείται κανονικά, αλλιώς το task αναγκάζεται να μπει σε κατάσταση αναμονής, μέχρι να εκτελεστεί η συγκεκριμένη εντολή. Στο Σχήμα 8, για παράδειγμα, το task Γ_1 θα τεθεί σε κατάσταση αναμονής αν κατά τη διάρκεια της εκτέλεσής του φτάσει στην εντολή u_{18} , ενώ δεν έχει εκτελεστεί ακόμα η u_9 του task Γ_2 .

Για τις *join ακμές* θα κάνουμε δυο παραδοχές: πρώτον ότι δεν μπορεί να καταλήγει μια *join ακμή* στην εντολή που δημιουργεί ένα task και δεύτερον ότι κάθε εντολή έχει το πολύ ένα σταθερό αριθμό από *join ακμές* που καταλήγουν σε αυτή. Μία ακόμη παραδοχή που κάνουμε είναι ότι το task-πατέρας θα μείνει ζωντανό μέχρι όλα τα tasks-παιδιά του να έχουν πεθάνει. Επίσης, κατά τη διάρκεια της εργασίας θα αναφερόμαστε σε *strict* και *fully strict* multitasking υπολογισμούς. Ως *strict* θα ορίζουμε τους υπολογισμούς που όλες οι *join ακμές* άγονται αποκλειστικά από τα παιδιά και καταλήγουν σε κάποιον από τους προγόνους τους, ενώ *ως fully strict* ορίζουμε τους υπολογισμούς στους οποίους όλες οι *join ακμές* άγονται μόνο από τα παιδιά και καταλήγουν στους γονείς τους. Το *dag* που έχει δοθεί ως παράδειγμα αντιπροσωπεύει ένα *fully strict* υπολογισμό.

Με βάση κάθε *dag* μπορούμε να φτιάξουμε σενάρια δρομολόγησης των εντολών ώστε τα tasks να εκτελούνται παράλληλα, δεδομένου ότι κάθε τέτοιο σενάριο θα σέβεται όλες τις εξαρτήσεις που εισάγει ο γράφος. Το σενάριο δρομολόγησης είναι ανεξάρτητο από τα χαρακτηριστικά του συστήματος στο οποίο εφαρμόζεται. Ο μόνος περιορισμός που εισάγει το σύστημα είναι το πλήθος των επεξεργαστών, μιας και σε κάθε βήμα μόνο μια εντολή μπορεί να εκτελείται σε έναν επεξεργαστή. Προσοχή θέλει το γεγονός ότι το σενάριο δρομολόγησης δε λαμβάνει υπόψη του οποιεσδήποτε καθυστερήσεις μπορεί να εισάγει το σύστημα, λόγου χάρη καθυστερήσεις εξαιτίας επικοινωνίας μεταξύ των επεξεργαστών, αλλά υπολογίζει μόνο τις καθυστερήσεις που οφείλονται στις εξαρτήσεις των εντολών που φαίνονται στο *dag*.

step	living threads						processor activity		
							p_1	p_2	p_3
1	$\mathbf{\Gamma_1}$						v_1		
2	$\mathbf{\Gamma_1}$						v_2		
3	$\mathbf{\Gamma_1}$	$\mathbf{\Gamma_2}$					v_3	v_{14}	
4	Γ_1	$\mathbf{\Gamma_2}$	$\mathbf{\Gamma_3}$			$\mathbf{\Gamma_6}$	v_4	v_6	v_{15}
5	Γ_1	$\mathbf{\Gamma_2}$	$\mathbf{\Gamma_3}$	$\mathbf{\Gamma_4}$		$\mathbf{\Gamma_6}$	v_5	v_9	v_{16}
6	$\mathbf{\Gamma_1}$	Γ_2		$\mathbf{\Gamma_4}$	$\mathbf{\Gamma_5}$	$\mathbf{\Gamma_6}$	v_7	v_{10}	v_{17}
7	$\mathbf{\Gamma_1}$	Γ_2		$\mathbf{\Gamma_4}$	Γ_5		v_8	v_{18}	
8	$\mathbf{\Gamma_1}$	Γ_2			$\mathbf{\Gamma_5}$			v_{19}	v_{11}
9	Γ_1	Γ_2			$\mathbf{\Gamma_5}$				v_{12}
10	Γ_1	$\mathbf{\Gamma_2}$							v_{13}
11	$\mathbf{\Gamma_1}$								v_{20}

Σχήμα 9: Μια δρομολόγηση του υπολογισμού του Σχήματος 8 σε ένα σύστημα τριών επεξεργαστών [24]

Ένα σενάριο δρομολόγησης του παραδείγματός μας δίνεται στο Σχήμα 9. Πρόκειται για τη δρομολόγηση των tasks (τα οποία αναφέρονται ως threads στον πίνακα) σε 3 επεξεργαστές. Η πρώτη στήλη είναι τα βήματα της εκτέλεσης, η δεύτερη τα tasks τα οποία είναι ζωντανά και η τρίτη οι εντολή που εκτελείται στον αντίστοιχο επεξεργαστή. Τα tasks που είναι με έντονη γραμματοσειρά εκτελούνται, ενώ τα άλλα βρίσκονται για κάποιο λόγο σε αναμονή. Για παράδειγμα στο βήμα 4, το task Γ_1 βρίσκεται στην ουρά κάποιου επεξεργαστή γιατί έχουν δημιουργηθεί τα tasks $\Gamma_2, \Gamma_3, \Gamma_6$, τα οποία εκτελούνται εκείνη τη στιγμή στους 3 επεξεργαστές. Ο αλγόριθμος δρομολόγησης στο συγκεκριμένο παράδειγμα επέλεξε σε περίπτωση συναγωνισμού των tasks, να εκτελείται πρώτα το παιδί, εκείνο δηλαδή που βρίσκεται σε χαμηλότερο επίπεδο στο *spawn tree*. Ένας άλλος λόγος που βρίσκεται σε αναμονή ένα task είναι όταν θέλει να εκτελέσει μια εντολή η οποία όμως έχει μια *join* εξάρτηση από μια άλλη εντολή ενός άλλου task. Στο Σχήμα 9 αυτό φαίνεται στο βήμα 9, όπου το task Γ_1 θέλει να εκτελέσει την εντολή u_{20}, η οποία περιμένει να εκτελεστεί πρώτα η εντολή u_{13} του task Γ_2 και επιπλέον το task Γ_2 περιμένει πρώτα να εκτελεστεί η εντολή u_{12} του task Γ_5.

Με βάση το *dag* μπορούμε να φτιάξουμε όλα τα πιθανά σενάρια δρομολόγησης, δηλαδή με ποια σειρά θα επιλέγονται τα tasks να εκτελούνται κάθε φορά και σε ποιον επεξεργαστή. Εμείς ενδιαφερόμαστε να δρομολογήσουμε τα tasks με τέτοιο τρόπο ώστε να αξιοποιούμε στο μεγαλύτερο δυνατό βαθμό όλους τους επεξεργαστές κι έτσι να αυξάνουμε

την απόδοση του συστήματος. Έχουν προταθεί διαφόρων ειδών αλγόριθμοι κι έχουν γίνει αρκετές αναλύσεις πάνω στην απόδοσή τους και αυτό το ζήτημα θα μελετήσουμε στις παρακάτω ενότητες.

3.2 Χρονική και Χωρική Μελέτη Αλγορίθμων Χρονοδρομολόγησης

Σε αυτήν την ενότητα θα μελετήσουμε την απόδοση -χρονική και χωρική- που έχουν κάποιοι αλγόριθμοι δρομολόγησης των tasks, βασισμένοι πάνω στο *dag* που κατασκευάσαμε στην προηγούμενη ενότητα. Παρόλο που ο παράλληλος προγραμματισμός άρχισε να εξελίσσεται σε επίπεδο software από τη δεκαετία του '90 κι έπειτα, οι βάσεις για την επίδοση τέτοιων αλγορίθμων έχουν τεθεί ήδη από τη δεκαετία του '70 με τις ανεξάρτητες εργασίες των Brent [27] και Graham [28].

Αρχικά, θα εισάγουμε δυο βασικές έννοιες: το *critical-path* και το *work*. Ως *critical-path* του υπολογισμού ορίζουμε το μήκος του μεγαλύτερου μονοπατιού του *dag* και ως *work* το πλήθος των κόμβων του. Έτσι, ο γράφος του Σχήματος 9 έχει *critical-path* μήκους 10 (η διαδρομή $u_1 u_2 u_3 u_6 u_7 u_8 u_{11} u_{12} u_{13} u_{20}$) και *work* μήκους 20. Για έναν υπολογισμό θα ορίσουμε ως $T(X)$ το χρόνο που χρειάζεται να πραγματοποιηθεί δεδομένου ενός σεναρίου δρομολόγησης X. Για P επεξεργαστές θα ορίζουμε

$$T_P = min T(X)$$

τον ελάχιστο χρόνο εκτέλεσης του υπολογισμού για το βέλτιστο σενάριο δρομολόγησης. Ως *work* χρόνο του υπολογισμού θα χρησιμοποιούμε επομένως το συμβολισμό T_1 , αφού ένας επεξεργαστής μπορεί να εκτελεί μια μόνο εντολή ανά βήμα, και ως *critical-path* χρόνο το συμβολισμό T_∞ , δεδομένου ότι για άπειρο πλήθος επεξεργαστών κάθε εντολή στο *critical-path* πρέπει να εκτελεστεί σειριακά, λόγω των εξαρτήσεών τους. Είναι προφανές ότι θα ισχύουν τα δυο παρακάτω όρια:

$$T_P \geq \frac{T_1}{P} \quad \text{και} \quad T_P \geq T_\infty$$

αφού P επεξεργαστές μπορούν να εκτελούν το πολύ P εντολές σε κάθε βήμα και η απόδοση θεωρητικά άπειρων επεξεργαστών είναι καλύτερη από πεπερασμένο πλήθος. Θα αποδείξουμε ότι για *άπληστους αλγορίθμους* ισχύει $T_P \leq \frac{T_1}{P} + T_\infty$. *Άπληστοι αλγόριθμοι* ορίζονται εκείνοι που σε κάθε βήμα της εκτέλεσης αν τουλάχιστον P εντολές είναι έτοιμες, τότε εκτελούνται οι P, αλλιώς αν είναι λιγότερες τότε εκτελούνται όλες, δεδομένων P επεξεργαστών. Έτσι προκύπτει το εξής θεώρημα:

Θεώρημα άπληστης δρομολόγησης (The greedy scheduling theorem)

Για κάθε multitasking υπολογισμό με *work* T_1 και *critical-path* T_∞ και για κάθε αριθμό επεξεργαστών P, κάθε άπληστος αλγόριθμος δρομολόγησης X των tasks επιτυγχάνει χρόνο

$$T(X) \leq \frac{T_1}{P} + T_\infty \quad (1)$$

Απόδειξη

Ας θεωρήσουμε το *dag* $G=(V,E)$, όπου V το πλήθος των κόμβων-εντολών και E το πλήθος των ακμών-εξαρτήσεων. Τότε θα ισχύει ότι *work* $T_1=|V|$ και το μεγαλύτερο μονοπάτι του *dag* έχει μήκος T_∞. Θεωρούμε επίσης ένα άπληστο σενάριο δρομολόγησης X των εντολών σε ένα σύστημα P επεξεργαστών και θα ονομάσουμε V'_i με i = 1,2,...,k το σύνολο των εντολών που εκτελούνται στο βήμα i (όπου $k=T(X)$). Έτσι, το σύνολο των κόμβων-εντολών V'_i δημιουργούν μια διαμέριση του V.

Για κάθε βήμα i της εκτέλεσης δημιουργείται η αλληλουχία γράφων $G_0, G_1, G_2, \ldots, G_k$ όπου $G_0=G$ και για κάθε i αφαιρούμε από το *dag* τη διαμέριση V'_i, οπότε προκύπτει γράφος G_i υπογράφος του G_{i-1} με κόμβους $V_i=V_{i-1}-V'_i$. Με άλλα λόγια σε κάθε βήμα δημιουργείται ένα νέο *dag,* υπογράφος του αρχικού, με την αφαίρεση όλων των κόμβων-εντολών που εκτελούνται από τους επεξεργαστές με βάση το σενάριο δρομολόγησης X και όλων των ακμών-εξαρτήσεων που τους ενώνουν με τους υπόλοιπους κόμβους. Σε κάθε βήμα της εκτέλεσης, θα δείξουμε ότι είτε μειώνεται το μέγεθος του *dag* είτε το μέγεθος του *critical-path.*

Ας θεωρήσουμε ότι σε ένα βήμα της εκτέλεσης υπάρχουν P εντολές έτοιμες να εκτελεστούν. Αυτό σημαίνει ότι $V'_i=P$. Σε αυτήν την περίπτωση $V_i=V_{i-1}-V'_i=V_{i-1}-P$ και επειδή $T_1=|V|$ προκύπτει ότι μπορούν να υπάρξουν το πολύ $\frac{T_1}{P}$ τέτοια βήματα, αφού αφαιρούνται κάθε φορά P εντολές από το *dag.* Αν θεωρήσουμε ότι σε κάποιο βήμα εκτέλεσης υπάρχουν λιγότερες από P εντολές έτοιμες να εκτελεστούν, δηλαδή $V'_i<P$, τότε, δεδομένου ότι η δρομολόγηση είναι άπληστη, το σύνολο V'_i των κόμβων που θα αφαιρεθούν περιέχει αναγκαστικά όλους τους κόμβους με βαθμό εισόδου 0, δηλαδή όλους τους κόμβους που δεν έχουν καμία εξάρτηση. Αυτοί οι κόμβοι ανήκουν υποχρεωτικά στο *critical-path* και επειδή κάθε υπογράφος έχει τουλάχιστον ένα τέτοιο (τον πρώτο ή αλλιώς τη ρίζα), τότε σε κάθε τέτοιο βήμα θα

μειώνεται τουλάχιστον κατά ένα το μέγεθος του *critical-path,* με αποτέλεσμα να χρειάζονται το πολύ T_∞ τέτοια βήματα.

Συνδυάζοντας τα δυο παραπάνω ευρήματα καταλήγουμε ότι ο συνολικός χρόνος εκτέλεσης είναι το πολύ το άθροισμα αυτών των δυο ορίων, δηλαδή

$$T(X) \leq \frac{T_1}{P} + T_\infty$$

Αυτό το θεώρημα μας αποκαλύπτει δυο συμπεράσματα. Πρώτον, εγκλωβίζει το χρόνο εκτέλεσης ανάμεσα σε δυο όρια:

$$max\left\{\frac{T_1}{P}, T_\infty\right\} \leq T_P \leq 2\, max\left\{\frac{T_1}{P}, T_\infty\right\} \quad (2)$$

αφού ο βέλτιστος χρόνος που μπορεί να επιτύχει οποιοσδήποτε αλγόριθμος δρομολόγησης είναι $\frac{T_1}{P}$ όταν η εφαρμογή μπορεί να παραλληλοποιηθεί εντελώς ή T_∞ εξαιτίας των εξαρτήσεων που μπορεί να υπάρχουν ανάμεσα στις εντολές κι έτσι να μην μπορεί να παραλληλοποιηθεί πέραν αυτού του βαθμού. Δεύτερον, το θεώρημα μας υποδεικνύει πότε ένας άπληστος αλγόριθμος μπορεί να πετύχει γραμμική επιτάχυνση σε σχέση με το πλήθος των επεξεργαστών -που είναι και η μέγιστη-, πότε δηλαδή μπορούμε να βρούμε έναν σενάριο δρομολόγησης έτσι ώστε $T(X) = \Theta(\frac{T_1}{P})$. Κι αυτό γίνεται όταν:

$$T_\infty < \frac{T_1}{P} \Rightarrow P < \frac{T_1}{T_\infty}$$

ο αριθμός των επεξεργαστών P δεν είναι μεγαλύτερος από το λόγο $\frac{T_1}{T_\infty}$, που σημαίνει ότι για έναν άπληστο αλγόριθμο η σχέση (2) θα γίνεται

$$T(X) \leq 2\frac{T_1}{P} \quad . \blacksquare$$

Τώρα θα μελετήσουμε τη χωρική απόδοση αλγορίθμων, δηλαδή το μέγεθος της μνήμης που χρειάζεται να δεσμεύσουν. Για αυτή τη μελέτη δεν μας ενδιαφέρουν οι κόμβοι-εντολές του *dag*, αλλά το *spawn tree* που προκύπτει, όταν αντιμετωπίζουμε τα tasks ως κόμβους. Κι αυτό γιατί μνήμη δεσμεύουμε όταν δημιουργούμε ένα task. Σε αυτό το δέντρο οι ακμές που συνδέουν τα tasks είναι οι *join ακμές*. Θα ορίσουμε ως *εγγραφή δραστηριοποίησης (activation frame)* το χώρο μνήμης που δεσμεύεται με τη δημιουργία ενός task ώστε να αποθηκεύει τις μεταβλητές του. Επίσης, ορίζουμε ως *βάθος δραστηριοποίησης (activation depth)* ενός task το άθροισμα των *εγγραφών δραστηριοποίησης* όλων των προγόνων του task, συμπεριλαμβανομένου και του ίδιου. Το *βάθος δραστηριοποίησης* για ένα multitasking

υπολογισμό είναι ίσο με το μέγιστο *βάθος δραστηριοποίησης* για κάθε task.

Θα συμβολίζουμε $S(X)$ το χώρο μνήμης που χρειάζεται ένα σύστημα P επεξεργαστών δεδομένου ενός σεναρίου δρομολόγησης X . Ο χώρος μνήμης $S(X)$ αντιπροσωπεύει το μέγιστο άθροισμα των *εγγραφών δραστηριοποίησης* των tasks που είναι ζωντανά σε κάποιο βήμα. Κάθε εκτέλεση σε P επεξεργαστές μπορεί να προσομοιωθεί με την ίδια εκτέλεση σε έναν επεξεργαστή χωρίς να χρησιμοποιείται επιπλέον χώρος, οπότε ισχύει $S_1 \leq S(X)$, όπου $S_1 = minS(X)$ για κάθε σενάριο δρομολόγησης X . Δηλαδή S_1 δηλώνουμε τον ελάχιστο χώρο μνήμης που θα χρησιμοποιήσει μια εκτέλεση σε έναν επεξεργαστή.

Θεώρημα (Χωρική Απόδοση Αλγορίθμων Δρομολόγησης)

Έστω A το *βάθος δραστηριοποίησης* ενός multitasking υπολογισμού και X ένα σενάριο δρομολόγησης της εκτέλεσης σε P επεξεργαστές. Τότε θα ισχύει:

$$S(X) \geq A$$

και πιο συγκεκριμένα

$$S_1 \geq A \quad .$$

Απόδειξη

Κατά τη διάρκεια οποιουδήποτε σεναρίου εκτέλεσης, κάποια στιγμή θα δημιουργηθεί το task που βρίσκεται στο χαμηλότερο επίπεδο του *spawn tree*. Εκείνη τη στιγμή, έχει δεσμευτεί στη μνήμη χώρος τουλάχιστον ίσος με το άθροισμα των *εγγραφών δραστηριοποίησης* κάθε task προγόνου μαζί με το τρέχων task, αφού έχουμε θεωρήσει ότι όταν ένα task είναι ζωντανό, τότε και όλοι οι πρόγονοί του είναι επίσης ζωντανοί. Με αυτόν τον τρόπο, όμως, έχουμε ορίσει το *βάθος δραστηριοποίησης,* οπότε για εκείνη τη στιγμή σίγουρα $S(X) \geq A$. Δεδομένου ότι το όριο ισχύει για κάθε σενάριο δρομολόγησης σε P επεξεργαστές, θα ισχύει και για το σενάριο εκτέλεσης που χρησιμοποιεί τον ελάχιστο δυνατό χώρο μνήμης, δηλαδή αυτόν για ένας επεξεργαστή, οπότε $S_1 \geq A$. ■

Το ερώτημα που προκύπτει από αυτό το θεώρημα είναι για ποιους αλγορίθμους δρομολόγησης μπορούμε να πετύχουμε το χαμηλότερο όριο όπου $S_1 = A$. Στη γενική περίπτωση αυτό δεν είναι εφικτό, καθώς μπορεί τα tasks να έχουν ένα κύκλο από εξαρτήσεις μεταξύ τους που να τα υποχρεώνει να είναι όλα ζωντανά σε κάποιο σημείο της εκτέλεσης ανεξάρτητα από τον αλγόριθμο δρομολόγησης. Γι' αυτό το λόγο θα εισάγουμε την έννοια του *depth-first υπολογισμού*, στον οποίο μπορεί να επιτευχθεί αυτό το όριο, στην περίπτωση της σειριακής εκτέλεσης σε έναν επξεργαστή. Ο *depth-first υπολογισμός* είναι

ένας multitasking υπολογισμός ο οποίος ακολουθεί μια από αριστερά προς τα δεξιά αναζήτηση των tasks του *spawn tree* και εκτελεί πρώτα ένα task από το οποίο εξαρτώνται άλλα. Μια σειριακή *depth-first* εκτέλεση ξεκινάει με τη ρίζα του *spawn tree* και εκτελεί όλες τις εντολές του μέχρι μια από αυτές να δημιουργήσει ένα νέο task ή μέχρι να τερματίσει το task. Στην περίπτωση γέννησης, το αρχικό task παραμερίζεται και τη θέση στην εκτέλεσή του παίρνει το task-παιδί, μέχρι αυτό να ολοκληρώσει την εκτέλεσή του, οπότε και επανέρχεται το task-πατέρας για να συνεχίσει από το σημείο που είχε σταματήσει. Αποδεικνύεται πολύ απλά ότι, αφού ανά πάσα στιγμή της εκτέλεσης υπάρχει μονάχα ένα μονοπάτι από τη ρίζα προς κάποιο task, ο χώρος που χρειάζεται είναι το πολύ ίσος με το *βάθος δραστηριοποίησης*. Επομένως, για κάθε *depth-first υπολογισμό* ισχύει $S_1 = A$.

Το θέμα είναι τι χωρική απόδοση έχουμε όταν πρόκειται για κάποιο σενάριο εκτέλεσης σε P επεξεργαστές. Αν συγκρίνουμε το χώρο που χρησιμοποιεί ένα σενάριο δρομολόγησης σε P επεξεργαστές με το βέλτιστο σενάριο δρομολόγησης σε έναν επεξεργαστή, θα δούμε ότι μπορεί να χρησιμοποιηθεί ακόμα και P φορές περισσότερος χώρος. Δηλαδή, μπορεί να έχουμε γραμμική αύξηση του χώρου μνήμης που χρησιμοποιούμε. Αυτό το γεγονός είναι ορατό αν αναλογιστούμε έναν υπολογισμό όπου το task της ρίζας είναι ένα επαναληπτικό loop που γεννά σε κάθε επανάληψή του από ένα παιδί. Όταν πρόκειται για εκτέλεση σε έναν επεξεργαστή, ο μέγιστος χώρος που θα χρησιμοποιηθεί είναι αυτός της ρίζας συν του παιδιού. Όταν το παιδί τερματίσει και ο έλεγχος επιστρέψει στη ρίζα, τότε ο χώρος που δέσμευε το παιδί ελευθερώνεται και ο πατέρας συνεχίζει την εκτέλεσή του γεννώντας ένα νέο παιδί. Στην περίπτωση, όμως, των P επεξεργαστών, μπορούν να εκτελεστούν ταυτόχρονα P επαναλήψεις αυτού του loop -μία σε κάθε επεξεργαστή. Αυτό σημαίνει ότι κάποια στιγμή, μπορεί να γεννηθούν ταυτόχρονα P παιδιά, οπότε ο χώρος μνήμης που θα δεσμευτεί θα είναι P φορές μεγαλύτερος απ' ότι στην εκτέλεση σε έναν επεξεργαστή. Αυτό σημαίνει ότι για μια δρομολόγηση X σε P επεξεργαστές θα ισχύει $S(X) = \Theta(S_1 P)$.

Συνολικά, αν συνδυάσουμε τη χρονική και τη χωρική απόδοση σεναρίων δρομολόγησης σε P επεξεργαστές προκύπτει

$$T(X)S(X) = O(T_1/P)O(S_1 P) = O(T_1 S_1) \quad (3)$$

Είναι, λοιπόν, σαν κάθε επεξεργαστής του συστήματος να χρειάζεται το χώρο που θα χρειαζόταν ένας επεξεργαστής, το οποίο τελικά είναι αρκετά δίκαιο. Το βασικό πρόβλημα που προκύπτει και έχει αποδειχθεί από τους R.D. Blumofe και C.E. Leiserson είναι ότι στη γενική περίπτωση multitasking υπολογιγμών είναι αδύνατον να επιτύχουμε ταυτόχρονα γραμμική επιτάχυνση στο χρόνο και γραμμική αύξηση στον αναγκαίο χώρο μνήμης [24].

3.3 Work Sharing Αλγόριθμοι

Εδώ θα μελετήσουμε θεωρητικά δυο αλγορίθμους χρονοδρομολόγησης παράλληλων εργασιών, οι οποίοι ανήκουν στην κατηγορία των work sharing αλγορίθμων. Στους work sharing αλγορίθμους, όποτε ένας επεξεργαστής γεννάει καινούργια tasks, ο δρομολογητής (scheduler) αναλαμβάνει να τα αναθέσει στους άλλους επεξεργαστές με στόχο να μοιράσει το υπολογιστικό φορτίο σε όλους τους επεξεργαστές κι έτσι κανείς να μη μένει ανεκμετάλλευτος.

Κάθε αλγόριθμος δρομολόγησης πρέπει να διαμορφώσει ένα σενάριο δρομολόγησης για P επεξεργαστές δεδομένου ενός multitasking υπολογισμού. Ένα ζήτημα είναι ότι ο υπολογισμός, δηλαδή το *dag* δεν είναι γνωστό εκ των προτέρων, αλλά δημιουργείται δυναμικά κατά τη διάρκεια εκτέλεσης του προγράμματος. Ο δρομολογητής τρέχει παράλληλα με το πρόγραμμα και λαμβάνει τις αποφάσεις του δυναμικά για το σε ποιον επεξεργαστή θα ανατεθούν τα ζωντανά tasks. Έτσι σε κάθε στιγμή της εκτέλεσης, ο δρομολογητής διαθέτει ένα σύνολο από ζωντανά tasks, τα οποία μπορεί να είναι είτε έτοιμα προς εκτέλεση είτε μπλοκαρισμένα λόγω κάποιου συμβάντος (πχ. κάποια εξάρτηση όπως είδαμε προηγουμένως). Κάθε task μπορεί να φέρει κάποια ειδική πληροφορία βάσει της οποίας ο δρομολογητής μπορεί τελικά να αποφασίσει τον τελικό προορισμό του, αλλά σίγουρα ο δρομολογητής δεν μπορεί να γνωρίζει το ποσοστό της εκτέλεσης που απομένει.

Όπως είπαμε στη γενική περίπτωση multitasking υπολογισμών είναι αδύνατο να επιτευχθεί γραμμική αύξηση ταχύτητας με γραμμική αύξηση χώρου (σχέση 3) από κάποιον αλγόριθμο [24]. Αυτό, όμως, δεν ισχύει όταν περιοριζόμαστε στην κατηγορία των *strict* υπολογισμών όπου όλες οι εξαρτήσεις των tasks οδεύουν από τα παιδιά προς τους προγόνους τους. Οι *strict* υπολογισμοί είναι και *depth-first*, αφού δεν υπάρχουν εξαρτήσεις μεταξύ παιδιών του ίδιου task. Όταν γεννιέται ένα task Γ σε ένα *strict* υπολογισμό, ένας επεξεργαστής μπορεί να ολοκληρώσει την εκτέλεση τη δική του και όλων των απογόνων του χρησιμοποιώντας μια *depth-first* δρομολόγηση, ακόμη κι αν δεν εκτελείται κανένα άλλο task σε κανέναν άλλο επεξεργαστή. Με άλλα λόγια, από τη στιγμή που δημιουργείται ένα task Γ, μέχρι τη στιγμή που θα πεθάνει, θα υπάρχει πάντα ένα τουλάχιστον task έτοιμο προς εκτέλεση. Κι αυτό ακριβώς είναι το γεγονός που μας επιτρέπει να σχεδιάσουμε αλγορίθμους που να αληθεύουν τη σχέση 3.

➔ *Αλγόριθμος GDF (Global Depth-First)*

Ο αλγόριθμος GDF διατηρεί όλα τα ζωντανά tasks σε μια καθολική ουρά, στην οποία έχουν πρόσβαση όλοι οι επεξεργαστές. Υψηλότερη προτεραιότητα έχει το task που βρίσκεται σε μεγαλύτερο βάθος στην ουρά. Ο δρομολογητής σε κάθε βήμα παίρνει το πολύ P έτοιμα tasks που υπάρχουν στην ουρά ανάλογα με το πόσοι επεξεργαστές είναι άδειοι και

ζητούν εργασία. Κάθε επεξεργαστής εκτελεί μια εντολή από το task που του έχει ανατεθεί και στο τέλος του βήματος, ο δρομολογητής επιστρέφει όλα τα tasks που παραμένουν ζωντανά (κάποια μπορεί να έχουν μόλις τερματίσει) και όσα μόλις δημιουργήθηκαν πίσω στην καθολική ουρά. Ο τρόπος που εισάγονται είναι αυτός που θα καθορίσει και την προτεραιότητά τους, γι' αυτό και πρώτα μπαίνουν τα νεοδημιουργηθέντα tasks.

Μιας και αυτός ο αλγόριθμος είναι άπληστος θα ικανοποιεί τη σχέση 1 και η χρονική επιτάχυνση θα έχει άνω όριο:

$$T(X) \leq \frac{T_1}{P} + T_\infty \quad .$$

Για τη χωρική απόδοσή του θα αποδείξουμε ότι χρησιμοποιεί χώρο το πολύ

$$S(X) \leq S_1 P$$

για P επεξεργαστές και για κάθε σενάριο δρομολόγησης X που μπορεί να δημιουργήσει ο GDF. Ισοδύναμα, θα δείξουμε ότι ποτέ η ουρά του δρομολογητή δεν περιέχει περισσότερα από P tasks (ενεργά ή σταματημένα).

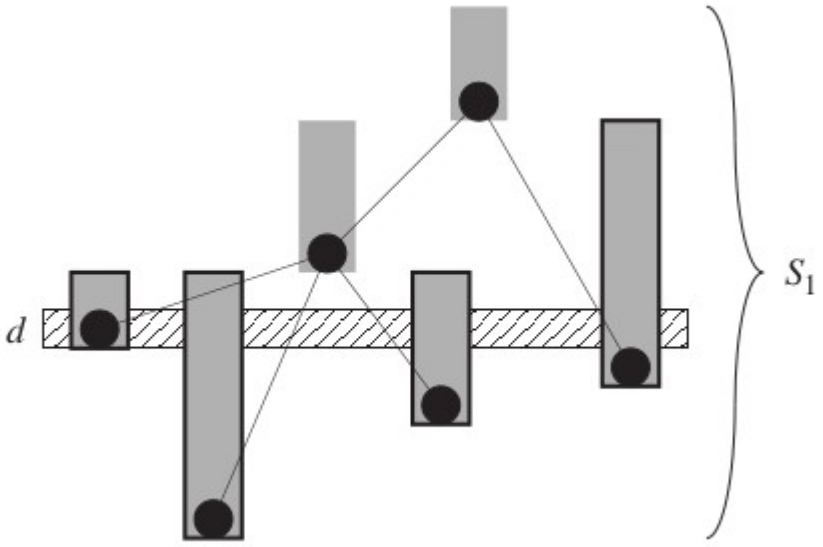
Απόδειξη

Θα λέμε ότι ένα task Γ συγκεντρώνει *βάθος δραστηριοποίησης d*, όταν για το *βάθος δραστηριοποίησής* του $A(\Gamma)$ ισχύει $A(\Gamma) \geq d$ και είτε το Γ είναι ρίζα του *spawn tree* είτε για τον πατέρα του Γ' ισχύει $A(\Gamma') < d$. Επίσης, θα συμβολίζουμε $s(t,d)$ τον αριθμό των ζωντανών tasks που έχουν *βάθος δραστηριοποίησης* τουλάχιστον *d* στην αρχή του βήματος *t*. Έτσι, ο συνολικός χώρος μνήμης που καταναλώνεται σε κάποιο βήμα *t* είναι:

$$s(t) = \sum_{d=1}^{S_1} s(t,d) \quad (4)$$

Με βάση τους καινούργιους συμβολισμούς, θα δείξουμε ότι $s(t) \leq S_1 P$ για κάθε βήμα *t* του αλγορίθμου του δρομολογητή.

Σχηματικά, μπορούμε να δούμε χωρικά το *spawn tree* του *dag* του σχήματος 8, στο παρακάτω σχήμα (Σχήμα 10). Κάθε task αναπαρίσταται ως ένα γκριζαρισμένο ορθογώνιο παραλληλόγραμμο το ύψος του οποίου δηλώνει το μέγεθος της *εγγραφής δραστηριοποίησής* του. Τοποθετείται στο σχήμα έτσι ώστε η κορυφή του να είναι ευθυγραμμισμένη με το τέλος της *εγγραφής δραστηριοποίησής* του πατέρα του, όπου βρίσκεται ένας μαύρος κόμβος. Στο σχήμα φαίνεται ότι τα tasks συγκεντρώνουν *βάθος δραστηριοποίησης d,* όπου d είναι το ελάχιστο βάθος που συγκεντρώνουν τα φύλλα του *spawn tree.*



Σχήμα 10: *Spawn tree* όπου εμφανίζεται το *βάθος δραστηριοποίησης*

Για την απόδειξη θα ακολουθήσουμε τον αλγόριθμο κατά βήμα και θα ακολουθήσουμε τη μέθοδο αναγωγής.

- Αρχικά για t = 1 ο GDF ξεκινάει με ένα μόνο ζωντανό task, τη ρίζα του *spawn tree,* οπότε για κάθε d ισχύει $s(1,d) \leq 1 \leq P$.
- Υποθέτουμε ότι για ένα οποιοδήποτε βήμα t του GDF ισχύει η σχέση $s(t,d) \leq P$.
- Θα αποδείξουμε ότι αυτή η σχέση ισχύει και για το βήμα t+1. Από την υπόθεση του δεύτερου βήματος προκύπτει ότι στο βήμα t υπάρχουν τουλάχιστον $s(t,d)$ task ζωντανά και έτοιμα προς εκτέλεση που συγκεντρώνουν *βάθος δραστηριοποίησής* μεγαλύτερο ή ίσο με d. Το ότι είναι έτοιμα προς εκτέλεση προκύπτει από τη θεώρηση ότι ο υπολογισμός είναι *strict* κι έτσι όλες οι εξαρτήσεις πηγαίνουν μόνο από τα παιδιά προς τους προγόνους. Δεδομένου ότι ο GDF είναι *depth-first* αλγόριθμος σημαίνει ότι μεγαλύτερη προτεραιότητα προς εκτέλεση έχουν τα tasks με μεγαλύτερο βάθος, οπότε το πολύ να εκτελεστούν $P - s(t,d)$ που συγκεντρώνουν *βάθος δραστηριοποίησής* μικρότερο από d. Για να αυξηθεί ο αριθμός των tasks που συγκεντρώνουν βάθος μεγαλύτερο από d πρέπει να δημιουργήσουν παιδιά τα tasks που συγκεντρώνουν μικρότερο βάθος. Άρα το πολύ να δημιουργηθούν $P - s(t,d)$ νέα tasks με *βάθος δραστηριοποίησης* μεγαλύτερο από d. Οπότε στην αρχή του βήματος t+1 το πολύ να υπάρχουν $s(t,d) + (P - s(t,d)) = P$ ζωντανά tasks να συγκεντρώνουν *βάθος δραστηριοποίησής* μεγαλύτερο ή ίσο με d. Οπότε $s(t+1,d) \leq P$. ■

Συμπερασματικά, ο αλγόριθμος GDF έχει:

$$T(X) \leq \frac{T_1}{P} + T_\infty \quad \text{χρονική απόδοση}$$

και

$$S(X) \leq S_1 P \quad \text{χωρική απόδοση.}$$

➔ *Busy-Leaves Αλγόριθμος*

Και σε αυτήν την περίπτωση μελετάμε ένα δυναμικό αλγόριθμο με την έννοια ότι για τη δρομολόγηση των tasks στο t βήμα, ο αλγόριθμος χρησιμοποιεί μόνο τη γνώση που έχει από τα προηγούμενα t-1 βήματα και καμία πληροφορία από τις εντολές που ακόμα δεν έχουν εκτελεστεί. Τα tasks αποθηκεύονται και εδώ σε μια καθολική ουρά, στην οποία έχουν ομοιόμορφη πρόσβαση όλοι οι επεξεργαστές.

Ο αλγόριθμος ξεκινάει με όλους τους επεξεργαστές άδειους και στην καθολική ουρά μόνο το task-ρίζα του *spawn tree*. Στην αρχή κάθε βήματος κάθε επεξεργαστής είτε θα είναι άδειος είτε θα εκτελεί κάποιο task. Οι άδειοι επεξεργαστές εκκινούν το βήμα κάνοντας απόπειρα να πάρουν ένα task από την ουρά. Αν υπάρχουν περισσότερα tasks έτοιμα προς εκτέλεση απ' ότι επεξεργαστές (η προσφορά μεγαλύτερη από τη ζήτηση) τότε στο τέλος του βήματος κάθε επεξεργαστής θα έχει από ένα task προς εκτέλεση. Αλλιώς, κάποιοι θα παραμείνουν άδειοι. Ο τρόπος με τον οποίο καθορίζεται η προτεραιότητά των επεξεργαστών για την πρόσβαση στην ουρά δεν μας απασχολεί εδώ. Καθένας επεξεργαστής εκτελεί τις εντολές ενός task με τη σειρά που ορίζει το *dag* μέχρι να συμβεί ένα από τα τρία: το task κάνει *spawn*, μπαίνει σε κατάσταση αναμονής ή πεθαίνει. Ανάλογα με την περίπτωση, ο Busy-Leaves αλγόριθμος ακολουθεί έναν από τους τρεις κανόνες:

(1) *Spawn:* Όταν ένα task-πατέρας Γ_a δημιουργεί ένα task-παιδί Γ_b , ο επεξεργαστής επιστρέφει το Γ_a πίσω στην καθολική ουρά στο τέλος του βήματος και εκκινεί το επόμενο βήμα εκτελώντας το Γ_b .

(2) *Αναμονή:* Όταν ένα task Γ_a μπει σε κατάσταση αναμονής (επειδή συναντάει κάποια εξάρτηση δεδομένων), ο επεξεργαστής επιστρέφει το Γ_a στην ουρά στο τέλος του βήματος και εκκινεί το επόμενο βήμα άδειος.

(3) *Θάνατος:* Όταν ένα task Γ_a πεθάνει, ο επεξεργαστής ελέγχει αν το task-πατέρας του Γ_b έχει κάποιο άλλο ζωντανό παιδί στην ουρά. Αν ναι, τότε ξεκινάει το

επόμενο βήμα άδειος. Αν όχι και αν επιπλέον κανείς άλλος επεξεργαστής δεν εκτελεί το Γ_b (αυτό σημαίνει ότι βρίσκεται στην ουρά), τότε αφαιρεί το Γ_b από την ουρά κι εκκινεί το επόμενο βήμα εκτελώντας το.

Πέρα από άπληστος για *strict* υπολογισμούς, ο αλγόριθμος συνδυάζει την "αρχή απασχολημένων φύλλων" (*busy-leaves property*), από την οποία και λαμβάνει το όνομά του: σε κάθε βήμα του αλγορίθμου, κάθε φύλλο του *spawn subtree* εκτελείται σε έναν επεξεργαστή. Ως *spawn subtree* του t βήματος ορίζεται το υποδέντρο του *spawn tree* που περιλαμβάνει αποκλειστικά τα tasks που είναι ζωντανά στο t βήμα του αλγορίθμου. Το να αποδείξουμε κάτι τέτοιο είναι αρκετά απλό.

Αρχικά, ο Busy-Leaves αλγόριθμος έχει μόνο ένα task-ρίζα, το οποίο και εκτελείται σε κάποιον επεξεργαστή. Σε οποιοδήποτε βήμα του αλγορίθμου αλλαγή στο task που εκτελείται σε έναν επεξεργαστή γίνεται σε τρεις περιπτώσεις όπως είδαμε. Σε καθεμία από αυτές ο αλγόριθμος ενεργεί έτσι ώστε τελικά στον επεξεργαστή να εκτελείται το φύλλο. Έτσι:

- Αν ένα task Γ_a κάνει *spawn* ένα παιδί, τότε παύει να είναι φύλλο ακόμη κι αν ήταν πριν, αφού τώρα έχει παιδί. Τη θέση του παίρνει το παιδί του, το οποίο είναι φύλλο του *spawn subtree* εκείνου του βήματος.
- Αν ένα task Γ_a μπει σε κατάσταση αναμονής σημαίνει ότι δεν ήταν φύλλο, δεδομένου ότι ο υπολογισμός είναι *strict.*
- Αν ένα task Γ_a πεθάνει, τότε ο πατέρας του μπορεί να έχει γίνει φύλλο αν αυτό ήταν το παιδί του. Ο αλγόριθμος ελέγχει αν υπάρχει στην ουρά ο πατέρας και αν ναι το εκτελεί. Αν δεν υπάρχει, σημαίνει ότι κάποιος άλλος επεξεργαστής το εκτελεί εκείνη τη στιγμή, οπότε σε κάθε περίπτωση το φύλλο εκτελείται.

Επομένως ο Busy-Leaves αλγόριθμος ικανοποιεί την αρχή απασχολημένων φύλλων.

Τώρα θα αποδείξουμε ένα θεώρημα για τις χωρικές απαιτήσεις που έχει ένας οποιοδήποτε αλγόριθμος που υπακούσει στην παραπάνω αρχή.

Θεώρημα (Χωρική Απόδοση Busy-Leaves Αλγορίθμου)

Για κάθε multitasking υπολογισμό με *βάθος δραστηριοποίησης* S_1 και για κάθε σενάριο δρομολόγησης X σε P επεξεργαστές, οι αλγόριθμοι που υπακούν στην αρχή απασχολημένων φύλλων χρησιμοποιούν το πολύ χώρο $S(X) \leq S_1 P$.

Απόδειξη

Σύμφωνα με την αρχή απασχολημένων φύλλων κάθε φύλλο του *spawn subtree* εκτελείται σε κάποιον επεξεργαστή. Αφού υπάρχουν P επεξεργαστές τότε σε κάθε βήμα το πολύ να

υπάρχουν P φύλλα. Επίσης, για κάθε φύλλο που εκτελείται, ο απαιτούμενος χώρος μνήμης που χρειάζεται να δεσμεύσει είναι το πολύ όσο το *βάθος δραστηριοποίησής* S_1 . Επομένως, ο συνολικός χώρος που μπορεί να χρησιμοποιήσει ένας οποιοσδήποτε αλγόριθμος που υπακούει στην αρχή απασχολημένων φύλλων είναι

$$S(X) \leq S_1 P \quad . \blacksquare$$

Τελικά, αν θέλουμε να δούμε την απόδοση της δρομολόγησης του Busy-Leaves αλγορίθμου συμπεραίνουμε ότι

$$T(X) \leq \frac{T_1}{P} + T_\infty \quad \text{αφού είναι άπληστος}$$

και

$$S(X) \leq S_1 P \quad \text{αφού υπακούει στην αρχή απασχολημένων φύλλων}$$

Παρατηρώντας τις θεωρητικές επιδόσεις των δυο αλγορίθμων που εξετάσαμε, διαπιστώνουμε ότι έχουν ίδια όρια. Αυτό, όμως, δε σημαίνει ότι είναι και το ίδιο αποδοτικοί. Στη μελέτη μας δε λαμβάνουμε υπόψη καθόλου τις καθυστερήσεις που μπορεί να έχουν οι επεξεργαστές όταν συναγωνίζονται να πάρουν ένα task από την ουρά. Αντίθετα, μας ενδιέφερε μόνο ο χρόνος που δρομολογούνται τα tasks. Αν, όμως, έπρεπε να διαλέξουμε ποιον από τους δυο θα υλοποιούσαμε με κριτήριο την απόδοση, θα διαλέγαμε τον Busy-Leaves έναντι του GDF. Το κριτήριο της επιλογής αυτής είναι ότι ο GDF στο τέλος κάθε βήματος επιστρέφει όλα τα tasks στην ουρά των εκτελέσιμων tasks και τα βάζει σε τέτοια σειρά που καθορίζει την προτεραιότητά τους. Αυτό, όμως, είναι αρκετά δαπανηρό σε χρόνο, καθώς πολλές φορές tasks που δε χρειάζεται να μετακινηθούν από τους επεξεργαστές πηγαίνουν στην ουρά και ύστερα ξαναγυρνάνε για εκτέλεση. Ο Busy-Leaves, όμως, δε κάνει τέτοιες άσκοπες μετακινήσεις. Κάθε επεξεργαστής θα εκτελεί ένα task και θα το γυρνάει στην ουρά μόνο στην περίπτωση που δημιουργηθεί ένα νέο task με υψηλότερη προτεραιότητα.

3.4 Work Sharing με τη βιβλιοθήκη task.h

Αξιοποιώντας τις συναρτήσεις διαχείρισης των tasks της βιβλιοθήκης task.h που περιγράφηκαν στην ενότητα 2.6, αποφασίσαμε να δημιουργήσουμε το δικό μας work sharing αλγόριθμο. Θυμίζουμε ότι ως work sharing αλγόριθμος θα ακολουθεί κι αυτός την ιδέα ότι οι επεξεργαστές μοιράζονται μεταξύ τους την εργασία (δηλαδή τα εν ενεργεία tasks) μέσω της προσφοράς: αυτός που έχει πολλά tasks θα δώσει σε αυτόν που δεν έχει.

Η συνάρτηση που υλοποιεί τον αλγόριθμο λέγεται scheduler(). Η πρώτη συνάρτηση που πρέπει να εκτελέσει ο χρήστης για να μπορεί να χρησιμοποιήσει τις συναρτήσεις της task.h είναι η task_init(). Αυτή δημιουργεί ένα pthread ανά διαθέσιμο επεξεργαστή. Κάθε pthread εκτελεί τη συνάρτηση scheduler(), σκοπός της οποίας είναι να δρομολογεί τα tasks αρχικά τοπικά στον επεξεργαστή κι έπειτα καθολικά με το να επικοινωνεί με τους άλλους επεξεργαστές, σύμφωνα με τις προδιαγραφές του work sharing αλγορίθμου μας.

Οι work sharing αλγόριθμοι που ως τώρα έχουν περιγραφεί (GDF και Busy-Leaves) έχουν μια καθολική ουρά στην οποία αποθηκεύονται τα tasks και που έχουν πρόσβαση όλοι οι επεξεργαστές. Ο αλγόριθμός μας, από την άλλη, διαθέτει μια δομή deque για την αποθήκευση των tasks, μοναδική και ατομική για κάθε επεξεργαστή. Ο τρόπος ονομασίας αυτής της δομής θα γίνει κατανοητός στο τέταρτο κεφάλαιο. Προς το παρόν, η deque λειτουργεί ως στοίβα για τον επεξεργαστή στον οποίο ανήκει, ο οποίος έχει δυνατότητα να κάνει push και pop tasks από την ουρά της. Πρόσβαση στη deque ενός επεξεργαστή έχουν και οι υπόλοιποι επεξεργαστές, αλλά μόνο για να κάνουν push κάποιο task. Αυτός είναι και ο τρόπος με τον οποίο κατανέμεται η εργασία.

Η ιδέα του αλγορίθμου μας είναι η εξής: ένας επεξεργαστής εκτελεί μόνο τα tasks που βρίσκονται στη deque του. Κάθε φορά εκτελεί το task που γεννήθηκε πιο πρόσφατα (με άλλα λόγια αυτό που βρίσκεται σε μεγαλύτερο βάθος στο spawn subtree εκείνου του βήματος), οπότε ανήκει στην κατηγορία των depth-first αλγορίθμων. Εργασία θα μοιραστεί μόνο με αδρανείς επεξεργαστές, αυτούς που δεν έχουν κανένα task στην ουρά τους κι επιπλέον δεν εκτελούν κανένα εκείνη τη στιγμή. Ένας επεξεργαστής P_1, λοιπόν, με πλεονάζουσα εργασία, μόλις βρει κάποιον αδρανή επεξεργαστή P_2, κάνει pop το task που βρίσκεται στην ουρά της deque του και το κάνει push στην άδεια deque του P_2. Ο P_2 ειδοποιείται ότι έχει πλέον εργασία και αρχίζει να εκτελεί το task που του δόθηκε. Όταν κι αυτός θα δημιουργήσει περισσότερα tasks, θα φροντίσει να τα μοιράσει σε όσους είναι αδρανείς. Ακολουθεί ο ψευδοκώδικας του work sharing αλγορίθμου μας:

```
void *scheduler(void *arg) {

 while(initial_task->state!=FINALIZED) {

  if (empty_deque) {
   if (current_task->state!=RUNNABLE)) {
      idle = 1;
      pthread_cond_wait(condition,mutex);
    }
    else {
     swapcontext(scheduler_uc,current_task->uc);
    }
  }
  else {
    new_task = pop_from_deque;
    if ((current_task->state==RUNNABLE) && (idle) {
     j = find_empty_scheduler();
     push(current_task,deque_j);
     pthread_cond_signal(condition);
     if_there_is_no_other_empty_scheduler idle=0;
    }
    else {
     push(current_task,current_deque);
    }
  current_task = new_task;
  swapcontext(scheduler_uc,current_task->uc);
  }
 }
 swapcontext(scheduler_uc,init_uc);
 return;
}
```

Ο αλγόριθμος δουλεύει ως ένα loop με συνθήκη εξόδου το αρχικό task που έχει δημιουργηθεί να μην έχει ακόμη τερματίσει. Ουσιαστικά, όταν επιστραφεί το αποτέλεσμα του πρώτου task που έχει δημιουργηθεί τότε έχει ολοκληρωθεί η εργασία του χρήστη, οπότε και ο αλγόριθμος επιστρέφει. Όσο βρισκόμαστε μέσα στο loop γίνονται τα εξής βήματα:

1. Ελέγχεται αν υπάρχει κάποιο task στη deque του επεξεργαστή ή όχι.
2. Αν η deque του επεξεργαστή είναι άδεια, ελέγχεται αν το task που μέχρι τώρα έτρεχε ο επεξεργαστής είναι ακόμα σε κατάσταση έτοιμο για εκτέλεση.
3. Αν όχι (που σημαίνει είτε ότι δεν υπήρχε προηγουμένως κάποιο task να εκτελείται

είτε ότι το task που εκτελούνταν πριν έχει τερματίσει ή έχει βρεθεί σε κατάσταση αναμονής), ο επεξεργαστής αδρανοποιείται εκτελώντας την pthread_cond_wait() και περιμένοντας να τον "ξυπνήσει" όποιος του δώσει εργασία. Επίσης, θέτει την τιμή μιας μεταβλητής *idle* ίση με 1 που είναι μια σημαία για το αν υπάρχει επεξεργαστής που δεν έχει εργασία. Η τιμή 1 στη C σημαίνει true ενώ η 0 false.

4. Αν το current_task του επεξεργαστή είναι έτοιμο για εκτέλεση, τότε γίνεται αλλαγή από το context του scheduler στο context του current_task μέσω της swapcontext. Με αυτόν τον τρόπο, γίνεται το context_switch και ο επεξεργαστής συνεχίζει να εκτελεί το task από το σημείο που είχε σταματήσει.
5. Αν τώρα η deque του επεξεργαστή δεν ήταν άδεια, αλλά είχε τουλάχιστον ένα task, τότε ο work sharing αλγόριθμος κάνει pop το task που βρίσκεται στην ουρά της deque και το αποθηκεύει προσωρινά σε μια μεταβλητή new_task.
6. Ακολουθεί πάλι έλεγχος για το αν το current_task του επεξεργαστή είναι έτοιμο για εκτέλεση ή όχι και για το αν υπάρχει αδρανής επεξεργαστής.
7. Αν ναι, τότε ο επεξεργαστής θα δώσει το current_task του σε έναν από τους επεξεργαστές που είναι άδειος και θα του στείλει σήμα μέσω της pthread_cond_signal ότι πλέον έχει task στη deque του. Κατόπιν, θα ελέγξει αν υπάρχει κάποιος άλλος επεξεργαστής που είναι άδειος και αν δεν βρει κανένα θα θέσει την τιμή της μεταβλητής *idle* ίση με 0.
8. Αν όχι, τότε ο επεξεργαστής θα κάνει push το current_task στη δική του deque.
9. Σε κάθε περίπτωση, θα αλλάξει την τιμή του current_task για να δείχνει στο task που είχε κάνει pop και θα κάνει context_switch στο καινούργιο task για να το εκτελέσει.
10. Όταν βγει έξω από το loop, που σημαίνει ότι θα έχει τερματίσει το πρώτο task που είχε δημιουργηθεί (ανεξάρτητα από το αν το αρχικό task τερματίστηκε στον συγκεκριμένο επεξεργαστή), τότε κάνει context_switch στο αρχικό context που είχε δημιουργηθεί από την task_init().

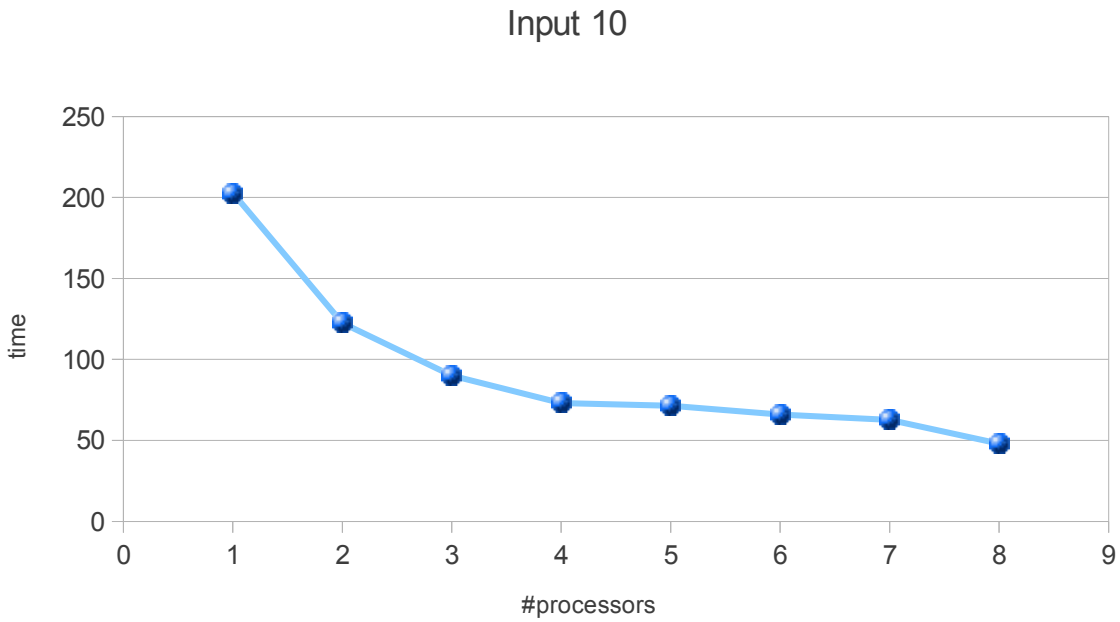
Μια παρατήρηση που μπορούμε να κάνουμε για τον παραπάνω ψευδοκώδικα είναι ο λόγος της χρήση της μοιραζόμενης μεταβλητής *idle,* ως σημαίας για το αν υπάρχουν ή όχι αδρανείς επεξεργαστές. Χωρίς την *idle*, θα χρειαζόταν να ελέγχουμε κάθε φορά όλες τις deque των επεξεργαστών, που αν και δεν είναι πολλές, υπήρχε περίπτωση να δημιουργήσει προβλήματα false sharing [29]. False sharing είναι το φαινόμενο όπου αλλάζουν συχνά μεταβλητές (στην περίπτωσή μας τα δεδομένα των deques), οπότε πρέπει να ενημερώνεται η κύρια μνήμη γι' αυτό. Έτσι, όταν ένας επεξεργαστής διαβάζει συχνά αυτές τις μεταβλητές, αντί να τις διαβάζει από την cache του, χρειάζεται να τις φέρει από την κύρια μνήμη, χάνοντας αρκετό χρόνο. Προς αποφυγή, λοιπόν, του false sharing χρησιμοποιείται η σημαία *idle.*

Κατόπιν θα μελετήσουμε την επίδοση του αλγορίθμου χρησιμοποιώντας ως κριτήριο της ταχύτητας εκτέλεσης τον αναδρομικό υπολογισμό αριθμών fibonacci, με τη διαφορά ότι τον τροποποιήσαμε ώστε κάθε task να εκτελεί ένα βαρύ υπολογιστικά φορτίο. Αυτό έγινε ώστε να φανεί η κλιμάκωση του χρόνου όταν αλλάζει το πλήθος των επεξεργαστών.

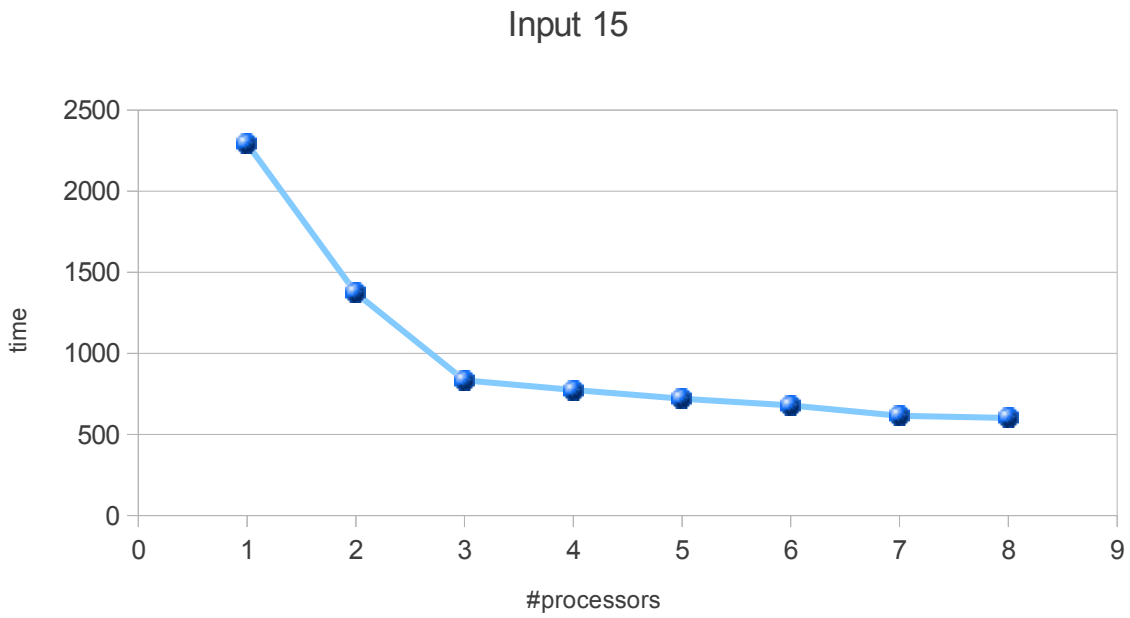
Χρησιμοποιήσαμε δυο διαφορετικές πλατφόρμες του εργαστηρίου cslab για να πάρουμε μετρήσεις: τα clones και τον dunnington. Τα clones είναι μηχανήματα που αποτελούνται από 2 CPUs με 4 πυρήνες η καθεμία, που βλέπουν μια κοινή μνήμη. Δηλαδή, συνολικά 8 επεξεργαστικές μονάδες κοινής μνήμης. Ο dunnington αποτελείται από 4 CPUs με 6 πυρήνες η καθεμία, οπότε συνολικά 24 επεξεργαστικές μονάδες κοινής μνήμης. Ας δούμε ξεχωριστά τις μετρήσεις σε κάθε μηχάνημα.

➔ *Clones*

Τα clones έχουν 8 επεξεργαστικές μονάδες, οπότε μπορούμε να παρακολουθήσουμε την επιτάχυνση του προγράμματος χρησιμοποιώντας μέχρι 8 threads. Υπενθυμίζουμε, ότι σηκώνουμε ένα pthread ανά διαθέσιμο επεξεργαστή.

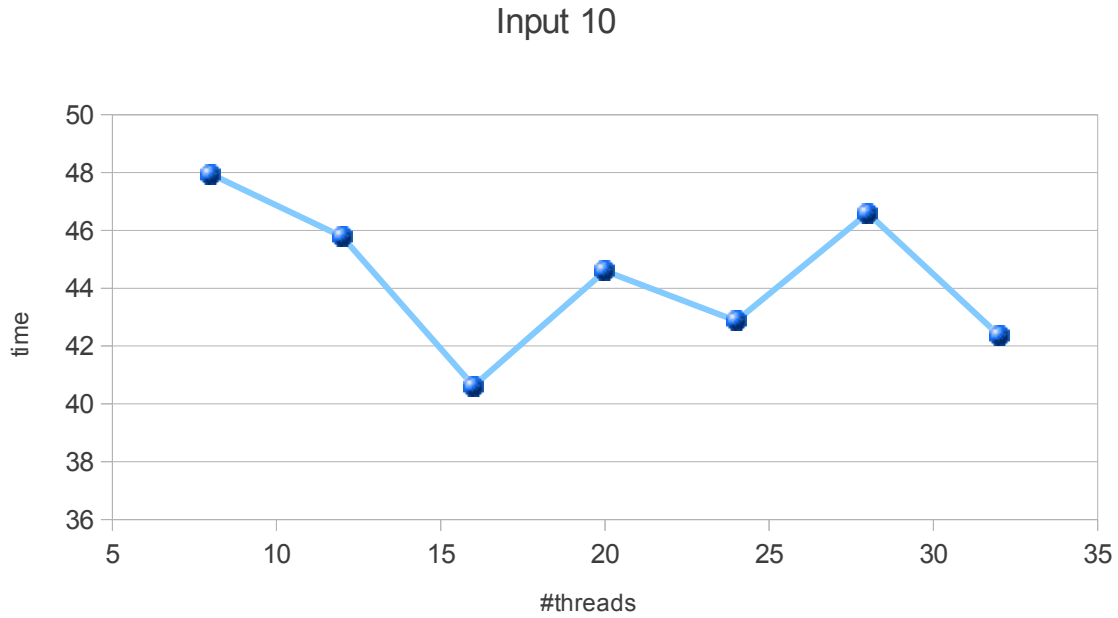


Για είσοδο ίση με 10, όπως στο παραπάνω διάγραμμα, δημιουργούνται 2^{10} tasks, τα οποία δρομολογούνται από τον work sharing αλγόριθμο. Παρατηρούμε ότι ο χρόνος αρχικά μειώνεται σχεδόν αντιστρόφως ανάλογα με την αύξηση των επεξεργαστών, αγγίζοντας το κάτω όριο $\frac{T_1}{P}$, αλλά όταν ξεπεράσουμε τους 4 επεξεργαστές το πρόγραμμα επιταχύνεται με μικρότερο ρυθμό.



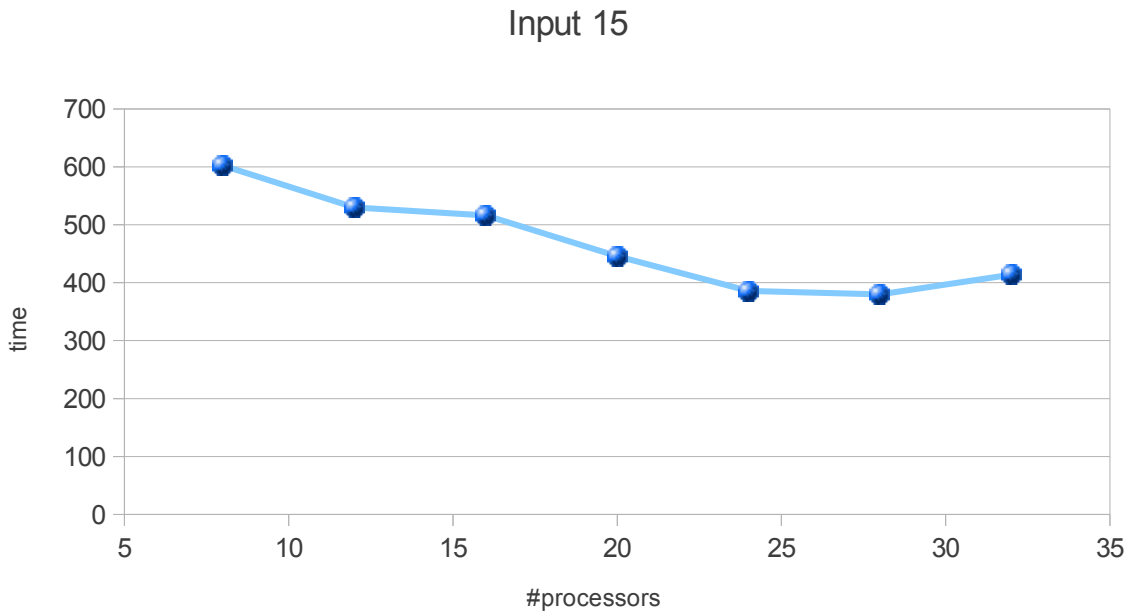
Παρόμοια συμπεριφορά παρατηρούμε και όταν αυξάνουμε την είσοδο από 10 σε 15, οπότε δημιουργούνται 2^{15} tasks. Αυτό σημαίνει ότι ο αλγόριθμός μας, ανεξάρτητα από το μέγεθος είσοδου (και πριν φυσικά φτάσει στα όρια παραλληλοποίησης του προγράμματος όπου δεν μπορεί να επιταχυνθεί άλλο) έχει ρυθμό επιτάχυνσης αρχικά γραμμικό ως προς το πλήθος των επεξεργαστών, που όμως μειώνεται καθώς αυτό αυξάνεται.

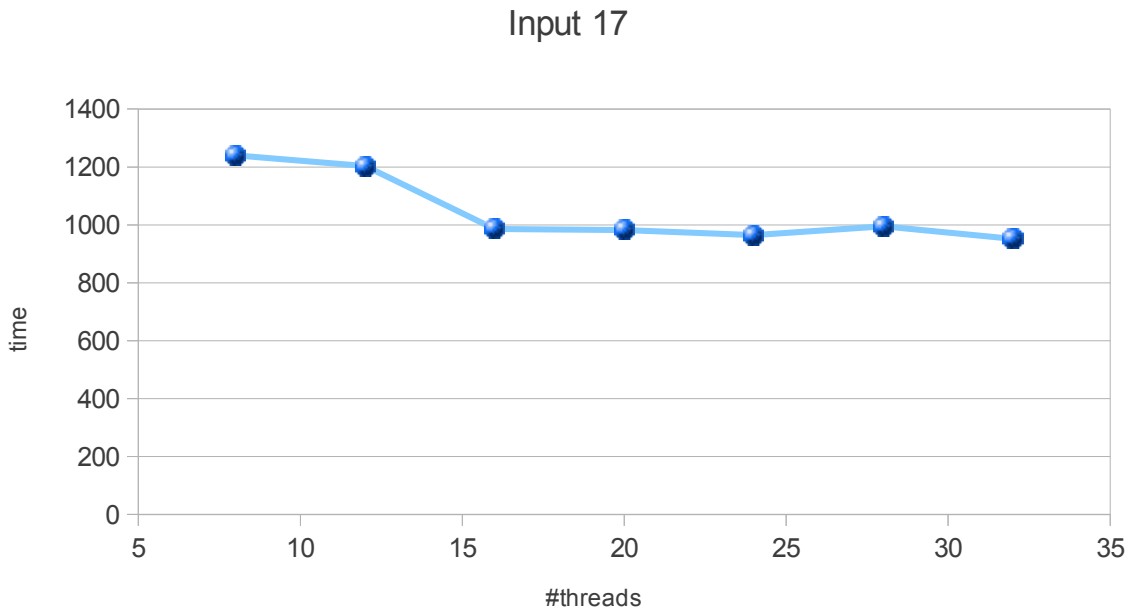
Τι θα γινόταν τώρα αν αυξάναμε τον αριθμό των threads;



Το πλήθος των επεξεργαστών των clones δεν μπορεί να ξεπεράσει τους 8, οπότε έχουμε

περισσότερα από ένα threads σε κάθε επεξεργαστή, τα οποία δρομολογούνται από το λειτουργικό σύστημα. Αυτό το φαινόμενο λέγεται oversubscription. Στο παραπάνω διάγραμμα βλέπουμε μια "περίεργη" συμπεριφορά, καθώς άλλοτε επιταχύνεται το πρόγραμμα ενώ άλλοτε επιβραδύνεται. Αφού δεν αυξάνεται το πλήθος των επεξεργαστών και παραμένει σταθερό ίσο με 8 δεν είναι λογικό να ζητούμε μια γραμμική μείωση χρόνου. Το θετικό, όμως, είναι ότι συνολικά η τεχνική του oversubscription βοηθάει ώστε να μειωθεί έστω και λίγο ο χρόνος εκτέλεσης: τα 8 threads χρειάζονται περισσότερο χρόνο εκτέλεσης από όλα τα άλλα.

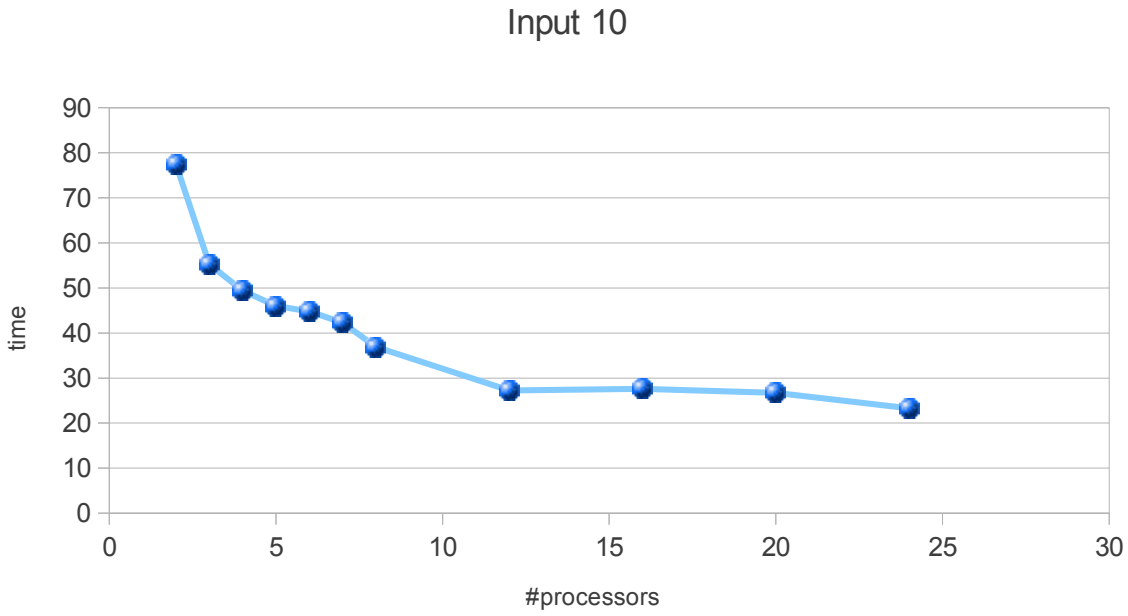




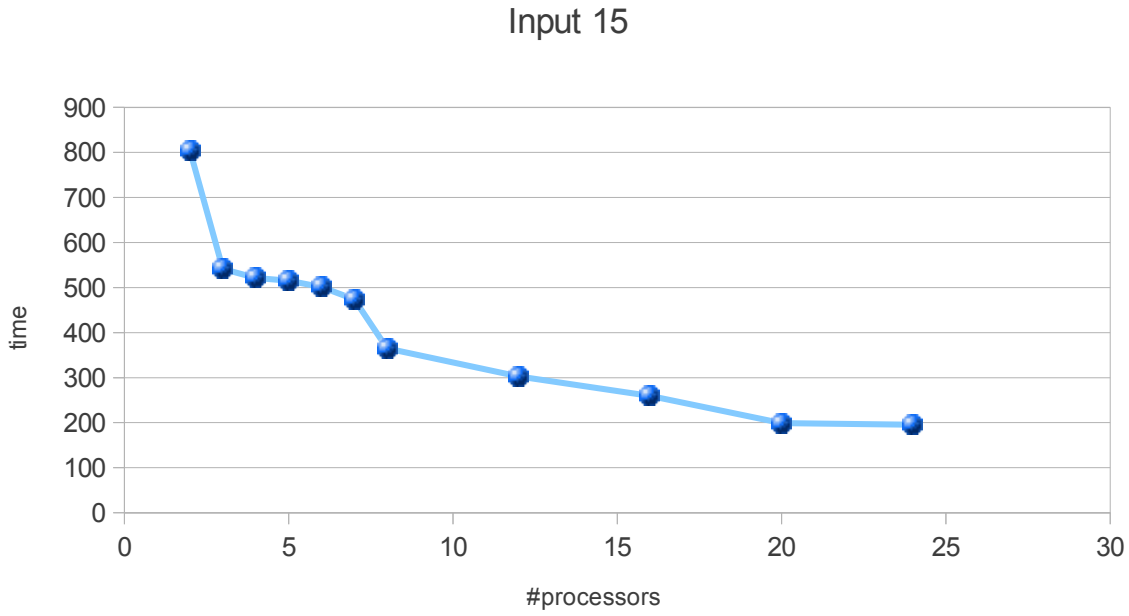
Τα αντίστοιχα διαγράμματα για είσοδο ίση με 15 και 17 παρέχουν κάποια χρήσιμα συμπεράσματα. Δείχνουν ότι η τεχνική του oversubscription, δουλεύει αρκετά καλά για το work sharing αλγόριθμο μιας και στη γενική περίπτωση το πρόγραμμα θα έχει καλύτερη επίδοση με την αύξηση των threads. Σίγουρα δεν έχουμε θεαματική επιτάχυνση, αλλά είναι μια πιθανή βελτιστοποίηση, ιδιαίτερα όσο αυξάνεται το μέγεθος εισόδου.

➔ *Dunnington*

Ο dunnington είναι μια διαφορετική πλατφόρμα καθώς έχει συνολικά 24 επεξεργατικές μονάδες, οπότε μπορούμε να πάρουμε μετρήσεις για μεγαλύτερο αριθμό επεξεργαστών.

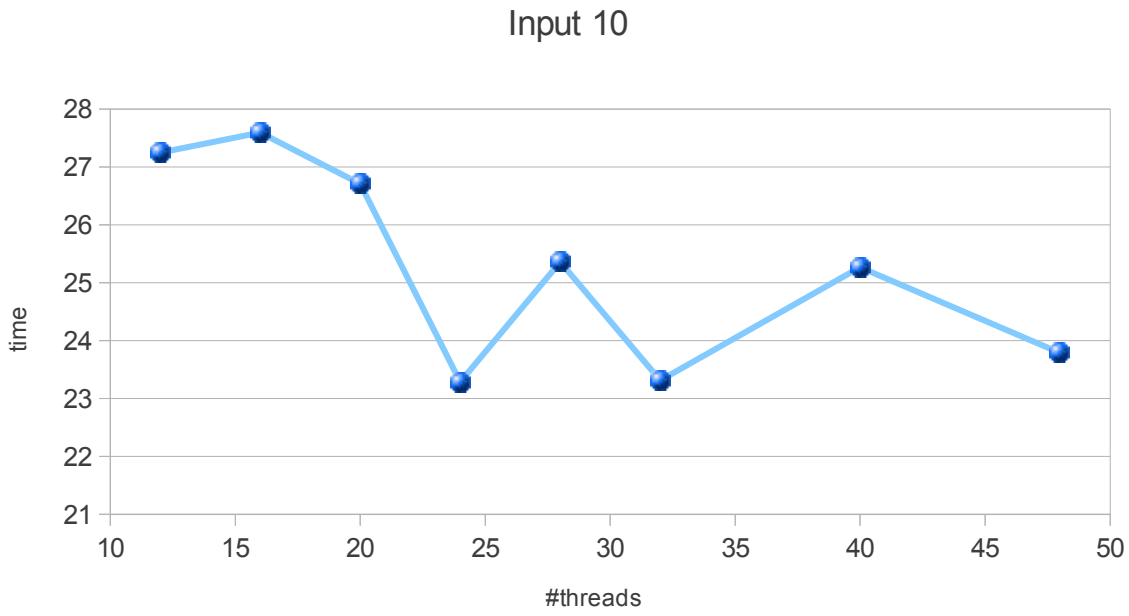


Για είσοδο 10, η εικόνα της χρονικής επίδοσης είναι η παραπάνω. Η επιτάχυνση είναι αρκετά μεγαλύτερη σε σχέση με εκείνη που παρατηρήσαμε στα clones. Μέχρι και τους 12 επεξεργαστές, το πρόγραμμα επιταχύνεται σχεδόν γραμμικά με ρυθμό βέβαια μεγαλύτερο από το $\frac{1}{P}$, αλλά όπως περιμέναμε από τη θεωρητική ανάλυση να υπακούει στον κανόνα $T(X) \leq \frac{T_1}{P} + T_\infty$. Αξιοσημείωτο είναι ότι όταν το πλήθος των επεξεργαστών γίνει μεγαλύτερο από 12, το πρόγραμμα σταματάει πρακτικά να επιταχύνει, αλλά δίνει παρόμοιους χρόνους εκτέλεσης. Αυτό οφείλεται στο ότι φτάνουμε στα όρια παραλληλοποίησης και ότι γι' αυτήν την είσοδο ο αλγόριθμος δεν μπορεί να δώσει καλύτερους χρόνους παρά την αύξηση των επεξεργαστών.



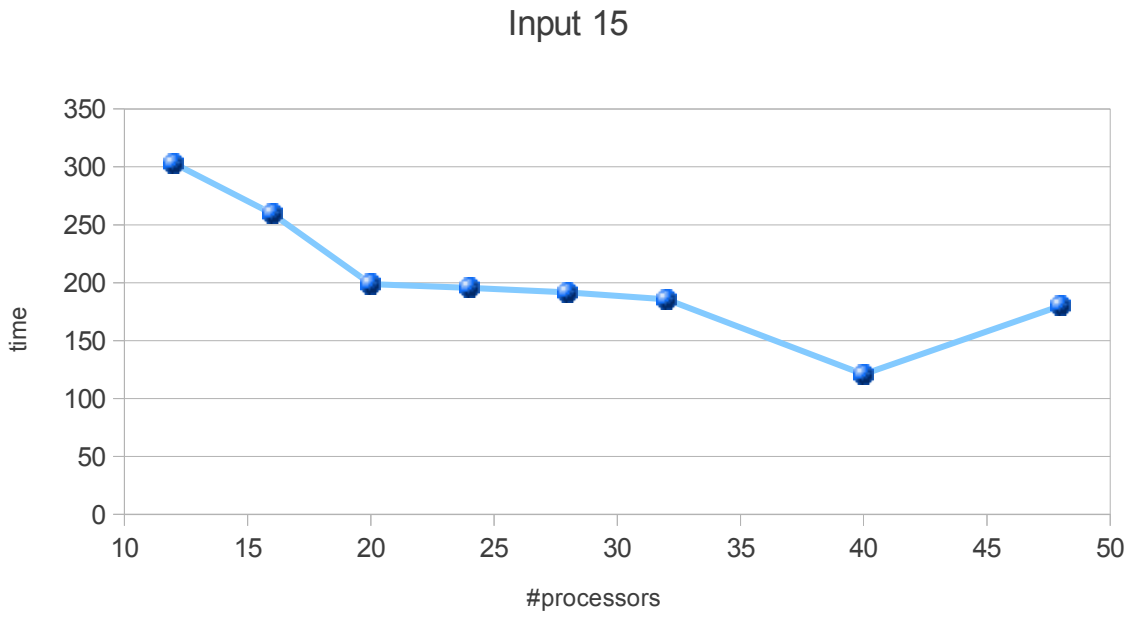
Η συμπεριφορά είναι παρόμοια όταν η είσοδος γίνει 15. Η διαφορά είναι ότι η επιτάχυνση πρακτικά σταματάει όταν το πλήθος των επεξεργαστών γίνει ίσο με 20, λογικό αφού αυξάνεται το πλήθος των δρομολογούμενων tasks.

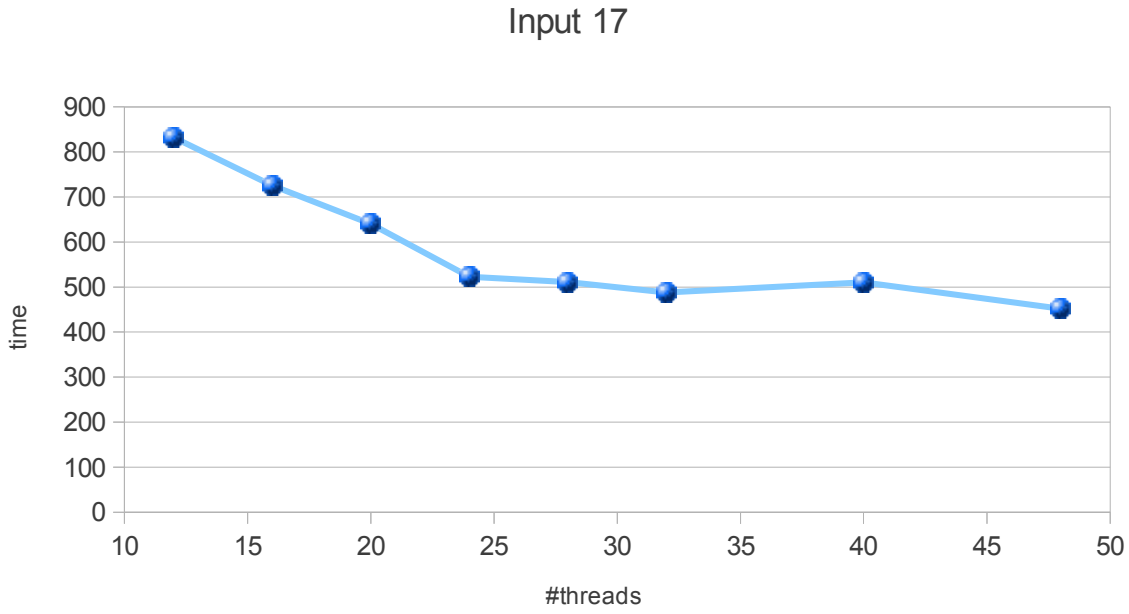
Ας δούμε και σε αυτήν την πλατφόρμα τι γίνεται στην περίπτωση του oversubscription.



Η περίπτωση που η είσοδος είναι ίση με 10 έχει ιδιαίτερη συμπεριφορά. Παρατηρούμε ότι τον καλύτερο χρόνο τον δίνει όταν τα threads είναι 24, ενώ το oversubscription ξεκινά από τη στιγμή που τα threads γίνουν περισσότερα από 24. Αυτή η τεχνική εδώ δε βοηθάει διότι ήδη έχουμε φτάσει στα όρια της παραλληλοποίησης και πλέον η αύξηση των threads που

οδηγεί σε συνολική αύξηση του χρόνου του context switch είναι επιβαρυντική. Οι χρόνοι που παίρνουμε είναι σχετικά ίδιοι μεταξύ τους (διαφέρουν κατά μερικά δευτερόλεπτα) επομένως το παραπάνω διάγραμμα δεν αποτελεί καλή ένδειξη για να μελετήσουμε τη συμπεριφορά του oversubscription, μιας και το πλήθος των tasks είναι αρκετά μικρό.





Όταν τώρα βάζουμε μεγαλύτερη είσοδο (15 και 17 για τα παραπάνω διαγράμματα), η συμπεριφορά είναι ίδια με εκείνη που παρατηρήσαμε στα clones. Άρα, μπορούμε να συμπεράνουμε ότι συνολικά η τεχνική του oversubscription δουλεύει αρκετά καλά για το work sharing αλγόριθμο, τον οποίο βελτιστοποιεί.

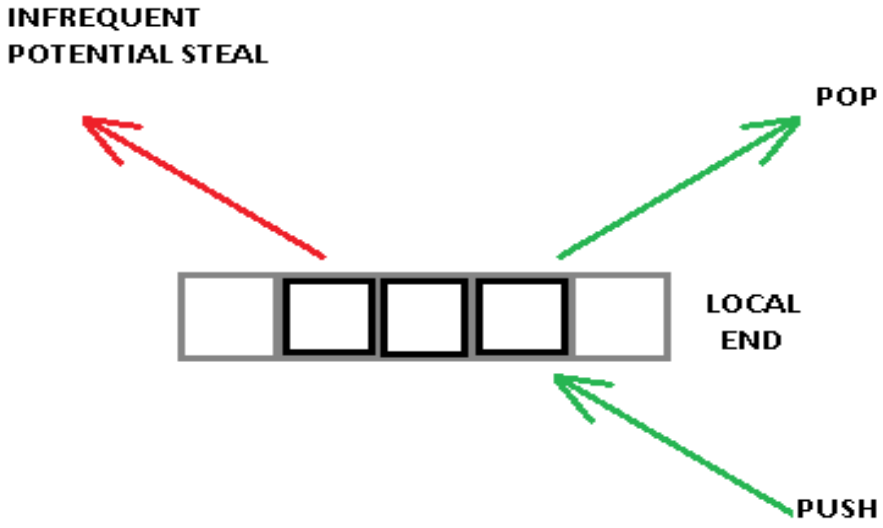
ΚΕΦΑΛΑΙΟ 4: WORK STEALING

4.1 Work Stealing Αλγόριθμος

4.1.1 Περιγραφή του Αλγορίθμου

Η φιλοσοφία των work stealing αλγορίθμων διαφέρει από εκείνη των work sharing στο γεγονός ότι ο δρομολογητής δεν επιχειρεί να κατανείμει τα tasks στους επεξεργαστές, αλλά οι ίδιοι οι επεξεργαστές παίρνουν την πρωτοβουλία. Οι άδειοι επεξεργαστές προσπαθούν να κλέψουν tasks από αυτούς που έχουν εργασία. Ιδιαίτερα σημαντικό στοιχείο είναι ότι η μεταφορά των tasks από έναν επεξεργαστή σε έναν άλλο είναι λιγότερο σπάνια στους work stealing δρομολογητές απ' ότι στους work sharing κι αυτό γιατί όταν όλοι οι επεξεργαστές έχουν εργασία, κανένα task δε μεταφέρεται στο work stealing. Η ιδέα του είναι πιο παλιά από τη δημιουργία των γλωσσών παράλληλου προγραμματισμού, ήδη από το 1980, και είχε αρχικά στόχο την αύξηση της απόδοσης στην επικοινωνία μεταξύ μονάδων επεξεργασίας [30].

Ο work stealing αλγόριθμος που θα μελετήσουμε σε αυτήν την ενότητα βασίζεται σε ιδέες που χρησιμοποιεί και ο Busy-Leaves αλγόριθμος που περιγράφηκε στην ενότητα 3.3. Μια βασική αλλαγή είναι ότι η καθολική ουρά του Busy-Leaves διαμοιράζεται εδώ σε κάθε επεξεργαστή και αντικαθίσταται από μια δομή deque (Σχήμα 11).



Σχήμα 11: Η Deque

Η deque είναι μια δομή δεδομένων που έχει δυο άκρα: μια κεφαλή (head) και μια ουρά (tail). Στη deque μπορεί να γίνει εισαγωγή tasks μόνο από την ουρά, αλλά εξαγωγή και από

τα δυο άκρα. Ο επεξεργαστής στον οποίο ανήκει η deque την αντιμετωπίζει ως στοίβα, δηλαδή κάνει push και pop από την ουρά της. Οι άλλοι επεξεργαστές έχουν το δικαίωμα να κλέψουν μόνο από την κεφαλή της deque και να τοποθετήσουν το κλεμμένο task στη δική τους. Η μέθοδος με την οποία λειτουργεί το work stealing είναι η ακόλουθη: ο άδειος επεξεργαστής γίνεται *κλέφτης* και επιχειρεί να κλέψει εργασία από έναν άλλο τυχαία επιλεγμένο επεξεργαστή που γίνεται *θύμα*. Ο κλέφτης παίρνει το task που βρίσκεται στην κεφαλή της deque του θύματός του, στην περίπτωση που αυτή δεν είναι άδεια και ξεκινάει να το εκτελεί. Αν είναι άδεια, τότε η κλοπή αποτυγχάνει και επαναλαμβάνει την ίδια διαδικασία διαλέγοντας στην τύχη κάποιον άλλον επεξεργαστή μέχρι τελικά να αποκτήσει εργασία.

Στη γενική περίπτωση, ένας επεξεργαστής, που δεν είναι άδειος, παίρνει το πρώτο task που βρίσκεται στην ουρά της deque του κι εκτελεί τις εντολές του μέχρι αυτό να κάνει *spawn* ή να μπει σε κατάσταση αναμονής ή να πεθάνει ή να ενεργοποιήσει ένα άλλο task που βρισκόταν σε αναμονή. Ο work stealing αλγόριθμος ακολουθεί τέσσερις κανόνες:

(1) *Spawn:* Όταν ένα task-πατέρας Γ_a που εκτελείται δημιουργεί ένα task-παιδί Γ_b , τότε το Γ_a γίνεται pushed πίσω στη deque κι ο επεξεργαστής εκτελεί το Γ_b .

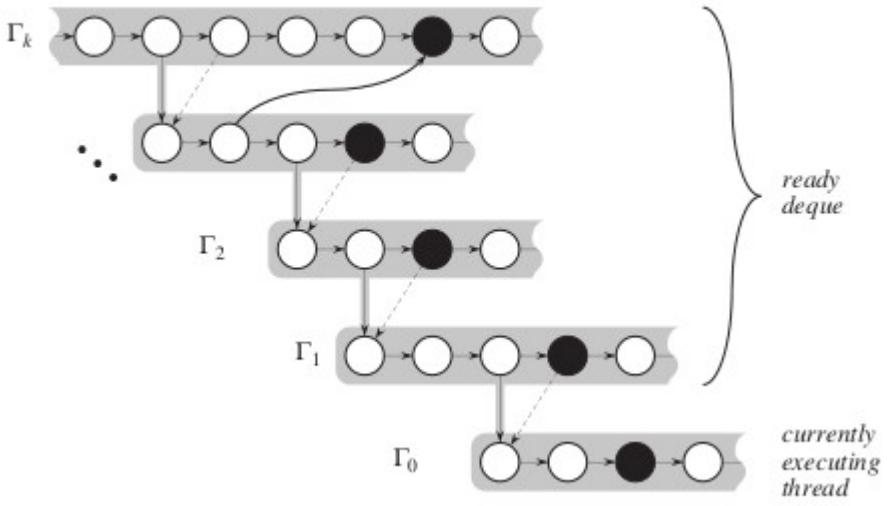
(2) Αναμονή: Όταν ένα task Γ_a μπει σε κατάσταση αναμονής, ο επεξεργαστής ελέγχει τη deque του. Αν αυτή έχει έστω ένα task, τότε κάνει pop το πρώτο που βρίσκεται στην ουρά της και εκτελεί αυτό. Αν η deque του είναι άδεια, ο επεξεργαστής ξεκινάει το work stealing: διαλέγει τυχαία έναν επεξεργαστή και αφαιρεί το task που βρίσκεται στην κεφαλή της deque του κι έπειτα ξεκινάει την εκτέλεση του κλεμμένου task. Αν αποτύχει (όταν η deque του υποψήφιου θύματος είναι επίσης άδεια) τότε επαναλαμβάνει αυτή τη διαδικασία μέχρι να κλέψει στο τέλος κάποιο task.

(3) Θάνατος; Όταν ένα task Γ_a πεθάνει, ο επεξεργαστής ακολουθεί την ίδια τακτική με τον κανόνα (2).

(4) Ενεργοποίηση: Όταν ένα task Γ_a ενεργοποιήσει ένα task Γ_b , το οποίο βρισκόταν σε αναμονή, με το να εκτελέσει κάποια εντολή από την οποία εξαρτιόταν το Γ_b , τότε το Γ_b που είναι πλέον έτοιμο για εκτέλεση γίνεται pushed στη deque του επεξεργαστή.

Επειδή ένα task μπορεί ταυτόχρονα να ενεργοποιήσει ένα εν αναμονή task και να πεθάνει ή να μπει το ίδιο σε κατάσταση αναμονής με την ίδια εντολή, ο αλγόριθμος

εφαρμόζει τον κανόνα 4 με μεγαλύτερη προτεραιότητά σε σχέση με τους κανόνες 2 και 3. Οι τρεις πρώτοι κανόνες αντιστοιχίζονται με τους αντίστοιχους κανόνες του Busy-Leaves. Η αναγκαιότητα του επιπλέον τέταρτου κανόνα για την ενεργοποίηση έγκειται στη διασφάλιση της "αρχής των απασχολημένων φύλλων", στην οποία υπακούει και ο work stealing αλγόριθμος.



Σχήμα 12: Η deque ενός επεξεργαστή p με το άνω task να βρίσκεται στην κεφαλή και το τελευταίο να είναι αυτό που εκτελείται στον επεξεργαστή.

Θεώρημα (Ακολουθία των tasks σε μια deque)

Έστω τυχαίος επεξεργαστής p κατά την εκτέλεση ενός οποιουδήποτε *fully strict multitasking υπολογισμού* που χρησιμοποιεί για δρομολόγηση εργασιών τον work stealing αλγόριθμο. Έστω ότι σε ένα τυχαίο βήμα t του αλγορίθμου ο p εκτελεί το task Γ_0 κι έχει στη deque του k tasks, τα $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$, με σειρά από την ουρά προς την κεφαλή (το Γ_1 βρίσκεται στην αρχή της ουράς της deque και το Γ_k στην κεφαλή της) (Σχήμα 12). Για k>0 ισχύουν τα δυο παρακάτω πορίσματα:

1. Για $i=1,2,\ldots,k$ το task Γ_i είναι ο πατέρας του Γ_{i-1} .

2. Αν $k>1$, τότε για $i=1,2,\ldots,k-1$ το task Γ_i δεν έχει εκτελεστεί από τη στιγμή που δημιούργησε το Γ_{i-1} .

Απόδειξη

Για την απόδειξη του θεωρήματος θα χρησιμοποιήσουμε τη μέθοδο της αναγωγής.

- Αρχικά για t = 1, μόνο ένας επεξεργαστής εργάζεται εκτελώντας το task-ρίζα, που είναι και το μόνο ζωντανό κι όλοι οι άλλοι επεξεργαστές είναι άδειοι. Η ισχύς των πορισμάτων είναι προφανής δεδομένου ότι οι deque είναι άδειες (k=0).
- Έστω ότι στο βήμα t κάποιος επεξεργαστής p εκτελεί ένα task Γ_0 και έχει στη deque του k tasks, τα $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$ με σειρά από την ουρά στην κεφαλή. Υποθέτουμε ότι σε αυτό το βήμα ισχύουν τα πορίσματα του θεωρήματος για k>0 ή αλλιώς k=0.
- Θα αποδείξουμε ότι τα πορίσματα 1 και 2 ισχύουν και στο βήμα t+1 του αλγορίθμου.

Έστω ότι στο βήμα t+1 ο επεξεργαστής p εκτελεί το task Γ'_0 (αν εκτελεί κάποιο και δεν είναι άδειος). Επίσης στη deque του έχει k' tasks, τα $\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_{k'}$. Η μόνη περίπτωση να έχει αλλάξει η κατάσταση της deque είναι να έχει εφαρμοστεί κάποιος από τους τέσσερις κανόνες που ακολουθεί ο work stealing αλγόριθμος. Ας εξετάσουμε χωριστά την ισχύ των πορισμάτων με βάση καθέναν απ' αυτούς χωριστά.

Κανόνας 1: Αν το Γ_0 έκανε στο τέλος του βήματος t *spawn* ένα παιδί, τότε ο επεξεργαστής p έκανε push στη deque του το Γ_0 και άρχισε να εκτελεί το παιδί. Αυτό σημαίνει ότι το Γ'_0 είναι το παιδί που δημιουργήθηκε. Στη deque αρχικά υπήρχαν k tasks και σε αυτά προστέθηκε το Γ_0 , οπότε στο βήμα t+1 η deque έχει k' = k +1 tasks, τα οποία είναι $\Gamma_0, \Gamma_1, \Gamma_2, \ldots, \Gamma_k$ και αντιστοιχούν στα $\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_{k'}$ ένα προς ένα (Σχήμα 13). Δηλαδή:

$$\Gamma'_i = \Gamma_{i-1} \quad \text{για} \quad i = 1,2,\ldots,k$$

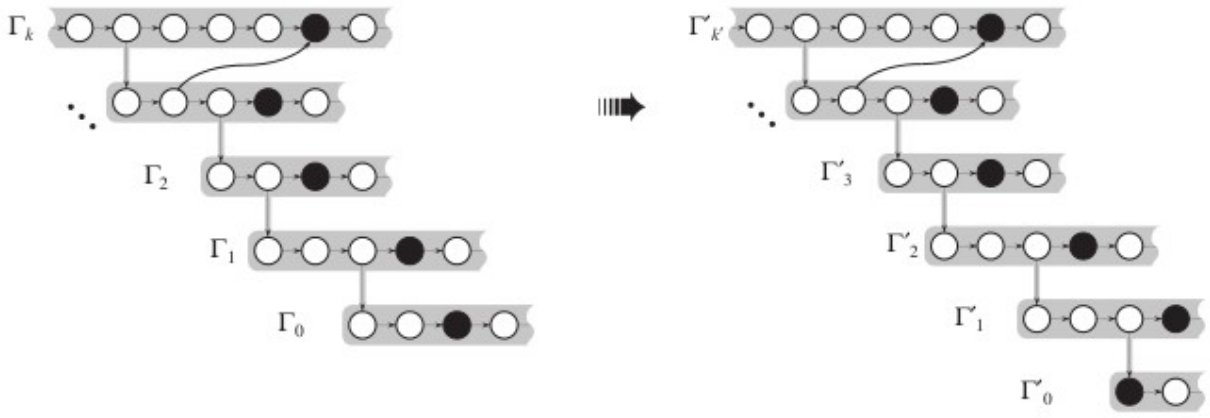
Επειδή, όμως στο δεύτερο βήμα της απόδειξης θεωρήσαμε ότι ισχύουν τα πορίσματα για τα $\Gamma_1, \Gamma_2, \ldots, \Gamma_k$,τότε προφανώς

$$\text{το} \quad \Gamma'_i \quad \text{είναι ο πατέρας του} \quad \Gamma'_{i-1} \quad \text{για} \quad i = 2,3,\ldots,k$$

Όσον αφορά τα Γ'_1 και Γ'_2 το πρώτο είναι ο πατέρας του δεύτερου γιατί το Γ_0 που στο προηγούμενο βήμα εκτελούνταν στον p είναι προφανές ότι γεννήθηκε από το πρώτο task που υπήρχε στη deque, δηλαδή το Γ_1 . Οπότε ισχύει το πόρισμα 1.

Επιπλέον, γνωρίζουμε ότι τα Γ'_i με $i = 2,3,\ldots,k$ δεν έχουν εργαστεί από τότε που

δημιούργησαν το παιδί τους από υπόθεση. Για i = 1, το Γ'_1 δεν έχει εργαστεί από τότε που δημιούργησε το παιδί του, μιας και το δημιούργησε μόλις στο προηγούμενο βήμα. Επομένως ισχύει και το πόρισμα 2.



Σχήμα 13: Η αντιστοιχία των tasks στη deque ενός επεξεργαστή p πριν (αριστερά) και μετά (δεξιά) το spawn. Το κατώτερο task είναι αυτό το οποίο εκτελεί εκείνη τη στιγμή ο επεξεργαστής

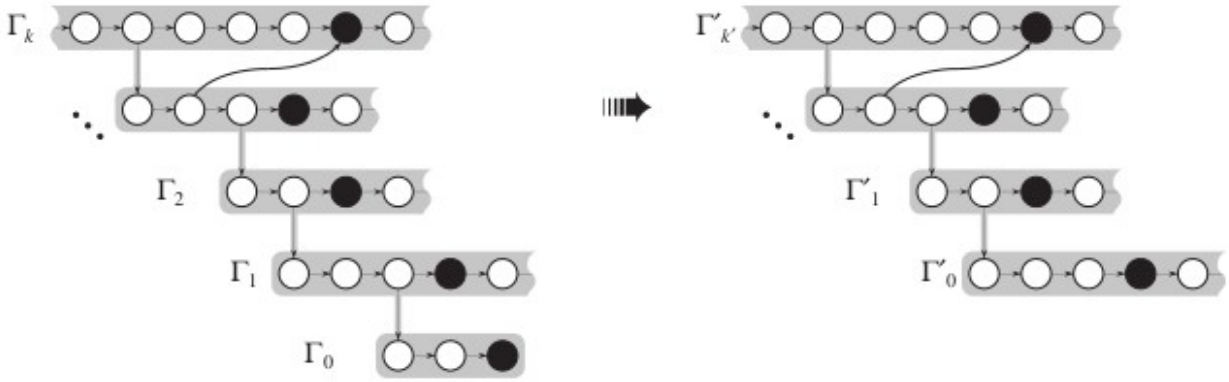
<u>Κανόνες 2 και 3:</u> Αν το Γ_0 μπήκε σε κατάσταση αναμονής ή πέθανε, τότε υπάρχουν δυο περιπτώσεις. Η πρώτη είναι η deque του p να είναι άδεια, οπότε k = 0. Τότε ο επεξεργαστής ξεκινάει το work stealing και όταν επιτύχει να κλέψει ένα task αρχίζει να εκτελεί εκείνο και έχει k' = 0, οπότε τα πορίσματα ισχύουν. Στη δεύτερη περίπτωση, η deque δεν είναι άδεια, οπότε k > 0. Τότε ο επεξεργαστής θα κάνει pop το task που βρίσκεται πρώτο στην ουρά της deque του, δηλαδή το Γ_1 και θα ξεκινήσει να εκτελεί αυτό. Η deque θα αποτελείται πλέον από k' = k-1 tasks, τα $\Gamma_2, \Gamma_3, \ldots, \Gamma_k$ που αντιστοιχούν στα $\Gamma'_1, \Gamma'_2, \ldots, \Gamma'_{k'}$ ένα προς ένα (Σχήμα 14). Δηλαδή

$$\Gamma'_i = \Gamma_{i+1} \quad \text{για} \quad i = 1,2,\ldots,k-1$$

Από υπόθεση τα πορίσματα ισχύουν για τα $\Gamma_2, \Gamma_3, \ldots, \Gamma_k$, οπότε

$$\text{το} \quad \Gamma'_i \quad \text{είναι ο πατέρας του} \quad \Gamma'_{i-1} \quad \text{για} \quad i = 1,2,\ldots,k-1$$

και ισχύει το πόρισμα 1 για το νέο βήμα, όπως επίσης και το πόρισμα 2, καθώς κανένα task που βρίσκεται στη deque δεν είχε εκτελεστεί από τότε που έκανε *spawn* το παιδί του.



Σχήμα 14: Η αντιστοιχία των tasks στη deque ενός επεξεργαστή p πριν (αριστερά) και μετά (δεξιά) από το θάνατο ή την αναμονή του task που εκτελούσε ο επεξεργαστής.

Κανόνας 4: Αν το Γ_0 ενεργοποιήσει κάποιο task, αυτό υποχρεωτικά θα είναι ο πατέρας του, δηλαδή το Γ_1 , λόγω της υπόθεσης ότι ο υπολογισμός είναι *fully strict*. Υπενθυμίζουμε ότι *fully strict* ορίζονται οι multitasking υπολογισμοί που οι εξαρτήσεις μεταξύ των tasks είναι αποκλειστικά από το παιδί προς τον πατέρα του. Αναγκαστικά, λοιπόν, στο προηγούμενο βήμα η deque ήταν άδεια και k=0. Αν ήταν k>0, δηλαδή η deque είχε tasks, τότε το πρώτο στην ουρά θα ήταν ο πατέρας του Γ_0 . Έτσι, όμως, το Γ_1 θα ήταν έτοιμο προς εκτέλεση κι όχι σε αναμονή, πράγμα άτοπο, αφού δε θα εφαρμοζόταν ο τρέχων κανόνας. Κατά την εφαρμογή του κανόνα 4, το Γ_1 θα γίνει pushed στη μέχρι τότε άδεια deque, οπότε $\Gamma'_0=\Gamma_1$. Είναι προφανές ότι το πόρισμα 1 ισχύει. Το δεύτερο δεν έχει νόημα να το εξετάσουμε αφού k'=1.

Κλοπή: Το τελευταίο ενδεχόμενο που μπορεί να προκύψει ανάμεσα στα βήματα t και t+1 του αλγορίθμου είναι κάποιος άλλος επεξεργαστής να κλέψει ένα task από τον p. Αυτό σημαίνει ότι η deque του p είχε τουλάχιστον ένα task στο βήμα t (k>0). Στο t+1 βήμα θα ισχύει k'=k-1. Αν εξακολουθεί η deque να έχει tasks (k'>0) κατ' αντιστοιχία με τις περιπτώσεις των κανόνων 2 και 3, και τα δυο πορίσματα του θεωρήματος θα ισχύουν. ■

Θα θέλαμε να επισημάνουμε δυο σημεία πάνω σε αυτό το θεώρημα:

Το task Γ_k δεν υπόκειται στο δεύτερο συμπέρασμα που αναφέρει ότι τα tasks δεν έχουν εκτελεστεί από τότε που έκαναν *spawn* το παιδί τους. Υπάρχει, όντως, μια περίπτωση να έχει εκτελεστεί και σε αυτή θα αναφερθούμε. Επειδή βρίσκεται στην κεφαλή της deque υπάρχει περίπτωση να κλαπεί από κάποιον άλλον επεξεργαστή και να έχει εκτελεστεί σε αυτόν. Αν σε κάποια στιγμή της εκτέλεσης του δημιουργήσει κάποιο παιδί ή πεθάνει τότε υπακούει στο δεύτερο πόρισμα του θεωρήματος. Αν, όμως, κάποια στιγμή μπει σε

κατάσταση αναμονής λόγω κάποιας εξάρτησης από το παιδί του, τότε θα επιστρέψει στην ουρά του p επεξεργαστή, μόλις ικανοποιηθεί η εξάρτηση. Για να γίνει κάτι τέτοιο, σημαίνει ότι στον p επεξεργαστή θα εκτελείται το task Γ_{k-1} κι έτσι η deque του p θα είναι άδεια, αφού αυτό ήταν το τελευταίο στη σειρά task προς εκτέλεση. Το Γ_k θα γίνει pushed στη deque και θα είναι το μοναδικό της task, το οποίο, όμως, θα έχει εκτελέσει μέρος του κώδικά του μετά το *spawn* για τη δημιουργία του Γ_{k-1} .

Η δεύτερη παρατήρηση αφορά το χωρική επίδοση που έχει ο work stealing αλγόριθμος. Σε κάθε βήμα του κάθε φύλλο του *spawn subtree* θα είναι έτοιμο προς εκτέλεση, επομένως είτε θα βρίσκεται στη deque κάποιου επεξεργαστή είτε θα εκτελείται σε κάποιον από αυτούς. Σύμφωνα, όμως, με το προηγούμενο θεώρημα, το να βρίσκεται κάποιο task στη deque συνεπάγεται ότι είναι πατέρας του αμέσως επόμενου task που βρίσκεται στη deque ή του task που εκτελείται εκείνη την ώρα, γεγονός που σημαίνει ότι κανένα φύλλο δε βρίσκεται στη deque κανενός επεξεργαστή σε κανένα βήμα. Άρα κάθε φύλλο του *spawn subtree* εκτελείται ανά πάσα στιγμή και ο work stealing αλγόριθμος ακολουθεί την αρχή απασχολημένων φύλλων. Από γνωστό θεώρημα, θα δεσμεύει κατά την εκτέλεσή του χώρο μνήμης το πολύ ίσο με $S_1 P$ όπου S_1 το *βάθος δραστηριοποίησής* του υπολογισμού. Για τη χρονική απόδοση του αλγορίθμου θα ασχοληθούμε παρακάτω, μιας και επιτυγχάνει καλύτερα όρια από τους αντίστοιχους work sharing αλγορίθμους.

4.1.2 Χρονική επίδοση του Work Stealing Αλγορίθμου

Σε αυτήν την ενότητα θα ασχοληθούμε με τη χρονική επίδοση του work stealing αλγορίθμου που περιγράψαμε κατ' αντιστοιχία με την ανάλυση που κάναμε για τους work sharing αλγορίθμους. Πέρα από το χρόνο δρομολόγησης των εργασιών, θα επεκτείνουμε τη μελέτη και στο κόστος επικοινωνίας των επεξεργαστών όταν συναγωνίζονται μεταξύ τους, μια μελέτη που δεν έγινε στην ενότητα του work sharing. Συνδυάζοντας τα αποτελέσματα, θα βρούμε ένα άνω όριο για το συνολικό χρόνο εκτέλεσης μιας εφαρμογής που δρομολογεί τις παράλληλες εργασίες της με το work stealing αλγόριθμο.

Ο προσδιορισμός του απαιτούμενου χρόνου εκτέλεσης ενός *fully strict multitasking υπολογισμού* με *work* T_1 και *critical-path* T_∞ σε ένα σύστημα P επεξεργαστών χρησιμοποιώντας τον work stealing αλγόριθμο της ενότητας 4.1.1 ακολουθεί την εξής μεθοδολογία: Θεωρούμε τρία καλάθια στα οποία κάθε επεξεργαστής θα τοποθετεί μία μονάδα στο τέλος κάθε βήματος ανάλογα με την ενέργεια που έχει πραγματοποιήσει στο συγκεκριμένο βήμα. Τα ονόματα των καλαθιών είναι ΕΡΓΑΣΙΑ, ΚΛΟΠΗ και ΑΝΑΜΟΝΗ κατ' αντιστοιχία με τις τρεις πιθανές ενέργειες των επεξεργαστών. Έτσι, αν κάποιος επεξεργαστής εκτελέσει μια εντολή, τότε θα τοποθετήσει μια μονάδα στο καλάθι ΕΡΓΑΣΙΑ στο τέλος του βήματος. Αν προσπαθήσει να κλέψει κάποιο task, επιτυχώς ή ανεπιτυχώς δεν

έχει σημασία, θα βάλει μια μονάδα στο καλάθι ΚΛΟΠΗ, ενώ αν περιμένει να ικανοποιηθεί το αίτημα της κλοπής του (αυτή η περίπτωση μπορεί να συμβεί όταν συναγωνίζεται με άλλους επεξεργαστές για το ίδιο task όπως θα δούμε παρακάτω), τότε μια μονάδα θα προστεθεί στο καλάθι ΑΝΑΜΟΝΗ.

Στο τέλος της εκτέλεσης θα μετρήσουμε όλες τις μονάδες που περιέχουν τα τρία καλάθια και θα τις διαιρέσουμε με το πλήθος των επεξεργαστών για να εξάγουμε το συνολικό χρόνο εκτέλεσης του υπολογισμού. Η εργασία μας είναι να φράξουμε το πλήθος των μονάδων κάθε καλαθιού, ώστε να βρούμε το άνω όριο του χρόνο υπολογισμού. Ας δούμε κάθε καλάθι χωριστά.

Αρχικά για το καλάθι ΕΡΓΑΣΙΑ. Τα πράγματα είναι αρκετά απλά, καθώς στο καλάθι προστίθεται μια μονάδα για κάθε εντολή που εκτελείται. Επειδή όλες οι εντολές εκτελούνται ακριβώς μια φορά και το συνολικό πλήθος εντολών είναι T_1 τότε στο τέλος το καλάθι εργασία θα έχει ακριβώς T_1 μονάδες. Άρα:

$$T_{ΕΡΓ} = T_1 \quad (1)$$

Συνεχίζουμε με το καλάθι ΚΛΟΠΗ. Ο υπολογισμός εδώ είναι αρκετά περίπλοκος, γι' αυτό θα δώσουμε μια απλοποιημένη εκδοχή του. Θα χρειαστεί να ορίσουμε τέσσερις έννοιες: το *γύρο κλοπής,* τον *επαυξημένο dag G',* την *critical εντολή* και την *ακολουθία καθυστέρησης*. Αναλυτικά:

Ως *γύρο κλοπής* ορίζουμε ένα σύνολο από 3P έως 4P συνεχόμενες απόπειρες κλοπής κάποιου task έτσι ώστε όλες οι απόπειρες ενός βήματος του αλγορίθμου να ανήκουν στον ίδιο γύρο. Ένας γύρος πρέπει να ξεκινάει, δηλαδή, με κάποιο βήμα του αλγορίθμου και να ολοκληρώνεται σε κάποιο άλλο. Οπότε, ο πρώτος γύρος περιέχει όλες τις απόπειρες κλοπής των βημάτων $1,2,\ldots,t_1$, ο δεύτερος γύρος των βημάτων $t_1+1, t_1+2\ldots, t_2$ κοκ. Δεδομένου ότι υπάρχουν P επεξεργαστές, εκ των οποίων τουλάχιστον ο ένας έχει εργασία σε κάθε βήμα του αλγορίθμου, για να ολοκληρωθεί ένας γύρος κλοπής απαιτούνται τουλάχιστον 4 βήματα, όπου θα έχουμε 4P-4 το πολύ απόπειρες κλοπής.

Ως *επαυξημένο dag G'* ορίζουμε το *dag* που προκύπτει όταν προσθέσουμε κάποιες επιπλέον ακμές στο *dag G* του υπολογισμού που κατασκευάσαμε στην ενότητα 3.1, τις οποίες θα ονομάζουμε *deque ακμές*. Η κατασκευή του γίνεται ως εξής: για κάθε σύνολο εντολών u, v και w του αρχικού dag για τις οποίες η (u,v) είναι μια *spawn ακμή* (δηλαδή η u είναι η εντολή του task-πατέρα που γεννάει το task-παιδί με πρώτη εντολή τη v) και η (u,w) είναι μια *ακμή εξάρτησης* (οι u και w είναι δηλαδή δυο συνεχόμενες εντολές του ίδιου task), θα δημιουργήσουμε την ακμή (w,v) και θα την ονομάζουμε *deque ακμή.* Ας δούμε κάποιες παρατηρήσεις για το γράφο G'. Πρώτον, ο G' θα παραμείνει dag λόγω της υπόθεσης ότι στη w δεν καταλήγει καμία *join ακμή*, όπως είδαμε στην ενότητα 3.1 όταν κατασκευάσαμε το *dag,* οπότε αποκλείεται η ύπαρξη κύκλου. Δεύτερον, αν το *critical-path* του G ήταν T_∞ , τότε του G' θα είναι το πολύ $2T_\infty$. Τρίτον, ο G' δεν έχει καμία σχέση με τη δρομολόγηση των tasks από το work stealing αλγόριθμο, αλλά είναι ένα αναλυτικό εργαλείο για τη μελέτη του χρόνου κλοπής και αυτή και μόνο είναι η χρησιμότητά του.

Ως *critical εντολή* ορίζουμε μια εντολή v όταν κάθε εντολή w που προηγείται από αυτήν στο G' έχει εκτελεστεί. Με άλλα λόγια, κάθε εντολή στο μονοπάτι w-v έχει εκτελεστεί. Είναι προφανές ότι η critical εντολή είναι έτοιμη προς εκτέλεση, αφού όλες οι εξαρτήσεις της έχουν ικανοποιηθεί. Αυτό βέβαια δε συνεπάγεται ότι και κάθε έτοιμη προς εκτέλεση εντολή είναι και critical. Ακόμη, με βάση τη σειρά των tasks στη deque, κανένα task που βρίσκεται στο βάθος της deque δεν μπορεί να έχει critical εντολές. Για την ακρίβεια, σε κάθε βήμα της εκτέλεσης ενός *fully strict multitasking υπολογισμού* κάθε critical εντολή είναι η έτοιμη προς εκτέλεση εντολή ενός task που έχει το πολύ 1 task από πάνω του στη deque του επεξεργαστή του. Η τελευταία πρόταση δεν είναι προφανής, αλλά θα χρειαστεί να την αποδείξουμε.

Αν θεωρήσουμε v_0 την critical εντολή ενός task Γ_0 , τότε επειδή η v_0 είναι έτοιμη προς εκτέλεση εξ ορισμού, τότε και το Γ_0 θα είναι έτοιμο προς εκτέλεση. Επομένως, το task είτε θα βρίσκεται στη deque κάποιου επεξεργαστή είτε θα εκτελείται σε αυτόν. Αν βρίσκεται στη deque θα θεωρήσουμε τα από πάνω του tasks $\Gamma_{1,}\Gamma_{2,}\ldots,\Gamma_k$, τα οποία όπως αποδείξαμε είναι πρόγονοί του. Το δεύτερο πόρισμα αυτού του θεωρήματος ανέφερε ότι κανένα task, εξαιρουμένου ίσως αυτού που βρίσκεται στην κεφαλή της deque, δεν έχει εκτελεστεί από τότε που εκτέλεσε το *spawn* για τη δημιουργία του παιδιού του. Για να είναι, όμως, μια εντολή critical πρέπει να έχει εκτελεστεί και η αμέσως επόμενη εντολή από το *spawn* οποιουδήποτε προγόνου. Επομένως, το Γ_0 το πολύ να έχει ένα task από πάνω του στη deque.

Ως *ακολουθία καθυστέρησης* ορίζουμε την τριάδα (U,R,Π) τέτοια ώστε:

- U = ($v_{1,}v_{2,}..,v_L$) είναι ένα μονοπάτι στο G', έτσι ώστε στη v_1 να μην εισέρχεται καμία ακμή (οπότε πρέπει να είναι η πρώτη εντολή του task-ρίζα) και να μην εξέρχεται καμία ακμή από τη v_L (οπότε πρέπει να είναι η τελευταία εντολή του task-ρίζα).
- R είναι ένας θετικός ακέραιος που δηλώνει το συνολικό πλήθος των γύρων κλοπής
- Π = ($\pi_{1,}\pi'_{1,}\pi_{2,}\pi'_{2,}..,\pi_L,\pi'_L$) είναι μια διαμέριση του R τέτοια ώστε

$$R=\sum_{i=1}^{L}(\pi_i+\pi'_i) \quad \text{και}$$

$$\pi'_i=\ 0\ \ \acute{\eta}\ \ 1\quad \text{για κάθε}\quad i=1,2,\ldots,L$$

Η διαμέριση αυτή γίνεται έτσι ώστε κάθε π_i γύρους κλοπής να παραμένει ίδια η critical εντολή v_i .

Με βάση τους παραπάνω ορισμούς, μπορούμε να κάνουμε δυο ισχυρισμούς:

a) Σε ένα *fully strict multitasking υπολογισμό* που δρομολογεί τις παράλληλες εργασίες με το work stealing αλγόριθμο, αν πραγματοποιηθούν τουλάχιστον 4PR απόπειρες

κλοπής tasks κατά τη διάρκεια της εκτέλεσης, τότε θα προκύψει τουλάχιστον μια ακολουθία καθυστέρησης.

b) Η πιθανότητα να παραμένει μια εντολή v_i critical σε ένα πλήθος r γύρων κλοπής είναι μικρότερη από την τιμή e^{-2r+3} .

Οι αποδείξεις των ισχυρισμών αυτών μπορούν να βρεθούν εδώ [31]. Αυτό που μας ενδιαφέρει είναι ότι είναι πολύ μικρή η πιθανότητα να παραμένει ίδια μια critical εντολή σε πολλούς γύρους κλοπής, επομένως μικρή είναι και η πιθανότητα να προκύψει τελικά κάποια ακολουθία καθυστέρησης, δεδομένου ότι η ακολουθία καθυστέρησης ορίζεται με βάση γύρους κλοπής που κρατούν την ίδια critical εντολή. Έτσι, ολοκληρώνοντας την ανάλυση, προκύπτει μέσα από τους ισχυρισμούς που κάναμε ότι θα πραγματοποιηθούν συνολικά $O(P(T_\infty+\log(1/\varepsilon)))$ απόπειρες κλοπής για κάθε ε>0 και με πιθανότητα τουλάχιστον 1-ε. Αυτό σημαίνει ότι ο αναμενόμενος αριθμός απόπειρων κλοπής είναι $O(PT_\infty)$ και τόσες θα είναι και οι μονάδες στο καλάθι ΚΛΟΠΗ. Δηλαδή:

$$T_{K\Lambda} \le O(P(T_\infty+\log(1/\varepsilon))) \quad \text{ή} \quad T_{K\Lambda} \le PT_\infty \quad (2)$$

Τέλος θα ασχοληθούμε με το καλάθι ΑΝΑΜΟΝΗ. Όπως είπαμε, η καθολική ουρά στην οποία είχαν πρόσβαση οι επεξεργαστές για να εισάγουν και να εξάγουν tasks, στο work stealing διαμοιράζεται σε καθένα επεξεργαστή ξεχωριστά με τη μορφή της deque. Αυτή η τεχνική οδηγεί στην αποφυγή του χρονικού κόστους των επεξεργαστών όταν συναγωνίζονται για ταυτόχρονη πρόσβαση στην καθολική ουρά. Αυτό, όμως, δεν εξαλείφει όλες τις περιπτώσεις συναγωνισμού. Για την ακρίβεια, στο work stealing υπάρχει περίπτωση δυο ή περισσότεροι επεξεργαστές να συναγωνίζονται να κλέψουν ταυτόχρονα το ίδιο task από το ίδιο θύμα. Ο συναγωνισμός αυτός πρέπει να επιλυθεί με κάποιον τρόπο. Για την ανάλυσή του θα χρησιμοποιήσουμε ένα μοντέλο ατομικής πρόσβασης για ένα ασύγχρονο, παράλληλο σύστημα P επεξεργαστών, του οποίου η μνήμη μπορεί να είναι μοιραζόμενη ή κατανεμημένη [32]. Επιλέγουμε όλες οι αιτήσεις να μπαίνουν σε μια σειρά από έναν *αντίπαλο*, ο οποίος θα καθορίζει τον τρόπο με τον οποίο θα εξυπηρετηθούν. Όταν υπάρχουν πολλές αιτήσεις για κλοπή του ίδιου task μόνο μια από αυτές μπορεί να εξυπηρετηθεί σε κάθε βήμα. Οι υπόλοιπες, αντί να απορρίπτονται [33], επιλέγουμε να εισέλθουν σε μια ουρά από έναν *αντίπαλο*. Ο αντίπαλος επιλέγει σε κάθε βήμα ποια αίτηση από αυτές που υπάρχουν στην ουρά θα εξυπηρετηθεί, με μοναδικό περιορισμό του ότι αν υπάρχει τουλάχιστον μια αίτηση κλοπής για μια deque, πρέπει να την ικανοποιήσει.

Ο αντίπαλος χρησιμοποιεί πάντα την καλύτερη στρατηγική ώστε να μεγαλώσει την καθυστέρηση των επεξεργαστών. Ανεξάρτητα, όμως, από το ποια στρατηγική θα επιλέξει και από το ποιοι επεξεργαστές θα κάνουν τις απόπειρες κλοπής αποδεικνύεται ότι ο συνολικός χρόνος αναμονής των επεξεργαστών θα είναι αναλογικός με το πλήθος M των αιτήσεων για κλοπή κάποιου task [31]. Πιο συγκεκριμένα: Η συνολική καθυστέρηση που προκαλείται από M αιτήσεις κλοπής που γίνονται τυχαία από P επεξεργαστές στο μοντέλο

ατομικής πρόσβασης είναι $O(M+PlogP+Plog(1/\varepsilon))$ με πιθανότητα 1-ε για κάθε ε>0. Επειδή το ε μπορεί να γίνει αρκετά μικρό και το πλήθος των επεξεργαστών αρκετά μικρότερο από τις συνολικές αιτήσεις (M>>P) , τελικά ο αναμενόμενος χρόνος αναμονής είναι M.

$$T_{AN} \le O(M+PlogP+Plog(1/\varepsilon)) \quad \text{ή} \quad T_{AN} \le M \quad (3)$$

Αν προσθέσουμε τις (1), (2) και (3) και τις διαιρέσουμε με το πλήθος των επεξεργαστών μπορούμε να βρούμε το άνω όριο για το συνολικό χρόνο του work stealing αλγορίθμου. Έτσι

$$T_P = \frac{T_{ΕΡΓ}+T_{ΚΛ}+T_{AN}}{P} = \frac{T_1}{P} + O(T_\infty + \log(1/\varepsilon)) + O(\frac{M}{P} + logP + \log(1/\varepsilon))$$

και αν δούμε τους αναμενόμενους χρόνους, καθώς επίσης και ότι ο χρόνος αναμονής είναι το πολύ ένας σταθερός αριθμός πολλαπλασιασμένος με το χρόνο κλοπής τότε προκύπτει η τελική ανίσωση:

$$T_P \le \frac{T_1}{P} + O(T_\infty)$$

4.2 Η Work-First Αρχή

Η work-first αρχή είναι μια αρχή για τον τρόπο κατανομής του κόστους δρομολόγησης των εργασιών έτσι ώστε να επιτευχθεί καλύτερη επίδοση στο χρόνο εκτέλεσης εφαρμογών. Αφορά τη Cilk καθώς πάνω σε αυτή στηρίχτηκε η υλοποίησή της, όπως θα δούμε στην επόμενη ενότητα. Σε αυτό την ενότητα θα αναλύσουμε το λόγο που επιλέχτηκε η work-fisrt αρχή από τη Cilk με στόχο την αύξηση της αποδοτικότητάς της.

Work-Fisrt Αρχή: Για την αύξηση της αποδοτικότητας δρομολόγησης και εκτέλεσης παράλληλων εργασιών χρειάζεται να ελαχιστοποιηθεί το κόστος δρομολόγησης που προκύπτει από το *work* κομμάτι του υπολογισμού. Πιο συγκεκριμένα, όλα τα κόστη δρομολόγησης να μεταφερθούν από το *work* στο *critical-path.*

Με βάση τις αναλύσεις που έχουμε κάνει για το χρόνο δρομολόγησης work sharing και work stealing αλγορίθμων μπορούμε να φανταστούμε ότι δεν είναι τυχαία η επιλογή αυτής της αρχής. Για την ακρίβεια, η work-first αρχή προκύπτει από τρεις υποθέσεις. Πρώτον, οι work stealing αλγόριθμοι που υλοποιούμε (και πιο ειδικά ο work stealing αλγόριθμος της

Cilk) ακολουθούν στην πράξη τις θεωρητικές αναλύσεις της προηγούμενης ενότητας. Δεύτερον, υπάρχει *parallel slackness* για τις εφαρμογές που εκτελούνται παράλληλα [34]. Τρίτον, το κόστος εκτέλεσης της C elision είναι συγκρίσιμο με εκείνο του αντίστοιχου C προγράμματος.

Η πρώτη υπόθεση έχει ήδη μελετηθεί αρκετά. Αφορά το άνω και κάτω φράξιμο του χρόνου εκτέλεσης ενός παράλληλου προγράμματος από το *work* και από το *critical-path.* Δηλαδή:

$$T_P \geq \frac{T_1}{P} \quad \text{και} \quad T_P \geq T_\infty \quad \text{(κάτω φράγμα - ενότητα 3.2)}$$

$$T_P \leq \frac{T_1}{P} + O(T_\infty) \quad \text{(άνω φράγμα – ενότητα 4.1.2)}$$

Το άνω φράγμα μπορούμε επίσης να το γράψουμε και με την παρακάτω ανισότητα:

$$T_P \leq \frac{T_1}{P} + c_\infty T_\infty \quad (4)$$

όπου c_∞ μια πολύ μικρή σταθερά. Η work-first αρχή υποδεικνύει την αύξηση του T_∞ αντί του T_1 δεδομένου ότι θα εξασφαλιστεί μια πολύ μικρή σταθερά c_∞ ώστε τελικά να επιτυγχάνεται γραμμική επιτάχυνση του χρόνου εκτέλεσης σε P επεξεργαστές. Δηλαδή το T_1 πρέπει να είναι όσο μικρότερο δυνατό γίνεται.

Η δεύτερη υπόθεση αφορά το λόγο που θεωρείται ότι η σταθερά c_∞ θα είναι αρκετά μικρή. Θα ορίσουμε δυο μεγέθη: το *average parallelism* (μέσο παραλληλισμό) $\bar{P}$ ενός προγράμματος και την *parallel slackness* (αργοπορία παραλληλισμού). Ως *average parallelism* ορίζουμε το λόγο του *work* χρόνου ως προς τον *critical-path* χρόνο, δηλαδή:

$$\bar{P} = \frac{T_1}{T_\infty}$$

Αυτό το μέγεθος αντιπροσωπεύει τη μέγιστη επιτάχυνση που μπορεί να αποκτήσει η συγκεκριμένη εφαρμογή. Ως *parallel slackness* ορίζουμε το λόγο του *average parallelism* ως προς το πλήθος P των επεξεργαστών. Δηλαδή

$$\text{parallel slackness} = \frac{\bar{P}}{P}$$

Η υπόθεση που κάνουμε είναι ότι ο παραλληλισμός ενός προγράμματος, δηλαδή το πλήθος των ανεξάρτητων tasks στα οποία μπορεί να διαιρεθεί, είναι πολύ μεγαλύτερος από το πλήθος των επεξεργαστών του συστήματος. Πιο μαθηματικά, αυτός ο λόγος είναι πολύ

μεγαλύτερος από τη σταθερά c_∞ , του συντελεστή του *critical-path* από το οποίο συνεπάγεται ότι και ο συνολικός χρόνος του *work* κομματιού είναι πολύ μεγαλύτερος από το συνολικό χρόνο του *critical-path:*

$$\frac{\bar{P}}{P} \gg c_\infty \Rightarrow \frac{T_1}{P} \gg c_\infty T_\infty$$

Με αυτήν την υπόθεση γίνεται φανερό ότι ο συνολικός χρόνος εκτέλεσης θα είναι ίσος με τον πρώτο όρο της ανισότητας (4) δεδομένου ότι υπάρχει αρκετή *parallel slackness.*

$$T_P \approx \frac{T_1}{P} \quad (5)$$

Αυτό φυσικά εξαρτάται από τη φύση κάθε εφαρμογής και το πόσο αυτή παραλληλοποιείται. Στη γενική περίπτωση, όμως, αυτό ισχύει και μάλιστα ο βαθμός παραλληλισμού των εφαρμογών συνήθως αυξάνει με την αύξηση του μεγέθους εισόδου. Αυτό που οφείλουμε να έχουμε υπόψη είναι ότι ο βαθμός παραλληλισμού είναι σχετικός και πρέπει να είναι τόσος ώστε να μπορεί η *parallel slackness* να είναι αρκούντως μεγαλύτερη της σταθεράς c_∞ που πρακτικά επηρεάζεται από το επιπλέον κόστος που προσθέτουμε στο *critical-path.*

Η τρίτη και τελευταία υπόθεση από την οποία πηγάζει η ιδέα της *work-first αρχής* είναι ότι το πρόγραμμα που προκύπτει από το cilk κώδικα αν αφαιρεθούν οι λέξεις-κλειδιά και αντικατασταθούν με τις αντίστοιχες κλήσεις στο RTS της Cilk, η C elision δηλαδή, έχει καλή και μετρήσιμη επίδοση. Αν ορίσουμε ως T_S το χρόνο εκτέλεσης της C elision σε έναν επεξεργαστή, τότε ο συντελεστής του *work* κόστους θα είναι

$$c_1 = \frac{T_1}{T_S}$$

Αν επεξεργαστούμε την ανίσωση (4) θα προκύψει ότι:

$$T_P \leq c_1 \frac{T_S}{P} + c_\infty T_\infty \approx c_1 \frac{T_S}{P}$$

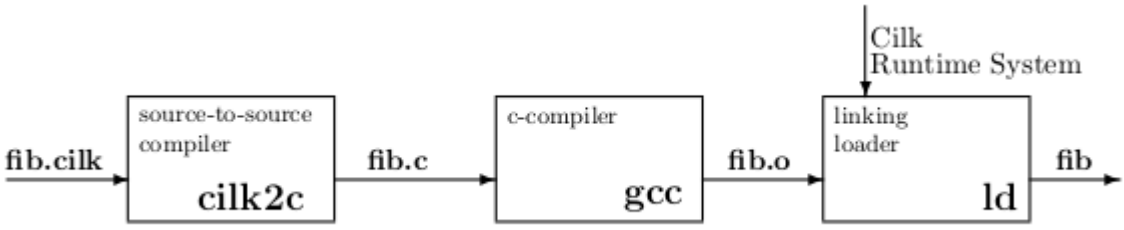
που στην πράξη είναι ίδια με την ισότητα (5).

Χρησιμοποιώντας πλέον τους ορισμούς που δώσαμε σύμφωνα με τις τρεις υποθέσεις που υποδεικνύουν τη *work-first αρχή*, μπορούμε να τη διατυπώσουμε με πιο μαθηματικό τρόπο: ελαχιστοποίηση του c_1 ακόμη κι αν χρειαστεί να αυξηθεί το c_∞ , γιατί το c_1 παίζει σημαντικότερο ρόλο στην απόδοση.

4.3 Υλοποίηση της γλώσσας Cilk

Μπορούμε χοντρικά να χωρίσουμε την υλοποίηση κάθε γλώσσας παράλληλου προγραμματισμού σε δυο κομμάτια: το μεταγλωττιστή και το RTS. Ο τρόπος που δομούνται είναι ξεχωριστός για κάθε γλώσσα και καθορίζει τα χαρακτηριστικά που θα έχει, άρα και για το ποιες εφαρμογές ενδείκνυται η χρησιμοποίησή της. Για γενικότερες πληροφορίες πάνω στις διαφορετικές προσεγγίσεις των γλωσσών παράλληλου προγραμματισμού παραπέμπουμε εδώ [35]. Σε αυτήν την ενότητα εμείς θα μελετήσουμε τον τρόπο που υλοποιείται η γλώσσα Cilk.

Η Cilk χρησιμοποιεί ένα *source-to-source* μεταγλωττιστή, ο οποίος μετατρέπει ένα πρόγραμμα γραμμένο σε Cilk σε ένα αντίστοιχο γραμμένο σε C, που καλεί συναρτήσεις από τη βιβλιοθήκη της Cilk. Πρόκειται για έναν ενδιάμεσο μεταγλωττιστή, ο οποίος κάνει type-checking, τρέχει γρήγορα, παράγει αποδοτικό κώδικα, χρησιμοποιώντας τεχνικές βελτιστοποίησης, είναι εύκολο να αναπτυχθεί, να επεκταθεί και να μεταφερθεί σε άλλες πλατφόρμες και το όνομά του είναι *cilk2c compiler* [36]. Για να ολοκληρωθεί η μεταγλώττιση ενός Cilk προγράμματος, ο κώδικας που παράγεται από τον cilk2c μεταγλωττίζεται εκ νέου από το gcc compiler που χρησιμοποιεί η C. Τέλος, για να δημιουργηθεί το εκτελέσιμο αρχείο, το assembly αρχείο που παράγεται από το gcc συνδέεται με το RTS της Cilk. Η διαδικασία που ακολουθείται (και η οποία γίνεται αυτόματα χρησιμοποιώντας ο προγραμματιστής την εντολή *cilkc*) παρουσιάζετα στο Σχήμα 15.



Σχήμα 15: Διαδρομή μεταγλώττισης ενός προγράμματος cilk με ονομασία fib.cilk

Αρχικά το πρόγραμμα που περιέχει τον κώδικα της cilk (στο σχήμα το fib.cilk) περνάει από τον ενδιάμεσο μεταγλωττιστή cilk2c για να παραχθεί το ενδιάμεσο πρόγραμμα (fib.c) που περιέχει κλήσεις συναρτήσεων και δομών του RTS της Cilk. Αυτό με τη σειρά του μεταγλωττίζεται από τον gcc και το ενδιάμεσο αρχείο σε assembly (fib.o) περνάει από έναν linker που το ενώνει με το RTS της Cilk ώστε τελικά να προκύψει το εκτελέσιμο αρχείο (fib).

Το RTS της Cilk αποτελείται από ένα σύνολο συναρτήσεων και δομών δεδομένων τα οποία απαρτίζουν το δρομολογητή της, που χειρίζεται τα υπάρχοντα tasks. Στόχος είναι η δρομολόγηση να αξιοποιήσει καλύτερα τις μονάδες του συστήματος. Η *work-first αρχή* οδήγησε στη δημιουργία ενός runtime δρομολογητή κοινής μνήμης εμπνευσμένο από την

εργασία του Dijkstra για τον αμοιβαίο αποκλεισμό κυκλικών εργασιών [37]. Ο δρομολογητής της Cilk χρησιμοποιεί ένα work stealing αλγόριθμο, μεταφέροντας σχεδόν όλα τα κόστη δρομολόγησης στο *critical-path.* Μόνο τα κόστη του αμοιβαίου αποκλεισμού (όταν δυο επεξεργαστές συναγωνίζονται για το ίδιο task) εμπίπτουν στο *work* κομμάτι και για να ελαχιστοποιηθούν, οι δημιουργείς της Cilk αποφάσισαν να μη χρησιμοποιήσουν κλειδώματα, αλλά ένα πρωτόκολλο στις βάσεις του πρωτοκόλλου του Dijkstra, το οποίο ονομάζεται THE.

Ας δούμε αναλυτικά αυτούς τους δυο πυλώνες στους οποίους στηρίζεται η υλοποίηση της Cilk.

4.3.1 Μεταγλώττιση στη Cilk

Ακολουθεί η περιγραφή της στρατηγικής που ακολουθεί ο ενδιάμεσος cilk2c μεταγλωττιστής για την παραγωγή του κώδικα σε C. Ο μεταγλωττιστής υπακούει στη *work-first αρχή,* οπότε είναι σχεδιασμένος να περιορίζει στο ελάχιστο δυνατό το κόστος από το *work* κομμάτι της εκτέλεσης. Μεταγλωττίζοντας μια cilk διεργασία μπορεί να δημιουργήσει δυο κλώνους της, ένα *fast clone* κι ένα *slow clone.* Ο *fast clone* χρησιμοποιείται για την τοπική δρομολόγηση των εργασιών στον ίδιο επεξεργαστή και δεν προσφέρει ιδιαίτερο παραλληλισμό, συμπεριφερόμενος περισσότερο όπως η C elision. Αντίθετα, ο *slow clone* δημιουργείται γιατί υποστηρίζει πλήρη παραλληλισμό ώστε να αξιοποιηθεί στην καθολική δρομολόγηση, αλλά γι' αυτόν ακριβώς το λόγο έχει μεγαλύτερο κόστος διαχείρισης.

Όταν καλείται μια cilk διεργασία μέσω της λέξης *spawn* ενεργοποιείται και εκτελείται ο *fast clone.* Μόλις κλαπεί μια διεργασία από έναν άλλο επεξεργαστή ο *fast clone* μετατρέπεται σε *slow clone* για να μπορεί να εκτελεστεί παράλληλα. Ο αριθμός των κλοπών των εργασιών είναι σχετικά μικρός, καθώς ο δρομολογητής της Cilk κλέβει πάντα από την κορυφή της deque, δηλαδή τα tasks που βρίσκονται σε υψηλότερο επίπεδο στο *spaw tree.* Η Cilk, μάλιστα, εγγυάται ότι αν υπάρχει αρκετή *slackness,* οι περισσότερες εργασίες θα εκτελούνται ως *fast clones*. Βασικός στόχος, λοιπόν, είναι η ελαχιστοποίηση του κόστους των *fast clones,* που είναι αυτοί που συνεισφέρουν στο κόστος του *work* και, μολονότι θα ήταν επιθυμητό ομοίως να ελαχιστοποιηθεί και το κόστος των *slow clones,* κάτι τέτοιο δεν είναι το ίδιο σημαντικό, αφού συνεισφέρει περισσότερο στο *critical-path* παρά στο *work.*

➔ *Fast Clone*

Ένας *fast clone* δεν έχει κλαπεί ποτέ, καθώς με το που κλαπεί ένα task, μετατρέπεται αυτόματα στο *slow clone*. Με βάση αυτήν τη θεώρηση προκύπτει ότι και κάθε task που έχει γεννηθεί από το συγκεκριμένο task έχει κι αυτό μεταγλωττιστεί ως *fast clone,* καθώς πρώτα

κλέβονται οι γονείς από τη deque σύμφωνα με τη στρατηγική του work stealing αλγορίθμου. Λαμβάνοντας υπόψη αυτό το γεγονός, οι σχεδιαστές της Cilk μπόρεσαν να κάνουν αρκετές παραδοχές και να απλοποιήσουν τη σχεδίαση του *fast clone,* μειώνοντας το πλήθος των επιπλέον εντολών και βελτιώνοντας έτσι την απόδοσή του.

Ο *fast clone* έχει παρόμοια δομή με τη συνάρτηση της C elision με μερικές επιπλέον εντολές. Αρχικά, δημιουργεί μια *εγγραφή δραστηριοποίησής* για την κληθείσα συνάρτηση και την αρχικοποιεί. Αυτή η εγγραφή είναι πολύ συγκεκριμένη για τις διεργασίες της Cilk και για την αρχικοποίησή της χρησιμοποιείται μια στατική δομή που λέγεται *υπογραφή,* η οποία περιγράφει τη συνάρτηση και περιέχει ένα δείκτη στο *slow clone*. Ένα *spawn* θα μεταφραστεί με την ακόλουθη σειρά εντολών:

- αποθήκευση της κατάστασης της συνάρτησης (του μετρητή προγράμματος και των μεταβλητών της) στην *εγγραφή δραστηριοποίησής*
- η εγγραφή γίνεται pushed στη deque του επεξεργαστή
- καλείται η νέα συνάρτηση-παιδί, όπως ακριβώς στη C. Μάλιστα, για την κληθείσα συνάρτηση-παιδί μπορούν να χρησιμοποιηθούν οι ίδιες τεχνικές βελτιστοποίησης που χρησιμοποιεί ο μεταγλωττιστής της C και για κάθε κανονική συνάρτηση, όπως το πέρασμα παραμέτρων και η επιστροφή τιμών μέσω καταχωρητών του επεξεργαστή κι όχι μέσω μνήμης.

Η συνάρτηση-παιδί εκτελείται κανονικά και όταν επιστρέψει γίνεται έλεγχος αν η συνάρτηση-πατέρας έχει κλαπεί. Αν ισχύει κάτι τέτοιο, επιστρέφεται μια dummy value και ο έλεγχος περνάει πίσω στο RTS της Cilk. Αν όχι, τότε συνεχίζει να εκτελείται κανονικά ο *fast clone* της εργασίας κάνοντας τις απαραίτητες ενέργειες ανάλογα με το εκάστοτε πρόγραμμα. Όλες οι δηλώσεις *sync* μεταφράζονται στον *fast clone* ως κενές εντολές (κοινώς με ;). Αυτή η απλοποίηση προκύπτει από το γεγονός ότι αφού δεν έχει κλαπεί ο πατέρας, αποκλείεται να έχει κλαπεί οποιοδήποτε από τα παιδιά του. Στην τοπική δρομολόγηση τα παιδιά εκτελούνται πριν από τους γονείς και, επειδή όλοι οι υπολογισμοί που κάνουμε είναι *strict,* κανένα από τα παιδιά του πατέρα μέχρι το *sync* δε θα βρίσκεται σε αναμονή, αλλά όλα θα έχουν τερματίσει. Επομένως το *sync* είναι μια τετριμμένη (και άχρηστη) δήλωση σε αυτές τις περιπτώσεις και μπορεί να αμεληθεί. Στο τέλος, και πριν επιστρέψει ο *fast clone,* απελευθερώνονται από τη μνήμη ο χώρος που δεσμεύτηκε για την *εγγραφή δραστηριοποίησής*.

Στο Σχήμα 16 δίνεται το παράδειγμα μιας cilk διεργασίας που υπολογίζει το n-οστό αριθμό fibonacci με χρήση αναδρομής και δίπλα είναι μια απλοποιημένη εκδοχή του *fast clone* που προκύπτει από το cilk2c μεταγλωττιστή (χωρίς τον κώδικα για το δεύτερο *spawn*). Όλα τα βήματα για τη δημιουργία του που περιγράφηκαν παραπάνω αντιστοιχίζονται στο παράδειγμα με τον ακόλουθο τρόπο:

- γραμμές 3-5: δημιουργία και αρχικοποίηση *εγγραφής δραστηριοποίησής* (fib_frame)
- γραμμές 12-15: μετάφραση του *spawn*
- γραμμές 16-17: έλεγχος αν η εργασία έχει κλαπεί
- γραμμή 19: η μετάφραση του *sync* σε κενή εντολή

- γραμμές 7-8, 20-21: αποδέσμευση της *εγγραφής δραστηριοποίησής* και επιστροφή της συνάρτησης.

```
cilk int fib (int n)
{
    if (n<2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
```

```
1  int fib (int n)
2  {
3      fib_frame *f;
4      f = alloc(sizeof(*f));
5      f->sig = fib_sig;
6      if (n<2) {
7          free(f, sizeof(*f));
8          return n;
9      }
10     else {
11         int x, y;
12         f->entry = 1;
13         f->n = n;
14         push(f);
15         x = fib (n-1);
16         if (pop(x) == FAILURE)
17             return 0;
18         ...
19         ;
20         free(f, sizeof(*f));
21         return (x+y);
22     }
23 }
```

Σχήμα 16: Αριστερά η διεργασία της Cilk για παραγωγή του ν-οστού αριθμού fibonacci με αναδρομικό τρόπο και δεξιά η απλοποιημένη εκδοχή του *fast clone* που παράγει ο cilk2c μεταγλωττιστής

➔ *Slow Clone*

Ο *slow clone* μοιάζει σε κάποια σημεία με το *fast clone*, αλλά συγκριτικά έχει πολύ υψηλότερο κόστος, αφού για τη δημιουργία του δεν μπορούν να γίνουν οι παραδοχές που γίνονται στο *fast clone*. Για τη δημιουργία του *slow clone* αλλάζουν πολλά πράγματα στον κώδικα λόγω της απαίτησης της υποστήριξης του παραλληλισμού. Ο *slow clone* δημιουργείται όταν κλαπεί ένα task. Αυτό σημαίνει ότι ένα μέρος του κώδικα της

συνάρτησης του task έχει ήδη εκτελεστεί. Έτσι, ο *slow clone* φροντίζει εξ αρχής να μεταφέρει τον έλεγχο στην τελευταία εντολή που έχει εκτελεστεί, δηλαδή είτε σε κάποια δήλωση *spawn,* οπότε από την τοπική δρομολόγηση το task θα είχε επιστρέψει στη deque για να εκτελεστεί πρώτα το παιδί του, είτε σε κάποια δήλωση *sync* όπου μπορεί να είχε μπλοκάρει περιμένοντας την επιστροφή της τιμής του παιδιού του. Αυτή η διαδικασία γίνεται μέσω μιας εντολής switch, η οποία ελέγχει την τιμή entry της *εγγραφής δραστηριοποίησής* της συνάρτησης, η οποία έχει περάσει ως όρισμα στη συνάρτηση του *slow clone.* Ανάλογα με τη τιμή του entry χρησιμοποιείται μια εντολή goto για να μεταφερθεί ο έλεγχος στο κατάλληλο σημείο. Αυτή η τεχνική μετάβασης είναι βασισμένη στη μηχανή του Duff [38], της οποίας αρχικός στόχος ήταν η αύξηση απόδοσης όταν εκτελούνται loops (loop unwinding), που όμως προφέρεται και για άλματα στη μέση ενός C κώδικα. Η μετάφραση των δηλώσεων *spawn* γίνεται ακριβώς όπως και στους *fast clones.* Εκεί που διαφέρουν είναι στη μετάφραση της δήλωσης *sync*. Στο *slow clone* γίνεται κλήση μιας μακροεντολής της _SYNC του RTS, η οποία ελέγχει αν έχουν επιστρέψει όλα τα παιδιά της συνάρτησης. Αν δεν έχουν επιστρέψει τότε ο *slow clone* επιστρέφει πίσω στο δρομολογητή και το task μπαίνει σε κατάσταση αναμονής μέχρι να ειδοποιηθεί ότι τα παιδιά του επέστρεψαν. Αν τη στιγμή του *sync* όλα τα παιδιά έχουν τερματίσει τότε αποθηκεύει την τιμή επιστροφής σε ένα προσωρινό χώρο μέσω της μακροεντολής _SET_RESULT κι έπειτα επιστέφει μια κενή τιμή.

```
struct _fib_frame {
  StackFrame header;
  struct {int n;} scope0;
  struct {int x;int y;} scope1;
};
int fib(int n)
{
  struct _fib_frame *_frame;
  _INIT_FRAME(_frame,sizeof(struct _fib_frame),_fib_sig);
  {
    if (n < 2)
      {_BEFORE_RETURN_FAST();return (n);}
    else {
      int x;int y;
      { _frame->header.entry=1;
        _frame->scope0.n=n;
        x=fib(n-1);
        _XPOP_FRAME_RESULT(_frame,0,x);
      }
      { _frame->header.entry=2;
        _frame->scope1.x=x;
        y=fib(n-2);
        _XPOP_FRAME_RESULT(_frame,0,y);
      }
      /* sync */;
      {_BEFORE_RETURN_FAST();return (x+y);}
    }
  }
}
```

```
static void _fib_slow(struct _fib_frame *_frame)
{
  int n;
  switch (_frame->header.entry) {
  case 1: goto _sync1;
  case 2: goto _sync2;
  case 3: goto _sync3;
  }
  n=_frame->scope0.n;
  {
    if (n < 2)
      {_SET_RESULT((n));_BEFORE_RETURN_SLOW();return;}
    else {
      { int _temp0;
        _frame->header.entry=1;
        _frame->scope0.n=n;
        _temp0=fib(n-1);
        _XPOP_FRAME_RESULT(_frame,/* return nothing */,_temp0);
        _frame->scope1.x=_temp0;
        if (0) { _sync1:; n=_frame->scope0.n; }
      }
      { int _temp1;
        _frame->header.entry=2;
        _temp1=fib(n-2);
        _XPOP_FRAME_RESULT(_frame,/* return nothing */,_temp1);
        _frame->scope1.y=_temp1;
        if (0) { _sync2:; }
      }
      { _frame->header.entry=3;
        if (_SYNC) {
          return;
          _sync3:;
        }
      }
      { _SET_RESULT((_frame->scope1.x+_frame->scope1.y));
        _BEFORE_RETURN_SLOW();return;
      }
    }
  }
```

Σχήμα 17: Αριστερά ο fast clone και δεξιά ο slow clone όπως παράγονται από το cilk2c μεταγλωττιστή με χρήση μακροεντολών του RTS της Cilk

Ο cilk2c μεταγλωττιστής παράγει τελικά τους δυο κλώνους (fast και slow) που φαίνονται στο Σχήμα 17 για τη συνάρτηση των fibonacci αριθμών. Η Cilk μάλιστα αναφέρει τη δημιουργία αυτών των κλώνων ως *nanoscheduling* για τους *fast clones* και *microscheduling* για τους *slow clones*. Στόχος αυτού του διαχωρισμού ήταν το τελικό κόστος του *nanoscheduling* να είναι αρκετά μικρό ώστε η απόδοση ενός cilk προγράμματος που εκτελείται σειριακά να μην έχει δραματικά μεγαλύτερο κόστος από το κόστος του αντίστοιχου c προγράμματος. Με βάση τα όσα έχουμε περιγράψει το επιπλέον κόστος προκύπτει, τελικά, από πέντε παράγοντες:

- από τη δέσμευση και αρχικοποίηση της *εγγραφής δραστηριοποίησής* στην αρχή κάθε

cilk διεργασίας

- από την αποθήκευση της κατάστασης της cilk διεργασίας πριν από κάθε *spawn*
- από τον έλεγχο έπειτα από κάθε *spawn* αν η cilk διεργασία έχει κλαπεί ή όχι
- από την αποδέσμευση της *εγγραφής δραστηριοποίησής* πριν από τον τερματισμό της cilk διεργασίας
- από τη χρησιμοποίηση μιας επιπλέον μεταβλητής για δείκτη στην *εγγραφή δραστηριοποίησής*.

Αυτό το επιπλέον κόστος προκύπτει μέσα από πειράματα ότι στην πράξη είναι αρκετά μικρό, κι ακόμη γίνεται αρκετά μικρό για προγράμματα των οποίων τα ανεξάρτητα tasks είναι αρκετά μεγάλα.

Στην ερώτηση αν μπορεί να μειωθεί κι άλλο το *work* κόστος, η απάντηση είναι θετική. Για την ακρίβεια κάτι τέτοιο έκανε η προηγούμενη έκδοση της Cilk, η σχεδίαση της οποίας έγινε με γνώμονα να υπηρετηθεί στα άκρα η *work-first* αρχή. Η Cilk-4, λοιπόν, δέσμευε μνήμη σε μορφή στοίβας (stack) κι όχι σωρού (heap) για τις *εγγραφές δραστηριοποίησής*, διότι το κόστος ήταν μικρότερο. Μάλιστα, κατέληξε να μειώσει τόσο το κόστος της δέσμευσης ώστε να είναι ελαφρώς μεγαλύτερο από το κόστος αύξησης του δείκτη στοίβας. Ακολουθώντας, όμως, αυτήν την τακτική η Cilk-4 έπρεπε να διαχειρίζεται και το χάρτη εικονικής μνήμης κάθε επεξεργαστή, γεγονός που οδηγούσε σε πολύπλοκα πρωτόκολλα για την αποδοτική διαχείριση των page fault [39]. Παρόλο που το κόστος αυτών των πρωτοκόλλων μεταβιβαζόταν στο *critical-path*, ήταν τόσο μεγάλο που παραβίαζε την παραδοχή του *parallel slackness*. Έτσι, ενώ η Cilk-4 κατάφερνε να ελαχιστοποιήσει το *work* κόστος και τα προγράμματα να εκτελούνται σε έναν επεξεργαστή σε παραπλήσιο χρόνο με τα αντίστοιχα της C, κάποιες φορές τα αποτελέσματα από την παράλληλη εκτέλεση σε πολλούς επεξεργαστές δεν παρουσίαζαν την επιθυμητή επιτάχυνση.

Η Cilk-5 έκανε κάποιες εξισορροπήσεις, αποφασίζοντας όλες οι *εγγραφές δραστηριοποίησής* να δεσμεύονται σε σωρό, τεχνική που αν και με μεγαλύτερο *work* κόστος, είχε πολύ μικρή συνεισφορά στο κόστος του *critical-path*. Ένα ακόμη αρνητικό του σωρού είναι ότι μπορεί λόγω κατακερματισμού να μην κάνει καλή διαχείριση μνήμης και τελικά να χρησιμοποιεί περισσότερη απ' ότι μια στοίβα. Παρά τα αρνητικά αυτά δεδομένα, σημαντικό είναι να δούμε ότι στις τελικές αποφάσεις απαιτείται αφενός ο συμβιβασμός των θεωρητικών αναλύσεων που έχουμε κάνει (και που πολλές φορές είναι αντικρουόμενες) και αφετέρου να μένουμε προσηλωμένοι στον κεντρικό στόχο της εφαρμογής, που στην υλοποίηση μιας γλώσσας παράλληλου προγραμματισμού είναι η επίτευξη μέγιστης απόδοσής σε συστήματα πολλών επεξεργαστών παρά λίγων. Τελικά η Cilk-5 δείχνει ότι εξισορροπεί σε πολύ καλό βαθμό όλες τις θεωρητικές αναλύσεις, προσφέροντας ταυτόχρονα καλή απόδοση και φορητότητα σε διαφορετικές πλατφόρμες.

4.3.2 Το πρωτόκολλο THE

Ακολουθώντας πάλι τη *work-first αρχή,* οι σχεδιαστές του RTS της Cilk έθεσαν στόχο την ελαχιστοποίηση του *work* κόστους και στην υλοποίηση του RTS, της "κόλλας" μεταξύ των *fast* και των *slow clones*. Επιλέχθηκε να υλοποιηθεί ένας work stealing αλγόριθμος σε μοντέλο κοινής μνήμης, μιας και η ίδια η Cilk είναι σχεδιασμένη για αρχιτεκτονικές κοινής μνήμης. Αμφότεροι κλέφτης και θύμα ενεργούν μαζί στη deque του θύματος, χωρίς να ανταλλάσσουν μεταξύ τους μηνύματα. Σύγκρουση μπορεί να προκύψει μόνο όταν κλέφτης και θύμα επιχειρούν ταυτόχρονα να πάρουν το ίδιο task από την ουρά.

Μια λύση για αυτή τη σύγκρουση είναι ένα κλείδωμα στη deque έτσι ώστε όποιος θέλει να έχει πρόσβαση σε αυτήν να την κλειδώνει. Για το κλείδωμα μπορούν να χρησιμοποιηθούν είτε hardware primitive τεχνικές, όπως Test-And-Set ή Compare-And-Swap [40], είτε κλήση στο OS. Και οι δύο αυτές μέθοδοι είναι αρκέτα δαπανηρές, καθώς όποτε χρειαστεί να γίνει μια ενέργεια στη deque (pop ή push) πρέπει να πληρωθεί το κόστος κλειδώματος.

Αποφασίστηκε, λοιπόν, να υιοθετηθεί το πρωτόκολλο THE [41], το οποίο είναι εμπνευσμένο από την εργασία του Dijkstra για αμοιβαίο αποκλεισμό εργασιών, για να επιλύσει το πρόβλημα συναγωνισμού. Το THE πήρε το όνομά του από τα ονόματα των τριών μεταβλητών που χρησιμοποιεί: T, H και E. Το πρωτόκολλο επιχειρεί να φέρει τα κόστη από το θύμα στον κλέφτη, καθώς οι ενέργειες του θύματος επηρεάζουν το *work* κόστος, ενώ εκείνες του κλέφτη το *critical-path* κόστος. Για να διευθετηθεί η προτεραιότητα μεταξύ των κλεφτών για το ποιος θα κλέψει ένα task από το ίδιο θύμα, χρησιμοποιείται ένα hardware κλείδωμα. Εδώ δε μας επηρεάζει το υψηλό κόστος του κλειδώματος γιατί αφενός περιλαμβάνεται στο *critical-path* και αφετέρου δε γίνεται ιδιαίτερα συχνά. Η προτεραιότητα ανάμεσα στο θύμα και στο μοναδικό κλέφτη καθορίζεται από το πρωτόκολλο THE, το οποίο καταφεύγει σε ένα "βαρύ" κλείδωμα μόνο όταν υπάρχει συναγωνισμός μεταξύ των δυο επεξεργαστών.

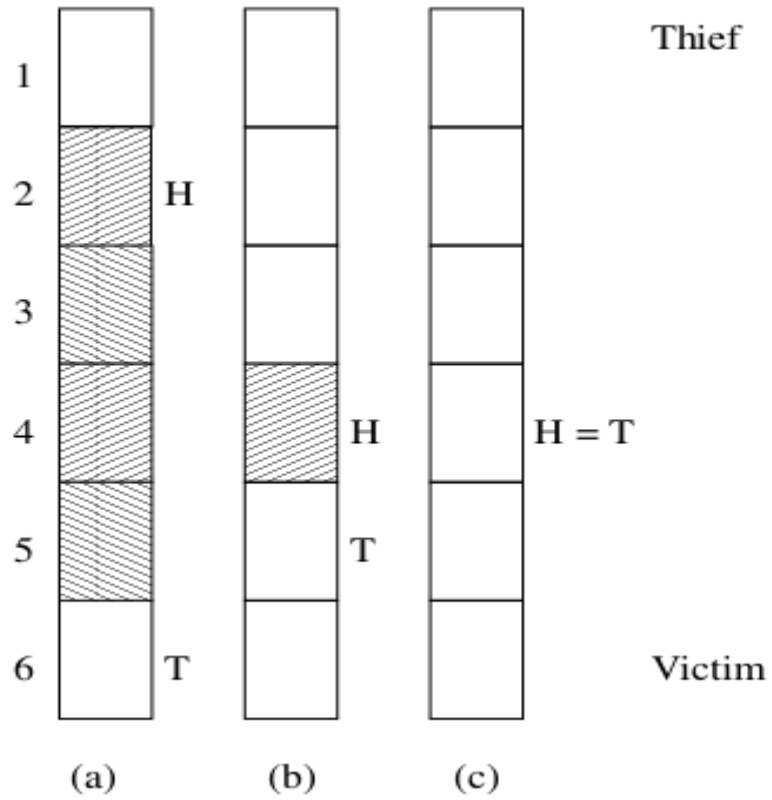
```
    Worker/Victim                          Thief
1   push() {                          1   steal() {
2     T++;                            2     lock(L);
3   }                                 3     H++;
                                      4     if (H > T) {
4   pop() {                           5       H--;
5     T--;                            6       unlock(L);
6     if (H > T) {                    7       return FAILURE;
7       T++;                          8     }
8       lock(L);                      9     unlock(L);
9       T--;                         10     return SUCCESS;
10      if (H > T) {                 11   }
11        T++;
12        unlock(L);
13        return FAILURE;
14      }
15      unlock(L);
16    }
17    return SUCCESS;
18  }
```

Σχήμα 18: Ψευδοκώδικας για την απλοποιημένη εκδοχή του πρωτοκόλλου THE. Αριστερά ο κώδικας του θύματος και δεξιά του κλέφτη.

Στο παραπάνω σχήμα δίνεται ο ψευδοκώδικας μιας απλοποιημένης εκδοχής του πρωτοκόλλου THE. Απουσιάζει μόνο η μεταβλητή E, την οποία θα εισάγουμε αργότερα. Ο ψευδοκώδικας αναφέρεται σε μοντέλα κοινής μνήμης που έχουν σειριακή συνέπεια [25]. Αν κάτι τέτοιο δεν ισχύει, πρέπει να εξασφαλιστεί ότι οι εντολές στις γραμμές 5-6 του θύματος και οι 3-4 του κλέφτη θα εκτελεστούν σειριακά, κάτι που μπορεί να γίνει μέσω κάποιου φράγματος μνήμης. Επιπλέον, στον κώδικα, η deque αντιμετωπίζεται ως ένας πίνακας από tasks (ή καλύτερα *εγγραφών δραστηριοποίησής* των tasks).

Τα ονόματα των μεταβλητών T και H προέρχονται από τα αρχικά των λέξεων tail (ουρά) και head (κεφαλή) και αναφέρονται στην ουρά και στην κεφαλή της deque. Και οι δυο μεταβλητές είναι αποθηκευμένες στην κοινή μνήμη, οπότε είναι ορατές απ' όλους τους επεξεργαστές. Το T δείχνει στο πρώτο άδειο κελί του πίνακα της deque ενώ το H στο πρώτο κελί του πίνακα. Κάθε deque θα διαθέτει κι ένα κλείδωμα L, μοναδικό για καθεμία, που θα επιτυγχάνεται μέσω κάποιου hardware primitive ή μέσω κάποιας κλήσης του OS.

Παρατηρώντας τον κώδικα του THE, γίνεται φανερό ότι η T μεταβάλλεται μόνο από το θύμα και η H μόνο από τον κλέφτη. Φυσιολογικό, αφού το θύμα αντιμετωπίζει τη deque ως στοίβα και κάνει push και pop τα tasks από την ουρά της (T), ενώ ο κλέφτης έχει δικαίωμα να κλέψει μόνο από την κεφαλή της (H). Έτσι, όποτε το θύμα κάνει push στη deque η T θα αυξάνεται κατά 1, ενώ όποτε κάνει pop θα μειώνεται κατά 1. Ο κλέφτης θα αυξάνει το H κατά 1 όποτε κλέβει ένα task. Προκύπτουν, λοιπόν, τρεις δυνατές περιπτώσεις (Σχήμα 19).



Σχήμα 19: Οι τρεις πιθανές περιπτώσεις του πρωτοκόλλου THE

Όποτε το θύμα θέλει να κάνει push στη deque ένα task, δεν υπάρχει κανένα πρόβλημα, αφού ταυτόχρονα μπορεί ο κλέφτης να κλέβει ένα task από την κορυφή. Επομένως, σε κάθε push απλά αυξάνεται το T (γραμμές 1-3 θύμα). Όποτε το θύμα θέλει να κάνει pop θα μειώσει το T κι έπειτα θα ελέγξει αν το H είναι μεγαλύτερο από το T. Αν το T είναι μεγαλύτερο, τότε βρισκόμαστε στην πρώτη περίπτωση όπου η deque έχει παραπάνω από ένα tasks, οπότε κλέφτης και θύμα μπορούν να συνυπάρχουν και να λάβουν και οι δυο tasks από τη deque ταυτόχρονα με απόλυτη ασφάλεια.

Στη δεύτερη περίπτωση, η deque περιέχει μόνο ένα task, που σημαίνει ότι το T είναι κατά ένα μεγαλύτερο από το H. Όταν το θύμα επιχειρήσει να κάνει pop το task, θα μειώσει το T (γραμμή 5) και αν εξακολουθεί να είναι μεγαλύτερο ή ίσο από το H (γραμμή 6), τότε μπορεί να το πάρει, καθώς κανένας κλέφτης δεν επιχειρεί να το κλέψει. Αντίστοιχα, όταν κάποιος κλέφτης επιχειρήσει να το κλέψει, θα αυξήσει το H (γραμμή 3) και αν εξακολουθεί να είναι μικρότερο από το T (γραμμή 4), δηλαδή το θύμα δεν επιχειρεί να το κάνει pop, τότε θα το κλέψει επιτυχώς. Σύγκρουση θα προκύψει όταν επιχειρήσουν ταυτόχρονα να πάρουν το task. Ο νικητής της διένεξης μεταξύ θύματος και κλέφτη είναι αυτός που δε θα εκτελέσει πρώτος τον έλεγχο της συνθήκης H>T (γραμμή 4 κλέφτης | γραμμή 5 θύμα). Αν εκτελεστεί πρώτα ο κλέφτης, τότε το H θα είναι μεγαλύτερο από το T, οπότε η συνθήκη θα ισχύει, το H

θα μειωθεί, θα ελευθερωθεί το κλείδωμα της deque και η κλοπή θα επιστρέψει με αποτυχία (γραμμές 5-8 κλέφτης). Αν εκτελεστεί πρώτα το θύμα, ικανοποιείται η συνθήκη του, οπότε αυξάνει το T, κλειδώνει τη deque και επαναλαμβάνει το πρωτόκολλο (γραμμές 7-14 θύμα). Αφού κλειδώσει την ουρά ουσιαστικά υπάρχουν δυο περιπτώσεις: είτε έχει υποχωρήσει ο κλέφτης οπότε το task ανήκει στο θύμα, είτε ο κλέφτης δεν έχει εκτελεστεί ακόμη, οπότε το θύμα υποχωρεί και επιστρέφει πίσω στο RTS της Cilk με αποτυχία. Το θύμα θα θεωρεί ότι η deque του είναι άδεια, πράγμα ορθό αφού ο κλέφτης θα πάρει το μοναδικό task της αν δε το έχει ήδη κάνει, κι έτσι θα μετατραπεί το ίδιο σε κλέφτη, επιχειρώντας να κλέψει task από κάποιον άλλον επεξεργαστή.

Στην τρίτη και τελευταία περίπτωση H και T είναι ίσα γιατί η deque είναι άδεια. Σε αυτήν την περίπτωση, ο κλέφτης πάντα θα αποτυγχάνει, το ίδιο και το θύμα, το οποίο επιστρέφει τον έλεγχο στο RTS της Cilk. Το πρωτόκολλο δεν οδηγεί ποτέ σε αδιέξοδο, γιατί κάθε επεξεργαστής κρατάει ένα το πολύ κλείδωμα κάθε φορά.

Αυτή ήταν η απλή εκδοχή του πρωτοκόλλου THE. Η πλήρης εκδοχή περιλαμβάνει μια επιπλέον μεταβλητή E, το όνομα της οποίας προέρχεται από τη λέξη exceptions. Είναι η μεταβλητή που επιτρέπει τη λειτουργία του *abort* στη Cilk. Όποτε εκτελεστεί ένα *abort*, όλα τα παιδιά του task που το εκτελεί θα εγκαταλειφθούν και δε θα εκτελεστούν ποτέ. Το RTS της Cilk διατρέχει σειριακά το *spawn tree* και μαρκάρει όλα τα παιδιά ως *aborted* και επιπλέον στέλνει σε όλους τους επεξεργαστές σήμα για την εγκατάλειψη των εν λόγω tasks. Όταν θα γίνει pop ένα *aborted* task το πρωτόκολλο THE φροντίζει μέσω της E να εγκαταλειφθεί το task και να επιστρέψει ο έλεγχος στο RTS.

Έτσι, το THE, όπως το περιγράψαμε μέχρι τώρα, επεκτείνεται χωρίζοντας τη μεταβλητή H σε δυο: στην H, η οποία είναι αυτή που δείχνει την κεφαλή της deque και στην E, η οποία δείχνει το σημείο της deque που το θύμα δεν μπορεί να περάσει όταν κάνει pop. Ενώ πριν είχαν θεωρηθεί ότι είναι το ίδιο σημείο (η κεφαλή της deque), πλέον μπορούν να μην είναι, καθώς το E μπορεί να δείχνει σε ένα task πιο κοντά στην ουρά, άρα σε ένα πρόγονο πιο κοντινό του task που εκτελείται. Από την πλευρά του θύματος η λογική είναι ίδια με προηγουμένως, μόνο που αντί για σύγκριση μεταξύ των H και T, το θύμα συγκρίνει το E και το T. Στον ψευδοκώδικα, λοιπόν, του σχήματος 18, ο κώδικας του θύματος τροποποιείται αλλάζοντας τη μεταβλητή H και κάνοντάς την E στη γραμμή 6. Επίσης, οι γραμμές 7-15 αντικαθίστανται από ένα *χειριστή εξαιρέσεων*, που καθορίζει το είδος της εξαίρεσης (είτε πρόκειται για συναγωνισμό task είτε για κάτι άλλο) και εκτελεί την ανάλογη ενέργεια. Ο κώδικας του κλέφτη αλλάζει έτσι ώστε πριν γίνει προσπάθεια για κλοπή να αυξάνει το E. Αν δεν γίνει η κλοπή, τότε το E επαναφέρεται στην αρχική τιμή του.

Όπως έχουμε ήδη πει, ο λόγος που υλοποιήθηκε το THE ήταν ο περιορισμός του *work* κόστους, του κόστους του θύματος, και αυτό ακριβώς επιτυγχάνει. Στη γενική περίπτωση, που δεν υπάρχει σύγκρουση κλέφτη θύματος, το επιπλέον κόστος είναι μόλις μερικές απλές εντολές (αν τις βλέπαμε σε assembly θα ήταν 2 loads, 1 store, 1 αφαίρεση και 1 conditional branch). Αυτές οι εντολές μπορούν να εκτελεστούν πολύ γρήγορα όταν οι μεταβλητές T και H βρίσκονται στην cache των επεξεργαστών. Η μόνη λοιπόν εντολή με μεγάλο κόστος είναι

το κλείδωμα που κάνει στη deque όταν συναγωνίζεται με τον κλέφτη. Αυτή η περίπτωση, όμως, όπως έχουμε δει είναι σχετικά σπάνια και εξαρτάται από το μήκος του *critical-path* T_∞ κι έτσι τελικά αυτό το κόστος μπορεί να θεωρηθεί ως μέρος του κόστους του *critical-path* c_∞ .

Αν θέλαμε να κατηγοριοποιήσουμε το πρωτόκολλο, θα επιλέγαμε να το θεωρήσουμε *non-blocking*, παρόλο που σε μια περίπτωση χρησιμοποιεί κλείδωμα μνήμης. Η επιλογή να τον κατατάξουμε στην κατηγορία αλγορίθμων που δε χρησιμοποιούν κλειδώματα έχει να κάνει με τη σπανιότητα εμφάνισης σύγκρουσης θύματος-κλέφτη. Αυτό σημαίνει ότι το THE είναι απίθανο να εμφανίσει spinlock προβλήματα, όπου το ένα task περιμένει το άλλο για να αποκτήσει ένα κλείδωμα. Η υλοποίηση πλήρως *non-blocking* work stealing αλγορίθμων είναι εφικτή [42] και το πρωτόκολλο THE μπορεί να τροποποιηθεί ώστε να γίνει πλήρως *non-blocking,* αλλά κάτι τέτοιο δεν προκαλεί ιδιαίτερη αύξηση στην επίδοση, ενώ αντίθετα είναι πιο περίπλοκο στην υλοποίηση.

4.4 Work Stealing με τη βιβλιοθήκη task.h

Κατ' αντιστοιχία με τους work sharing αλγορίθμους, αποφασίσαμε να αξιοποιήσουμε τις συναρτήσεις της task.h με στόχο να υλοποιήσουμε το δικό μας work stealing αλγόριθμο. Ως work sharing, ακολουθεί την ιδέα ότι οι επεξεργαστές μοιράζονται μεταξύ τους εργασία μέσω της κλοπής: ο αδρανής επεξεργαστής θα επιχειρήσει να κλέψει task από εκείνον που έχει.

Ο αλγόριθμος που υλοποιήσαμε αποτελεί μια απλοποιημένη εκδοχή του work stealing αλγορίθμου που εκτελεί η Cilk. Πιο συγκεκριμένα, όταν είναι άδειοι οι επεξεργαστές, επιχειρούν να αποκτήσουν πρόσβαση στην ουρά κάποιου άλλου επεξεργαστή για να κλέψουν κάποιο task. Η επιλογή του επεξεργαστή-θύματος είναι εντελώς τυχαία (γίνεται μέσω της συνάρτησης rand_r()). Για να αποφύγουμε το συναγωνισμό πάνω στα ίδια δεδομένα, το πρόβλημα που επιλύει το πρωτόκολλο THE όπως είδαμε, εμείς χρησιμοποιούμε μια απλούστερη ιδέα. Επιτρέπουμε πρόσβαση για κλέψιμο στη deque κάποιου επεξεργαστή όταν αυτός έχει τουλάχιστον δυο tasks σε αυτήν. Ουσιαστικά, είναι σαν να διαλέγουμε εκ των προτέρων ως νικητή του συναγωνισμού το θύμα κι όχι τον κλέφτη.

Ο αλγόριθμός μας ακολουθεί τον παρακάτω ψευδοκώδικα, που υλοποιείται στη συνάρτηση scheduler:

```
void *scheduler(void *arg) {

  while(initial_task->state!=FINALIZED) {

    if (empty_deque) {
      if (current_task->state!=RUNNABLE)) {
        victim = choose_random_scheduler;
        if(victim_deque_size>1) {
          current_task = steal_from_head_of_victim;
          swapcontext(scheduler_uc,current_task->uc);
        }
      }
      else  swapcontext(scheduler_uc,current_task->uc);
    }
    else {
      new_task = pop_from_deque;
      if (current_task->state==RUNNABLE) push(current_task,current_deque);
      current_task = new_task;
      swapcontext(scheduler_uc,current_task->uc);
    }
  swapcontext(scheduler_uc,init_uc);
  return;
}
```

Η δομή του αλγορίθμου είναι παρόμοια με τη δομή του work sharing. Έχουμε ένα ατέρμονο loop με συνθήκη εξόδου τον τερματισμό του αρχικού task. Τα βήματα που εκτελεί είναι τα ακόλουθα:

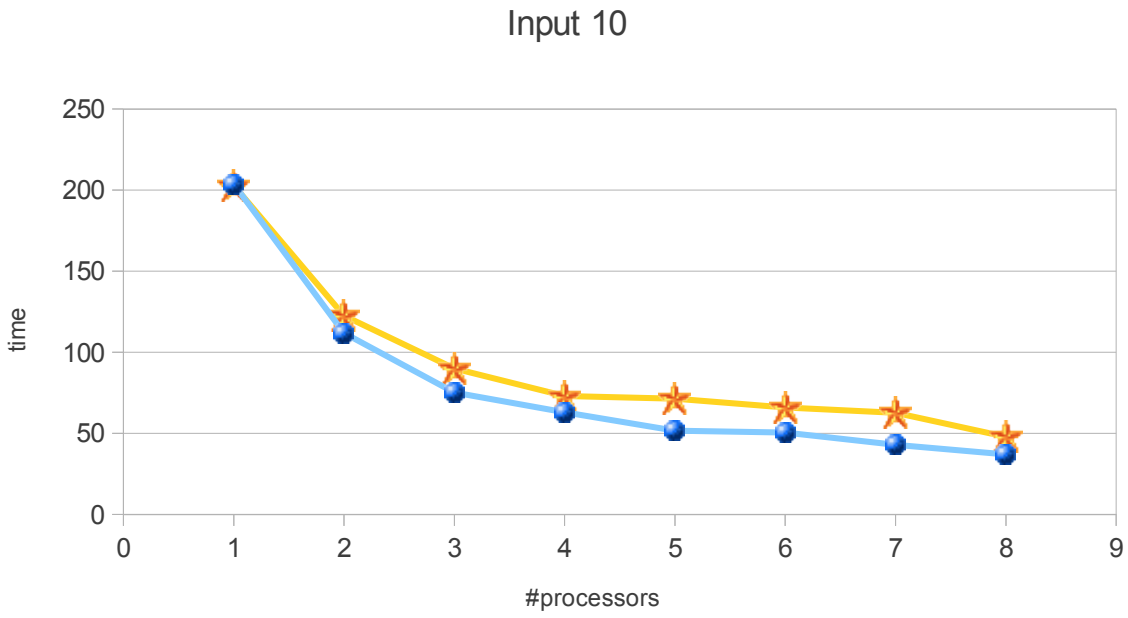
1. Ελέγχεται αν υπάρχει κάποιο task στη deque του επεξεργαστή ή όχι.
2. Αν η deque του επεξεργαστή είναι άδεια ελέγχεται αν το task που έτρεχε μέχρι εκείνη τη στιγμή στον επεξεργαστή είναι έτοιμο για εκτέλεση.
3. Αν δεν είναι ή αν δεν υπάρχει καθόλου, τότε ο επεξεργαστής επιχειρεί να κλέψει ένα task. Διαλέγει τυχαία έναν άλλο επεξεργαστή και αν η deque του έχει τουλάχιστον δυο task, κλέβει αυτό που βρίσκεται στην κεφαλή της. Αλλιώς, επανέρχεται στην αρχή του loop.
4. Αν το τρέχον task ήταν έτοιμο προς εκτέλεση, τότε κάνει context switch στο context του task για να το εκτελέσει.
5. Αν τώρα η deque του επεξεργαστή δεν ήταν άδεια, αλλά είχε τουλάχιστον ένα task, τότε ο work stealing αλγόριθμος κάνει pop το task που βρίσκεται στην ουρά της deque και το αποθηκεύει προσωρινά σε μια μεταβλητή new_task.

6. Στη συνέχεια γίνεται pushed στη deque το τρέχον task του επεξεργαστή, αν αυτό είναι έτοιμο προς εκτέλεση.
7. Έπειτα η τιμή του τρέχοντος task αλλάζει για να δείχνει στο task που είχε γίνει pop και μετά θα γίνει context_switch στο καινούργιο task για να εκτελεστεί.
8. Όταν βγει έξω από το loop, που σημαίνει ότι θα έχει τερματίσει το πρώτο task που είχε δημιουργηθεί (ανεξάρτητα από το αν το αρχικό task τερματίστηκε στον συγκεκριμένο επεξεργαστή), τότε γίνεται context_switch στο αρχικό context που είχε δημιουργηθεί από την task_init().

Ας δούμε τώρα την επίδοση που έχει ο αλγόριθμός μας. Αντίστοιχα με τον work sharing αλγόριθμο, αρχικά παρουσιάζουμε μετρήσεις σε δυο διαφορετικά μηχανήματα: clones και dunnington. Ταυτόχρονα με τις μετρήσεις για τον work stealing αλγόριθμο θα δείχνουμε στα διαγράμματα και τη συμπεριφορά του work sharing αλγορίθμου, όπως περιγράφηκε στην ενότητα 3.4 ώστε να γίνεται και η μεταξύ τους σύγκριση. Η γαλάζια γραμμή (με τους κύκλους) αντιπροσωπεύει τις μετρήσεις του work stealing, ενώ η πορτοκαλί γραμμή (με τα αστεράκια) τις μετρήσεις του work sharing.

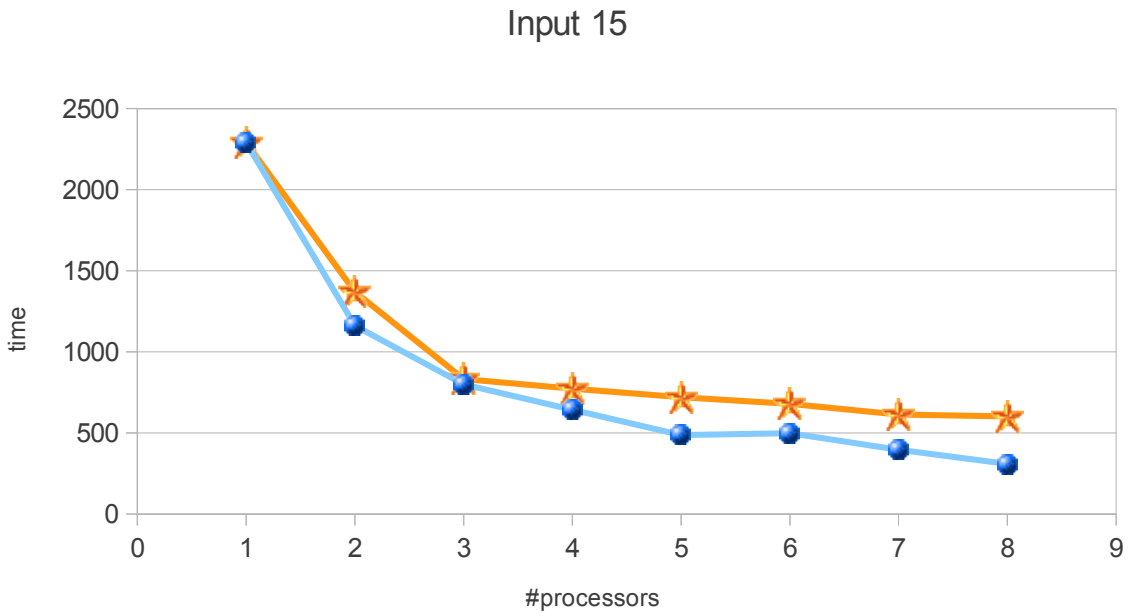
➔ *Clones*

Θυμίζουμε ότι τα clones έχουν συνολικά 8 επεξεργαστικές μονάδες και ότι σηκώνουμε ένα pthread ανά διαθέσιμο επεξεργαστή.



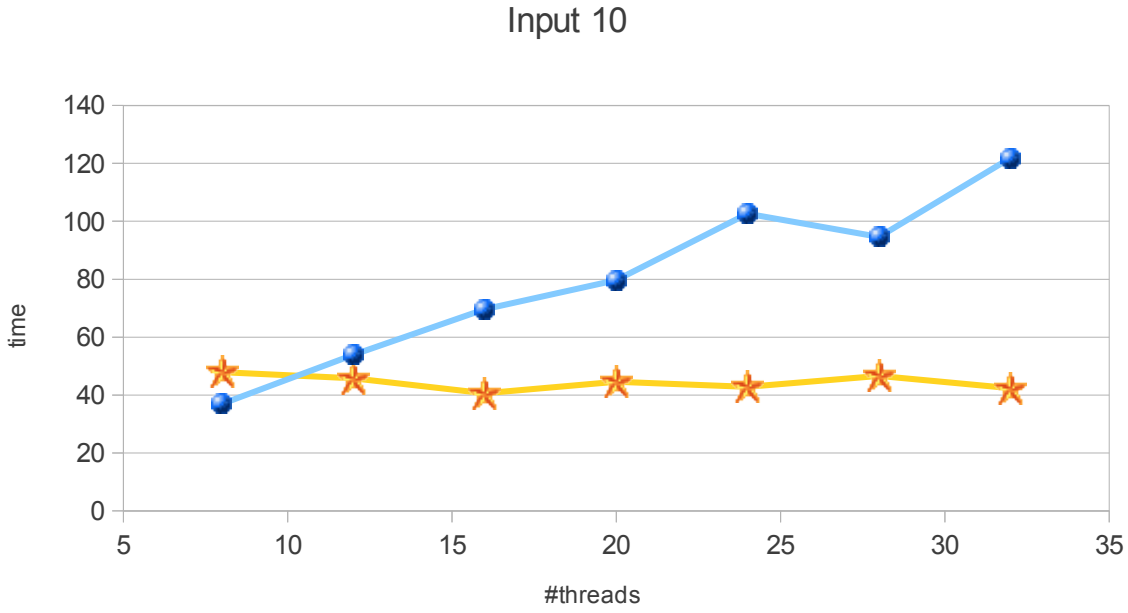
Για είσοδο ίση με 10, το πρόγραμμα επιταχύνει αρχικά με ρυθμό αντιστρόφως ανάλογο του πλήθους των επεξεργαστών. Η συμπεριφορά είναι σχεδόν ίδια με τον work sharing. Απλώς

παρατηρείται μια ελαφρή μείωση του χρόνου εκτέλεσης, χωρίς όμως να είναι ιδιαίτερα σημαντική.



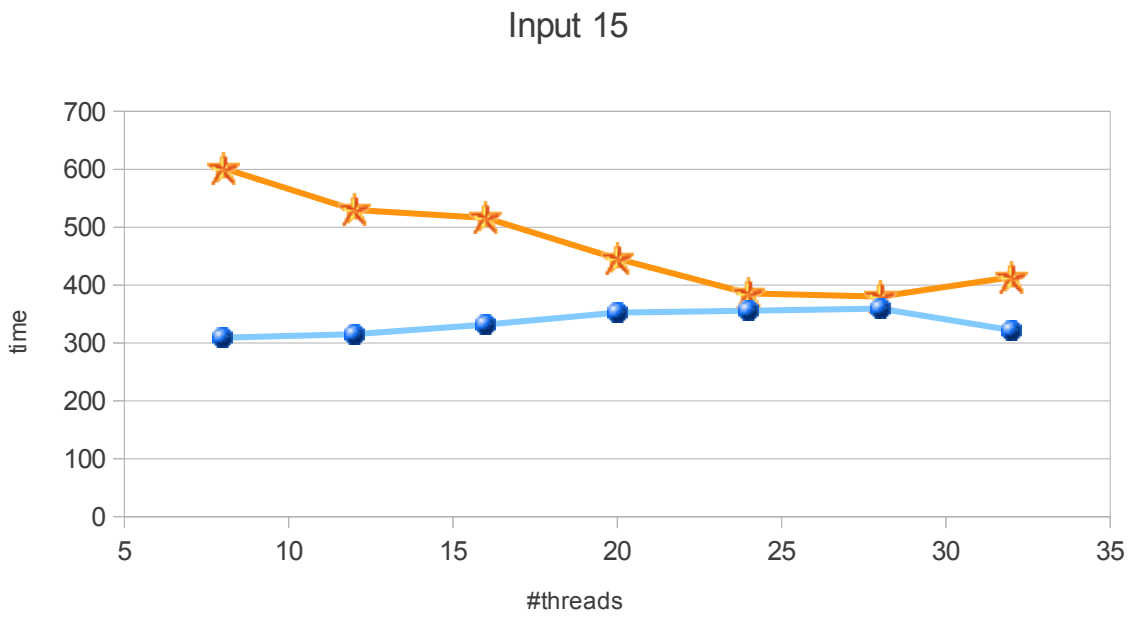
Με την αύξηση της εισόδου σε 15, ο work stealing αλγόριθμος επιταχύνει με μεγαλύτερο ρυθμό σε σχέση με το work sharing όσο αυξάνεται το πλήθος των επεξεργαστών. Άρα, η stealing τεχνική είναι αποδοτικότερη σε σχέση με τη sharing κι αυτό δημιουργεί μείωση του χρόνου εκτέλεσης αρκετά σημαντική όσο αυξάνεται το μέγεθος της εργασίας, δηλαδή το πλήθος των tasks.

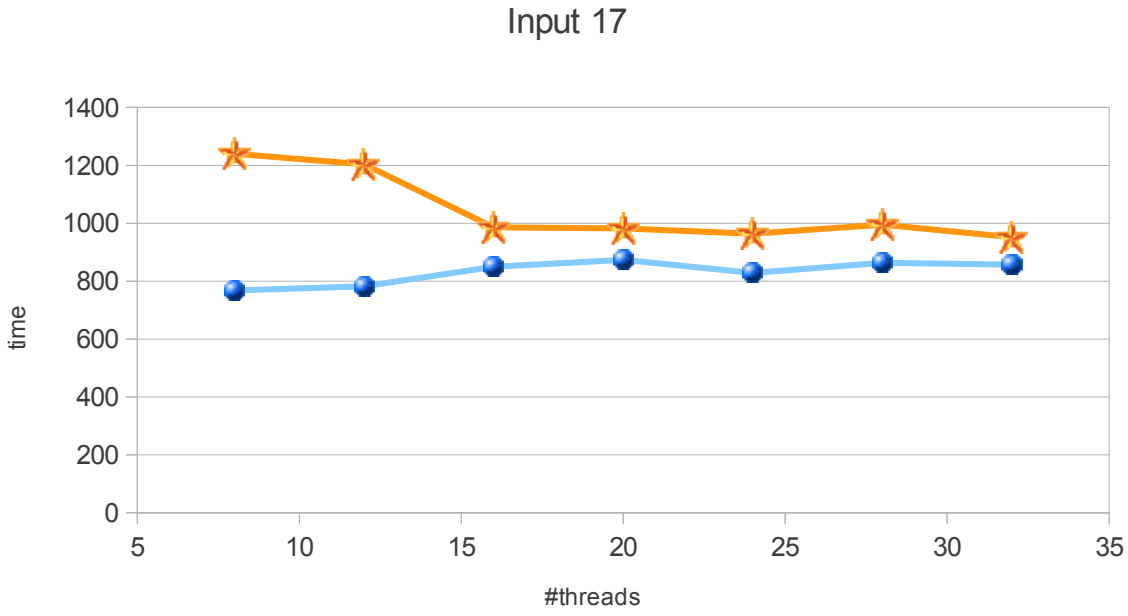
Ας δούμε τώρα πώς συμπεριφέρεται ο work stealing στο oversubscription.



Το παραπάνω διάγραμμα είναι ιδιαίτερα ενδιαφέρον. Για είσοδο 10, ενώ ο work sharing αλγόριθμος παρουσιάζει μια μικρή επιτάχυνση, όπως σχολιάσαμε στην ενότητα 3.4, ο work stealing αλγόριθμος δίνει αντίθετες μετρήσεις, καθώς ο χρόνος εκτέλεσης αντί να μειωθεί αυξάνεται. Μάλιστα, σε αυτήν την περίπτωση είναι λιγότερο αποδοτικός και από τον work sharing. Γιατί, όμως, έχουμε αυτή τη συμπεριφορά;

Η απάντηση βρίσκεται στο ότι ο μεν work stealing είναι optimized αλγόριθμος, ενώ ο work sharing όχι. Όταν αυξάνουμε το πλήθος των threads ανά επεξεργαστή δίνουμε τη δυνατότητα στο λειτουργικό σύστημα να κάνει context switch μεταξύ αυτών κατά τη διάρκεια της εκτέλεσης, όπως περιγράψαμε στο 2.2. Ο work sharing αλγόριθμος επωφελείται από το context switch του λειτουργικού, ώστε κάθε φορά να εκτελείται το thread που έχει tasks προς εκτέλεση και όχι αυτό που περιμένει μέχρι να του έρθει εργασία. Αυτό αντισταθμίζει το overhead εξαιτίας του context switch και τελικά οι χρόνοι που προκύπτουν είναι ελαφρώς καλύτεροι. Αντίθετα, ο work stealing είναι ένας ήδη optimized αλγόριθμος. Το context switch από το λειτουργικό δεν τον εξυπηρετεί, αφού θα υπάρχουν φορές που θα εκτελούνται μόνο threads που προσπαθούν να κλέψουν tasks κι όχι εκείνα που έχουν task έτοιμα προς εκτέλεση. Αν σε αυτή την καθυστέρηση προσθέσουμε και το overhead από το context switch, τότε τελικά ο χρόνος αντί να μειώνεται αυξάνεται, δίνοντάς μας την εικόνα της παραπάνω γραφικής παράστασης.

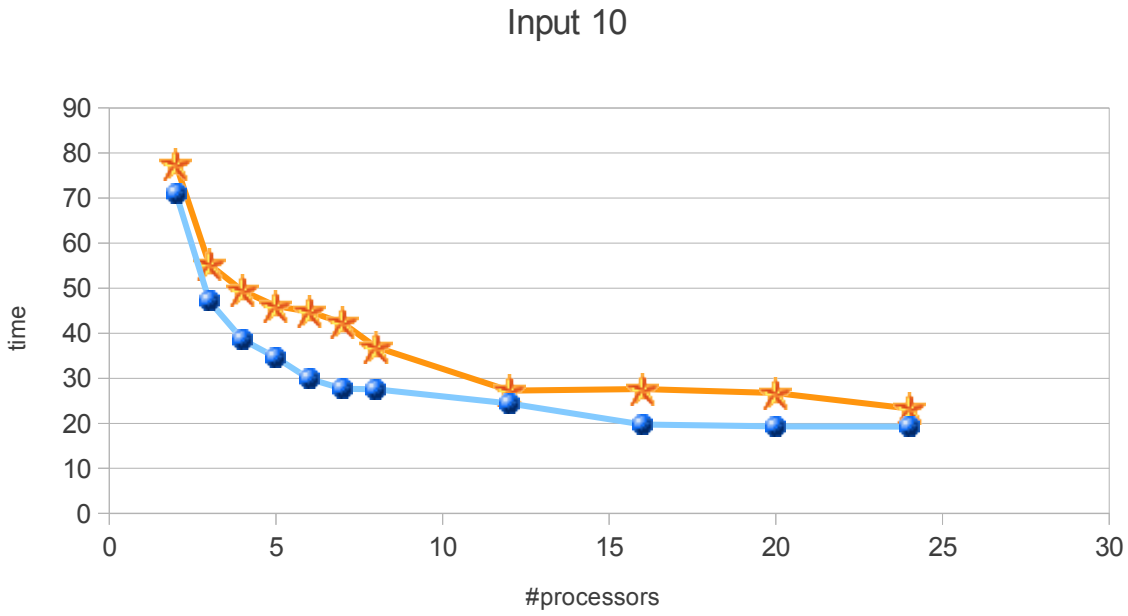




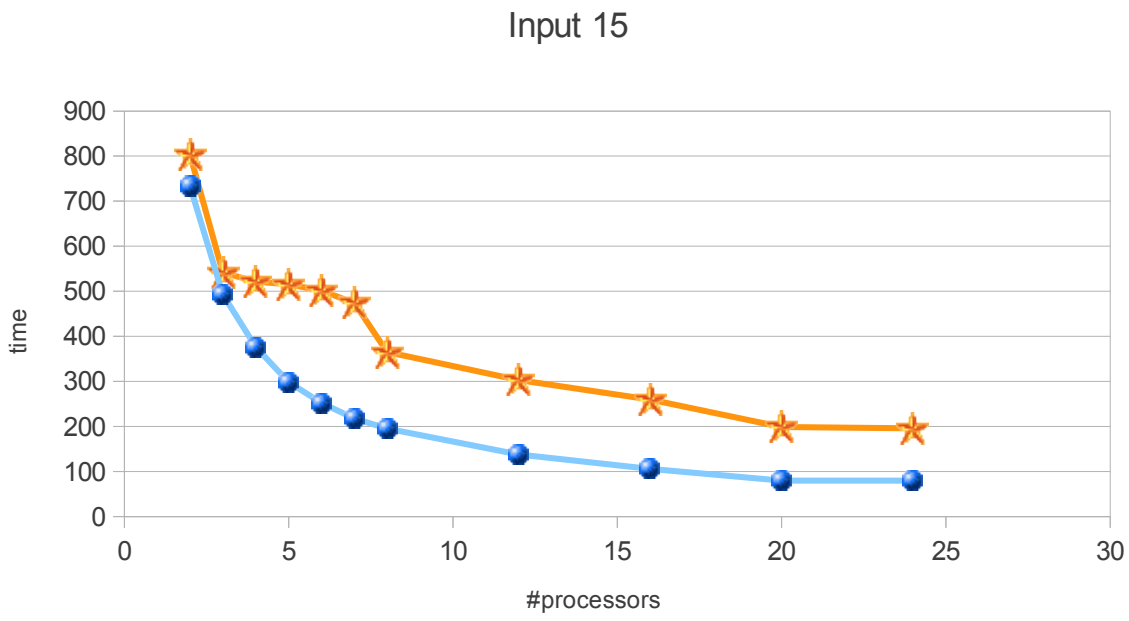
Αντίστοιχες παρατηρήσεις μπορούμε να κάνουμε και με την αύξηση της εισόδου. Η διαφορά είναι ότι ακόμα και σε περίπτωση oversubscription, ο work stealing αλγόριθμος είναι αποδοτικότερος από τον work sharing, παρόλο που το πρόγραμμα επιβραδύνεται.

➔ *Dunnington*

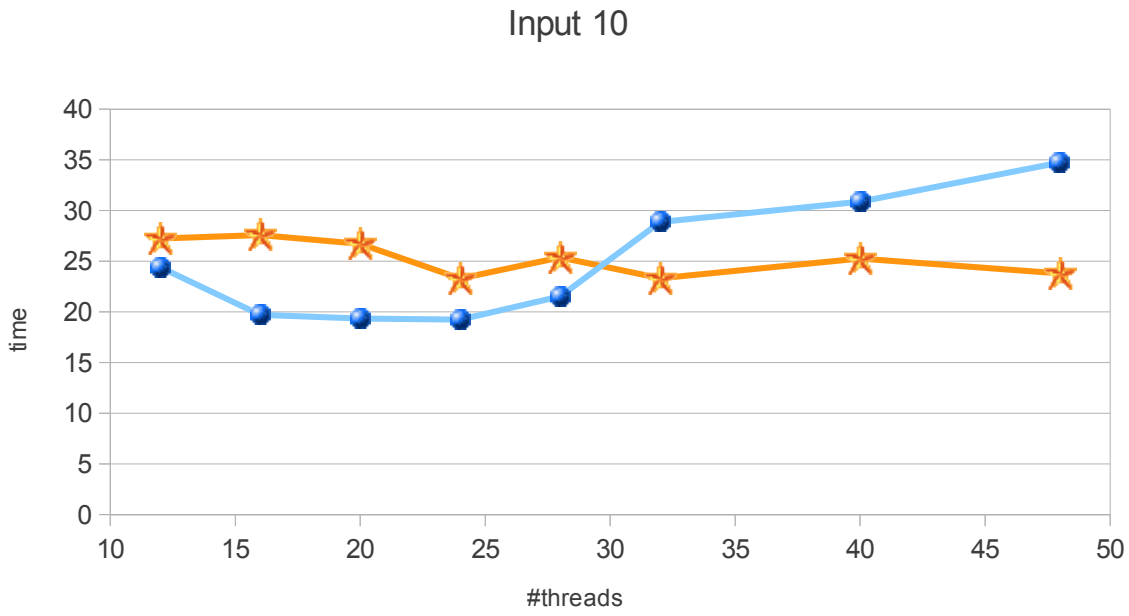
Παρουσιάζουμε παρακάτω τα αποτελέσματα των μετρήσεων στη πλατφόρμα dunnington, που έχει 24 επεξεργαστικές μονάδες.

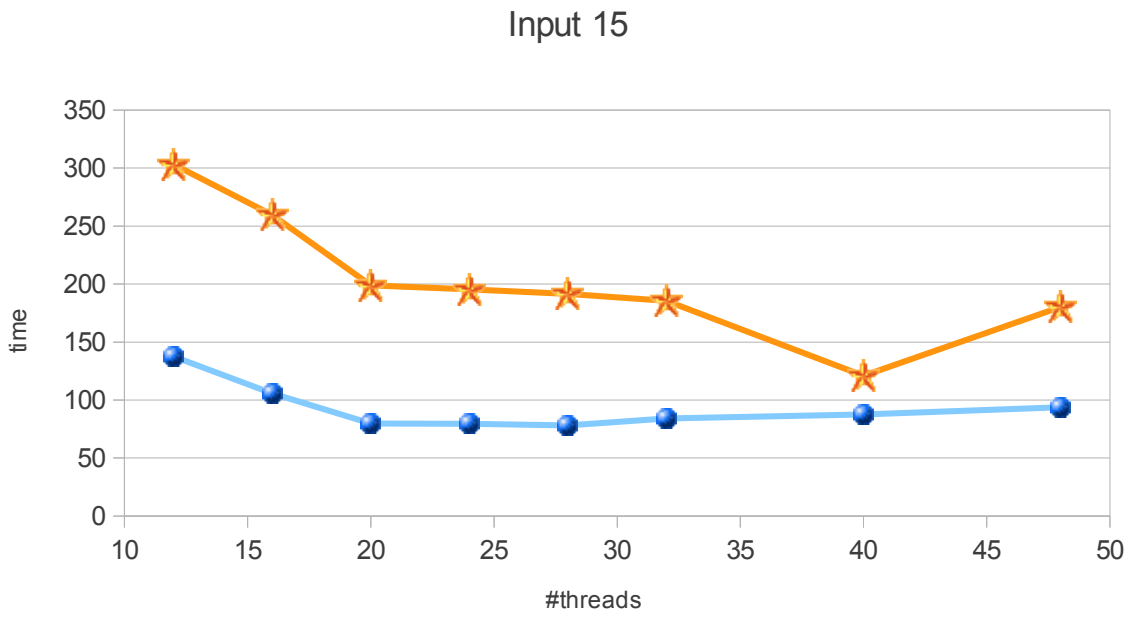


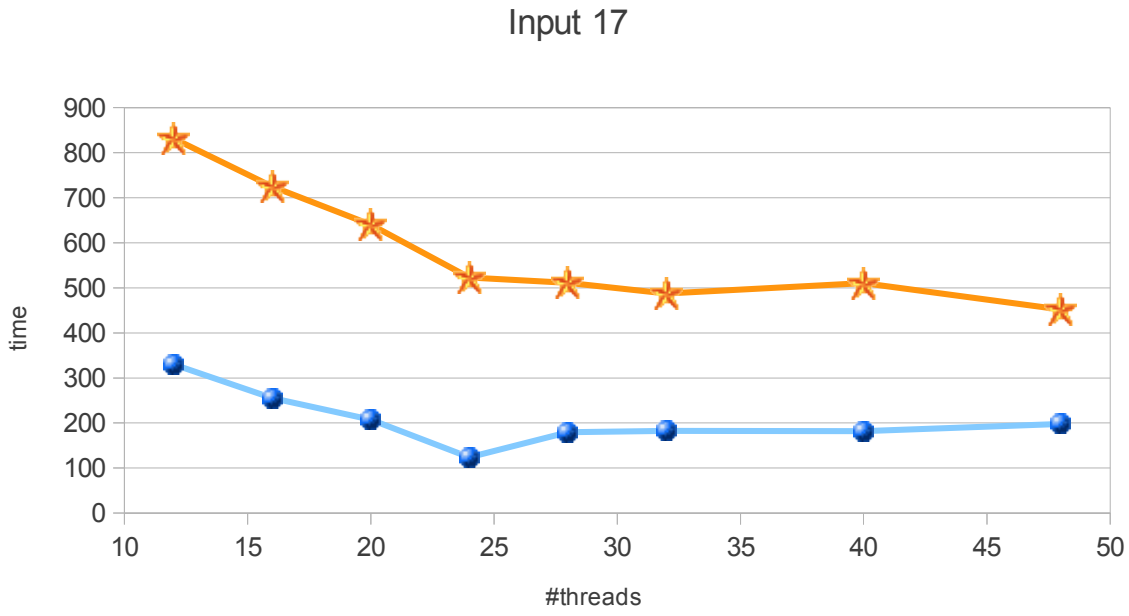
Και σε αυτό το διάγραμμα παρατηρούμε ότι ο work stealing αλγόριθμος είναι καλύτερος, αν και όχι σημαντικά. Αυτό, όμως, οφείλεται στο μικρό πλήθος των tasks (θυμίζουμε 2^{10}). Η πραγματική διαφορά στην επίδοση της δρομολόγησης που προσφέρουν γίνεται φανερή όταν αυξήσουμε την είσοδο και γίνει ίση με 15 (2^{15} tasks). Στο παρακάτω διάγραμμα γίνεται ορατό αυτό το συμπέρασμα.



Ας ελέγξουμε κι εδώ τι γίνεται με το oversubscription.







Και τα τρία αυτά διαγράμματα δίνουν τις ίδιες ακριβώς πληροφορίες και εξάγουμε τα ίδια συμπεράσματα όπως και στην περίπτωση των clones. Επομένως, ενώ για το work sharing η τεχνική του oversubscription αποτελεί μια βελτιστοποίηση, για το work stealing αντίθετα δημιουργεί μικρές καθυστερήσεις στην εκτέλεση. Παρ' όλ' αυτά ο work stealing αλγόριθμος έχει σαφώς πολύ καλύτερους χρόνους από τον work sharing, επαληθεύοντας όλες τις θεωρητικές αναλύσεις που έχουμε κάνει σε αυτό το κεφάλαιο.

ΚΕΦΑΛΑΙΟ 5: ΕΠΙΛΟΓΟΣ

Σε αυτήν την εργασία προσπαθήσαμε να αξιοποιήσουμε θεωρητικά μοντέλα ώστε να μελετήσουμε αλγορίθμους διαχείρισης και εκτέλεσης παράλληλων εργασιών. Χρειάστηκε να δημιουργήσουμε αρχικά μια πρότυπη βιβλιοθήκη επεξεργασίας ανεξάρτητων εργασιών, την task.h, μια απλοποιημένη εκδοχή της βιβλιοθήκης pthread.h. Έχοντας τις συναρτήσεις ως εργαλείο υλοποιήσαμε δυο αλγορίθμους: ένα work sharing και ένα work stealing. Οι πειραματικές τους ενδείξεις μας έδειξαν ενδιαφέροντα αποτελέσματα, καθώς αρκετά από αυτά συνέπιπταν με τις θεωρητικές μελέτες και οι αλγόριθμοι είχαν την αναμενόμενη συμπεριφορά.

Για να γίνει πλήρης η μελέτη μας θα έπρεπε να διεξάγουμε πειράματα και σε άλλα προγράμματα, όπου τα tasks δεν είναι ίδια μεταξύ τους, όπως γίνεται στον τροποποιημένο αλγόριθμο του fibonacci που χρησιμοποιήσαμε. Έτσι θα παρατηρούσαμε τη συμπεριφορά των δυο αλγορίθμων σε ανομοιογενή προγράμματα και θα είχαμε μια καθολική εικόνα για αυτούς.

Κάποια βήματα, ακόμη, με τα οποία θα μπορούσε να συνεχιστεί αυτή η εργασία είναι η βελτίωση των υπάρχοντων αλγορίθμων. Θα μπορούσαν να φτιαχτούν work sharing αλγόριθμοι που μοιράζουν με διαφορετικό τρόπο τα tasks (για παράδειγμα ο επεξεργαστής με τα λιγότερα tasks να δέχεται κάθε φορά το task που μοιράζεται) και να συγκρίνουμε τις επιδόσεις τους, ώστε να βρούμε την βέλτιστη μέθοδο. Επίσης, θα μπορούσε να βελτιωθεί ο work stealing αλγόριθμος με το να μη γίνεται τυχαία η επιλογή του θύματος, αλλά μέσω κάποιου αλγορίθμου, αλλά και να υλοποιηθεί το πρωτόκολλο THE ώστε να ελεγχθεί πειραματικά πόσο επιταχύνεται η διαδικασία.

Τέλος, ένα επόμενο βήμα, θα ήταν η απόπειρα υλοποίησης ενός μοντέλου όπως της Cilk. Δηλαδή έχοντας τη βιβλιοθήκη task.h και τον ή τους αλγορίθμους δρομολόγησης των tasks να υλοποιηθεί ένας ενδιάμεσος μεταγλωττιστής στα πρότυπα του cilk2c, ώστε όλη η δουλειά των συναρτήσεων των tasks να κωδικοποιείται σε ορισμένες λέξεις-κλειδιά. Υπενθυμίζουμε ότι όλη αυτή η δουλειά δεν αποσκοπεί στο να βελτιώσει ήδη υπάρχοντα εργαλεία παράλληλου προγραμματισμού (όπως η Cilk), αλλά έχει διδακτικό σκοπό.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Wilson, Gregory V (1994). “The History of the Development of Parallel Computing” Virginia Tech/Norfolk State University, Interactive Learning with a Digital Library in Computer Science

[2] Amdahl, Gene M. (1967). “Validity of the single processor approach to achieving large scale computing capabilities”. Proceeding AFIPS '67 (Spring) Proceedings of the April 18–20, 1967

[3] J.L. Gustafson, Reevaluating Amdahl’s Law. Comm. ACM, Vol.31, No.5, 1988, pp. 532-533

[4] “An Overview of Static Pipelining”, Ian Finlayson, Gang-Ryung Uh, David Whalley and Gary Tyson, Department of Computer Science Florida State, Department of Computer Science University Boise State University [2012]

[5] Flynn, M. (1972). "Some Computer Organizations and Their Effectiveness". *IEEE Trans. Comput.* C–21: 948.

[6] Duncan, Ralph, "A Survey of Parallel Computer Architectures", IEEE Computer. February 1990, pp. 5-16.

[7] Myricom. Myricom, Inc. http://www.myri.com

[8] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000

[9] The OpenMP® API specification for parallel programming, http://openmp.org/wp/

[10] “Evaluation of OpenMP Task Scheduling Strategies” , Alejandro Duran, Julita Corbalán, and Eduard Ayguadé, Barcelona Supercomputing Center Departament d’Arquitectura de Computadors
Universitat Politècnica de Catalunya, 2008

[11] Aoyama, Yukiya, Nakano, “Practical MPI Programming”, Jun 1999

[12] Chapter 2 – Cuda Programming Model, David Kirk/NVIDIA and Wen-mei Hwu, 2006-2008

[13] "The Asynchronous Partitioned Global Address Space Model", Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky Olivier Tardieu, IBM, 2010

[14] Cilk 5.4.6 Reference Manual, Supercomputing Technologies Group, MIT Labaratory for Computer Science, 1998

[15] "Memory management in C: The heap and the stack", Leo Ferres Department of Computer Science Universidad de Concepción, leo@inf.udec.cl, October 7, 2010

[16] "The thorny problem of the cactus stack", Matteo Frigo, April 2009

[17] Silberschatz A., Peterson J., Galvin P., "Operating System Concepts", Addison - Wesley, 1991.

[18] "Computer Architecture: A Quantitative Approach, 3th Edition", John Hennessy, David Patterson, 2005

[19] "Intel Hyper-Threading Technology", Technical User's Guide

[20] "The Problem with Threads", Edward A. Lee, Professor, Chair of EE, Associate Chair of EECS
EECS Department, University of California at Berkeley, 10 January 2006

[21] Lewis, Bill and Daniel J. Berg., "Multithreaded Programming with Pthreads",California: Prentice Hall, 1998

[22] "Pthreads Library Interface" , Frank Mueller, Department of Computer Science , Florida State University, July 12, 1995

[23] http://pubs.opengroup.org/onlinepubs/007908799/xsh/ucontext.h.html, The Single UNIX ® Specification, Version 2, Copyright © 1997 The Open Group

[24] Robert D. Blumofe and Charles E. Leiserson. "Space-efficient scheduling of multithreaded
computations". *SIAM Journal on Computing*, 27(1):202-229, February 1998

[25] Leslie Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs”, IEEE Transactions on Computers, C-28(9):690-691, September 1979

[26] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall, “Dag-Consistent Distributed Shared Memory”, MIT Labaratory for Computer Science, 1996

[27] Richard P. Brent, “The Parallel Evaluation of General Arithmetic Expressions”, J. Assoc. Comput. 21 March 1974

[28] R. L. Graham, “Bounds on Multiprocessing Timing Anomalies”, Siam Journal on Applied Mathematics, 1969

[29] “False Sharing and its Effect on Shared Memory Performance”, William J. Bolosky, Microsoft Research Laboratory, Michael L. Scott, Computer Science Department University of Rochester, 1993

[30] “Executing functional programs on a virtual tree of processors”, F. Warren Burton and M. Ronan Sleep, In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, October 1981

[31] “Scheduling Multithreaded Computations by Work Stealing”, Robert D. Blumofe, The University of Texas at Austin, Charles E. Leiserson, MIT Laboratory for Computer Science, September 1999

[32] “An atomic model for message-passing”, Pangfeng Liu, William Aiello, and Sandeep Bhatt. In Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures, June 1993

[33] “Randomized parallel algorithms for backtrack search and branch-and-bound computation”, Richard M. Karp and Yanjun Zhang, Journal of the ACM, July 1993.

[34] “A bridging model for parallel computation”, Leslie G. Valiant, Communications of the ACM, August 1990

[35] “Parallel Languages: Design & Implementation part I: parallel execution”, Kornilios Kourtis, February 25, 2011

[36] “A type-checking preprocessor for Cilk 2, a multithreaded C language”, Robert C.

Miller, Master's Thesis, Department of Electrical Engineering and Computer Science, MIT, May 1995

[37] "Solution of a problem in concurrent programming control", E.W. Dijkstra, Communication of the ACM, 8(9):569, September 1965

[38] http://www.lysator.liu.se/c/duffs-device.html, Tom Duff, Duff's Device, Usenet Posting, 10 November 1983

[39] "VLSI support for a cactus stack oriented memory organization", Per Stenström, Department of Computer Engineering, University of Lund, January 1988

[40] "Wait-free synchronization", Maurice Herlihy, Digital Equipment Corporation, January 1991

[41] "The Implementation of the Cilk-5 Multithreaded Language", Matteo Frifo, Charles E. Leiserson, Keith H. Randall, MIT Laboratory for Computer Science, June 1998

[42] "Thread Scheduling for Multiprogrammed Multiprocessors", Nimar S. Arora, Robert D. Blumofe, C. Greg Plaxton, Department of Computer Science, University of Texas at Austin, June 1998