Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Παράλληλοι αλγόριθμοι ταξινόμησης σε πολυπύρηνους επεξεργαστές**
Ονοματεπώνυμο Φοιτητή	**Κοντόπουλος Σωκράτης**
Πατρώνυμο	**Στυλιανός**
Αριθμός Μητρώου	**ΜΠΠΛ/ 09048**
Επιβλέπων	**Κωνσταντόπουλος Χαράλαμπος, Επίκουρος Καθηγητής**

Ημερομηνία παράδοσης: **Φεβρουάριος 2015**

Τριμελής Εξεταστική Επιτροπή

(υπογραφή) (υπογραφή) (υπογραφή)

Περιεχόμενα

0. Περίληψη

Στο παρόν θα μελετηθούν παράλληλοι αλγόριθμοι ταξινόμησης. Το κείμενο είναι χωρισμένο σε τρεις βασικές κατηγορίες. Στην πρώτη κατηγορία αναφέρονται όλα αυτά τα στοιχεία τα οποία ανέδειξαν τον παράλληλο προγραμματισμό σαν μία εναλλακτική διαδρομή εξέλιξης των υπολογιστικών συστημάτων. Επίσης, παρουσιάζεται ένα παράδειγμα μέσα από το οποίο θα αποκτήσουμε μία βασική κατανόηση για το πώς λειτουργεί ένα παράλληλο πρόγραμμα, τι δυσκολίες δημιουργούνται και πως μπορούμε να μετρήσουμε την απόδοσή του.

Το δεύτερο μέρος είναι η αποκλειστική παρουσίαση των βιβλιοθηκών *'MPI'* και *'PTHREADS'* τις οποίες θα χρησιμοποιήσουμε. Πιο συγκεκριμένα, θα παρουσιάσουμε την φιλοσοφία της κάθε βιβλιοθήκης καθώς και τις πιο βασικές τις συναρτήσεις με τις οποίες συνοδεύονται.

Τέλος, το τρίτο και τελευταίο κύριο μέρος είναι η υλοποίηση κάποιων από τους πιο βασικούς αλγορίθμους σύμφωνα με τα προαναφερθέντα περιεχόμενα και καταμέτρηση χρόνου εκτέλεσης, απόδοσης καθώς και φόρτου εργασίας των πυρηνών – διεργασιών – νημάτων.

Επίσης, να αναφέρουμε πως σε αυτό το σύγγραμα, θα παρουσιάσουμε την αρχιτεκτονική του υπολογιστικού συστήματος και πιο συγκεκριμένα του επεξεργαστή και λειτουργικού συστήματος που δοκιμάσαμε να εκτελέσουμε τα προγράμματα και τα τελικά συμπεράσματα, μαζί με κάποιες προτάσεις για την επέκταση και βελτιστοποίηση του πονήματος.

1. Εισαγωγή

Είναι πασίγνωστο πλέον ότι οι αλγόριθμοι ταξινόμησης έχουν αναλυθεί σε υπέρμετρο βαθμό και αυτό δεν περιορίζεται μόνο σε σειριακή μορφή, αλλά και σε παράλληλη. Δεν θα μπορούσε να γίνει και διαφορετικά αφού έχει ήδη γίνει η μετάβαση της τεχνολογίας από την μονοπύρηνη εποχή της στην πολυπύρηνη. Σε συζητήσεις μεταξύ ανθρώπων με θέμα ηλεκτρονικούς υπολογιστές ή και κινητά ακόμη - οι οποίοι δεν έχουν και μεγάλη σχέση με την πληροφορική και την τεχνολογία παρά γνωρίζουν ότι ακούν και διαβάζουν από τα μέσα ενημέρωσης (περιοδικά, internet) – ακούμε συχνά να αναφέρονται στο πόσους πυρήνες έχει η συσκευή του καθενός και να γίνεται ένα είδος σύγκρισης με βάση αυτούς τους 'πυρήνες'. Θα λέγαμε ότι ο αριθμός των πυρήνων τείνει να γίνει μονάδα μέτρησης δυναμικότητας μίας ηλεκτρονικής συσκευής. Δεν εξελίχθηκαν τυχαία τα πράγματα με αυτό τον τρόπο. Την υπολογιστική δύναμη που μας δίνουν οι πολλοί πυρήνες την εκμετελλεύονται πλέον και τα προγράμματα τα οποία με τους σωστούς αλγορίθμους καταφέρνουν να την αξιοποιούν στο μέγιστο βαθμό, κάτι που αυτόματα σημαίνει πως το έδαφος είναι πρόσφορο για την ανάπτυξή τους. Έτσι ακόμη και αν δεν είναι πάντα ορατό *'δια γυμνού οφθαλμού',* οι ταχύτητες σχεδόν όλων των υπολογιστικών υπολογισμών έχουν αυξηθεί σε υπερδιπλάσιο βαθμό άρα και ο χρόνος αναμονής αποτελεσμάτων έχει μειωθεί εξίσου σε υπερδιπλάσιο βαθμό. Εν ολίγοις, τα πράγματα εξελίσσονται γρηγορότερα αλλά και με ταχύτερους ρυθμούς (χαρακτηριστικό των ημερών μας όπως θα μπορούσε να παρατηρήσει ο καθένας μας).

1.1 Η ανάγκη για παράλληλο προγραμματισμό

Ανεβαίνοντας ως κοινωνία τα σκαλοπάτια της εξέλιξης είχαμε και έχουμε να αντιμετωπίσουμε πληθώρα προβλημάτων που περιλαμβάνουν διαδικασίες οι οποίες απαιτούν μεγάλα κόστη σε χρόνο ή ενέργεια. Για παράδειγμα πρέπει να δούμε μακρύτερα στο σύμπαν μας, πρέπει να κάνουμε μεγαλύτερες και πολυπλοκότερες αναλύσεις στην οικονομία μας, πρέπει να ενημερωνόμαστε με τεράστιο όγκο πληροφοριών και να εκπαιδευόμαστε γρηγορότερα (σχεδόν

άμεσα) κ.α. Με λίγα λογια πρέπει να έχουμε πάντα και υπό οποιεσδήποτε συνθήκες την απαιτούμενη υπολογιστική ισχύ για να ανεβούμε ακόμη ένα σκαλοπάτι.

Υπό την έννοια της υπολογιστικής ισχύος όμως, πως θα μπορούσαμε να ανεβαίνουμε ολοένα και πιο ψηλά στην κλίμακα, εφόσον οι δυνατότητες, που μας προσέφερε ο δρόμος που είχαμε επιλέξει, είναι όχι μόνο πεπερασμένες αλλά και έχουν ουσιαστικά εξαντληθεί; Είναι γνωστό ότι όσο μεγαλύτερη είναι η πυκνότητα ενός κυκλώματος, τόσο περισσότερο αυξάνεται η ισχύς του, από την άλλη όμως τόσο περισσότερο αυξάνεται η θερμοκρασία του. Αυτή η όλο και μεγαλύτερη πυκνότητα υλικού δεν μπορεί να συνεχιστεί για πάντα ενώ επίσης αυτή η αύξηση της θερμοκρασίας συνιστά ταυτόχρονα αλλοίωση της απόδοσης ισχύος. Έπρεπε είτε να αλλάξουμε δρόμο και να μπούμε σε έναν άλλο, ή να διαμορφώσουμε τον ήδη υπάρχοντα δρόμο στα δικά μας *'μέτρα'* μεταβάλλοντας την ίδια του την φύση. Και οι 2 επιλογές μοιάζουν εφικτές, η κάθε μία έχοντας τα δικά της προτερήματα.

Η πιο προσιτή και ταυτόχρονα αυτή που θα έχει τα γρηγορότερα αποτελέσματα στην προσπάθειά μας για ανέλιξη και επίτευξη περισσοτέρων επιστημονικών στόχων είναι η παράλληλη λειτουργία πολλών πυρήνων. Άρα δεν χρειάζεται να ανυσηχούμε για την επιπλέον συμπύκνωση των 'τρανζίστορς' ενός πυρήνα διότι μπορούμε πλέον να χρησιμοποιούμε πολλούς και αυτοί να λειτουργούν ταυτόχρονα πολλαπλασιάζοντας έτσι την ισχύ του συστήματός μας.

Να αλλάξουμε τον δρόμο μας δεν θα ήταν από μόνο του αρκετό για τα αποτελέσματα που επιθυμούμε κι έτσι έπρεπε να αλλάξουμε όλα τα παρελκόμενα ενός επεξεργαστή, όπως το λογισμικό, το οποίο θα πρέπει να εκμεταλλεύεται τον ίδιο τον επεξεργαστή στο έπακρο. Για την ακρίβεια δεν θα αλλάξουμε το ίδιο το λογισμικό, απλά θα το τροποποιήσουμε ώστε να το εντάξουμε στην λογική των πολυπύρηνων επεξεργαστών κι έτσι οι αλγόριθμοι και τα προγράμματα θα ξαναγραφούν ώστε να μπορούν να εκμεταλλεύονται την δομή τους.

Σε αυτό το σημείο είναι ενδιαφέρον να αναφέρουμε και μια άλλη επιλογή η οποία μελετάται. Εάν επιθυμούμε να επιμείνουμε σε σύστημα με έναν πυρήνα τότε μια πιθανή λύση θα ήταν η αλλαγή της ύλης από την οποία είναι κατασκευασμένος. Έτσι, χρησιμοποιώντας άλλα υλικά κατασκευής με διαφορετικές ιδιότητες και αλλάζοντας την δόμηση των δεδομένων, θα μπορούσαμε να αυξήσουμε την υπολογιστική δύναμη ενός συστήματος. Το πιο χαρακτηριστικό παράδειγμα τέτοιου υπολογιστή είναι ο κβαντικός υπολογιστής. Η αποτελεσματικότητά του τείνει να είναι συντριπτικά μεγαλύτερη από αυτή των πολυπύρηνων επεξεργαστών με την μόνη διαφορά ότι η ανάπτυξη τέτοιων συστημάτων είναι ακόμη σε πολύ πρώιμο στάδιο.

Συνεπώς, μέχρι να ολοκληρωθεί μια τέτοια *'μεγάλη ανακάλυψη'* επιβάλλεται η επιλογή ενός εναλλακτικού μονοπατιού, και δεν είναι παρά αυτού των πολυπύρηνων συστημάτων, το οποίο και θα αναλύσουμε στην συνέχεια.

1.2 Πως αναπτύσσουμε ένα παράλληλο πρόγραμμα

Υπάρχουν αρκετοί τρόποι γραφής ενός παράλληλου προγράμματος, με δύο από αυτούς να είναι οι πιο διαδεδομένοι:

1. Παραλληλοποίηση διεργασίων
2. Παραλληλοποίηση δεδομένων

Για να εξηγηθούν κατάλληλα οι τρόποι παραλληλοποίησης και να γίνει ευκολότερα κατανοητός ο κάθε τρόπος από τους αναγνώστες του συγγράματος θα χρησιμοποιηθεί ένα παράδειγμα και για τις δύο περιπτώσεις. Το παράδειγμα αφορά στο πώς μία ομάδα (έστω τεσσάρων) καθηγητών μπορεί να διορθώσει ένα σύνολο (έστω είκοσι) γραπτών ενός διαγωνισμού με τέσσερα θέματα το κάθε γραπτό.

1. Διόρθωση διαγωνίσματος με παραλληλοποίηση διεργασιών από τους καθηγητές

Σε αυτή την πρώτη περίπτωση όπως αναφέρεται στον τίτλο, τα θέματα - διεργασίες (4) θα διαμοιραστούν και στους τέσσερις καθηγητές - πυρήνες. Με αυτόν τον τρόπο κάθε καθηγητής - πυρήνας θα έχει να διορθώσει ένα θέμα – διεργασία σε κάθε ένα από τα είκοσι γραπτά. Δηλαδή

ο καθηγητής - πυρήνας Α πρέπει να διορθώσει το θέμα - διεργασία Α, ο καθηγήτης - πυρήνας Β το θέμα - διεργασία Β κ.ο.κ.

2. Διόρθωση διαγωνίσματος με παραλληλοποίηση δεδομένων από τους καθηγητές

Αυτή είναι η δεύτερη περίπτωση και με βάση τον τίτλο συμπεραίνουμε ότι οι καθηγητές αυτή την φορά θα διαμοιράσουν μεταξύ τους τα γραπτά – δεδομένα ισομερώς. Έτσι αντιστοίχως, ο καθηγητής – πυρήνας Α θα διορθώσει τα πρώτα 5 γραπτά – δεδομένα, ο καθηγητής Β τα δεύτερα 5 γραπτά – δεδομένα κ.ο.κ.

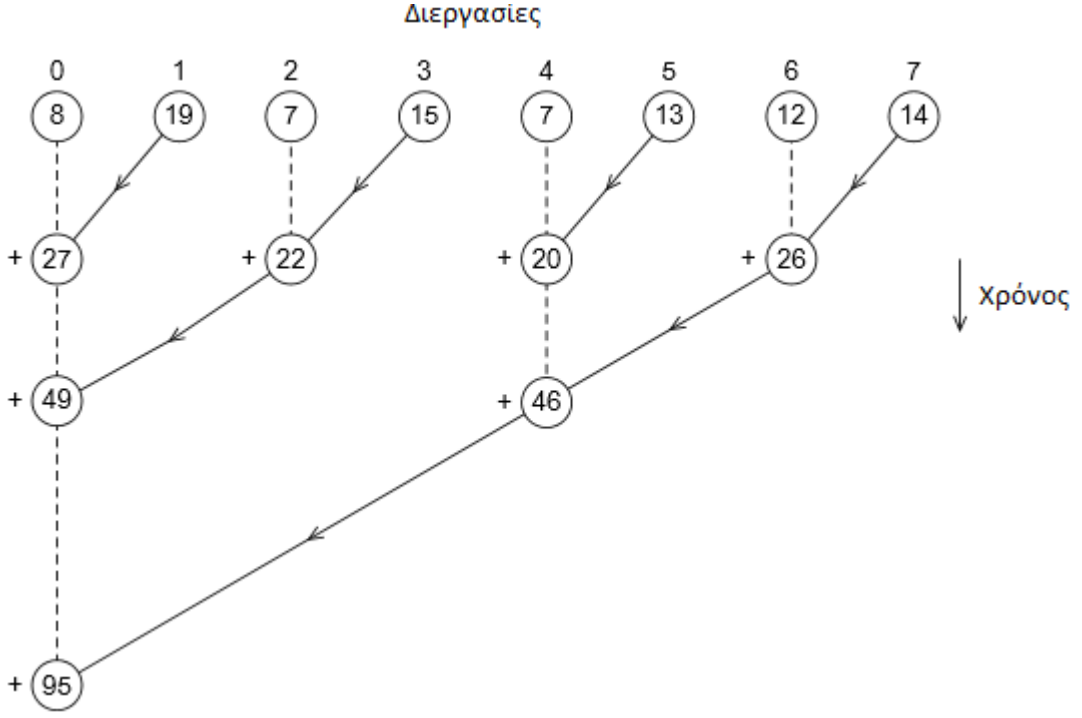
Στην συνέχεια θα επιχειρήσουμε να αναπτύξουμε ένα δεύτερο παράδειγμα ώστε να γίνουν πιο κατανοητά τα παραπάνω αλλά και για να αναδείξουμε και κάποιες άλλες πτυχές μίας παραλληλοποίησης ενός αλγορίθμου – τρόπου επίλυσης κάποιου ζητήματος.

1.3 Παράδειγμα πρόσθεσης 8 αριθμών

Έστω ότι θελουμε να κάνουμε μία πρόσθεση 8 αριθμών και να βρούμε το αποτέλεσμά της.

Εάν χρησιμοποιήσουμε ένα μονοπύρηνο σύστημα τότε θα πρέπει ο ένας και μοναδικός πυρήνας να διαβάσει έναν αριθμό, στην συνέχεια να διαβάσει τον επόμενο αριθμό, στην συνέχεια να κάνει την πράξη, στην συνέχεια να διαβάσει έναν τρίτο αριθμό, στην συνέχεια να κάνει την πρόσθεση με το αποτέλεσμα της πρώτης πράξης κ.ο.κ. Η χρήση όλων αυτών των συνδέσμων *“στην συνέχεια”* φανερώνει μία ενιαία και συνεχόμενη ροή των πραγμάτων, στην συγκεκριμένη περίπτωση πρόσθεση αριθμητικών ψηφίων. Εφόσον ο μοναδικός πυρήνας θα προσθέτει κάθε φορά κι από έναν αριθμό κι έχουμε 8 αριθμούς τότε είναι φανερό ότι θα χρειαστούν 8 φάσεις – προσθέσεις, όπως γίνεται φανερό κι ότι ένας πυρήνας αναλαμβάνει να διεκπεραιώσει όλη την *'δουλειά'.*

Αντίστοιχα αυτή η πράξη πρόσθεσης 8 αριθμών θα μπορούσε να γίνει πολύ πιο απλά αλλά και ταυτόχρονα πιο γρήγορα. Πως θα το πετυχαίναμε αυτό; Με την παραλληλοποίηση των διεργασιών (εικ.1) και ανάθεση κάθε διεργασίας σε έναν πυρήνα.



Εικόνα 1

8 διεργασίες – πυρήνες που συνεργάζονται για την πρόσθεση 8 αριθμών

Έτσι έστω ότι έχουμε 8 πυρήνες, και ότι όλοι διαβάζουν κι από έναν αριθμό ταυτόχρονα. Επίσης ταυτόχρονα κάθε ζυγός πυρήνας (το 0 το συμπεριλαμβάνουμε στους ζυγούς) θα κάνει την πρόσθεση με τον αριθμό που έχει διαβάσει ο διπλανός του (μονός) πυρήνας και θα κρατήσει το αποτέλεσμα. Στην δεύτερη φάση θα έχουν απομείνει μόνο τέσσερις (αποτελέσματα) αριθμοί και με τον ίδιο τρόπο θα προστεθούν για να πάμε στην τρίτη φάση όπου θα μείνουν δύο αριθμοί και τέλος στην τέταρτη φάση όπου θα μείνει το τελικό αποτέλεσμα.

Με μια γρήγορη ματιά βγάζουμε το συμπέρασμα ότι στην δεύτερη περίπτωση αθροίζονται τα 8 νούμερα στο μισό περίπου χρόνο σε σχέση με την πρώτη αφού το πλήθος των φάσεων είναι λογαριθμικό του πλήθους των στοιχείων προς ταξινόμηση. Αυτό μοιάζει να είναι αληθές αλλά στην πραγματικότητα δεν είναι. Για να την δούμε ολόκληρη αυτή την πραγματικότητα θα πρέπει να υλοποιήσουμε τις λογικές - αλγορίθμους σε δύο αντίστοιχα προγράμματα και να κάνουμε τις μετρήσεις – συγκρίσεις στους χρόνους εκτέλεσης τους.

Εκτελώντας τα προγράμματα σε ηλεκτρονικό υπολογιστή και παίρνοντας τις μετρήσεις βλέπουμε ότι το πρόγραμμα που εκμεταλλεύεται έναν πυρήνα παράγει αποτέλεσμα αρκετά γρηγορότερα από το αντίστοιχο που εκμεταλλεύεται και τους 8 πυρήνες. Γιατί να συμβαίνει αυτό; Δεν θα έπρεπε ο χρόνος που χρειάζονται 8 πυρήνες να προσθέσουν έναν πίνακα 8 αριθμών να είναι γρηγορότερος;

Η απάντηση στα παραπάνω ερωτήματα βρίσκεται στην επικοινωνία μεταξύ των πυρήνων. Αναφέρθηκε προηγουμένως ότι κάθε πυρήνας διαβάζει κι από έναν αριθμό και ότι για να γίνει η πρόσθεση μεταξύ δύο αριθμών πρέπει ο ζυγός πυρήνας να πάρει και το ψηφιο που έχει αποθηκευμένο και ο διπλανός – μονός - πυρήνας. Αυτή λοιπόν η επικοινωνία ή συνεργασία μεταξύ των δύο πυρήνων στοιχίζει όσον αφορά στο χρόνο, υπάρχει δηλαδή μία χρονοκαθυστέρηση προκειμένου να πραγματοποιηθεί. Κάποιες φορές είναι τόσο πολύ κρίσιμη (*'critical)'* αυτή η επικοινωνία και καθυστερεί τόσο πολύ το εκτελέσιμο πρόγραμμα να παράξει αποτέλεσμα, που τελικά δεν συμφέρει να παραλληλοποιηθεί, και η σειριακή υλοποίησή του μοιάζει ιδανικότερη.

1.4 Άλλα προβλήματα εγγραφής εφαρμογής με παράλληλη λογική

Ένα ιδεατό σενάριο θα ήταν να υπήρχε ένας μεταγλωτιστής ο οποίος θα έπαιρνε το σειριακό πρόγραμμα και θα το τροποποιούσε σε παράλληλο. Και όμως υπάρχει. Αυτό που μπορούν να κάνουν οι όποιοι μεταγλωτιστές έχουν δημιουργηθεί είναι να παίρνουν μία δομή από ένα σειριακό πρόγραμμα και να το μεταγλωτίζουν εύκολα σε παράλληλο. Αυτό που δεν μπορούν να κάνουν και είναι ο λόγος που δεν τους προτιμούμε μέχρι να αναπτυχθούν κατάλληλότερα εργαλεία, είναι να *'παραλληλοποιούν'* όλη την ακολουθία του προγράμματος με επιτυχία. Για να το αντιληφθεί κάποιος καλά ας πάρουμε ξανά το παράδειγμα με την πρόσθεση των 8 αριθμών. Ενώ επισημάνθηκε ότι και οι 8 πυρήνες μπορούν να λειτουργήσουν παράλληλα και να διατελέσουν τον σκοπό τους ο οποίος ειναι η πρόσθεση, δεν επισημάνθηκε πως πρέπει και οι 8 πυρήνες όταν εκτελούν τις πράξεις τους να βρίσκονται στην ίδια φάση. Δηλαδή δεν γίνεται όποιος ολοκληρώνει πρώτος τους υπολογισμούς του να μεταβαίνει στην επόμενη φάση να κάνει τους υπολογισμούς που του αντιστοιχούν στην τρέχουσα φάση και την ίδια στιγμή οι υπόλοιποι 7 να βρίσκονται στην προηγούμενη φάση προσπαθώντας να ολοκληρώσουν τις δικές τους πράξεις. Το πρόβλημα και εδώ είναι εμφανές. Δεν μπορεί ένας πυρήνας να χρησιμοποιήσει ένα αποτέλεσμα από έναν άλλο ο οποίος δεν βρίσκεται στην ίδια φάση γιατί πολύ απλά υπάρχει πάρα πολύ σοβαρός κίνδυνος αυτό το αποτέλεσμα τελικά να καταλήξει με λανθασμένη τιμή. Αυτό το είδος του προβλήματος είναι πολύ σημαντικό και πρέπει να γινει κατανοητό, για αυτό το λόγο θα επεξηγηθεί περισσότερο και θα παρουσιαστεί μέσα από τους υλοποιημένους αλγορίθμους στην συνέχεια. Ο συγχρονισμός λοιπόν, αποτελεί ένα καίριο και σημαντικό πρόβλημα που πρέπει κάθε φορά που γράφουμε ένα παράλληλο πρόγραμμα να το λαμβάνουμε υπ' όψιν πάρα πολύ σοβαρά.

Συνοψίζοντας, τα δύο (λογικά) προβλήματα που προκύπτουν μέχρι στιγμής είναι:

- Επικοινωνία μεταξύ των πυρήνων
- Συγχρονισμός μεταξύ των πυρήνων

Φυσικά υπάρχουν και άλλα προβλήματα τα οποία θα τα δούμε να παρουσιάζονται μπροστά μας στην συνέχεια και κατά την υλοποίηση των αλγορίθμων σε κώδικα. Αυτά θα τα αναλύσουμε επί τόπου όταν τα συναντήσουμε και θα εξηγήσουμε τους τρόπους αντιμετώπισής τους.

1.5 Παράλληλο υλικό (hardware)

Με μία μικρή ανασκόπηση γνωρίζουμε ότι η λογική κατασκευής του Ηλεκτρονικού Υπολογιστή προήλθε από το μοντέλο του '*VON NEUMANN*'. Κεντρική μονάδα επεξεργασίας, μνήμη και η διασύνδεση αυτών είναι τα βασικά συστατικά που αποτελούν τον υπολογιστή με την μορφή που τον γνωρίζουμε σήμερα ενώ η διαδικασία λειτουργίας του είναι η ίδια. Δηλαδή εισάγουμε δεδομένα και με βάση τα εξαρτήματα και την οδηγία που εμείς δίνουμε στον Υπολογιστή αυτά υπόκεινται σε μία συνεχή ροή μέχρι να βγουν από την μονάδα εξόδου είτε αυτούσια είτε επεξεργασμένα. Αυτή η ροή είναι ενιαία και συνεχόμενη άρα και σειριακή. Βγάζοντας ένα γρήγορο συμπέρασμα μπορούμε να πούμε ότι το σύστημα *'VON NEUMANN'* δεν είναι ένα παράλληλο σύστημα και στην πραγματικότητα όταν αναπτύχθηκε δεν προοριζόταν να μετεξελιχθεί σε ένα σύστημα που να επεξεργάζεται παράλληλα τα δεδομένα. Αυτό δεν σημαίνει όμως και ότι δεν μπορεί να τροποποιηθεί.

Για να συμβεί αυτή η τροποποίηση πρέπει να αναπτύξουμε λίγο περισσότερο το μοντέλο του *'VON NEUMANN'*. Η κεντρική μονάδα επεξεργασίας αποτελείται από επιπλέον δύο μέρη. Την μονάδα *ελέγχου (control unit)* και την αριθμητική-λογική μονάδα *(ALU-Arithmetic and Logic Unit)*. Η μονάδα ελέγχου είναι υπεύθυνη για το ποιες οδηγίες θα εκτελεστούν κατά την διάρκεια του προγράμματος και η αριθμητική-λογική μονάδα (ALU) είναι υπεύθυνη για αυτήν την τρέχουσα εκτέλεση των οδηγιών. Για να είμαστε πιο ακριβείς, ο διαχωρισμός των συστημάτων συμβαίνει με βάση την λειτουργία αυτών των δύο μονάδων της κεντρικής μονάδας επεξεργασίας και τον τρόπο που αξιοποιούν τις εισερχόμενες οδηγίες και τα δεδομένα. Ας μην το αναλύσουμε βαθύτερα όμως κι ας δούμε ποιοι είναι οι εμφανέστεροι παράγοντες και όπου χρειαστεί η συμβολή αυτών των δύο όρων των μονάδων θα ανατρέξουμε ξανά στα ονόματά τους.

Η οδηγία και τα δεδομένα που εισάγουμε σε έναν υπολογιστή είναι οι δύο όροι με τους οποίους μπορούμε να διαχωρίσουμε ένα ηλεκτρονικό υπολογιστικό σύστημα. Με κριτήριο αυτούς τους δύο όρους και με βάση την ταξινόμηση του Φλυν *(Flynn's taxonomy)* ένα παράλληλο ηλεκτρονικό υπολογιστικό σύστημα χωρίζεται σε δύο κατηγορίες. Η πρώτη κατηγορία περιέχει το *'SIMD'* σύστημα και η δεύτερη το *'MIMD'* σύστημα. Και τα δύο συστήματα είναι παράλληλα και έχουν την δυνατότητα να επεξεργαστούν παράλληλα τα εισαγόμενα δεδομένα. Η μόνη τους διαφορά εντοπίζεται στον αριθμό των οδηγιών που εφαρμόζονται πάνω στα δεδομένα.

1.5.1 SIMD σύστημα

Το πρώτο σύστημα παράλληλου hardware είναι όπως προαναφέρθηκε το **SIMD**. **SIMD** είναι ακρωνύμιο του ***'single instruction multiple data'***. Δηλαδή μία ενιαία οδηγία εφαρμόζεται σε πολλαπλά δεδομένα παράλληλα. Αναλυτικότερα, σε ένα κοινό SIMD σύστημα μία οδηγία μεταδίδεται με την βοήθεια της μονάδας ελέγχου, ή control unit όπως αναφέραμε, στην αριθμητική και λογική μονάδα (ALU) η οποία είναι υπεύθυνη για την εφαρμογή της στα δεδομένα. Συνεπώς, εάν η εφαρμογή χρησιμοποιεί τα δεδομένα παράλληλα θα πρέπει να υπάρχουν αντίστοιχες ALU ώστε να διαμοιράζει την οδηγία στα δεδομένα.

1.5.2 MIMD σύστημα

Το δεύτερο σύστημα και μεταφράζοντας το ακρωνύμιό του **(Multiple Instructions Multiple Data)** μας παραπέμπει σε πολλαπλές οδηγίες του συστήματος που εφαρμόζονται σε πολλαπλές ομάδες δεδομένων. Τυπικά, τα MIMD συστήματα αποτελούνται από πλήρως ανεξάρτητους επεξεργαστές ή πυρήνες και κάθε ένας από αυτούς έχει την δική του μονάδα ελέγχου και την δική του αριθμητική – λογική μονάδα. Επιπλέον, γίνεται ορατό πως οι

επεξεργαστές των MIMD συστημάτων δεν είναι συγχρονισμένοι μεταξύ τους και κατ'επέκταση έχουν τον δικό τους ανεξάρτητο χρόνο εκτέλεσης των πράξεων.

Τέλος, και τα MIMD χωρίζονται σε δύο κύριες κατηγορίες. Τα ***συστήματα διαμοιρασμένης μνήμης ή shared memory systems*** και τα ***συστήματα κατανεμημένης μνήμης ή αλλιώς distributed memory systems***. Και τα δύο αυτά συστήματα θα αναλυθούν στην συνέχεια ωστόσο το σημαντικό και κοινό στοιχείο που αφορούν και στα δύο συστήματα είναι η μνήμη.

1.6 Σκοπός

Ο σκοπός του πονήματος αυτού είναι να παρουσιαστούν τα όπλα της σημερινής επιστήμης της πληροφορικής, δηλαδή οι βιβλιοθήκες και οι αντίστοιχες συναρτήσεις του παράλληλου προγραμματισμού, για την πλήρη αξιοποίηση των πολυπύρηνων συστημάτων. Για να γίνουν περισσότερο κατανοητές οι έννοιες και οι συναρτήσεις που θα αναλυθούν στην συνέχεια, προσπαθήσαμε να τις χρησιμοποίησουμε στην ανάπυξη των βασικότερων αλγορίθμων ταξινόμησης.

1.7 Αλγόριθμοι και βιβλιοθήκες

Οι αλγόριθμοι που αποφασίστηκε να αναπτυχθούν και να παρουσιαστούν είναι οι εξής:

- **Quicksort**
- **Radix sort**
- **Merge sort**
- **Bitonic sort**
- **Odd/Even Transposition**

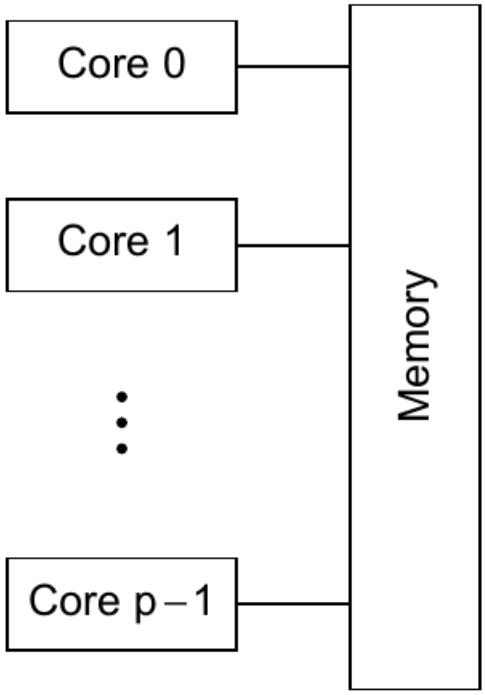
Αυτή η επιλογή έγινε με 2 κριτήρια. Το πρώτο ήταν η κυριότητα που καταλαμβάνουν στον κόσμο του προγραμματισμού και της πληροφορικής καθώς και στο πλήθος των εφαρμογών που τους συναντούμε. Το δεύτερο κριτήριο ήταν με βάση τις έρευνες που έχουν γίνει πάνω σε αυτούς τους αλγορίθμους από εξέχοντες επιστήμονες, και τα συμπεράσματά τους τα οποία βοήθησαν να γίνει κατανοητή η *'παραλληλοποίησή'* τους σε αρκετά ικανοποιητικό βαθμό.

Η γλώσσα προγραμματισμού που χρησιμοποιήθηκε στο σύγγραμα αυτό είναι η Γλώσσα C, ενώ οι βιβλιοθήκες που χρειάστηκαν ήταν η **mpi** και η **pthreads**. Ωστόσο, η επιλογή των προαναφερθείσων βιβλιοθηκών δεν έγινε με τυχαίο τρόπο. Οι πιο κοινές βιβλιοθήκες που χρησιμοποιούμε σήμερα είναι:

- **MPI**
- **PTHREADS**
- **OPENMP**

Ας δούμε όμως γιατί καταλήξαμε συγκεκριμένα στην επιλογή αυτών των 2 βιβλιοθηκών της C:

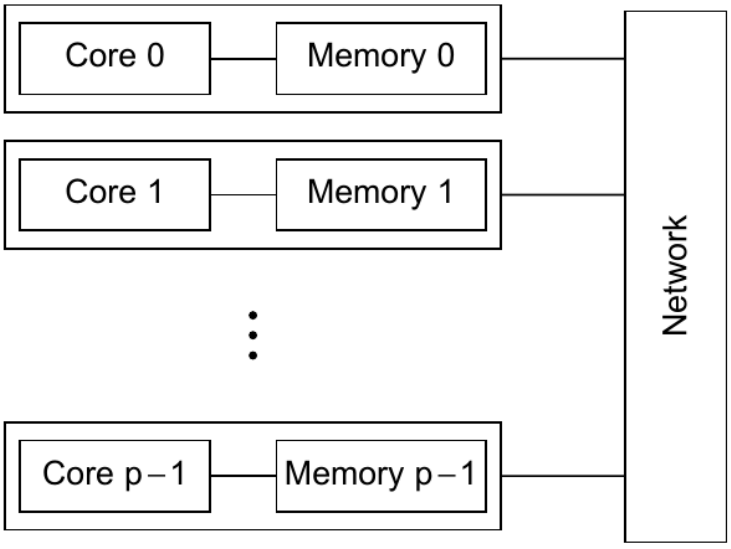
Ο παράλληλος προγραμματισμός είναι χωρισμένος σε 2 κατηγορίες, με βάση την μνήμη που χρειάζεται το ηλεκτρονικό σύστημα για να λειτουργήσει παράλληλα τους πυρήνες του. Έτσι, η πρώτη κατηγορία συστημάτων είναι η *'shared memory systems'* (εικ.2) κατά την οποία στους πυρήνες



Εικόνα 2

Υπολογιστικό σύστημα p πυρήνων τα οποία χρησιμοποιούν μία κοινόχρηστη μνήμη

διαμοιράζεται μία κοινή μνήμη για όλους. Από την άλλη πλευρά, εάν κάθε πυρήνας έχει και την δική του κατανεμημένη μνήμη τότε προκύπτει η δεύτερη κατηγορία συστημάτων, η αποκαλούμενη ως συστήματα κατανεμημένης μνήμης (εικ.3). Στα κατανεμημένα συστήματα τα οποία συνήθως αποτελούνται απο υπολογιστές τον ρόλο του πυρήνα παίρνει ο κάθε επεξεργαστής του υπολογιστή του δικτύου. Άρα όταν αναφερόμαστε σε πυρήνες σε αυτού του τύπου τα συστήματα πάντοτε θα εννοούμε επεξεργαστές.



Εικόνα 3

Υπολογιστικό σύστημα p πυρήνων κατά το οποίο κάθε πυρήνας έχει και την δική του αποκλειστική μνημη

Η '*MPI*' λοιπόν εκμεταλλεύεται συστήματα κατανεμημένης μνήμης για την λειτουργία των πυρήνων/επεξεργαστών, κάτι που συνήθως το βλέπουμε σε δίκτυα υπολογιστών, όπου κάθε υπολογιστής - πυρήνας έχει και την δική του μνήμη για να χρησιμοποιήσει. Εν αντιθέσει με την *'MPI'* βιβλιοθήκη, στις άλλες 2 (**PTHREADS** και **OPENMP**) οι υπολογιστές - πυρήνες χρησιμοποιούν 'shared memory' – διαμοιρασμένη μνήμη – για να λειτουγήσουν παράλληλα.

Η κάλυψη και των 2 τύπων συστημάτων είναι και ο λόγος που χρησιμοποιήθηκαν αυτές οι 2 βιβλιοθήκες, για την ανάλυση και υλοποίηση των αλγορίθμων του πονήματος αυτού.

Όπως προαναφέραμε ο διαχωρισμός των συστημάτων γίνεται με βάση την μνήμη. Αυτή είναι και η ειδοποιός διαφορά στην οποία αξίζει να αναφερθούμε εκτενέστερα. Εάν καταλάβουμε πως λειτουργεί η μνήμη και η κρυφή μνήμη (cache) τότε θα είμαστε σε θέση να προβλέψουμε και να επιλύσουμε αρκετά και σημαντικά προβλήματα πριν καν εμφανιστούν.

Σε αυτό το σημείο δεν πρέπει να παραλείψουμε και την χρήση των υβριδικών συστημάτων.

Ένα υβριδικό σύστημα προκύπτει από την ένωση (συνήθως με Ethernet) πολλαπλών κόμβων(όπου κόμβος μπορεί να θεωρηθεί ένα υπολογιστικό *'shared memory'* σύστημα). Έτσι ένας αλγόριθμος μπορεί να αναπτυχθεί υβριδικά, δηλαδή να χρησιμοποιήθούν βιβλιοθήκες και από τις 2 κατηγορίες ώστε να μπορέσει να επιτελέσει τον σκοπό του επιτυχημένα.

1.8 Η συνοχή της κρυφής μνήμης (Cache Coherence)

Είναι ευρέως γνωστό πως ένας προγραμματιστής δεν μπορεί να διαχειριστεί στον απόλυτο βαθμό την κρυφή μνήμη *'cache memory'*. Μόνο το ηλεκτρομηχανολογικό (hardware) μέρος του Υπολογιστή (hardware) μπορεί να το κάνει. Σε ένα σύστημα διαμοιρασμένης μνήμης (shared memory system), αυτό μπορεί να έχει σοβαρές και αρνητικές επιπτώσεις. Ας ανατρέξουμε στην παράγραφο που παρουσιάσαμε συνοπτικά τα προβλήματα που σύμφωνα με την λογική θα παρουσιαστούν κατά την συγγραφή ενός προγράμματος και ας δούμε το θέμα του συγχρονισμού των πυρήνων.

Εκεί αναφέραμε πως όταν τα δεδομένα είναι διαχωρισμένα έστω σε ισόποσες ομάδες προκειμένου να επεξεργαστούν από τους πυρήνες τότε ο συγχρονισμός είναι σημαντικός λόγος του ότι κάποιος πυρήνας δεν μπορεί να εκτελέσει πράξεις σε μία μεταγενέστερη φάση του προγράμματος εάν δεν έχουν φτάσει σε αυτή τη φάση κι οι υπόλοιποι πυρήνες. Ας δούμε ένα παράδειγμα για να το καταλάβουμε καλύτερα και να αντιληφθούμε που εμπεριέχεται κι η κρυφή μνήμη σε όλο αυτό το θέμα.

Ας υποθέσουμε ότι έχουμε ένα σύστημα διαμοιρασμένης μνήμης -shared memory system- το οποίο αποτελείται από δύο πυρήνες κάθε ένας από τους οποίους έχει την δική του κρυφή μνήμη. Όσο η *'εργασία'* και των δύο πυρήνων είναι να διαβάζουν απλά δεδομένα τότε δεν υπάρχει κανένα πρόβλημα. Όταν όμως χρειαστεί να εκτελέσουν μία πράξη με μία έστω X μεταβλητή και η οποία εμπεριέχεται στην κρυφή μνήμη και των δύο πυρήνων με διαφορετική τιμή τότε πως ξέρουμε ποια από τις δύο τιμές είναι η σωστή που θα πρέπει να διαβάσουν οι πυρήνες; Ας δούμε τον πίνακα 1 για να έχουμε καλύτερη άποψη του προβλήματος.

Υποθέτουμε ότι το X είναι η μεταβλητή που προαναφέραμε και που χρησιμοποιούν οι δύο πυρήνες ενώ η αρχική της τιμή είναι έστω 2. Το Y0 είναι μία ιδιωτική μεταβλητή που ανήκει στον πρώτο πυρήνα (ΠΥΡΗΝΑΣ 0) και οι Y1 και Z1 είναι δύο ιδιωτικές μεταβλητές που ανήκουν στον πυρήνα 1. Τώρα υποθέτουμε πως οι καταστάσεις (statements) εκτελούνται όπως φαίνονται παρακάτω στους αντίστοιχους χρόνους.

ΦΑΣΗ ΠΡΟΓΡΑΜΜΑΤΟΣ	ΠΥΡΗΝΑΣ 0	ΠΥΡΗΝΑΣ 1
0	Y0 = X	Y1 = 3 * X
1	X = 7	ΕΣΤΩ ΑΔΡΑΝΗΣ (IDLE)
2	ΕΣΤΩ ΑΔΡΑΝΗΣ (IDLE)	Z1 = 4 * X

Πίνακας 1

Παράδειγμα απρόβλεπτης κατάστασης όταν μία μεταβλητή είναι κοινή ανάμεσα σε πολλαπλούς πυρήνες

Αφού το Χ το αρχικοποιήσαμε δίνοντας του την τιμή 2 τότε στην ΦΑΣΗ 0 του πυρήνα 0 το Υ0 θα είναι 2. Αντίστοιχα στην ίδια φάση του πυρήνα 1 θα είναι 3 * 2 δηλαδή Υ1 = 6. Μέχρι εδώ όλα καλά μέχρι να μπούμε στην ΦΑΣΗ1. Σε αυτή τη φάση εκτελείται μία δήλωση (statement) στον πυρήνα 0 το οποίο είναι το διάβασμα της μεταβλητής Χ. Αυτή τώρα στην κρυφή μνήμη του πυρήνα 0 γίνεται 7 ενώ στην κρυφή μνήμη του πυρήνα 1 παραμένει 2 μιας και αυτός στην ΦΑΣΗ 1 παραμένει αδρανής. Τώρα στην ΦΑΣΗ 2 ο πυρήνας 0 μένει με την σειρά του αδρανής άρα το Χ έχει την τιμή που πήρε στην προηγούμενη φάση του δηλαδή 7 και ο πυρήνας 1 πρέπει να εκτελέσει την πράξη Ζ1 = 4 * Χ. Το Χ στην δεύτερη φάση τι τιμή θα έχει; Μία γρήγορη απάντηση θα ήταν 7 φυσικά διότι ήταν η τελευταία ενημέρωση της τιμής του και ο πυρήνας 0 εξέπεμψε αυτή την τιμή και στους υπόλοιπους (ΠΥΡΗΝΑΣ 1) με αποτέλεσμα να ανανεωθεί η τιμή του και άρα το Ζ1 σε αυτή τη περίπτωση να γίνει 28. Μία άλλη όμως λέει πως θα παραμείνει 2 αφού στην κρυφή μνήμη του πυρήνα 1 δεν έχει αλλάξει η τιμή και συνεπώς το Ζ1 θα ήταν 8. Τι ισχύει τελικά;

Αυτό είναι ένα πρόβλημα που καλούνται να λύσουν οι προγραμματιστές αφού και δεν μπορούν να επέμβουν στην κρυφή μνήμη (cache memory) και δεν γνωρίζουν πότε θα γίνει ανανέωση όλων των κρυφών μνημών έτσι ώστε όλες να συγχρονιστούν. Αυτό το πρόβλημα είναι γνωστό ως πρόβλημα συνοχής της κρυφής μνήμης ή πιο γνωστό **(cache coherence problem)**. Το πρόβλημα συνοχής της μνήμης μπορεί να λυθεί σε επίπεδο *'hardware'* ή σε επίπεδο λειτουργικού συστήματος και ο προγραμματιστής το μόνο που μπορεί να δει είναι τις ενημερωμένες τιμές στην δική του κρυφή μνήμη.

Το πρόβλημα της συνοχής της κρυφής μνήμης **(cache coherence problem)** μπορεί να αντιμετωπιστεί σε έως έναν βαθμό με δύο τρόπους. Ο ένας είναι να κατασκοπεύουμε συνεχώς την ροή της και να εκπέμπουμε συνεχώς και σε όλους τους πυρήνες τα αποθηκευμένα δεδομένα και ο άλλος να χτίζουμε μία δομή δεδομένων ή αλλιώς κατάλογο **(directory)** ο οποίος θα αποθηκεύει συνεχώς την κατάσταση της κρυφής μνήμης και την διαμοιράζει μόνο στους πυρήνες που πρέπει.

Η συνεχής κατασκοπεία της συνοχής της κρυφής μνήμης **(snooping cache coherence)** βασίζεται στην λογική των διαύλων **(BUS)**. Όταν δηλαδή πολλοί πυρήνες είναι συνδεδεμένοι σε έναν κοινό δίαυλο και ο ένας πυρήνας από αυτούς εκπέμψει ένα σήμα αλλαγής μίας μεταβλητής τότε θα ενημερωθούν και οι υπόλοιποι πυρήνες που είναι συνδεδεμένοι στον δίαυλο αυτόν.

Το μεινονέκτημα της συνεχούς παρακολούθησης της συνοχής της κρυφής μνήμης εντοπίζεται στην εφαρμογή της σε υπολογιστές με πολυπήρηνους επεξεργαστές οι οποίοι χρησιμοποιούν κοινόχρηστη μνήμη. Έτσι η εκπομπή συνεχόμενων ενημερώσεων κάθε φορά που έγκειται μια αλλαγή σε μία τιμή μίας μεταβλητής περισσότερο καθυστερεί και υποβαθμίζει το σύστημα παρά το διευκολύνει. Η λύση όπως προαναφέραμε είναι στην δημιουργία της δομής δεδομένων εκείνης στην οποία αποθηκεύεται η κάθε γραμμή της κρυφής μνήμης και γίνεται ορατή μονάχα στους πυρήνες εκείνους που την χρειάζονται.

1.9 Συνολική απόδοση παράλληλου προγράμματος

Ένα σημαντικότατο στοιχείο που οφείλουμε να λαμβάνουμε υπόψιν κατά την εγγραφή ενός παράλληλου προγράμματος (και όχι μόνο παράλληλου) είναι η απόδοσή του. Συγκεκριμένα, δεν έχει αρκετό νόημα να συγγράψουμε ένα σωστό, από άποψη λογικής και συντακτικού, πρόγραμμα όταν την ίδια στιγμή αυτό αργεί να παράξει αποτελέσματα κατά την εκτέλεση του. Πως όμως αξιολογούμε ένα πρόγραμμα; Με ποια σχέση κάνουμε αυτή την αξιολόγηση; Ποιοι είναι αυτοί οι όροι τους οποίους χρησιμοποιούμε για να εξηγήσουμε και περιγράψουμε την αξιολόγηση; Αυτά είναι μόνο μερικά ερωτήματα που πρέπει να απαντηθούν ώστε να έχουμε πιο ολοκληρωμένη άποψη για τον τρόπο που πρέπει να προσσεγγίζουμε την υλοποίηση ενός παράλληλου αλγορίθμου.

1.10 Επιτάχυνση και βαθμός απόδοσης

Η επιτάχυνση και ο βαθμός απόδοσης είναι δύο όροι οι οποίοι μας βοηθούν να αξιολογήσουμε την συνολική απόδοση ενός συστήματος. Με διαφορετικά λόγια θα μπορούσαμε να πούμε πως είναι μονάδες μετρήσεως. Ας δούμε όμως πως μπορούμε να τις υπολογίσουμε.

Αρχικά, για να επιτύχουμε την βέλτιστη επιτάχυνση, το ιδανικό που πρέπει να κάνουμε είναι να διαμοιράσουμε τα δεδομένα ισόποσα στους επεξεργαστές – πυρήνες. Επίσης, οι επεξεργαστές – πυρήνες κατά την επεξεργασία των δεδομένων δεν θα πρέπει να αναλαμβάνουν άλλες εργασίες. Εάν ο αριθμός των πυρήνων είναι p και έχοντας σαν δεδομένο ότι ισχύουν τα παραπάνω αξιώματα, μπορούμε να ισχυριστούμε ότι το παράλληλο πρόγραμμα μας είναι p φορές γρηγορότερο από το αντίστοιχο σειριακό. Εάν συμβολίσουμε τον χρόνο που χρειάζεται ένα σειριακό πρόγραμμα για να εκτελέσει τις πράξεις του ως T_{serial} και τον αντίστοιχο χρόνο του παράλληλου ως $T_{parallel}$ τότε η επιτάχυνση ορίζεται από τον τύπο $S = T_{ser} / T_{par}$ και εάν ο ιδανικός χρόνος παραλληλοποίησης είναι ο $T_{parallel} = T_{serial} / p$, τότε η βέλτιστη επιτάχυνση ορίζεται από τον τύπο ως S = p και ονομάζεται διαφορετικά *'γραμμική επιτάχυνση'.*

Στην πράξη είναι αδύνατο να επιτύχουμε γραμμική επιτάχυνση σε ένα πρόγραμμα παράλληλα υλοποιημένο διότι είτε σε ένα σύστημα κοινόχρηστης μνήμης θα χρειαστεί να χρησιμοποιούμε επιπλέον εργασίες στους πυρήνες ώστε όλοι τους να διαβάζουν πάντα την ανανεωμένη τιμή από τις μεταβλητές που διαχειρίζονται, είτε σε ένα σύστημα κατανεμημένης μνήμης θα πρέπει να επιφορτώνουμε τους επεξεργαστές με επιπλέον 'δουλειά' όπως συνεχόμενης εκπομπής τιμών μεταβλητών έτσι ώστε να είναι ορατές από όλους τους υπόλοιπους επεξεργαστές του συστήματος. Είναι φανερό ότι αυτές οι λειτουργικές επιβαρύνσεις **($T_{overhead}$)** δεν υπάρχουν σε ένα σειριακό πρόγραμμα όπως επίσης είναι φανερό πως ο χρόνος εκτέλεσης ενός παράλληλου προγράμματος αυξάνεται (αρκετα) και δεν είναι ποτέ ο ιδανικός. Επίσης γίνεται εύκολα αντιληπτό πως όσο αυξάνεται ο αριθμός των πυρήνων ή των επεξεργαστών τόσο αυξάνεται (χρονικά) και το κόστος επικοινωνίας τους αντίστοιχα.

Αν ορίσουμε ως S την επιτάχυνση του παράλληλου προγράμματος και θεωρώντας πάντα πως η βέλτιστη – γραμμική επιτάχυνση είναι ο αριθμός των πυρήνων p τότε η γενική σχέση που ορίζει την επιτάχυνση ενός παράλληλου προγράμματος είναι:

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

Τύπος 1

Ο τύπος που αποδίδει την επιτάχυνση (speedup) ενός παράλληλου προγράμματος

και εάν αντικαταστήσουμε το $T_{parallel}$ με T_{serial} / p και απαλείφοντας το T_{serial} τότε έχουμε ως τύπο ορισμού της βέλτιστης ή γραμμικής επιτάχυνσης το S = p το οποίο είναι ασυνήθιστο να υλοποιηθεί τουλάχιστον σε μεγάλους και πολύπλοκους αλγορίθμους.

Επιπλέον, όσο οι πυρήνες αυξάνονται περιμένουμε η επιτάχυνση να γίνεται όλο και ένα μικρότερο κλάσμα της γραμμικής επιτάχυνσης. Διαφορετικά θα μπορούσαμε να πούμε ότι ο λόγος S / p μειώνεται όσο οι πυρήνες πληθαίνουν. Αυτός ο λόγος μας δείχνει τον βαθμό απόδοσης ενός παράλληλου προγράμματος. Αντικαθιστώντας το S με τον τύπο που έχουμε ορίσει παραπάνω έχουμε:

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

Τύπος 2

Ο τύπος που μας αποδίδει τον βαθμό απόδοσης ενός παράλληλου προγράμματος

Το Ε προκύπτει από την αγγλική λέξη **efficiency** που σημαίνει βαθμός απόδοσης και είναι ο δεύτερος σημαντικός όρος για την μέτρηση της συνολικής απόδοσης ενός παράλληλου προγράμματος.

Μέχρι στιγμής έχουμε αναλύσει λεπτομερώς την περίπτωση που ένα πρόγραμμα έχει γραμμική επιτάχυνση. Αναφέραμε όμως ότι σχεδόν ποτέ – ειδικά όταν η πολυπλοκότητα αυξάνεται – δεν έχουμε περίπτωση γραμμικής επιτάχυνσης σε ένα παράλληλο πρόγραμμα. Πάντα στους χρόνους εκτέλεσης υπεισέρχεται και ο χρόνος των λειτουργικών δαπανών (overhead) οι οποίες είναι αναγκαίες για την επικοινωνία των πυρήνων ή των διεργασιών. Έτσι, πρέπει πάντα να τον υπολογίζουμε όταν θέλουμε να είμαστε ακριβοδίκαιοι για την απόδοση του προγράμματός μας. Η τιμή που παίρνει αυτή η μονάδα μέτρησης (χρόνος λειτουργικών δαπανών) είναι ανάλογη του συνολικού αριθμού των πυρήνων ή των διεργασιών που λαμβάνουν μέρος στο παράλληλο πρόγραμμα μας. Συνεπώς, όσο μεγαλύτερος είναι αυτός ο αριθμός πυρήνων – διεργασιών τόσο μεγαλύτερη θα είναι και η τιμή του κόστους της επικοινωνίας.

Τελικά, ο χρόνος εκτέλεσης ενός παράλληλου προγράμματος ($T_{parallel}$) κατά το οποίο διαμοιράζουμε τον όγκο των δεδομένων ισόποσα σε κάθε νήμα - πυρήνα προκύπτει από το άθροισμα του χρόνου του ενός νήματος με την τιμή των λειτουργικών δαπανών (overhead) όλων των νημάτων που επιβάλλεται για την σωστή επικοινωνία. Η τελική μορφή που παίρνει ο τύπος εύρεσης του χρόνου εκτέλεσης είναι:

$$T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$$

Τύπος 3

Ο χρόνος που χρειάζεται ένα παράλληλο πρόγραμμα για να ολοκλρώσει όλες τις εργασίες του

Υπάρχουν όμως και άλλες περιπτώσεις παραλληλοποίησης όπου για παράδειγμα ένας αλγόριθμος να χρησιμοποιεί αναδρομή για την εύρεση αποτελέσματος. Τότε ο πρώτος όρος του «τύπου 3» αλλάζει. Αυτή την περίπτωση θα την εξετάσουμε με τον αλγόριθμο *'Merge Sort'* στον οποίο χρησιμοποιήσαμε αναδρομή για την ταξινόμηση των ακεραίων αριθμών.

1.11 Πολυπλοκότητα ενός αλγορίθμου

Αναφέραμε προηγουμένως πως όταν αυξάνεται η πολυπλοκότητα ενός αλγορίθμου τότε είναι σχεδόν σίγουρο ότι δεν θα μπορούμε να επιτύχουμε γραμμική επιτάχυνση. Τι σημαίνει όμως ο όρος *'πολυπλοκότητα'*;

Η πολυπλοκότητα ενός αλγορίθμου, όπως μας παραπέμπει και ο ίδιος ο όρος, είναι το πόσο πολύπλοκος είναι ένας αλγόριθμος όπου πολύπλοκος εννοείται πόσες πράξεις έχει συνήθως να επιτελέσει πριν βγάλει το επιθυμητό αποτέλεσμα. Περισσότερες πράξεις σημαίνει και περισσότερες διαφορετικές μεταβλητές, και περισσότερη επικοινωνία μεταξύ των πυρήνων – διεργασιών. Η εικόνα 4 στην συνέχεια είναι ένα απλό και εντελώς τυχαίο παράδειγμα του πως μεταβάλλεται η επιτάχυνση και η εικόνα 5 αντίστοιχο παράδειγμα του πως μεταβάλλεται ο βαθμός απόδοσης όταν σε έναν αλγόριθμο πολλαπλασιασμού δισδιάστατου πίνακα με ένα διάνυσμα επιλέξαμε να μειώσουμε στο μισό το μέγεθος του πίνακα και στην συνέχεια να διπλασιάσουμε το μέγεθος του πίνακα.

1. Την πρώτη φορά χρονομετρούμε τον πολλαπλασιασμό του μισού μεγέθους δισδιάστατου πίνακα με διάνυσμα

2. Την δεύτερη φορά χρονομετρούμε τον πολλαπλασιασμό του κανονικού μεγέθους του δισδιάστατου πίνακα με ένα διάνυσμα

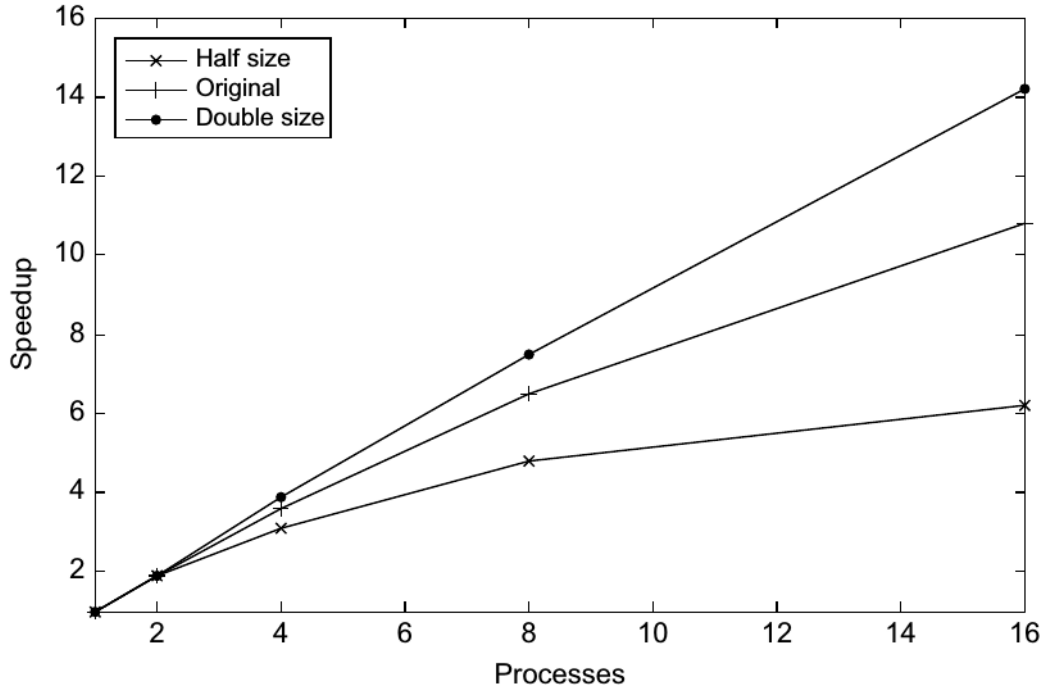
3. Την τρίτη φορά χρονομετρούμε τον πολλαπλασιασμό του διπλάσιου μεγέθους του δισδιάστατου πίνακα με ένα διάνυσμα

Έτσι ο ψευδοκώδικας που προκύπτει είναι:

```
multiplyMatrixVector(void* rank)
{
        myrank = rank;                          // Δήλωση των πυρήνων
        i, j;                                   // Δήλωση των μεταβλητών
        local m = m/thread count;               // Δήλωση του εύρους των τιμών κάθε πυρήνα
        my first row = my rank*local m;         // Δήλωση της πρώτης τιμής
        my last row = (my rank+1)*local m − 1;  // Δήλωση της τελευταίας τιμής
}
for (i = my first row; i <= my last row; i++)   /************************************/
{                                               /*                                  */
        y[i] = 0.0;                             /*      Ο πολλαπλασιασμός           */
        for (j = 0; j < n; j++)                 /*      ενός πίνακα                 */
        y[i] += A[i][j]*x[j];                   /*      με διάνυσμα                 */
}                                               /*                                  */
                                                /************************************/
```

Ψευδοκώδικας 1

Πολλαπλασιασμός πίνακα με διάνυσμα

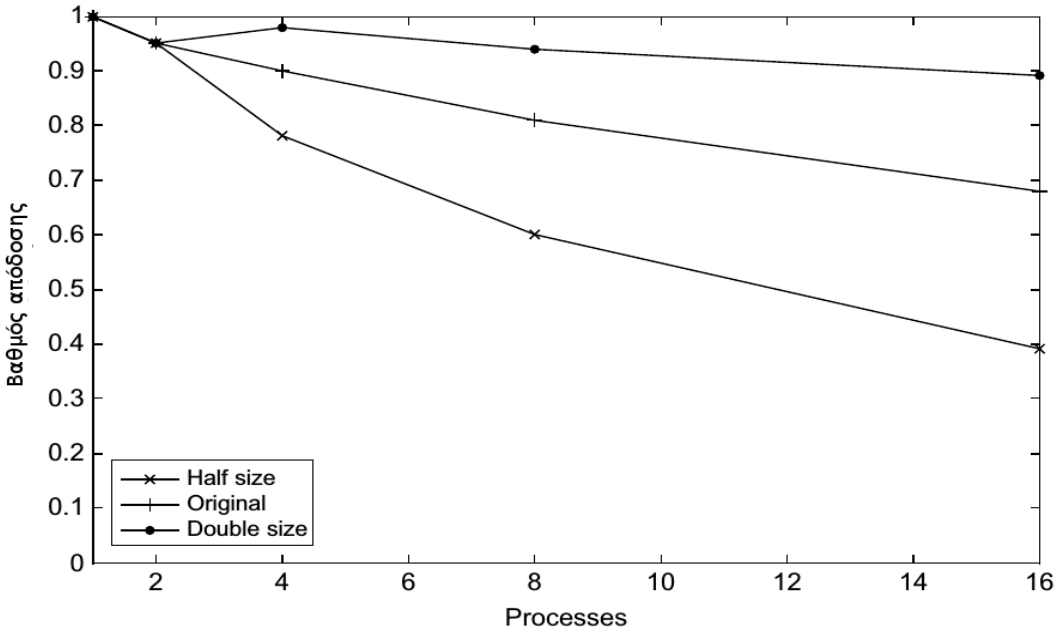


Εικόνα 4

Ρυθμός επιτάχυνσης παράλληλου προγράμματος πολλαπλασιασμού πίνακα με διάνυσμα με μέγεθος πίνακα μισό, κανονικό και διπλάσιο[1]

Όταν εννοούμε αρχικό, μισό και διπλάσιο μέγεθος του προβλήματος που παραλληλοποίησαμε εννοούμε πως θα χρονομετρήσουμε στην αρχή τον πολλαπλασιασμό του διανύσματος με τον

αρχικό πίνακα, στην συνέχεια θα χρονομετρήσουμε τον πολλαπλασιασμό του διανύσματος με το μισό μέγεθος του αρχικού πίνακα και τέλος θα χρονομετρήσουμε τον πολλαπλασιασμό του διανύσματος με το διπλάσιο μέγεθος του αρχικού πίνακα



Εικόνα 5

Βαθμός απόδοσης παράλληλου προγράμματος πολλαπλασιασμού πίνακα με διάνυσμα με μέγεθος πίνακα αρχικό, μισό και διπλάσιο[1]

1.12 Κλιμάκωση ενός παράλληλου προγράμματος

Ένας αλγόριθμος λέγεται κλιμακούμενος εάν αυξήσουμε τον όγκο των δεδομένων που εισάγουμε και ταυτόχρονα αυξήσουμε ανάλογα τους πυρήνες – διεργασίες τότε η αποδοτικότητα του προγράμματος παραμένει ίδια. Εάν δηλαδή καταφέρουμε να βρούμε έναν ρυθμό αύξησης του μεγέθους του προβλήματος έτσι ώστε αυτό να έχει συνεχώς την ίδια αποδοτικότητα τότε το πρόγραμμα το ονομάζουμε κλιμακωτό (scalable) και άρα σωστά βελτιστοποιημένο.

Ανατρέχοντας τον τύπο που υπολογίζουμε την συνολική απόδοση ενός παράλληλου προγράμματος $E = T_{serial} / T_{parallel} + T_{overhead}$ θεωρούμε πως το $T_{overhead}$ ισούται με 1, το T_{serial} ισούται με n όπου n μετριέται σε *microseconds* και ουσιαστικά είναι το μέγεθος του προβλήματος. Αντικαθιστώντας με ότι έχουμε μάθει ως τώρα έχουμε:

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

Τύπος 4

Ο τύπος της συνολικής απόδοσης ενός παράλληλου προβλήματος

Με $T_{parallel}$ = n / p + 1 η τελική μορφή της συνολικής απόδοσης είναι:

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}$$

Τύπος 5

Ανάπτυξη του τύπου συνολικής απόδοσης

Τώρα για να μπορέσουμε να υπολογίσουμε εάν ένα πρόγραμμα είναι κλιμακούμενο θα αυξήσουμε τους πυρήνες ή διεργασίες p κατά k φορές και θέλουμε να υπολογίσουμε τον παράγοντα x τον οποίο χρειαζόμαστε για να αυξήσουμε το μέγεθος του προβλήματος τόσο ώστε η αποδοτικότητα *'E'* να παραμείνει σταθερή. Έτσι λοιπόν θα έχουμε:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}$$

Τύπος 6

Προσθήκη του αριθμού k των πυρήνων / διεργασιών και του μεγέθους x

Όπως αναφέραμε, λύνοντας ως προς x θα βρούμε την τιμή εκείνη κατά την οποία αν αυξήσουμε το πρόβλημα τότε θα προκύψει ένα κλιμακούμενο πρόγραμμα (scalable). Αρχικά, αν υποθέσουμε πως το x ισούται με το k τότε θα υπάρχει ένας κοινός παράγοντας του *'k'* στον παρονομαστή και τότε μπορούμε να μειώσουμε το κλάσμα και να πάρουμε τον τύπο 7 όπως φαίνεται παρακάτω.

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}$$

Τύπος 7

Απόδειξη ότι αυξάνοντας του πυρήνες / διεργασίες και ταυτόχρονα αυξάνοντας με ίδια τιμή το μέγεθος του προβλήματος, η συνολική του απόδοση μπορεί να παραμείνει σταθερή

Μόλις αποδείξαμε πως εάν αυξήσουμε το μέγεθος του προβλήματος με τον ίδιο ρυθμό που αυξάνουμε τους πυρήνες – διεργασίες τότε η συνολική απόδοση θα παραμείνει αμετάβλητη και το πρόγραμμά μας θα είναι κλιμακούμενο.

Σε αυτό το σημείο θα πρέπει να αναφέρουμε πως ένας αλγόριθμος μπορεί να είναι κλιμακούμενος με διάφορους τρόπους. Για παράδειγμα, εάν αυξήσουμε τους πυρήνες – διεργασίες και η συνολική απόδοση του παράλληλου προγράμματος μείνει σταθερή χωρίς να αυξήσουμε το μέγεθος του προβλήματος τότε λέμε πως το πρόγραμμά μας είναι *'ισχυρά'* κλιμακούμενο *(strongly scalable)*. Από την άλλη, αν έχουμε το πρόβλημα του παραδείγματος

όπου το μέγεθός του αυξάνεται με τον ίδιο ρυθμό που αυξάνονται οι διαθέσιμες διεργασίες τότε το πρόβλημα αυτό χαρακτηρίζεται ελαφρώς κλιμακούμενο *(weakly scalable)*.

1.13 Πως θα χρονομετρούμε την ώρα εκτέλεσης του αλγορίθμου

Η χρονομέτρηση της εκτέλεσης ενός αλγορίθμου είναι μία καθοριστική διαδικασία για την σύγκριση της απόδοσης ενός σειριακού αλγορίθμου με έναν παράλληλο. Σε αυτό το σημείο πρέπει να κάνουμε αναφορά της διαδικασίας χρονομέτρησης από την οποία μπορούμε να πάρουμε πολλά συμπεράσματα για την συμπεριφορά των υλοποιήσεών μας με χρήση παράλληλου προγραμματισμού.

Φυσικά, υπάρχουν αρκετοί τρόποι για να χρονομετρήσουμε την εκτέλεση ενός προγράμματος. Θα μπορούσαμε να χρονομετρήσουμε όλο το πρόγραμμα, δηλαδή από την πρώτη γραμμή του κώδικά του έως και την τελευταία. Επίσης, θα μπορούσαμε να χρονομετρήσουμε διαφορετικά κομμάτια όπως για παράδειγμα, κάποιες μεθόδους, κάποιες συναρτήσεις και κάποιες συναρτήσεις επικοινωνίας μεταξύ διεργασιών ή πυρήνων και να κάνουμε μετά την πρόσθεση. Ακόμη, θα μπορούσαμε να χρονομετρήσουμε μόνο τα κομμάτια κώδικα τα οποία έχουμε παραλληλοποιήσει ή ακόμη και το κάθε νήμα ή την κάθε διεργασία και να επιλέξουμε τον πιο αργό χρόνο μέσα στον οποίο υποτίθεται πως θα έχουν εκτελέσει και τα υπόλοιπα νήματα/διεργασίες τις εργασίες του. Σε αυτή την συγγραφή χρησιμοποιήσαμε διάφορους τρόπους χρονομέτρησης οι οποίοι μας οδήγησαν σε πολύ χρήσιμα συμπεράσματα και τελικά καταλήξαμε σε έναν έτσι ώστε να υπάρχει ομοιομορφία στα αποτελέσματα. Πριν όμως αναφέρουμε την αιτία και τον τρόπο που χρησιμοποιήσαμε για να χρονομετρήσουμε τον χρόνο εκτέλεσης ας δούμε δύο παραδείγματα διαφορετικών χρονομετρήσεων.

Αρχικά, θεωρούμε σαν δεδομένο ότι θα χρησιμοποιήσουμε την βιβλιοθήκη *'pthreads'* και άρα θα πρέπει να συγχρονίσουμε τα νήματα ή ***'threads'***, προκειμένου να είναι αξιόπιστη η χρονομέτρηση. Αφού φτάσουν όλα τα νήματα στο ίδιο σημείο του κώδικα (το σημείο που θέλουμε να χρονομετρήσουμε) το επόμενο βήμα είναι να μετρήσουμε τον χρόνο που θα χρειαστεί κάθε νήμα να επιτελέσει τις πράξεις του. Όταν τελειώσει αυτή η διαδικασία και πάρουμε τους χρόνους του κάθε νήματος θα χρειαστεί να επιλέξουμε το πιο *'τεμπέλικο'* νήμα έτσι ώστε να το εντάξουμε στο συνολικό χρόνο εκτέλεσης και συνάμα στην συνάρτηση συνολικής αποδοσης του προγράμματος αφού όπως προαναφέραμε μέσα σε αυτόν τον χρόνο θα έχουν τελειώσει με τις δικές τους εντολές τα υπόλοιπα νήματα - διεργασίες. Η λέξη κλειδί για αυτό το σκοπό είναι η αγγλική λέξη ***'barrier'*** όπου barrier σημαίνει φράγμα. Και το φράγμα το χρησιμοποιούμε για να σταματήσουμε την ροή των threads έτσι όπως ακριβώς ένα φράγμα σταματάει την ροή ενός ποταμού μέχρι όλα τα νήματα να φτάσουν στο ίδιο σημείο. Από εκείνη την στιγμή λοιπόν θα ξεκινήσει η χρονομέτρηση του παράλληλου προγράμματός μας μέχρι και την γραμμή του κώδικα που εμείς επιθυμούμε.

Τέλος.η έννοια του thread θα αναλυθεί στην ενότητα παρουσίασης της βιβλιοθήκης 'pthreads' και γιατί οι επιστήμονες επέλεξαν το **νήμα (thread)** ως όρο.

```
/* Συγχρονισμός όλων των threads */
Barrier() // Το φράγμα το οποίο θα σταματήσει την ροή των threads από περαιτέρω ανάπτυξη
Αρχικός_χρόνος = Τωρινός_χρόνος(); //Παίρνουμε τον τωρινό χρόνο σε μία μεταβλητή
/* Κώδικας που θέλουμε να χρονομετρήσουμε*/
Τελικός_χρόνος = Τωρινός_χρόνος(); //Παίρνουμε τον τωρινό χρονο σε μία άλλη μεταβλητή
Χρόνος_εκτέλεσης_προγράμματος = Τελικός_χρονος – Αρχικός_χρόνος; // Χρόνος εκτέλεσης
```

Ψευδοκώδικας 2

Τρόπος μέτρησης χρόνου εκτέλεσης ενός τμήματος κώδικα

Στον παράνω ψευδοκώδικα χρονομετρούμε ένα κομμάτι κώδικα το οποίο είναι ίδιο σε κάθε νήμα άρα εάν έχουμε 4 νήματα θα πάρουμε 4 χρόνους για το κομμάτι τους. Εμείς θα επιλέξουμε

το πιο αργό νήμα και θα συμπεράνουμε πως μέσα σε αυτόν τον χρόνο και τα υπόλοιπα νήματα θα έχουν ολοκληρώσει τις λειτουργίες τους. Στην συνέχεια μπορούμε να χρονομετρήσουμε και άλλο κομμάτι κώδικα με τον ίδιο ακριβώς τρόπο και στο τέλος να κάνουμε την πρόσθεση ώστε να έχουμε τον συνολικό χρόνο των κομματιών που επιθυμούμε.

Ο επόμενος τρόπος είναι να μετρήσουμε τους χρόνους από την έναρξη και δημιουργία έως την λήξη των νημάτων η οποίες πραγματοποιούνται μέσα από την κύρια συνάρτηση *'main()'*.

```
begin = clock()
        for(i=0; i<NUMTHREADS; i++)
        {
                threads declaration
                threads_create()
        }
        for(i=0; i<NUMTHREADS; i++)
        {
                thread_join()
        }
end = clock()
time_spent = (double) (end-begin) / CLOCKS_PER_SEC

print_time()

return 0;
```

Ψευδοκώδικας 3

Τρόπος μέτρησης χρόνου εκτέλεσης όλης της διαδικασίας παραλληλοποίησης

Παρατηρούμε πως και οι δύο περιπτώσεις ψευδοκώδικα χρησιμοποιούν την συνάρτηση *'clock()'* και την *'MACRO'* εντολή *'CLOCKS_PER_SEC'* για την χρονομέτρηση των τμημάτων τους. Η συνάρτηση *'clock()'* επιστρέφει την τιμή από έναν μετρητή ο οποίος αυξάνεται με συγκεκριμένο ρυθμό. Αυτός ο ρυθμός εξαρτάται από την υλοποίηση της συνάρτησης και δεν είναι πάντα ο ίδιος όπως ο ρυθμός του ρολογιού του συστήματος. Επίσης, η μακροεντολή '*CLOCKS_PER_SEC'* μας αναφέρει τον ρυθμό, σε Hertz, στον οποίο αυτός ο μετρητής αυξάνεται ή με άλλα λόγια μας λέει την ανάλυση του μετρητή σε μία μονάδα χρόνου. Για παράδειγμα εάν το '*CLOCKS_PER_SEC'* είναι ρυθμισμένο στην τιμή 1000, τότε κάθε αλλαγή στην επιστρεφόμενη τιμή από την συνάρτηση *'clock()'* αναπαριστάται με 1ms. Αυτό περιορίζει την ακρίβεια στην αληθινή χρονομέτρηση.

Για να επιτύχουμε περισσότερη ακρίβεια και αληθινό χρόνο εκτέλεσης θα χρησιμοποιήσουμε την εντολή του *'linux', 'time'* με τον εξής τρόπο:

```
time ./myParQuicksort
```

1.14 Σύνοψη

Θεωρώντας πως έχουμε εξηγήσει τα σημαντικότερα κομμάτια του παζλ υλοποίησης ενός παράλληλου αλγορίθμου, ήρθε η ώρα να τα βάλουμε στην σειρά και να τα ενώσουμε μεταξύ τους ώστε το έργο να φανεί ολοκληρωμένο.

Ο ολοκληρωμένος σχεδιασμός ενός παράλληλου προγράμματος από ένα σειριακό πραγματοποιείται ακολουθώντας τα παρακάτω βήματα.

1. Διαίρεση του όγκου των δεδομένων σε ισόποσα τμήματα
2. Επιλογή του είδους της επικοινωνίας μεταξύ των πυρήνων / νημάτων ή επεξεργαστών
3. Ανάθεση αυτών των εργασιών στα threads ή στις διεργασίες – processes

Κατασκευάζοντας τους αλγορίθμους ταξινόμησης σε παράλληλα προγράμματα θα γίνει περισσότερο ορατός ο παραπάνω σχεδιασμός.

1.15 Σχετικά θέματα με τον σχεδιασμό ενός παράλληλου προγράμματος

Ό,τι έχουμε αναφέρει προηγουμένως είναι και αυτά που θα χρησιμοποιηθούν στην πορεία και στο κύριο μέρος της εργασίας. Σε αυτό το σημείο θα αναφέρουμε κάποιες έννοιες και κάποιους όρους που είτε δεν έχουν αναφερθεί καθόλου, είτε δεν έχουν περιγραφεί, είτε θα αναφέρονται κατά την υλοποίηση των αλγορίθμων όπου παρουσιάζεται η ανάγκη, έτσι ώστε οι ενδιαφερόμενοι αναγνώστες να έχουν μια πλήρη εικόνα των επιμέρους μερών του σχεδιασμού ενός παράλληλου προγράμματος.

Αυτοί οι όροι είναι:

Αρχιτεκτονική von Neumann [13]**:** Οι σύγχρονοι ηλεκτρονικοί υπολογιστές σχεδιάζονται με βάση τις αρχές που διατυπώθηκαν τη δεκαετία του 1940 από τον Τζον φον Νόιμαν στο Ινστιτούτο Προηγμένων Επιστημών στο Πανεπιστήμιο του Πρίνστον. Αυτές οι θεμελιώδεις αρχές, οι οποίες αναφέρονται παρακάτω, συνιστούν την **αρχιτεκτονική φον Νόιμαν**.

- Τα δεδομένα και οι εντολές των εκτελούμενων προγραμμάτων αποθηκεύονται σε μια μοναδική μνήμη εγγραφής-ανάγνωσης.
- Τα περιεχόμενα της μνήμης αυτής μπορούν να διευθυνσιοδοτηθούν κατά κελί, χωρίς να μας ενδιαφέρει ο τύπος των δεδομένων που περιέχεται εκεί.
- Η εκτέλεση των εντολών του προγράμματος πραγματοποιείται σειριακά (εκτός και αν υπάρχει ρητή διακλάδωση), από μια εντολή στην επόμενη.

- **'Συμφόρηση' του von Neumann** [10]**:** Ο δίαυλος μεταξύ της μνήμης του προγράμματος και της μνήμης των δεδομένων οδηγεί στην συμφόρηση του Von Neumann, δηλαδή στον περιορισμένο ρυθμό μετάδοσης των δεδομένων μεταξύ της κεντρικής μονάδας επεξεργασίας και της μνήμης.
- **Instruction-level parallelism (ILP)** [11]: είναι μία τεχνική εκτέλεσης εντολών η οποία μετράει πόσες εργασίες σε ένα υπολογιστικό σύστημα μπορούν να εκτελεστούν ταυτόχρονα.
- **Pipelining** [12]. Τεχνική παράλληλης επεξεργασίας που προσπαθεί να προσομειώσει την γραμμή παραγωγής από μία βιομηχανική μονάδα. Συγκεκριμένα βρίσκει εφαρμογή στις περιπτώσεις που ένας υπολογισμός μπορεί να χωριστεί σε μια σειρά από διαδοχικά έργα και εκμεταλλευόμαστε το γεγονός ότι οι επεξεργαστές μπορούν να εκτελούν ταυτόχρονα τα διαφορετικά έργα που προέρχονται από ανεξάρτητους υπολογισμούς του ίδιου προγράμματος.

1. Αρχιτεκτονική επεξεργαστή

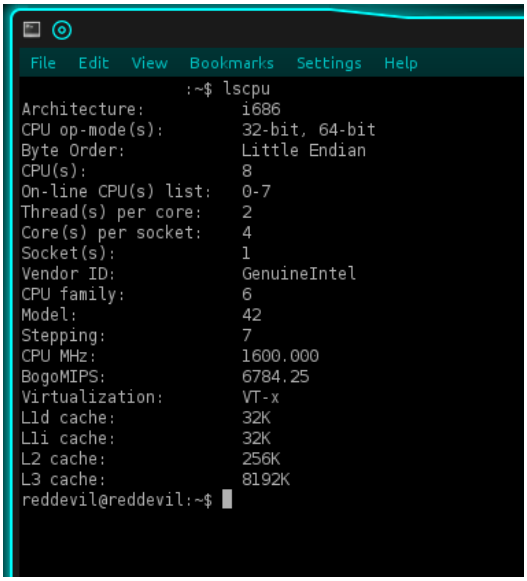
2.1 Γενικά

Το κυριότερο μέρος της αναφοράς μας έως τώρα συμπεριλάμβανε το λογισμικό που θα χρησιμοποιήσουμε για την υλοποίηση των παράλληλων αλγορίθμων. Θα πρέπει όμως να κάνουμε μια αναφορά και στο υλικό (hardware) του οποίου οι δυνατότητες είναι πεπερασμένες και ουσιαστικά ορίζουν τα όρια μέσα στα οποία θα κινηθούμε χτίζοντας της εφαρμογές μας.

2.2 Στοιχεία επεξεργαστή και λειτουργικού συστήματος

Ο επεξεργαστής πάνω στον οποίο έγιναν όλες οι μετρήσεις και οι εκτελέσεις των προγραμμάτων είναι ο ***intel i7 2600k*** και το λειτουργικό σύστημα που τρέχει είναι το ***linux*** με την έκδοση ***ubuntu 14.04***.

Ανοίγοντας το τερματικό **'terminal'** και δίνοντας σε αυτό την εντολή ***'lscpu'*** παρατηρούμε τα εξής στοιχεία από τα οποία αποτελείται ο επεξεργαστής μας.



Εικόνα 6

Χαρακτηριστικά επεξεργαστή πάνω στον οποίο υλοποιήθηκαν οι αλγόριθμοι

Ακόμη πιο αναλυτικά μπορούμε να παραθέσουμε τα στοιχεία που διέπουν κάθε core ξεχωριστά και τα οποία είναι:

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 42
model name : Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
stepping : 7
microcode : 0x1a
cpu MHz : 1600.000

cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx rdtscp lm constant_tsc arch_perfmon pebs bts xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 popcnt tsc_deadline_timer aes xsave avx lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid
bogomips : 6784.78
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management: 1600.00

Αυτές οι πληροφορίες του πρώτου πυρήνα είναι ίδιες και για τους υπόλοιπους 7.

2. Προγραμματισμός με MPI

3.1 Γενικά

Έχει ήδη αναφερθεί στο πρώτο κεφάλαιο και πιο συγκεκριμένα στην εισαγωγή πως τα υπολογιστικά συστήματα που χρησιμοποιούν παράλληλο προγραμματισμό χωρίζονται σε αυτά που χρησιμοποιούν μία ενιαία κοινόχρηστη μνήμη (shared-memory systems) και σε αυτά που κάθε επεξεργαστής – πυρήνας έχει την δική του ιδιωτική μνήμη (distributed-memory systems). Από την οπτική γωνία του προγραμματιστή μπορούμε να πούμε πως ένα σύστημα με κατανεμημένη μνήμη συνήθως αποτελείται από ένα δίκτυο υπολογιστών.

Σε αυτό το κεφάλαιο θα δούμε πως ένα σύστημα κατανεμημένης μνήμης χρησιμοποιεί την τεχνική του περάσματος ενός μηνύματος από μία διεργασία σε μία άλλη ή σε όλες τις υπόλοιπες. Αυτή η τεχνική εάν μεταφραστεί μας δίνει την ονομασία της βιβλιοθήκης *‘MPI’* δηλαδή *‘Message-Passing Interface’*.

Περιληπτικά, αυτή η διαδικασία περάσματος του μηνήματος από διεργασία σε διεργασία μεταξύ διαφορετικών επεξεργαστών γίνεται είτε με την χρήση δύο συναρτήσεων *‘Send()’* και *‘Receive()’* όπου *‘Send()’* χρησιμοποιεί ο επεξεργαστής ο οποίος θέλει να

αποστείλει την μεταβλητή της ή τα δεδομένα της σε κάποιον άλλο επεξεργαστή ο οποίος με την σειρά του χρησιμοποιεί την συνάρτηση *'Receive()'*. Επίσης, θα μάθουμε πως και με ποιες συναρτήσεις μας δίνεται η δυνατότητα να πετυχαίνουμε την επικοινωνία – αυτή την φορά - μεταξύ μίας διεργασιας και όλων των υπολοίπων. Αυτού του τύπου η μαζική επικοινωνία ονομάζεται ομαδική επικοινωνία ***(collective communication)***.

3.2 Τύποι επικοινωνίας

Στην βιβλιοθήκη *MPI* υπάρχουν 2 τύποι επικοινωνίας από επεξεργαστή σε επεξεργαστή. Η από σημείο σε σημείο επικοινωνία ή '*Point to point communication'* η οποία πραγματοποιείται αποκλειστικά από έναν επεξεργαστή σε έναν άλλο κατ' αντιστοιχία με τις συναρτήσεις *MPI_Send()* και *MPI_Recv()* και η ομαδική επικοινωνία '*Collective communication'* στην οποία όλοι οι επεξεργαστές που την χρησιμοποιούν μπορούν να διαμοιράσουν μεταξύ τους τα αποτελέσματα χρησιμοποιώντας τις βέλτιστες δυνατές τεχνικές και δομές επικοινωνίας όπως δομής πεταλούδας και δενδρικής δομής με αποτέλεσμα να επιτυγχάνονται αναλογικά πολύ καλύτεροι χρόνοι εκτέλεσης του προγράμματος.

3.3 Συνοψη των εντολών της MPI

Καθώς θα αναλύουμε τις συναρτήσεις της *'MPI'* και κυρίως τα ορίσματά της θα παρατηρήσουμε πως στο τέλος κάθε ορίσματος έχουμε την ετικέτα *'In'* ή *'Out'.* Η σημασία των δύο ετικετών έχει να κάνει με το όρισμα το οποίο φέρει την πληροφορία η οποία προορίζεται για αποστολή (Output) ή για λήψη (Input).

3.3.1 Point-to-point επικοινωνία

- **MPI_Send**

  Σύνταξη: int MPI_Send(

  ```
  void* msg_buf_p          /*In*/
  int msg_size             /*In*/
  MPI_Datatype msg_type/*In*/
  int dest                 /*In*/
  int tag                  /*In*/
  MPI_Comm communicator/*In*/   );
  ```

Στην παραπάνω σύνταξη της συνάρτησης **MPI_Send** το πρώτο όρισμα είναι δείκτης στην περιοχή μνήμης που είναι αποθηκευμενα τα δεδομένα που θα αποσταλλούν ενώ τα επόμενα δύο ορίσματα (msg_size, msg_type) περιγράφουν τα δεδομένα που θα αποσταλούν, δηλ το συνολικό μέγεθος στην μνήμη που θα δεσμευτεί, ο όγκος δεδομένων προς αποστολή σε bytes καθώς και ο τύπος (datatype) του μηνύματος αντίστοιχα. Τα 3 τελευταία ορίσματα (dest, tag, communicator) υποδηλώνουν αντίστοιχα την διεργασία για την οποία προορίζονται τα δεδομένα, μία ετικέτα για να διαχωρίζει την χρήση των απεσταλμένων δεδομένων καθώς και την ομάδα διεργασιών που επικοινωνουν μεταξύ τους. Τέλος, το In φανερώνει είσοδο στην συνάρτηση ενώ το Out έξοδο από την συνάρτηση δεδομένων.

- **MPI_Recv:**

  Σύνταξη: int MPI_Recv (

  ```
  void* msg_buf_p          /*Out*/
  int buf_size             /*In*/
  MPI_Datatype buf_type    /*In*/
  int source               /*In*/
  ```

```
int tag                 /*In*/
MPI_Comm communicator/*In*/
MPI_Status* status_p    /*In*/  );
```

Όπως και στην '*MPI*'_Send έτσι κι εδώ τα πρώτα 6 ορίσματα κάνουν την ίδια εργασία δηλαδή·

msg_buf_p δείκτης στην περιοχή μνήμης που είναι αποθηκευμενα τα δεδομένα που θα ληφθούν.

buf_size δηλώνει τον όγκο των δεδομένων που θα λειφθούν.

buf_type δηλώνει τον τύπο του μηνύματος

source δηλώνει την διεργασία από την οποία το μήνυμα πρέπει να ληφθεί

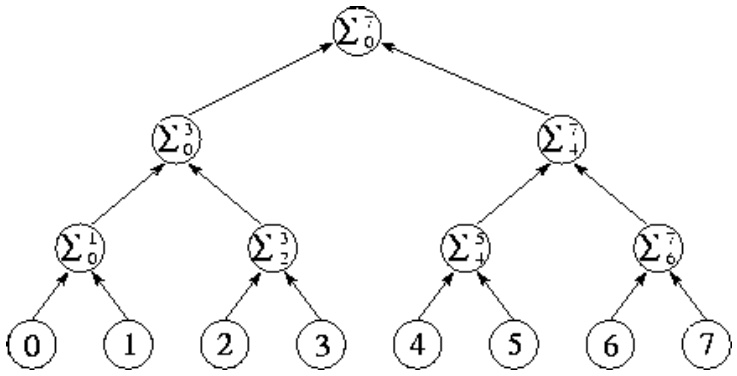
tag ο ακέραιος ο οποίος μαρτυράει την χρήση για την οποία προορίζονται τα δεδομένα και,

MPI_Comm η ομάδα διεργασιών που επικοιωνουν μεταξύ τους.

Η *'MPI_Recv'* σε αντίθεση με την '*MPI_Send*' έχει 7 ορίσματα. Το 7ο όρισμα είναι το *'MP*I_Status' το οποίο μας δίνει πληροφορίες για την συνάρτηση *'receive'* που μόλις ολοκληρώθηκε. Επίσης, η *'MPI_Status'* μπορεί να επιστρέψει ένα μήνυμα λάθους μέσω και ενός τρίτου ορίσματος *'MPI_Error'* που φέρει μέσα της.

3.3.2 Collective επικοινωνία

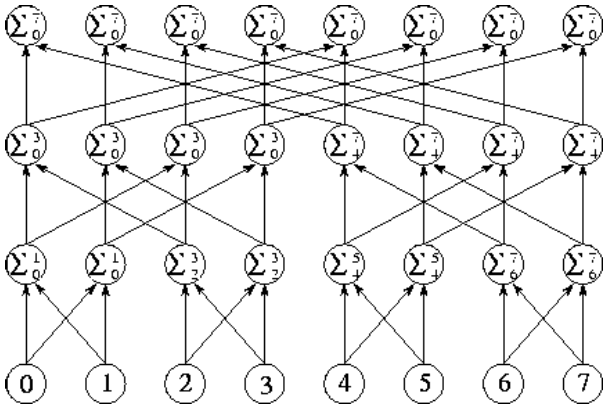
Ένα συχνό πρόβλημα που αντιμετωπίζουμε στον προγραμματισμό με χρήση επικοινωνίας από σημείο σε σημείο είναι ο φόρτος των πράξεων ο οποίος δεν κατανέμεται ισόποσα σε όλες τις διεργασίες και τους επεξεργαστές με αποτέλεσμα να "δουλεύουν" περισσότερο από κάποιους άλλους. Επειδή κάποια μοτίβα επικοινωνίας μεταξύ των διεργασιών συμβαίνουν συχνά, οι σχεδιαστές του *'MPI'* όρισαν συναρτήσεις που υλοποιούν αυτά τα μοτίβα.



Εικόνα 7

Επικοινωνία δενδρικής δομής

Σε αυτό το είδος επικοινωνίας, οι μισές διεργασίες στέλνουν τις τιμές τους στις υπόλοιπες μισές διεργασίες και αυτές με την σειρά τους, αφού εκτελέσουν τις πράξεις που τους αντιστοιχούν, επαναλαμβάνουν την αποστολή των τιμών τους μέχρι όλες να καταλήξουν στην κύρια διεργασία.



Εικόνα 8

Επικοινωνία δομής πεταλούδας

Η δομή επικοινωνίας 'πεταλούδα' είναι ένα πιο εξελιγμένο είδος επικοινωνίας κατά το οποίο οι διεργασίες μπορούν κι ανταλλάσουν μεταξύ τους τα μερικά αποτελέσματά τους.

Αναλυτικότερα οι συναρτήσεις οι οποίες χρησιμοποιούν αυτούς τους δύο τύπους επικοινωνίας είναι οι εξής:

- **MPI_Reduce:** Η πρώτη συνάρτηση που συναντούμε κι η οποία χρησιμοποιεί την δενδρική δομή επικοινωνίας είναι η *'MPI_Reduce()'*. Σκοπός της είναι να συγχωνεύσει τα αποτελέσματα σε ένα και να το αποθηκεύσει σε μία συγκεκριμένη διεργασία για περεταίρω χρήση. Χρησιμοποιώντας αυτή τη συνάρτηση επιτυγχάνουμε να ελατώσσουμε τον χρόνο εκτέλεσης του προγράμματος πάνω από το μισό διότι μειώνεται σημαντικά το πλήθος των πράξεων στην Κύρια διεργασία αλλά και οι μεταφορές των τιμών από τις άλλες διεργασίες στην Κύρια διεργασία.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Reduce (
        void* input_data_p      /*In*/
        void* output_data_p     /*Out*/
        int count               /*In*/
        MPI_Datatype datatype   /*In*/
        MPI_Op operator         /*In*/
        int dest_process        /*In*/
        MPI_Comm comm           /*In*/  );
```

Σε αυτή την συνάρτηση και στο πέμπτο όρισμά της συναντάμε ένα νέο τύπο. Το *'MPI_Op'* είναι ένας προκαθορισμένος τύπος όπως το *'MPI_Comm'* ή το *'MPI_Datatype'* και αναλόγως την λειτουργία που θέλουμε η *'MPI_Reduce()'* να επιτελέσει καλούμε και τον αντίστοιχο τύπο της. Έτσι αν θέλουμε να περάσουμε το άθροισμα των τιμών από μία διεργασία στην διεργασία 0 παραδείγματος χάριν καλούμε το *'MPI_SUM'* κι αυτό αναλαμβάνει να κάνει όλες τις πράξεις που χρειάζονται κ.ο.κ. Τα υπόλοιπα κατά σειρά ορίσματα διενεργούν τις εξής λειτουργίες.

void* input_data_p: Περιοχή μνήμης η οποία λαμβάνει τα προς επεξεργασία δεδομένα

void* output_data_p: Περιοχή μνήμης η οποία περιέχει τα επεξεργασμένα δεδομένα int count: Δηλώνει το πλήθος των δεδομένων

MPI_Datatype datatype: Δηλώνει τον τύπο των δεδομένων

int dest_process: Δηλώνει την διεργασία προορισμού των δεδομένων

MPI_Comm comm: Ονοματίζει τον δίαυλο επικοινωνίας

- **MPI_Allreduce:** Η MPI_Allreduce χρησιμοποιεί τη **Butterfly επικοινωνία** και όπως θα παρατηρήσουμε και στην συνέχεια η ειδοποιός διαφορά της σύνταξής της με αυτής της MPI_Reduce εντοπίζεται στο **dest_process** το οποίο στην MPI_Allreduce συνάρτηση δεν υπάρχει. Άρα, μόνο από αυτό το στοιχείο αντιλαμβανόμαστε πως όταν κληθεί η MPI_Allreduce, θα λάβει ένα σύνολο δεδομένων – τιμών, στη συνέχεια θα εκτελέσει την λειτουργία την οποία θα της έχουμε προσδιορίσει και τέλος θα κάνει το αποτέλεσμα ορατό και διαθέσιμο σε όλες τις διεργασίες μίας ομάδας επικοινωνίας και όχι μόνο σε μία (αυτή που ορίζει η MPI_Reduce).

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Allreduce (
        void* input_data_p      /*In*/
        void* output_data_p     /*Out*/
        int count               /*In*/
        MPI_Datatype datatype   /*In*/
        MPI_Op operator         /*In*/
        MPI_Comm comm           /*In*/  );
```

- **MPI_Bcast (Broadcast):** Η συνάρτηση MPI_Bcast έχει δημιουργηθεί για να κάνει ορατές τις τιμές που επιλέγει ο χρήστης από την Κύρια Διεργασία στις υπόλοιπες διεργασίες και ουσιαστικά να απαλείψει τις πολύ χρονοβόρες επαναλήψεις των MPI_Recv.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Bcast (
        void* input_data_p      /*In*/
        int count               /*In*/
        MPI_Datatype datatype   /*In*/
        int source_proc         /*In*/
        MPI_Comm comm           /*In*/  );
```

και το είδος επικοινωνίας στο οποίο ανήκει είναι η “Tree-structured”.

Ένα σημαντικό χαρακτηριστικό της *‘MPI_Bcast’* το οποίο την διαφοροποιεί από άλλες συναρτήσεις εξόδου δεδομένων είναι πως τα αποτελέσματα έστω της Κύριας διεργασίας ή διεργασίας 0 είναι όρατα και αποστέλλονται σε **ΟΛΕΣ** τις διεργασίες της ίδιας ομάδας. Όμως, τις περισσότερες φορές, εμείς χρειαζόμαστε κάποια δεδομένα να καταλήγουν σε συγκεκριμένες διεργασίες κάτι το οποίο δεν έχει δυνατότητα να κάνει η *‘MPI_Bcast()’* αλλά η επόμενη συνάρτηση που θα παρουσιάσουμε.

- **MPI_Scatter:** Η συνάρτηση *‘MPI_Scatter()’* επιλύει το πρόβλημα που μόλις αναφέραμε παραπάνω. Αυτή η συνάρτηση μπορεί και μοιράζει τα δεδομένα από μία διεργασία σε όλες τις υπόλοιπες διεργασίες που ανήκουν στην ίδια ομάδα.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Scatter (
        void* send_buf_p        /*In*/
        int send_count          /*In*/
        MPI_Datatype send_type  /*In*/
```

```
    void* recv_buf_p            /*Out*/
    int recv_count              /*In*/
    MPI_Datatype recv_type      /*In*/
    int src_proc                /*In*/
    MPI_Comm comm               /*In*/  );
```

Τα πρώτα 3 ορίσματα αντιστοίχως και σε άλλες Collective συναρτήσεις δηλώνουν την περιοχή μνήμης που δεσμεύονται τα δεδομένα, το μέγεθος και το είδος των δεδομένων που πρόκειται να σταλλούν. Τα επόμενα 3, δηλώνουν την περιοχή της μνήμης που πρέπει να δεσμεύσει κάθε διεργασία καθώς το είδος και το μέγεθος των δεδομένων που θα ληφθούν. Το 7ο όρισμα δηλώνει από ποια διεργασία "φεύγουν τα δεδομένα" (συνήθως αυτή είναι η διεργασία 0) και το 8ο όρισμα ορίζει τον δίαυλο – είδος επικοινωνίας που θα ανοιχτεί.

- **MPI_Gather: Η** MPI_Gather είναι η αντίστροφη διαδικασία συλλογής των διασκορπισμένων δεδομένων που έχουν προέλθει από την συνάρτηση *'MPI_Scatter'*.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Gather (
    void* send_buf_p        /*In*/
    int send_count          /*In*/
    MPI_Datatype send_type/*In*/
    void* recv_buf_p        /*Out*/
    int recv_count          /*In*/
    MPI_Datatype recv_type/*In*/
    int dest_proc           /*In*/
    MPI_Comm comm           /*In*/  );
```

Σε αυτό το σημείο πρέπει να επισημάνουμε ότι υπάρχει ένας περιορισμός για την *'MPI_Gather'* ο οποίος είναι ακριβώς ίδιος και για την *'MPI_Scatter'*. Τα δεδομένα που θα ληφθούν ή διαμοιραστούν μεταξύ των διεργασιών θα πρέπει να είναι ακριβώς ισόποσα σε κάθε διεργασία

- **MPI_Scatterv:** Επεκτείνει την λειτουργικότητα της συνάρτησης *'MPI_Scatter'* επιτρέποντας να σταλεί ένας μεταβαλλόμενος αριθμός δεδομένων σε κάθε διεργασία αφού πλέον το όρισμα *'recv_count'* είναι πίνακας. Επιπλέον προσδίδει περισσότερη ευελιξία για το από που θα παρθούν τα δεδομένα από τις διεργασίες στις οποίες προβλέπεται να διαμοιραστούν.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Scatterv(
    void * send_buf_p,      /*In*/
    int *send_count,        /*In*/
    int * displs,           /*In*/
    MPI_Datatype sendtype, /*In*/
    void *  recv_buf_p,     /*Out*/
    int recv_count,         /*In*/
    MPI_Datatype recv_type,/*In*/
    int root,               /*In*/
    MPI_Comm comm           /*In*/);
```

- **MPI_Gatherv:** Επεκτείνει την λειτουργικοτητα της συνάρτησης *'MPI_Gather'*

επιτρέποντας δεχτεί έναν μεταβαλλόμενο αριθμό δεδομένων από κάθε διεργασία αφού πλέον το όρισμα *'recv_count'* είναι πίνακας. Επιλέον δίνει περισσότερη ευελιξία για το που θα αποθηκευτούν τα δεδομένα στην διεργασία παρέχοντας το νέο όρισμα *'displs'*.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Gatherv (
        void * send_buf_p,        /*In*/
        int send_count,           /*In*/
        MPI_Datatype sendtype, /*In*/
        void * recv_buf_p,        /*Out*/
        int * recv_counts,                /*In*/
        int * displs,             /*In*/
        MPI_Datatype recv_type,/*In*/
        int root,                 /*In*/
        MPI_Comm comm         /*In*/ );
```

- **MPI_Allgather:** Χρησιμοποιώντας την συνάρτηση *'MPI_Gather'* πετυχαίνουμε να συγκεντρωθούν μαζί τα δεδομένα μίας ομάδας διεργασιών σε μία. Κάποιες φορές όμως αυτό δεν είναι αρκετό και πρέπει όλα τα δεδομένα όλων των διεργασιών να συγκεντρωθούν σε όλες τις διεργασίες. Αυτή την λειτουργία επιτυγχάνει να εκτελέσει η συνάρτηση *'MPI_Allgather'.*

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Allgather (
        void* send_buf_p        /*In*/
        int send_count          /*In*/
        MPI_Datatype send_type/*In*/
        void* recv_buf_p        /*Out*/
        int recv_count          /*In*/
        MPI_Datatype recv_type/*In*/
        MPI_Comm comm       /*In*/  );
```

- **MPI_Alltoall:** Μία από τις σπουδαιότερες συναρτήσεις που μας προσφέρει η MPI βιβλιοθήκη είναι η *'MPI_Alltoall'***.** Σύμφωνα με αυτήν, όλες οι διεργασίες μπορούν να στείλουν διαφορετικά δεδομένα σε κάθε άλλη διεργασία. Αν και εκ πρώτης όψεως φαίνεται πως η λειτουργία της είναι ίδια με αυτή της προηγούμενης συνάρτησης, τελικά δεν είναι. Η διαφορά εντοπίζεται στην επιπλέον λειτουργία που έχει η συνάρτηση *'MPI_Alltoall'* αυτή του διαμοιρασμού όλων των δεδομένων. Για να είμαστε περισσότερο κατανοητοί η *'MPI_Alltoall'* κάνει έναν συνδυασμό λειτουργιών *'MPI_Scatter'* και *'MPI_Gather'*.

operation	send buf size	recv buf size
MPI_Allgather	sendcnt	n_procs * sendcnt
MPI_Alltoall	n_procs * sendcnt	n_procs * sendcnt

Σχήμα 1

Διαφορά MPI_Allgather με MPI_Alltoall

Η συνταξη της συνάρτησης είναι η εξής:

```
int MPI_Alltoall(
```

```
void* send_buf_p        /*In*/
int send_count          /*In*/
MPI_Datatype send_type /*In*/
void* recv_buf_p        /*Out*/
int recv_count,         /*In*/
MPI_Datatype recv_type,/*In*/
MPI_Comm comm           /*In*/ );
```

- **MPI_Barrier:** Όταν μία διεργασία συναντήσει αυτή τη συνάρτηση στο πρόγραμμά μας τότε θα σταματήσει την εκτέλεσή της έως ότου όλες οι υπόλοιπες φτάσουν στο ίδιο σημείο. Ο κυριότερος λόγος ύπαρξης της *'MPI_Barrier'* στην βιβλιοθήκη της *'MPI'* είναι ο συγχρονισμός των διεργασιών. Κάποιες φορές πρέπει να είμαστε σίγουροι ότι όλες οι διεργασίες έχουν ολοκληρώσει μία φάση πριν περάσουν στην επόμενη ώστε να αποφύγουμε λανθασμένες μεταβολές της τιμής κάποιων ή κάποιας μεταβλητής. Επίσης, ένα άλλο είδος χρήσης ο οποίος αφορά εμάς τους προγραμματιστές, είναι να μπορούμε να παίρνουμε τον σωστό χρόνο εκτέλεσης του παράλληλου κομματιού της εφαρμογής που χρειαζόμαστε. Αυτό γίνεται διότι, όλες οι διεργασίες φτάνουν στο ίδιο σημείο άρα κι όλες ξεκινούν τον υπολογισμό των δεδομένων τους περίπου το ίδιο χρονικό σημείο. Αυτό το χρονικό σημείο καταγράφεται με ειδικές συναρτήσεις που περιέχονται μέσα στην C και αφαιρείται από τον χρόνο που θα κάνει η πιο αργή διεργασία να τελειώσει τις πράξεις της. Άλλες σημαντικές χρήσεις της MPI_Barrier και γενικότερα της Barrier θα μελετηθουν στον πολυνηματικό προγραμματισμό.

Η σύνταξη της συνάρτησης είναι η εξής:

```
int MPI_Barrier(MPI_Comm comm );
```

3.4 Διαφορές Point to point επικοινωνίας με Collective επικοινωνία

1. Όλες οι συναρτήσεις που επικοινωνούν πρέπει να είναι του ίδιου τύπου. Για παράδειγμα η συνάρτηση *'MPI_Recv()'* δεν μπορεί να λάβει τα δεδομένα της συνάρτησης *'MPI_Reduce()'*. Εάν συμβεί αυτό τότε το πρόγραμμα θα υποπέσει σε σφάλμα.
2. Σε όλες τις *'Collective'* επικοινωνίες τα ορίσματα πρέπει να είναι του ίδιου τύπου. Δηλαδή δεν μπορεί μία διεργασία να περάσει το 0 σαν την dest_process και άλλη διεργασία να περάσει το 1. Και σε αυτό το παράδειγμα το πρόγραμμα θα υποπέσει σε σφάλμα και μπορεί να "κρεμάσει".
3. Το όρισμα *'output_data_p'*, όταν εκτελεστεί μία διεργασία, θα χρησιμοποιηθεί μόνο από την *'dest_process'*. Εντούτοις, όλες οι διεργασίες πρέπει να περάσουν κάποιο όρισμα σύμφωνα με το *'output_data_p'*, ακόμη κι αν αυτό είναι το κενό (NULL).
4. Ενώ οι 'από σημείο σε σημείο' επικοινωνίες βασίζονται στο όρισμα *'tag'* και στον δίαυλο επικοινωνίας, οι *'collective'* επικοινωνίες βασίζονται μόνο στον δίαυλο επικοινωνίας.

4. Προγραμματισμός με pthreads

4.1 Γενικά

Η δεύτερη κατά σειρά 'γλώσσα προγραμματισμού' με την οποία θα ασχοληθούμε είναι η **pthreads**. Η pthreads ή αλλιώς POSIX threads όπως και η mpi δεν είναι καθε αυτού γλώσσα προγραμματισμού αλλά μία 'standard' βιβλιοθήκη της C για τύπου UNIX λειτουργικά συστήματα.

Όπως αντιλαμβανόμαστε και από την ονομασία της, βασικό συστατικό είναι το **thread** που, όπως και η **process** για την mpi, μας χαρίζει την δυνατότητα της παραλληλίας. Για την ακρίβεια ένα νήμα είναι ελαφρώς 'ελαφρύτερο' από μία διεργασία λόγω των υπολογιστικών πόρων που οι διεργασίες χρειάζονται να δεσμεύσουν για την μεταξύ τους επικοινωνια (**overhead**).

Thread (νήμα) ή **thread of control** είναι μία ακολουθία καταστάσεων ή αλλιώς μία ανεξάρτητη ροή εντολών η οποία έχει προγραμματιστεί να τρέχει μέσα σε μία διεργασία και την οποία διαχειρίζεται το λειτουργικό σύστημα.

Αντίθετα από την *'mpi'*, η βιβλίοθήκη *'pthreads'* χρησιμοποιεί την μνήμη του συστήματος για την δημιουργία των νημάτων, την αποθήκευση και ενημέρωση των μεταβλητών. Αυτός είναι ο λόγος που τα προγράμματα τα οποία χρησιμοποιούν αυτή τη βιβλιοθήκη κατατάσσονται στα **Shared Memory Programs**.

4.2 Ανάλυση της βιβλιοθήκης

Το πρώτο πράγμα που πρέπει να μας έρχεται στο μυαλό ακούγοντας ή ξεκινώντας να υλοποιήσουμε κάποιο πρόγραμμα με την *'pthreads'* βιβλιοθήκη είναι ο τρόπος με τον οποίο δημιουργούνται τα νήματα από την κύρια *(main)* συνάρτηση. Αυτό γιατί, εν αντιθέσει με την '*mpi*', δεν χρειάζεται να ορίζουμε συγκεκριμένο τμήμα μέσα στον κώδικα ο οποίος θα αναγράφει την εκκίνηση της διαδικασίας παραλληλοποίησης του προγράμματος. Η κύρια (main) συνάρτηση του προγράμματος δημιουργεί τα νήματα με την συνάρτηση *'pthread_create()'* – την οποία θα εξηγήσουμε στην συνέχεια – και αυτά στην συνέχεια θα εκτελέσουν τις εντολές τους παράλληλα.

Ξεκινώντας από την προηγούμενη παραδοχή της δημιουργίας των νημάτων, βρισκόμαστε τώρα στην θέση να αναλύσουμε και τις συναρτήσεις από τις οποίες αποτελείται η βιβλιοθήκη '*pthreads*'. Αυτές χωρίζονται σε 3 τμήματα.

1. **Διαχείριση νημάτων (Thread Management):** Αποτελείται από τις συναρτήσεις και τις μεταβλητές αυτών οι οποίες επηρεάζουν άμεσα τα threads κάνοντας εργασίες όπως δημιουργία, σύνδεση, αποσύνδεση.

2. **Κρίσιμες περιοχές (Critical Sections):** Αποτελείται από 3 τεχνικές οι οποίες επιλύουν το σημαντικότατο πρόβλημα που δημιουργείται στον προγραμματισμό πάνω σε σύστημα διαμοιρασμένης μνήμης, δηλαδή αυτό της προσπάθειας να χρησιμοποιηθεί μία κοινή για όλους μεταβλητή σε παραπάνω από ένα νήμα. Πιο συγκεκριμένα αυτές είναι η **busy-waiting** (η οποία είναι μία τεχνική που την χρησιμοποιούμε γενικά στον προγραμματισμό)**,** η χρήση του **αμοιβαίου αποκλεισμού** (**mutex)** και η χρήση των **σημαφόρων (semaphores)**.

3. **Συγχρονισμός (Synchronization):** Οι συναρτήσεις εκείνες οι οποίες διαχειρίζονται τον συγχρονισμό των threads έτσι ώστε να μπορούν όλα να βρίσκονται στην ίδια κατάσταση και στο ίδιο σημείο του προγράμματος όταν αυτό χρειάζεται.

4.2.1 Διαχείριση νημάτων (Thread management)

Προϋπόθεση για την διαχείριση των νημάτων είναι η δημιουργία και η ύπαρξή τους. Έτσι, η βιβλιοθήκη *'pthreads'* μας εξοπλίζει με συναρτήσεις υπεύθυνες για την δημιουργία αυτών αλλά και την διαχείρισή τους.

- **pthread_create (thread, attr, start_routine, arg)**

Σύνοψη συνάρτησης: int pthread_create (pthread_t *restrict thread,
const pthread_attr_t *restrict attr,
void *(*start_routine)(void*),
void *restrict arg);

Περιγραφή: Η **pthread_create** όπως φανερώνει και το όνομά της είναι υπεύθυνη για την δημιουργία των επιμέρους νημάτων εντός μίας διεργασίας. Κάθε νήμα που δημιουργείται έχει τα δικά του χαρακτηριστικά (attributes) τα οποία καθορίζονται από το δεύτερο όρισμα της συνάρτησης (attr). Το τρίτο όρισμα εκκινεί την ρουτίνα – συνάρτηση ενώ το τέταρτο όρισμα περνάει στην συνάρτηση ένα όρισμα. Εξαιτίας της μορφής της συνάρτησης 'pthread_create' μόνο ένα όρισμα μπορεί να περάσει μέσα από αυτήν. Όμως τις περισσότερες φορές χρειάζεται να περάσουμε παραπάνω από ένα. Η τεχνική που χρησιμοποιούμε για την αποστολή 2 ή περισσοτέρων ορισμάτων από την συνάρτηση *'pthread_create'* σε κάποια άλλη συνάρτηση είναι η ομαδοποίηση των ορισμάτων μέσω της δομής *'struct'* της *'C'*.

- **pthread_attr_init (attr)**

Σύνοψη συνάρτησης: int pthread_attr_init(pthread_attr_t *attr);

Περιγραφή: Η συνάρτηση *'pthread_attr_init()'* αρχικοποιεί την δομή που περιέχουν τα χαρακτηριστικά των νημάτων τα οποία προκειται να δημιουργηθούν κατά την εκτέλεση της εφαρμογής.

Οι δύο προηγούμενες συναρτήσεις είναι υπεύθυνες για την δημιουργία και την αρχικοποίηση των νημάτων που επιθυμούμε να χρησιμοποιήσουμε στην πορεία του προγράμματός μας. Όλες οι επόμενες αφορούν στην διαχείριση των νημάτων και είναι οι εξής:

- **pthread_exit (status)**

Σύνοψη συνάρτησης: void pthread_exit (void *value_ptr);

Περιγραφή: Η '*pthread_exit* τερματίζει το καλούμενο νήμα.

- **pthread_cancel (thread)**

Σύνοψη συνάρτησης: int pthread_cancel (pthread_t thread);

Περιγραφή: Αυτή η συνάρτηση ζητάει από το νήμα, έστω με όνομα *thread,* να ακυρωθεί. Η κατάσταση και ο τύπος ακύρωσης του νήματος προσδιορίζουν το πότε θα λάβει χώρα η ακύρωση.

- **pthread_attr_destroy (attr)**

Σύνοψη συνάρτησης: int pthread_attr_destroy(pthread_attr_t *attr);

Περιγραφή: Όπως μας παραπέμπει και το όνομα της συνάρτησης, η *'pthread_attr_destroy()'* πρόκειται να καταστρέψει ένα χαρακτηριστικό αντικείμενο το οποίο έχει δημιουργηθεί κατά την εφαρμογή του προγράμματός μας. Όταν αυτό συμβεί τότε τα χαρακτηριστικά αντικείμενα ενός νήματος μπορούν να ξανα-αρχικοποιηθούν με την *'pthread_attr_init'***.**

- **pthread_join (threadid, status)**

Σύνοψη συνάρτησης: int pthread_join(pthread_t thread, void **value_ptr);

Περιγραφή: Η συνάρτηση *'pthread_join()'* αναστέλλει το νήμα που την καλεί μέχρι να ολοκληρώσει την λειτουργία του το νήμα με αναγνωριστικό *'threadid'*.

- **pthread_detach (threadid)**

Σύνοψη συνάρτησης: int pthread_detach (pthread_t thread);

Περιγραφή: Η συνάρτηση *'pthread_deach()* κάνει ανεξάρτητο ένα νήμα και δεν χρειάζεται η διεργασία που το δημιούργησε να περιμένει να ολοκληρωθεί κάνοντας join().

- **pthread_attr_get/setdetatchstate (attr, detachstate)**

Σύνοψη συναρτήσεων: int pthread_attr_getdetachstate (const pthread_attr_t *attr,
int *detatchstate);
int pthread_attr_setdetachstate (pthread_attr_t *attr,
int detatchstate);

Περιγραφή: Αρχικά να τονίσουμε ότι το χαρακτηριστικό detachstate ελέγχει εάν το **thread** έχει δημιουργηθεί σε κατάσταση αποσύνδεσης η όχι. Αν συμβαίνει αυτό, τότε δεν μπορεί να χρησιμοποιήθεί το αναγνωριστικό του νέου **thread** που μόλις δημιουργήσαμε από τις συναρτήσεις pthread_join ή pthread_detach γιατί θα υποπέσει η εφαρμογή μας σε λάθη (errors). Καλώντας αυτές τις εντολές μπορούμε να πάρουμε / ορίσουμε την κατάσταση, από το χαρακτηριστικό detachstate, στην οποία βρίσκεται το **thread** κατά την δημιουργία του.

Για το τέλος αφήσαμε 2 συναρτήσεις της pthreads βιβλιοθήκης οι οποίες δεν ανήκουν σε κάποια κατηγορία.

Αυτές είναι οι:

- **pthread_self():**

Σύνοψησυνάρτησης: pthread_t pthread_self(void);

Περιγραφή: Αυτή η συνάρτηση όταν καλείται επιστρέφει το ID του thread από το οποίο έχει κληθεί.

- **pthread_equal(thread1, thread2):**

Σύνοψησυνάρτησης: int pthread_equal(pthread_t t1, pthread_t t2);

Περιγραφή: Χρησιμοποιώντας αυτή τη συνάρτηση έχουμε την δυνατότητα να συγκρίνουμε τα IDs των νημάτων που επιθυμούμε και τα περνάμε σαν ορίσματα.

4.2.2 Κρίσιμες περιοχές (Critical Sections)

Το κεφάλαιο των κρίσιμων τμημάτων του κώδικα είναι το σημαντικότερο κεφάλαιο στον πολυνηματικό (multithreading) προγραμματισμό και γενικά στον παράλληλο προγραμματισμό σε συστήματα κοινόχρηστης μνήμης**.** Η σπουδαιότητα του έγκειται στο πρόβλημα το οποίο πολλοί προγραμματιστές καλούνται να επιλύσουν, αυτό της ταυτόχρονης ανάθεσης τιμής σε μία μεταβλητή από 2 και παραπάνω νήματα (threads). Για να γίνει περισσότερο κατανοητό το παρακάτω σχήμα μας δείχνει πως μπορεί να προκύψει αυτή η κατάταξη και στην συνέχεια εμείς με την βοήθεια των ρουτινών που μας παρέχει η βιβλιοθήκη της *'pthreads'* θα προσπαθήσουμε να το επιλύσουμε.

Έστω ότι έχουμε σε έναν λογαριασμό 1000 Ευρώ και προσπαθούμε από 2 άλλους λογαριασμούς να κάνουμε ταυτόχρονα κατάθεση 200 Ευρώ. Το πρόβλημα έγκειται στο ότι όταν το γρηγορότερο νήμα ενημερώσει τον λογαριασμό τότε στο βήμα 2 του αργότερου νήματος δεν εμφανίζεται αυτή η ενημέρωση. Κατά συνέπεια όταν αυτό κάνει την κατάθεση δεν θα την κάνει στον ενημερωμένο λογαριασμό αλλά σε αυτόν που έχει διαβάσει στο βήμα 2.

Βήμα	Νήμα 1	Νήμα 2	Ποσό
1	Λογαριασμός = 1000		1000
2		Λογαριασμός = 1000	1000
3		Κατάθεση = 200	1000
4	Κατάθεση = 200		1000
5	Ενημέρωση λογαριασμού = 1000+200		1200
6		Ενημέρωση λογαριασμού = 1000+200	1200

Πίνακας 2

Πρόβλημα ενημέρωσης μεταβλητής ταυτόχρονα από δύο νήματα

Για να αντιμετωπίσουμε αυτού του είδους το πρόβλημα η pthreads μας παρέχει 2 είδη τεχνικών επίλυσης ενώ υπάρχει επίσης η προγραμματιστική τεχνική του *'busy-waiting'*.

- **Busy – Waiting:** Όταν ένα νήμα**,** έστω το νήμα 0**,** θέλει να εκτελέσει την δήλωση 'Ενημέρωση λογαριασμού = 1000 + 200' τότε πρέπει να είμαστε σίγουροι με κάποιο τρόπο ότι κανένα άλλο νήμα δεν βρίσκεται ήδη σε κατάσταση εκτέλεσης της ίδιας δήλωσης. Στην συνέχεια, και όταν αρχίσει η εκτέλεση της εντολής από το νήμα 0, αυτό με την σειρά του θα καταστήσει σαφές ότι κανένα άλλο νήμα δεν θα μπορέσει να φτάσει αυτή τη γραμμή κώδικα και να εκτελέσει την πράξη μέχρι το νήμα 0 ολοκληρώσει την εργασία του. Με τον ίδιο τρόπο, όταν ολοκληρωσει τις πράξεις του το νήμα 0 θα 'ειδοποιήσει' τα υπόλοιπα ότι πλέον μπορούν να εισέρθουν σε αυτό το κομμάτι κώδικα.

Για να επιτευχθεί αυτός ο τρόπος πρέπει να χρησιμοποιήσουμε μία μεταβλητή τύπου flag η οποία όταν παίρνει την τιμή του εκάστοτε νήματος τότε δεν θα επιτρέπει στα υπόλοιπα να συνεχίσουν την εκτέλεση του κομματιού που θέλουμε να διαφυλλάξουμε από την ταυτόχρονη χρήση τους.

Συνοπτικότερα αυτή η εντολή είναι η while (flag != my_rank); και χρησιμοποιούμε το ερωτηματικό (;) στο τέλος της εντολής διότι αυτός ο τύπος χρήσης της εντολής ελέγχου *'while'*

δεν έχει σώμα και κατ'επέκταση δεν θα προχωράει η ροή του προγράμματος (στην περίπτωσή μας το νήμα) στην εκτέλεση του υπολοίπου κώδικα μέχρι να ισχύσει η ισότητα.

Τέλος, ένα μειονέκτημα της μεθόδου *'Busy-Waiting'* είναι ότι υπάρχει μια μικρή πιθανότητα να επιτρέψει δύο εργασίες να εισέλθουν στην κρίσιμη περιοχή λαμβάνοντας ανεπιθύμητα αποτελέσματα από την εκτέλεση του προγράμματος.

- **Mutexes:** Το μειονέκτημα του προηγούμενου τρόπου είναι η συνεχής κατανάλωση πόρων του συστήματος λόγω της διάρκειας εκτέλεσης του βρόγχου *'while'* μέχρι να μην ισχύσει η η συνθήκη του *'while'*. Αν αναλογιστεί κανείς ότι πρωταρχικός μας στόχος και σκοπός είναι η βέλτιστη ταχύτητα εκτέλεσης μιας εφαρμογής και άρα η μικρότερη κατανάλωση πόρων συστήματος η μέθοδος *'Busy-Wait'* έρχεται σε αντίθεση. Το **mutex (mutual exclusive – αμοιβαίος αποκλεισμός)** έρχεται να το επιλύσει αυτό το πρόβλημα της συνεχόμενης κατανάλωσης πόρων. Ένα *'mutex'* είναι μία ειδική μεταβλητή και μαζί με μερικές ειδικές συναρτήσεις που μας παρέχει η βιβλιοθήκη '*pthreads*' μας βοηθάει να επιτύχουμε καλύτερο αποτέλεσμα από την προηγούμενη τεχνική του **busy – waiting**. Χωρίς να εξάγει διαφορετικού είδους αποτέλεσμα το **mutex** παρέχει αμοιβαίο αποκλεισμό στο νήμα αποκλείοντας τα υπόλοιπα από την εκτέλεση της / των γραμμών κώδικα της κρίσιμης περιοχής την ίδια στιγμή που αυτό έχει υπεισέρθει στην περιοχή αυτή.

Οι εντολές και τα χαρακτηριστικά που το καθορίζουν είναι τα εξής:

1. **pthread_mutex_init (mutex, attr)**

Συνοψη συνάρτησης: int pthread_mutex_init(pthread_mutex_t *restrict
mutex, const pthread_mutexattr_t *restrict attr);

Περιγραφή: Αρχικοποίηση ενός *'mutex'* το οποίο θα χρησιμοποιήσουμε. Εάν στην θέση των χαρακτηριστικών (attr) του περάσουμε την τιμή *'NULL'* τότε αυτά θα πάρουν τα προεπιλεγμένα χαρακτηριστικά. Σημαντική πληροφορία που πρέπει να γνωρίζουμε πριν χρησιμοποιήσουμε / αρχικοποιήσουμε κάποιο *'mutex'* είναι πως στην αρχική τους μορφή είναι ξεκλείδωτα 'unlock' οπότε το πρώτο νήμα που θα διατρέξει τον κώδικα του θα κλειδώσει την είσοδο αφήνοντας απ' έξω τα υπόλοιπα.

2. **pthread_mutex_destroy(mutex)**

Σύνοψησυνάρτησης: int pthread_mutex_destroy(pthread_mutex_t *mutex);

Περιγραφή συνάρτησης: Η συνάρτηση pthread_mutex_destroy() καταστρέφει το εκείνο το mutex στο οποίο αναφέρεται από το όρισμα mutex. Στην πραγματικότητα αυτό που συμβαίνει κατά την κλήση του είναι να αφαιρέσει την αρχικοποίηση που έχει υποστεί ένα mutex κατά την δημιουργία του.

3. **pthread_mutexattr_int/destroy(attr):**

Σύνοψη συνάρτησης: int pthread_mutexattr_init/destroy (pthread_mutexattr_t *attr);

Περιγραφή συναρτήσεων: Αντίστοιχα με τις *'pthread_mutex_int()'* και *'pthread_mutex_destroy()'* αυτές οι δύο συναρτήσεις μπορούν να αρχικοποιήσουν και καταστρέψουν ένα χαρακτηριστικό όρισμα ενός *'mutex'*.

Όπως αναφέραμε και προηγουμένως οι καταστάσεις στις οποίες μπορεί να βρεθεί ένα *'mutex'* είναι 3.

Οι συναρτήσεις που χρησιμοποιούμε για την επίτευξη της παραπάνω τεχνικής είναι οι εξής:

1. pthread_mutex_lock (mutex):

Σύνοψη συνάρτησης: int pthread_mutex_lock(pthread_mutex_t *mutex);

Περιγραφή συνάρτησης: Εάν ένα νήμα διατρέξει αυτό το κομμάτι του κώδικα και καλέσει αυτή την συνάρτηση τότε το αντικείμενο *'mutex'* το οποίο ορίζεται από το όρισμα *mutex θα κλειδώσει την είσοδο του. Εάν η είσοδος είναι ήδη κλειδωμένη τότε το νήμα θα περιμένει πρώτα να ξεκλειδώσει ώστε να εισέρθει στην κρίσιμη περιοχή και στην συνέχεια να το ξανακλειδώσει.

2. pthread_mutex_trylock (mutex):

Σύνοψη συνάρτησης: int pthread_mutex_trylock (pthread_mutex_t *mutex);

Περιγραφή συνάρτησης: Η συνάρτηση αυτή είναι περίπου ισοδύναμη με την pthread_mutex_lock(). Αν το αντικείμενο *'mutex'* που αναφέρεται από το όρισμα *mutex είναι ήδη κλειδωμένο είτε από το ίδιο ή από κάποιο άλλο νήμα τότε η κλήση θα επιστρέψει αμέσως. Εάν ο τύπος του *'mutex'* έχει ορισθεί να είναι αναδρομικός *'(PTHREAD_MUTEX_RECURSIVE)'* και το *'mutex'* ανήκει στο τρέχον νήμα τότε ο μετρητής του mutex_lock θα αυξηθεί κατά ένα και η συνάρτηση *'pthread_mutex_trylock()'* θα επιστρέψει επιτυχώς.

3. Pthread_mutex_unclock(mutex):

Σύνοψη συνάρτησης: int pthread_mutex_unlock (pthread_mutex_t *mutex);

Περιγραφή συνάρτησης: Απλά κάνοντας την μετάφραση της συνάρτησης κατανοούμε πως με την κλήση της, το αντικείμενο *'mutex'* που αναφέρεται από το όρισμα *mutex θα ξεκλειδώσει την είσοδο για το επόμενο νήμα.

Σαφώς και η τεχνική με την χρήση των *'mutexes'* είναι καλύτερη σε σχέση με αυτή του busy-waiting. Στο μοναδικό σημείο που 'υστερεί' είναι στον έλεγχο των νημάτων που θα προσεγγίσουν την κρίσιμη περιοχή του κώδικα. Στην πραγματικότητα δεν υπάρχει έλεγχος από τον χρήστη και αυτό το θέμα είναι υπό τον έλεγχο του λειτουργικού συστήματος. Όμως τι γίνεται όταν σε κάποια προβλήματα πρέπει να έχουμε αναγκαστικά έλεγχο των νημάτων που θα εκτελέσουν τις πράξεις μας;

- **Σημαφόροι (Semaphores):** Όταν το αποτέλεσμα που αναζητούμε σε μία εφαρμογή μπορεί να βρεθεί 'άσχετα' με την σειρά των πράξεων και την σειρά των τελεστών μέσα σε αυτές, τότε δεν μας ενδιαφέρει ιδιαίτερα ο έλεγχος των νημάτων που θα συνεργαστούν για την ανάθεση μίας τιμής σε μία μεταβλητή. Κατ'επέκταση η χρήση των mutexes μπορεί να κάνει αυτό που χρειαζόμαστε όσον αφορά στην διασφάλιση ότι ένα νήμα την φορά θα ανανεώνει την τιμή της μεταβλητής που βρίσκεται στο κρίσιμο τομέα του κώδικά μας. Όμως, θα συναντήσουμε προβλήματα (π.χ. Πολλ/σμός πινάκων) στα οποία η σειρά με την οποία θα εκτελεστούν οι πράξεις έχει σημασία για το τελικό αποτέλεσμα και όπως εύκολα κατανοούμε η χρήση των mutexes είναι λανθασμένη. Οι σημαφόροι* έχουν την δυνατότητα να μας δώσουν αυτό τον έλεγχο που χρειαζόμαστε.

Σημαντικό: Οι συναρτήσεις που αποτελούν τους σημαφόρους δεν βρίσκονται ορισμένες μέσα στις βιβλιοθήκες της C ούτε στην βιβλιοθήκη pthreads.h. Για να μπορέσουμε να τις χρησιμοποιήσουμε πρέπει να επικαλεστούμε στην αρχή της εφαρμογής μας την επικεφαλίδα *'semaphore.h'*.

Αυτές οι συναρτήσεις είναι οι εξής:

1. **sem_init(sem):**

Σύνοψη συνάρτησης: int sem_init(sem_t *sem, int pshared, unsigned int value)

Περιγραφή συνάρτησης: Αρχικά τονίζουμε ότι πρέπει να δηλώσουμε τον σημαφόρο που θα χρησιμοποιήσουμε. Αυτό γίνεται με την εντολή sem_t *sem_name*; Η sem_init(); αρχικοποιεί τον σημαφόρο που ορίσαμε προηγουμένως. Η πρώτη μεταβλητή sem δείχνει στον σημαφόρο που αρχικοποιούμε. Η pshared είναι μία σημαία η οποία μας υποδεικνύει αν ο σημαφόρος πρέπει να διαμοιραστεί από τις υπόλοιπες διεργασίες. Τέλος, η τιμή (value) είναι μια αρχικοποιημένη τιμή την οποία φέρει ο σημαφόρος.

2. **sem_wait(sem):**

Σύνοψη συνάρτησης: int sem_wait(sem_t *sem)

Περιγραφή συνάρτησης: Η συνάρτηση κλειδώνει τον σημαφόρο που αναφέρεται από το όρισμα sem εκτελώντας μία λειτουργία κλειδώματος. Όσο η τιμή του είναι 0, αυτός θα παραμείνει κλειδωμένος μέχρι κάποιο thread του αλλάξει αυτή τη τιμή.

3. **sem_post(sem):**

Σύνοψη συνάρτησης: int sem_post(sem_t *sem)

Περιγραφή συνάρτησης: Για να αλλάξουμε-αυξήσουμε τη τιμή του ορίσματος value ενός σημαφόρου τότε καλούμε αυτή τη συνάρτηση. Άρα, κατά συνέπεια ο σημαφόρος ξεκλειδώνει και κάποιο άλλο thread μπορεί να εισέρθει στην κρίσιμη περιοχή.

4. **sem_getvalue(sem, value):**

Σύνοψη συνάρτησης: int sem_getvalue(sem_t *sem, int *valp)

Περιγραφή συνάρτησης: Για να δούμε την τρέχουσα τιμή ενός σημαφόρου καλούμε την sem_getvalue(). Αυτό γίνεται παίρνοντας την τιμή και αναθέτοντάς την στην θέση μνήμης που δείχνεται από το valp.

5. **sem_destroy(sem):**

Σύνοψη συνάρτησης: int sem_destroy(sem_t *sem)

Περιγραφή συνάρτησης: Η sem_destroy() καταστρέφει έναν σημαφόρο. Προϋπόθεση για το συμβάν αυτό, είναι η μη αναμονή κάποιας διεργασίας μέσα σε αυτό το σημαφόρο.

4.2.3 Συγχρονισμός - Synchronization

Ένα ακόμη πρόβλημα που χρήζει προσοχής είναι ο συγχρονισμός των νημάτων ως προς την εκτέλεση του προγράμματος. Κάποιες φορές θα χρειαστεί να έχουμε όλα τα νήματα στο ίδιο ακριβώς σημείο ώστε να μπορέσουμε να διεξάγουμε σωστά αποτελέσματα ή συμπεράσματα. Για αυτό το λόγο ο συγχρονισμός των νημάτων είναι εξίσου σημαντικό θέμα όσο κι τα προηγούμενα που εξηγήσαμε. Οι ευρέως γνωστές τεχνικές είναι η χρήση των **φραγμάτων barriers,** η χρήση της busy-waiting και ενός mutex, η χρήση των σημαφόρων και τέλος η χρήση των **μεταβλητών συνθήκης (condition variables)**. Επειδή οι συναρτήσεις που ανήκουν στη δεύτερη και τρίτη τεχνική συγχρονισμού έχουν προαναφερθεί, εμείς θα εστιάσουμε στη χρήση των φραγμάτων και των μεταβλητών συνθήκης.

Ξεκινώντας από τα barriers οι συναρτήσεις που συναντούμε είναι οι εξής:

1. **pthread_barrier_init**

Σύνοψη συνάρτησης: int pthread_barrier_init(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned count)

Περιγραφή συνάρτησης: Είναι η συνάρτηση αρχικοποίησης ενός barrier όπου κατά την εκτέλεσή της δεσμεύει τις πηγές εκείνες που χρειάζεται ώστε να χρησιμοποιηθεί το barrier εκείνο που 'δείχνει' το όρισμα *barrier.* Επίσης, την ίδια στιγμή, αρχικοποιούνται και τα χαρακτηριστικά *attr* του αντικειμένου.

2. **pthread_barrier_wait**

Σύνοψη συνάρτησης: int pthread_barrier_wait(pthread_barrier_t *barrier**)**

Περιγραφή συνάρτησης: Η *'pthread_barrier_wait'* συγχρονίζει τα νήματα σταματώντας την λειτουργία τους έως ότου όλα συναντήσουν αυτή την συνάρτηση.

3. **pthread_barrier_destroy**

Σύνοψη συνάρτησης: int pthread_barrier_destroy(pthread_barrier_t *barrier)

Περιγραφή συνάρτησης: Διαγράφει το φράγμα που είναι ορισμένο ως *barrier* μέσα στη συνάρτηση. Επίσης, αναιρεί τις αρχικοποιήσεις των χαρακτηριστικών του ίδιου φράγματος.

Τα **condition variables** είναι μία μέθοδος συγχρονισμού κάπως καλύτερη απ'ότι τα φράγματα. Πιο συγκεκριμένα, μία μεταβλητή συνθήκης είναι ένα αντικείμενο δεδομένων που επιτρέπει σε ένα νήμα να αναστείλει την λειτουργία του μέχρι να συμβεί κάποιο γεγονός ή να ισχύσει μία συνθήκη (μεταβλητή συνθήκης). Όταν αυτή η συνθήκη ισχύσει τότε κάποιο άλλο νήμα στέλνει σήμα στο ανεσταλμένο thread ότι μπορεί να συνεχίσει την εκτέλεσή του κι έτσι το ενεργοποιεί.

Πριν περιγράψουμε τις συναρησεις ωφείλουμε να αναφέρουμε ότι οι condition variables έχουν τύπο pthread_cond_t .

Οι συναρτήσεις που τις περιγράφουν είναι οι εξής:

1. **pthread_cond_wait(condition, mutex):**

Σύνοψη συνάρτησης: int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex)

Περιγραφή συνάρτησης: Αυτή η συνάρτηση χρησιμοποιείται συνδυαστικά με μία συνάρτηση αμοιβαίου αποκλεισμού (mutex). Το νήμα που θα συναντήσει το *'mutex'* θα αναστείλει την λειτουργία του μέχρι να ισχύσει μία συνθήκη και να του επιτραπεί η είσοδος στην κρίσιμη περιοχή.

2. **pthread_cond_signal(condition):**

Σύνοψη συνάρτησης: int pthread_cond_signal(pthread_cond_t *cond)

Περιγραφή συνάρτησης: Η κλήση της συνάρτησης ενεργοποιεί ένα νήμα να συνεχίσει την εκτέλεση του το οποίο μέχρι πρότινος είχε αποκλειστεί.

3. pthread_cond_broadcast(condition):

Σύνοψη συνάρτησης: int pthread_cond_broadcast(pthread_cont_t *cond)

Περιγραφή συνάρτησης: Η κλήση της ενεργοποιεί όλα τα νήματα τα οποία έχουν υπεισέρθει σε κατάσταση αναμονής. Μία παρατήρηση σε αυτό το σημείο που κάνουμε είναι η χρησιμοποίηση της λέξης *'broadcast'* από την συνάρτηση κι η λειτουργία αυτής η οποία είναι όμοια με την αντίστοιχη *'MPI_Broadcast'* της βιβλιοθήκης *'MPI'*.

5. Αλγόριθμοι

Μέχρι τώρα, το περιεχόμενο του πονήματος μας παρουσιάζει και εξηγεί όλα όσα θα πρέπει να γνωρίζουμε για να κάνουμε τα πρώτα μας βήματα στον παράλληλο προγραμματισμό. Γνωρίζοντας πλέον τα βασικότερα βρισκόμαστε σε θέση να παραλληλοποιήσουμε κάποια κομμάτια βασικών αλγορίθμων.

Το μοτίβο που θα χρησιμοποιήσουμε σε αυτό το σημείο είναι το εξής:

1. Εισαγωγή - Περιγραφή του αλγορίθμου
2. Θεωρητική ανάλυση των κομματιών του αλγορίθμου που θα παραλληλοποιηθούν
3. Ψευδοκώδικας
4. Η παραλληλοποίηση των κομματιών
5. Η μέτρηση του χρόνου
6. Σύγκριση με σειριακό αλγόριθμο, υπολογισμός επιτάχυνσης - απόδοσης
7. Τελικό συμπέρασμα

Η υλοποίηση των αλγορίθμων θα γίνει αρχικά σε *'mpi'* και στην συνέχεια σε *'pthreads'* βιβλιοθήκη και οι αλγόριθμοι που υλοποιήσαμε είναι οι εξής:

- Quicksort
- Merge sort
- Odd/Even transposition
- Bitonic sort
- Radix sort

Οι αλγόριθμοι χρησιμοποιήθηκαν μόνο σε μία από τις δύο βιβλιοθήκες. Με αυτόν τον τρόπο καταφέραμε να αναλύσουμε και αναδείξουμε περισσότερους αλγορίθμους.

5.1 Ο αλγόριθμος Bitonic Sort σε MPI

Ο πρώτος αλγόριθμος που θα παραλληλοποιήσουμε - χρησιμοποιώντας αρχικά την βιβλιοθήκη *MPI* και στην συνέχεια την βιβλιοθήκη *pthreads* **-** είναι ο *Bitonic sort*.

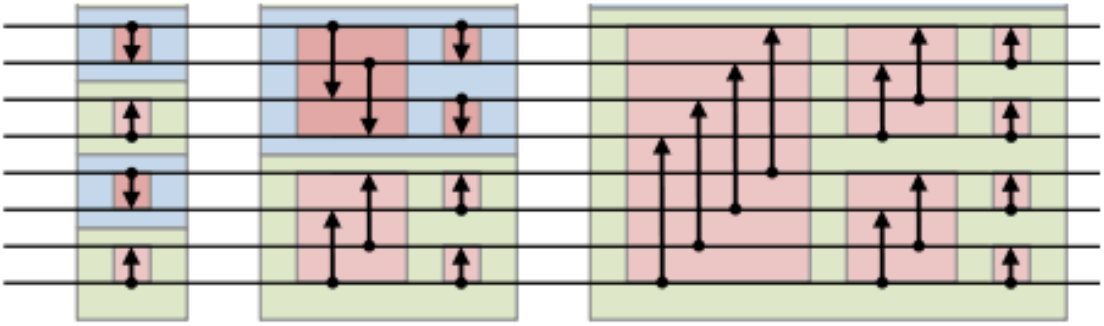
Ο *Bitonic sort* είναι ένας αλγόριθμος ο οποίος βασίζεται σε μία *"διτονική"* bitonic *αναζήτηση*. Αυτός ο αλγόριθμος αναπτύχθηκε για πρώτη φορά απο τον *K.E. Batcher* το 1968 και μέχρι σήμερα αυτή η μέθοδος με την διτονική ακολουθία χρησιμοποιείται σε πολλά προγράμματα και εφαρμογές.

5.1.1 Εισαγωγή – Περιγραφή του αλγορίθμου

Ο *Bitonic sort* χρησιμοποιεί μία *"διαίρει και βασίλευε"* προσέγγιση αρκετά όμοια με αυτή του αλγορίθμου *Merge sort.* Η ταξινόμηση μίας ακολουθίας από *N* στοιχεία χρειάζονται log N στάδια. Για παράδειγμα, εάν θέλουμε να ταξινομήσουμε με αύξουσα σειρά 8 νούμερα τότε $log_2 8=3$ τα στάδια από τα οποία θα περάσει ο αλγόριθμος μέχρι να ολοκληρώσει την ταξινόμησή του. Φυσικά, κάθε στάδιο περιέχει επιμέρους φάσεις.

Όπως βλέπουμε και στην εικόνα 10, στο πρώτο στάδιο ο αλγόριθμος θα συγκρίνει και θα ταξινομήσει εναλλάξ (αύξουσα – φθίνουσα – αύξουσα – φθίνουσα) τα γειτονικά νούμερα. Στο δεύτερο στάδιο ο αλγόριθμος θα κάνει ακριβώς τις ίδιες ενέργεις με την μόνη διαφορά ότι οι οι επιμερους φάσεις θα είναι δύο. Στην πρώτη φάση συγκρίνονται οι αριθμοί που έχουν απόσταση δύο και στην δεύτερη φάση συγκρίνονται πάλι τα γειτονικά στοιχεία. Αυτό το μοτίβο θα επαναλαμβάνεται μέχρι τα στάδια να γίνουν ίσα με τον λογάριθμο του πλήθους των αριθμών με βάση το 2.

Στο πρόγραμμά μας θα χρησιμοποιήσουμε εκατοντάδες - χιλιάδες νούμερα (τα οποία θα δημιουργηθούν τυχαία από το πρόγραμμα) και αυτό θα συμβεί γιατί μόνο με συγκρίσεις αρκετών αριθμών μπορούμε να δούμε διαφορά στην εκτέλεση του προγράμματος άρα και στην απόδοση του.



Εικ. 10

Ταξινόμηση του Bitonic sort για 8 νούμερα κατά αύξουσα σειρά

5.1.2 Ανάλυση του αλγορίθμου

Το κομμάτι που θα παραλληλοποιηθεί είναι αυτό των συναρτήσεων συγχώνευσης MergeLow() και MergeHigh().

5.1.3 Ψευδοκώδικας

```
int main()
{
   int i, j;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
   MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
  // Αρχικοποίηση πίνακα για παραγωγή τυχαίων αριθμών
   array_size = atoi(argv[1]) / num_processes;
   array = (int *) malloc(array_size * sizeof(int));
  //Γεννήτρια τυχαίων αριθμών
   srand(time(NULL));
   for (i = 0; i < array_size; i++)
   {
     array[i] = rand() % (atoi(argv[1]));
   }
   MPI_Barrier(MPI_COMM_WORLD);
  // Έναρξη παράλληλου προγράμματος
   int dimensions = (int)(log_2(num_processes));
```

```
  if (process_rank == MASTER)
  {
    printf("Number of Processes spawned: %d\n", num_processes);
    timer_start = MPI_Wtime();
  }
 // Bitonic Sort
  for (i = 0; i < dimensions; i++)
  {
    for (j = i; j >= 0; j--)
    {
 // (Το id είναι άρτιος και το j bit είναι 0)
 // Η' (το id is περιττός και το j είναι 1)
        if (((process_rank >> (i + 1)) % 2 == 0 && (process_rank >> j) % 2 == 0) ||
((process_rank >> (i + 1)) % 2 != 0 && (process_rank >> j) % 2 != 0))
        {
          MergeLow(j);
        }
        else
        {
          MergeHigh(j);
        }
    }
  }
  MPI_Barrier(MPI_COMM_WORLD);
  if (process_rank == MASTER)
  {
    timer_end = MPI_Wtime();
    printf("Displaying sorted array (only 10 elements for quick verification)\n");
 // Εκτύπωση αποτελεσμάτων
    for (i = 0; i < array_size; i++)
    {
        if ((i % (array_size / OUTPUT_NUM)) == 0)
        {
          printf("%d ",array[i]);
        }
    }
    printf("\n\n");
    printf("Time Elapsed (Sec): %f\n", timer_end - timer_start);
  }

  MPI_Finalize();
  return 0;
}
```

5.1.4 Η παραλληλοποίηση του Bitonic Sort

Στις πρώτες γραμμές του προγράμματος δηλώνουμε τις βιβλιοθήκες οι οποίες θα μας εξοπλίσουν με τις συναρτήσεις που θα χρησιμοποιήσουμε στην πορεία. Πολλές από αυτές τις βιβλιοθήκες τις καλέσαμε σε κάθε πρόγραμμα που υλοποιήσαμε έτσι η επεξήγησή τους θα γίνει μία φορά. Αυτές είναι:

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <mpi.h>
#include <string.h>
```

Σε αυτό το σημείο πρέπει να ορίσουμε και τις παγκόσμιας ορατότητας μεταβλητές **(global variables)**

Αυτές είναι οι εξής:

```
#defineMAX 8192
#defineTOTAL 16384
```

Ας εξηγήσουμε τον ρόλο όλων των παραπάνω ορισμών για μία φορά.

Η *'stdlib.h'* είναι μία βιβλιοθήκη που γενικά περιέχει τις συναρτήσεις υπολογισμού του μήκους των χαρακτήρων και εμείς θα χρειαστούμε την συνάρτηση *'sizeof()'* την οποία θα χρησιμοποιήσουμε για να δεσμεύσουμε ακριβώς την μνήμη που χρειαζόμαστε για κάθε πίνακα που θα συμπεριλάβουμε.

Η*'stdio.h'* είναι η κλασσική βιβλιοθήκη η οποία περιέχει τις συναρτήσεις εισόδου και εξόδου ενός προγράμματος όπως η *'printf'*, *'scanf'* κλπ. Στην δική μας περίπτωση θα χρειαστούμε την *'printf'*.

Η *'math.h'* είναι η βιβλιοθήκη η οποία περιέχει τις μαθηματικές συναρτήσεις. Εδώ πρέπει να αναφέρουμε πως δημιουργήσαμε την δική μας λογαριθμική συνάρτηση οπότε δεν την χρειαζόμαστε, αλλά κατά την διάρκεια της συγγραφής – επεξήγησης και παραμετροποίησης του κώδικα χρειάστηκε να βάλουμε τις λογαριθμικές συναρτήσεις που περιέχονται στην *'math.h'* για να μπορέσουμε να συγκρίνουμε το αποτέλεσμά της και θεωρήσαμε σωστό να την συμπεριλάβουμε.

Η*'time.h'* είναι η βιβλιοθήκη εκείνη η οποία περιέχει τις συναρτήσεις που αφορούν τον χρόνο άρα την χρειαζόμαστε για την χρονομέτρηση του προγράμματός μας.

Η *'mpi.h'* φυσικά για όλες τις συναρτήσεις που χρησιμοποιούμε κατά την παραλληλοποίηση της εφαρμογής.

Η *'string.h'* είναι μία βιβλιοθήκη που γενικά περιέχει τις συναρτήσεις υπολογισμού του μήκους των χαρακτήρων καθώς και κάποιες για την χρήση της μνήμης *'memory'.*Εμείς αυτή που θα χρησιμοποιήσουμε είναι η *'malloc()'.*

Σε αυτό το σημείο ορίζουμε και τις μεταβλητές οι οποίες θα είναι ορατές σε όλες τις διεργασίες καθόλη την διάρκεια εκτέλεσης του προγράμματος. Αυτές είναι η MAX όπου MAX θα ισούται με 8192 και TOTAL η οποία θα ισούται με 16384. MAX σημαίνει η μέγιστη τιμή του πλήθους των αριθμών τους οποίους μπορεί να ταξινομήσει – επεξεργαστεί κάθε μία διεργασία. Ενώ *'TOTAL'* σημαίνει η συνολική τιμή του πλήθους των αριθμών οι οποίοι θα ταξινομηθούν από το πρόγραμμα.

Στην συνέχεια του προγράμματος ακολουθεί η δήλωση των μεταβλητών και των συναρτήσεων. Αυτές είναι:

```
int temp[MAX];
int array1[MAX];
int mergeLow (int elements, int array[], int temp[]);
int mergeHigh (int elements, int array[], int temp[]);
int log2(intx);
```

Οι πίνακες temp[] και array1[] είναι δύο προσωρινοί πίνακες. Ο πίνακας *'temp[]'* είναι ένας προσωρινός πίνακας που χρειαζόμαστε ενώ ο *'array1[]'* είναι ο τελικός ταξινομημένος πίνακας ενώ οι ταξινομημένοι αριθμοί του πίνακα *'array1[]'* θα μεταφερθούν στον πίνακα *'array[]*.

Στην συνέχεια ακολουθεί η εκκίνηση της *'main()'* συνάρτησης και ουσιαστικά ολόκληρου του προγράμματος παράλληλης ταξινόμησης με τον αλγόριθμο bitonicsort. Σε αυτό το σήμειο για να μην γίνει κουραστική η ανάγνωση της εργασίας αλλά και για να γίνει ευκολότερο κατανοητό το κομμάτι της παραλληλοποίησης θα εξηγήσουμε μόνο τα κομμάτια και τις δηλώσεις μεταβλητών που προορίζονται για το παράλληλο μέρος.

Αρχικοποίηση της βιβλιοθήκης MPI

Μέχρι στιγμής έχουμε δηλώσει τις μεταβλητές και τις συναρτήσεις που θα χρησιμοποιήσουμε στην συνέχεια του προγράμματος και δεν σχετίζονται άμεσα με την *'mpi'* βιβλιοθήκη. Τώρα έφτασε η ώρα να δηλώσουμε και τις αντίστοιχες συναρτήσεις και μεταβλητές της βιβλιοθήκης *'mpi'*. Αυτοί οι ορισμοί και αυτές οι δηλώσεις γίνονται πάντα μέσα στον κορμό – σώμα *'body'* της *'main()'* συνάρτησης. Έτσι προκύπτει ο εξής κώδικας:

```
int main(intargc, char *argv)
{                                          // Από ξεκινάει η συνάρτηση main()
        int comm_sz;
        int my_rank;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
```

Η κλήση της συνάρτησης *'MPI_Init()'* ενημερώνει το *'MPI'* σύστημα να κάνει όλες τις απαραίτητες ρυθμίσεις που χρειάζονται για την συγγραφή ενός *'MPI'* προγράμματος. Για παράδειγμα μπορεί να δεσμεύσει χώρο για ανταλλαγή μηνυμάτων ή να αποφασίσει ποια διεργασία θα πάρει ποιον βαθμό.

Είναι σημαντικό να τονιστεί ότι πριν από αυτή την κλήση της συνάρτησης δεν μπορούμε να χρησιμοποιήσουμε καμία άλλη της βιβλιοθήκης *'MPI'*.

Τα ορίσματα *'argv'* και *'argc'* εάν δεν χρησιμοποιηθούν από το πρόγραμμα μπορούν να αντικατασταθούν από το κενό *'NULL'*. Μία λογική σκέψη θα ήταν αφού ορίζουμε πότε ξεκινάει το σώμα του MPI να μπορούμε να ορίσουμε και πότε τελειώνει αυτό. Μία τέτοια συνάρτηση μας την δίνει η MPI βιβλιοθήκη και είναι η MPI_Finalize();

Επίσης, είναι εξίσου σημαντικό να τονιστεί ότι μετά από αυτή την κλήση της συνάρτησης *'MPI_Finalize()'* δεν μπορούμε να χρησιμοποιήσουμε καμία άλλη συνάρτηση της βιβλιοθήκης MPI.

Έπειτα από την *'MPI_Init()'* ακολουθεί η *'MPI_Comm_rank()'* και η *'MPI_Comm_size()'*. Πριν δούμε ακριβώς τι κάνουν αυτές οι δύο συναρτήσεις ας θυμηθούμε την λειτουργία των *'communicators'*. Μία ομάδα επικοινωνίας στην '*MPI'* βιβλιοθήκη ουσιαστικά είναι μία ομάδα διεργασιών οι οποίες μπορούν να ανταλλάζουν μηνύματα μεταξύ τους. Ένας από τους σκοπούς της συνάρτησης *'MPI_Init()'* είναι να ορίσει σε έναν δίαυλο από ποιες διεργασίες αποτελείται τις οποίες ενεργοποιεί ο χρήστης όταν ξεκινάει να *'τρέχει'* το πρόγραμμα. Αυτή η ομάδα

επικοινωνίας ονομάζεται *'MPI_COMM_WORLD'*. Πιο συγκεκριμένα η κλήση των συναρτήσεων αυτών συλλέγουν πληροφορίες για τον *'communicator'*. Και στις δύο συναρτήσεις, το πρώτο όρισμα είναι ένας *'communicator'* και έχει τον ειδικό τύπο *'MPI_Comm'*. Η συνάρτηση *'MPI_Comm_size()'* επιστρέφει στο δεύτερο όρισμά της τον αριθμό των διεργασιών που βρίσκονται μέσα στον *'communicator'*, ενώ η *'MPI_Comm_rank'* επιστρέφει στο δεύτερο όρισμά της τον βαθμό της κληθείσας διεργασίας. Οι μεταβλητές που χρησιμοποιούμε συχνότερα είναι η comm_sz για τον αριθμό των διεργασίων και η my_rank για τον βαθμό της τρέχουσας διεργασίας.

Συνεχίζοντας, έφτασε η στιγμή να ορίσουμε την μνήμη που θα χρειαστεί να δεσμεύσουμε για να διαβάσουμε τον πίνακα *'array'* ο οποίος θα ταξινομηθεί. Επίσης, θα δηλώσουμε και τις υπόλοιπες μεταβλητές που μας ενδιαφέρουν όπως τα στοιχεία που θα διαβάσει η κάθε διεργασία, οι μεταβλητές του χρόνου που θα χρησιμοποιήσουμε για να καταμετρήσουμε την απόδοση του προγράμματος αλλά και μία ενδιάμεση μνήμη *'buffer'* για κάθε διεργασία για την αποθήκευση των δεδομένων και η οποία θα μας βοηθήσει στο να γίνει ακόμη γρηγορότερη η ταξινόμηση.

Έτσι έχουμε:

```
elements = (TOTAL/comm_sz);
array = (int*) malloc(elements * sizeof(int));
MPI_Buffer_attach (buf,MAX);
```

Από τις παραπάνω τρεις γραμμές βλέπουμε πως ενώ είπαμε πως θα ορίσουμε και τις μεταβλητές της καταμέτρησης του χρόνου εκτέλεσης του προγράμματος δεν τις έχουμε αναφέρει. Αυτό έγινε εσκεμμένα διότι ο κάθε ένας μπορεί να χρησιμοποιήσει την δική του τακτική καταμέτρησης. Εμείς, έχουμε ήδη αναφέρει ποια μέθοδος χρησιμοποιήθηκε σε όλα τα προγράμματα προηγουμένως στο σύγγραμμα.

Από αυτό το σημείο και έπειτα θα διευκρινίσουμε τον ρόλο της κύριας διεργασίας στην οποία έχουμε δώσει τον βαθμό μηδέν (0) αλλά και τον ρόλο των υπόλοιπων διεργασιών μέσα στο πρόγραμμα. Αναλυτικότερα έχουμε:

```
if (my_rank == 0)
{
        temp1 = (int*) malloc(elements * sizeof(int));
        for (j = 1; j < comm_sz; j++)
        {
        for (i = 0; i < elements; i++)
        {
                temp1[i] = rand()%TOTAL;
        }
        MPI_send (temp1, elements, MPI_INT, j , 2, MPI_COMM_WORLD);
        }
        free (temp1);
}
else
{
        MPI_Recv (array, elements, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
```

```
}
```

Από το παραπάνω τμήμα κώδικα γίνεται κατανοητό ότι η διεργασία με βαθμό 0 θα δεσμεύσει μνήμη για τον προσωρινό πίνακα *'temp1'* τόση όση είναι και το μέγεθος των στοιχείων της μεταβλητής *'elements'*. Στην συνέχεια για κάθε ένα στοιχείο του πίνακα θα δίνεται μία αντίστοιχη τυχαία τιμή η οποία μπορεί να φτάσει έως το 16384. Αφού λοιπόν συμπληρωθούν τόσοι πίνακες όσες είναι και οι διεργασίες μας σειρά έχει η αποστολή αυτών των πινάκων *'temp1'* σε αυτές τις διεργασίες. Αυτό γίνεται με την MPI_send (temp1, elements, MPI_INT, j , 2, MPI_COMM_WORLD);. Πρέπει να σημειώσουμε πως η διεργασία με αναγνωριστικό μηδέν (0) έχει δημιουργήσει έναν πίνακα array[] για να τον ταξινομήσει η ίδια παράλληλα με τους προσωρινούς πίνακες που δημιούργησε και έστειλε στις υπόλοιπες διεργασίες. Με το πέρας της αποστολής καταστρέφουμε τον πίνακα temp1[] για να μην καταναλώνει άλλο από την μνήμη μας. Οι υπόλοιπες διεργασίες θα λάβουν αυτά τα πακέτα με την εντολή MPI_Recv(array, elements, MPI_INT, 0, 2, MPI_COMM_WORLD, &status); τα οποία θα τα ταξινομήσουν στο εσωτερικό τους.

Αξίζει να σημειωθεί ότι η ετικέτα της επικοινωνίας είναι το 2. Επειδή θα χρησιμοποιήσουμε και άλλες *MPI_Send* και *MPI_Recv* θα δούμε ότι αυτή η ετικέτα θα αλλάξει βαθμό για να μην 'μπερδευτούν' οι διεργασίες και πάρουν λάθος δεδομένα.

Στην συνέχεια θα παρατηρήσουμε πως οι διεργασίες αλληλεπιδρούν ανταλλάσσοντας δεδομένα ώστε να γίνει η τελική ταξινόμηση.

Έτσι έχουμε:

```
void mergeLow() // Δημιουργεί μία νέα λίστα στέλνοντας του μεγαλύτερους αριθμούς και
                    λαμβάνοντας του μικρότερους

void mergeHigh() // Δημιουργεί μία νέα λίστα στέλνοντας του μικρότερους αριθμούς και
                    λαμβάνοντας του μεγαλύτερους
```

```
for ( i = 2, mask = 2; i <= comm_sz; i *= 2, mask = mask << 1)
{
        dim = log_2(i);
        mask2 = 1 << (dim-1);
        if ((rank & mask) == 0)
        {
                for (j = 0; j < dim; j++)
                {
                        partner = rank ^ mask2;
                        if (rank < partner)
                        {
                                MPI_Sendrecv (array, elements, MPI_INT, partner, 3, temp,
                                elements, MPI_INT, partner, 3, MPI_COMM_WORLD, &status);
                                mergeLow(elements. array, temp);
                        }
                        else
                        {
                                MPI_Sendrecv (array, elements, MPI_INT, partner, 3, temp,
                                elements, MPI_INT, partner, 3, MPI_COMM_WORLD, &status);
                                mergeHigh(elements. array, temp);
```

```
                }
                mask2 = mask2 >> 1;
        }
    else
    {
        for (j = 0; j < dim; j++)
        {
            partner = rank ^ mask2;
            if (rank > partner)
            {
                MPI_Sendrecv (array, elements, MPI_INT, partner, 3, temp,
                elements, MPI_INT, partner, 3, MPI_COMM_WORLD, &status);
                mergeLow(elements, array, temp);
            }
            else
            {
                MPI_Sendrecv (array, elements, MPI_INT, partner, 3, temp,
                elements, MPI_INT, partner, 3, MPI_COMM_WORLD, &status);
                mergeHigh(elements, array, temp);
            }
            mask2 = mask2 >> 1;
        }
    }
}
```

Ξεκινώντας από την πρώτη γραμμή του κώδικα, σκοπός μας είναι να ορίσουμε από πόσες φάσεις ή διαστάσεις *'dim'* θα περάσει το πρόγραμμα έως ότου ταξινομήσει τελικά τους αριθμούς. Επίσης, θα πρέπει να γνωρίζουμε αυτές οι φάσεις από πόσες εσωτερικές φάσεις αποτελούνται.

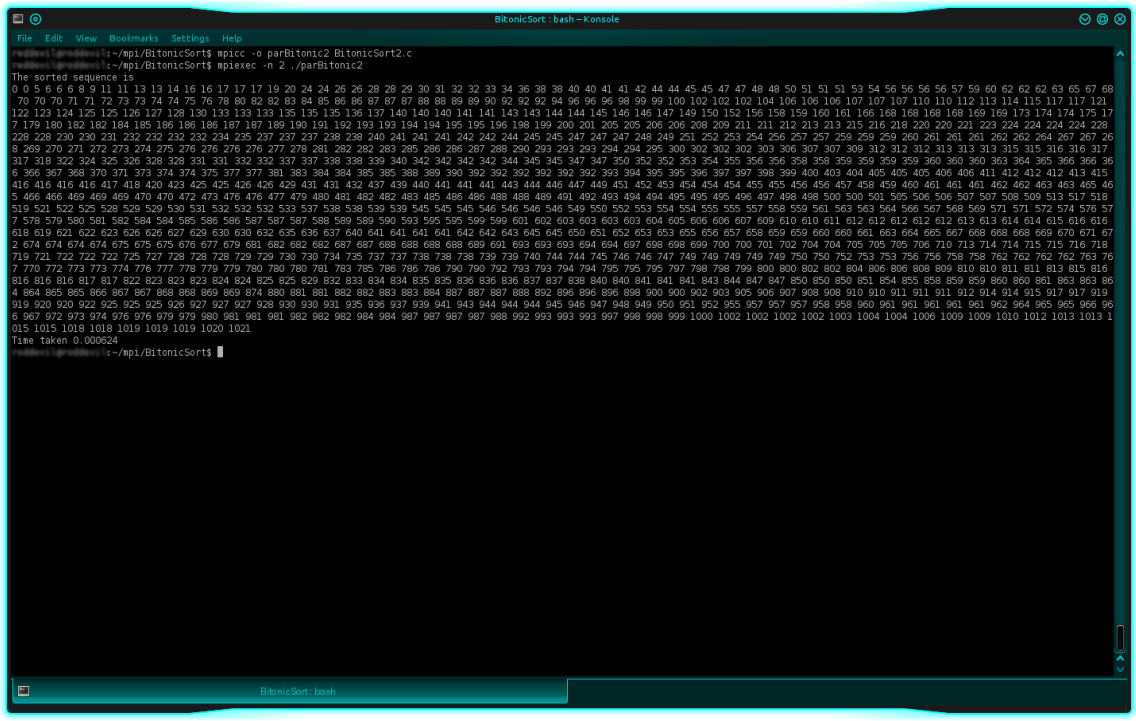
Σε προηγούμενο σημείο αυτού του κεφαλαίου είπαμε ότι μπορούμε να υπολογίσουμε τις συνολικές φάσεις από τις οποίες θα περιέλθει ο αλγόριθμος αφού γνωρίζουμε ήδη την συμπεριφορά του και με την χρήση του λογαρίθμου έχουμε για παράδειγμα: **$\log_2 8=3$**. Άρα η πρώτη φάση θα υπολογιστεί από τον λογάριθμο με τύπο **$\log_2 2$** το οποίο ισούται με ένα. Η επόμενη θα υπολογιστεί πολλαπλασιάζοντας επί δύο (X2) το δύο (2). Αυτό θα γίνει τέσσερα (4) και άρα το $\log_2 4 = 2$ κ.ο.κ. μέχρι να φτάσουμε στο σημείο το $\log_2 X$ να ισούται με τον αριθμό των διεργασιών μας δηλ. 2, 4 ή 8.

Το *'dim'* λοιπόν θα πάρει την τιμή 1 στην αρχή και το *'mask2'* επίσης την τιμή 1.

5.1.5 Η μέτρηση του χρόνου εκτέλεσης και της απόδοσης

Μετά από την παραλληλοποίηση του αλγορίθμου μπορούμε να μετρήσουμε την απόδοσή του. Χρονομετρώντας τον χρόνο εκτέλεσης και συγκρίνοντάς τον με αυτόν ενός σειριακού αλγορίθμου μπορούμε να βγάλουμε χρήσιμα συμπεράσματα. Επίσης, η χρονομέτρηση όπως προαναφέραμε έγινε τόσο για δύο όσο και για τέσσερις και οκτώ πυρήνες για ένα συγκεκριμένο εύρος αριθμών. Τα αποτελέσματα που πήραμε μετά από δέκα συγκρίσεις φαίνονται παρακάτω στις εικόνες.

Πρέπει να σημειωθεί ότι κάναμε δέκα συγκρίσεις και επιλέξαμε την χαμηλότερη τιμή από αυτές. Επίσης ενδεικτικά παρουσιάζουμε παρακάτω ένα *'print screen'* ως απόδειξη λειτουργίας του προγράμματος. Αυτό θα συμβεί και στις υπόλοιπες υλοποιήσεις των αλγορίθμων.



Εικόνα 10.1

Ταξινόμηση πίνακα με 1024 αριθμούς δημιουργώντας 2 διεργασίες

Ο παρακάτω πίνακας συνοψίζει τους χρόνους εκτέλεσης του παράλληλου τμήματος του προγράμματος για 9 περιπτώσεις.

Διεργασίες/Αριθμούς	2	4	8
1024	0,000624	0,000406	0,000425
4096	0,001034	0,001478	0,001019
16384	0,006037	0,004832	0,003229

Πίνακας 3

Χρόνος που χρειάστηκε να εκτελεστεί το παράλληλο τμήμα του προγράμματος

5.1.6 Τελικό συμπέρασμα

Το συμπέρασμα για να είναι το ακριβέστερο δυνατό θα πρέπει οι μετρήσεις να γίνονται στον ίδιο υπολογιστή για πολλές μέρες και αυτές να είναι όσες περισσότερες γίνεται. Στην συγκεκριμένη βιβλιοθήκη και με τον συγκεκριμένο *'compiler'* το ιδανικό θα ήταν να πραγματοποιηθεί σε ένα δίκτυο υπολογιστών αυτές οι μετρήσεις. Εμείς ορίσαμε τις συνθήκες με τις οποίες πήραμε τα προηγούμενα αλλά και τα επόμενα αποτελέσματα για να είμαστε όσο κοντά γίνεται σε σωστά συμπεράσματα.

Αυτό που παρατηρούμε είναι ότι το πρόγραμμα συμπεριφέρεται έτσι όπως θα περιμέναμε να συμπεριφερθεί και με βάση την λογική. Δηλαδή, τις περισσότερες φορές, για κάθε ομάδα αριθμών, όσες περισσότερες διεργασίες χρησιμοποιήσαμε τόσο γρηγορότερα ταξινομήθηκε ο πίνακας. Αυτό που είναι περισσότερο εμφανές είναι πως σε όλες τις μετρήσεις που κάναμε με οκτώ διεργασίες στο ίδιο πλήθος αριθμών τα αποτελέσματα ήταν γρηγορότερα σε σχέση με όλες τις μετρήσεις χρησιμοποιώντας δύο διεργασίες. Τις δύο από τις τρεις φορές όταν χρησιμοποιήσαμε τέσσερις διεργασίες πήραμε αναμενόμενα αποτελέσματα και σε σχέση με τους δύο αλλά και σε σχέση με τις οκτώ διεργασίες. Μόνο μία φορά η χρήση των δύο διεργασιών έφερε γρηγορότερη ταξινόμηση από την αντίστοιχη χρήση των τεσσάρων, όπως επίσης και μία φορά η χρήση των οκτώ διεργασιών έφερε λίγο καθυστερημένη ταξινόμηση από την αντίστοιχη των τεσσάρων. Αυτό ωφείλεται σε τρεις λόγους κυρίως οι οποίοι έχουν επισημανθεί νωρίτερα στην εργασία.

Ο πρώτος έχει να κάνει με το πλήθος των αριθμών. Όσο μικρότερο είναι αυτό το πλήθος τόσο περισσότερη είναι και η πιθανότητα η χρήση πολλών επεξεργαστών-διεργασιών να επιφέρει μεγαλύτερη καθυστέρηση. Από την άλλη είναι ξεκάθαρο πως η συμπεριφορά του αλγορίθμου σε πίνακα με πολύ μεγάλο πλήθος ακεραίων είναι αναμενόμενη. Ο δεύτερος λόγος είναι επειδή το παράλληλο πρόγραμμα τρέχει σε έναν υπολογιστή γενικής χρήσης όπου ταυτόχρονα τρέχουν και άλλες υπηρεσίες στο παρασκήνιό του και αυτές δημιουργούν απρόβλεπτες καταστάσεις στο πρόγραμμα δίνοντάς μας κάποιες φορές ψευδή εικόνα.

5.2 Ο αλγόριθμος Odd/Even transposition σε MPI

5.2.1 Εισαγωγή - Περιγραφή του αλγορίθμου

Ο επόμενος αλγόριθμος ταξινόμησης που θα αναλύσουμε είναι ο *'Odd/Even transposition'*. Αυτός είναι ένας αλγόριθμος ο οποίος έχει σχεδιαστεί αρχικά για παράλληλα υπολογιστικά συστήματα το 1972 από τον Habermann[14]. Εμείς θα εξετάσουμε μία παραλλαγή του συγκεκριμένου αλγορίθμου η οποία προήλθε από τους Baudet και Stevenson[15]. Σύμφωνα με αυτή την παραλλαγή κάθε διεργασία θα κάνει μία ταξινόμηση στην υπολίστα που έχει χρεωθεί από την αρχική λίστα σε κάθε βήμα. Στην συνέχεια, θα πραγματοποιείται συγχώνευση και ανταλλαγή περιττών και άρτιων αριθμών και το ανάποδο μεταξύ των γειτονικών διεργασιών.

5.2.2 Ανάλυση του αλγορίθμου

Σε αυτόν τον αλγόριθμο τα κομμάτια που θα παραλληλοποιήσουμε είναι δύο. Το πρώτο έχει να κάνει με τον τρόπο κατά τον οποίο θα διαμοιραστούν τα τμήματα του πίνακα προς ταξινόμηση σε κάθε διεργασία ώστε στην συνέχεια να εκτελεστούν οι συγχωνεύσεις. Το δεύτερο κομμάτι που θα παραλληλοποιηθεί είναι ο τρόπος με τον οποίο θα ζευγαρώνουν οι γειτονικές διεργασίες(περιττές/άρτιες) εναλλάξ.

5.2.3 Ψευδοκώδικας

Στην συνέχεια παρουσιάζεται ο ψευδοκώδικας του *'Odd/Even Transposition'*. Αρχικά για την συνάρτηση *'oddEven()'* η οποία τρέχει σε όλες τις διεργασίες. Οι δύο επόμενοι ψευδοκώδικες αφορούν στο κομμάτι του αλγορίθμου που παραλληλοποιήσαμε. Τα κομμάτια που εκτελούνται σε σειριακή μορφή θα περιγραφούν παρακάτω συνοπτικά και αυτά είναι οι συναρτήσεις *'merge()'* και *'split()'*.

```
oddEven()
{
        int my_rank, size, i
        int *local_array
```

```
        MPI_Scatter(array, local_array, INT)
        merge_sort(n/size, local_array)
        for (i = 1; i <= size; i++)
        {
                if ((my_rank is Odd) && (my_rank <= size-1) && (my_rank isnot root))
                {
                        pair(my_rank with next_rank)
                }
                else if ((my_rank is Even)  && (rank isnot root))
                {
                        pair(previous_rank with my_rank)
                }
                gather(all_locals_a)
                MPI_Success
        }
}
```

Αντίστοιχα ο ψευδοκώδικας της συνάρτησης *'pair()'* είναι:

```
pair(local_n, local_a, sendtag, recvtag, communicator)
{
        if(rank=sendrank)
        {
                MPI_Send(int local_array, int local_n, recvrank, mergetag, communication)
                MPI_Recv(int local_array, int local_n, recvrank, sortedtag, communication);
        }
        else //(rank = receiverank)
        {
                MPI_Recv(int remote,int local_n, sendrank, mergetag, communication);
                merge(local_a, local_n);
                int theirstart = 0, mystart = localn;
                if (sendrank > rank)
                {
                        theirstart = localn;
                        mystart = 0;
                }
                MPI_Send(&(all[theirstart]), local_n, sendrank, sortedtag, communicator);
                for (i=mystart; i<mystart+localn; i++)
                {
                        local_a[i-mystart] = all[i];
                }
        }
}
```

5.2.4 Η παραλληλοποίηση του Odd/Even Transposition

Στην αρχή δηλώνουμε τις βιβλιοθήκες οι οποίες θα μας παρέχουν τις συναρτήσεις που χρειαζόμαστε για την ανάπτυξη του προγράμματος, τις μεταβλητές εκείνες που χρειάζεται να είναι ορατές από όλες τις συναρτήσεις και τέλος τις συναρτήσεις που θα χρησιμοποιήσουμε. Οι μεταβλητές που δηλώσαμε περιορίζονται στις μεταβλητές που θα χρειαστούμε για την χρονομέτρηση του παράλληλου τμήματος του αλγορίθμου. Οι συναρτήσεις που αναπτύχθηκαν είναι η συνάρτηση *'merge()'* η οποία είναι υπευθυνη για την συγχώνευση, η *'split()'* η οποία είναι υπεύθυνη για την διάσπαση του πίνακα σε μικρότερους πίνακες, η *'pair()'* η οποία είναι υπεύθυνη για το ζευγάρωμα των διεργασίων καθώς και η *'oddEven()'* από την οποία ξεκινάει ο διαχωρισμός του αρχικού πίνακα σε υποπίνακες και ο διαμοιρασμός αυτών σε όλες τις διεργασίες μέχρι και την τελική συλλογή των υποπινάκων αφού έχουν πραγματοποιθεί οι συγχωνεύσεις και οι ανταλλαγές των αριθμών. Οι συναρτήσεις με παράλληλη μορφή όπως αναφέραμε και στην προηγούμενη υποενότητα τις οποίες θα αναλύσουμε και μελετήσουμε την συμπεριφορά τους είναι οι δύο τελευταίες.

```
void pair(int local_n, int *local_a, int sendrank, int recvrank, MPI_Comm comm)
{
        int rank, i;
        int remote[local_n];
        int all[2*local_n];
        const int mergeTag = 1;
        const int sortedTag = 2;

        MPI_Comm_rank(comm., &rank);
        if (rank = sendrank)
        {
            MPI_Send(local_a, local_n, MPI_INT, recvrank, mergetag, MPI_COMM_WORLD);
            MPI_Recv(local_a, local_n, MPI_INT, recvrank, sortedtag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        }
        else
        {
                MPI_Recv(remote, local_n, MPI_INT, sendrank, mergetag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                merge(local_a, local_n, remote, local_n, all);
                int theirstart = 0, mystart = local_n;
                if(sendrank > rank)
                {
                        theirstart = local_n;
                        mystart = 0;
                }
                MPI_Send(&(all[theirstart]), local_n, MPI_INT, sendrank, sortedtag,
MPI_COMM_WORLD);
                for (i = mystart; i < mystart + local_n; i++)
                {
                        local_a[i-mystart] = all[i];
                }
```

```
        }
}

int oddEven(int n, int *a, int root, MPI_Comm comm)
{
        int rank, size, i;
        int *local_a;       // αυτό το local_a είναι το ίδιο με της προηγούμενης συνάρτησης

        MPI_Comm_rank (comm, &rank);
        MPI_Comm_size (comm., &size);

        local_a = (int *) calloc(n/size, sizeof(int));

        MPI_Scatter(a, n/size, MPI_INT, local_a, n/size, MPI_INT, root, comm);
        merge_sort(n/size, local_a);

        for (i = 1; i <= size; i++)
        {
                if ((i + rank) % 2 == 0)
                {
                        if(rank < size - 1)
                        {
                                pair(n/size, local_a, rank, rank+1, comm);
                        }
                else if(rank > 0)
                {
                        pair(n/size, local_a, rank-1, rank, comm);
                }
        }
        MPI_Gather (local_a, n/size, MPI_INT, a, n/size, MPI_INT, root, comm);
        if (rank == root)
        {
                printf("Sorted: ", a);
        }
        return MPI_SUCCESS;
}
```

Στην συνέχεια ακολουθεί η συνάρτηση *'main()'* η οποία αρχικοποιεί το πρόγραμμα για να λάβει συναρτήσεις από την βιβλιοθήκη *'MPI'*, αναθέτει τυχαίους αριθμούς στον πίνακα *'a[]'*, καλεί την *'oddEven()'* και τέλος εμφανίζει τον χρόνο εκτέλεσης του προγράμματος. Ας δούμε όμως και αυτές τις εντολές:

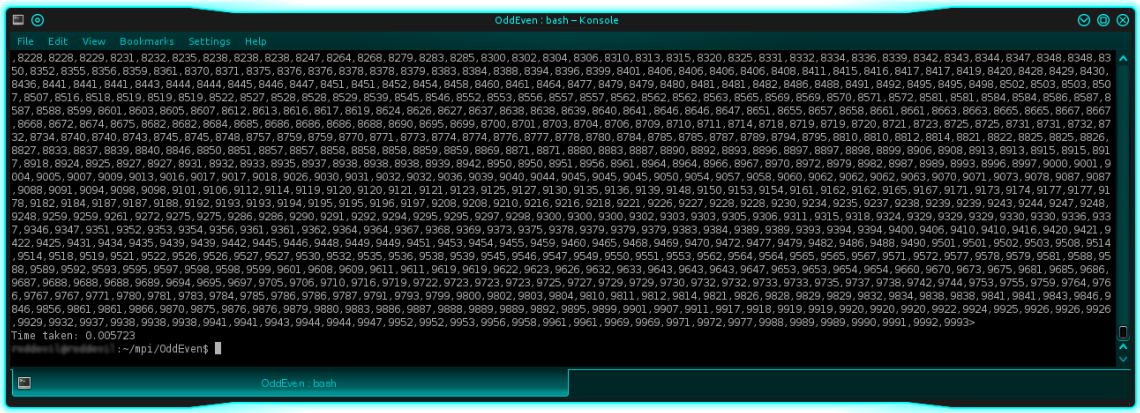
```
int main(int argc, char **argv)
{
        srand((double) time (NULL));
        int i;
```

```
        begin = clock();
        MPI_Init(&argc, &argv);
        int a[MAX];

        for (i = 0; i < MAX; i++)
        {
                a[i] = rand()%10001;
        }
        oddEven(MAX, a, 0, MPI_COMM_WORLD);
        MPI_Finalize();
        end = clock();

        time_spent = (double) (end-begin) / CLOCKS_PER_SEC;
        printf("Time taken: %f", time_spent);
}
```

5.2.5 Η μέτρηση του χρόνου εκτέλεσης

Οι μετρήσεις που πήραμε ήταν οι χαμηλότερες ανάμεσα στις 10 που πραγματοποιήσαμε συνολικά σε κάθε μία περίπτωση. Οι περιπτώσεις που θα εξετάσουμε είναι οι εξής. Για 1000 αριθμούς προς ταξινόμηση με 1, 2, 4 και 8 διεργασίες. Για 4000 αριθμούς με 1, 2, 4 και 8 διεργασίες. Τέλος, για 16000 αριθμούς προς ταξινόμηση με 1, 2, 4 και 8 διεργασίες.

Η παρακάτω εικόνα είναι αντιγραφή της οθόνης μας όταν εκτελέσαμε το πρόγραμμα και παρουσιάζει τον χρόνο εκτέλεσης του αλγορίθμου *'Odd/Even Transposition'* σε μία περίπτωση (πλ. αριθμών = 4096 και πλ. διεργασιών = 2). Αυτό που χρονομετρήσαμε και παρουσιάζουμε εμείς είναι το κομμάτι κώδικα μεταξύ έναρξης και λήξης της *'mpi'*.



Εικόνα 11.1

Χρόνος ταξινόμησης πίνακα 4000 αριθμών με χρήση 2 διεργασιών

Από τους χρόνους εκτέλεσης των διεργασιών παρατηρούμε ότι όσες περισσότερες προσθέταμε τόσο περισσότερο αυξανόταν ο χρόνος εκτέλεσής τους, όμως όχι σε απογοητευτικό βαθμό. Εδώ πρέπει να επαναλάβουμε ότι αυτή η συμπεριφορά είναι φυσιολογική διότι το σύστημα μας μετατρέπει τις διεργασίες σε νήματα και έτσι υπάρχει μία επιπλέον μικρή καθυστέρηση και άρα δεν μπορούμε να βγάλουμε κάποιο ασφαλές συμπέρασμα για την συμπεριφορά του

αλγορίθμου. Στους επόμενους τρεις αλγορίθμους όμως που θα μπορέσουμε να εξάγουμε συμπεράσματα με περισσότερη ασφάλεια.

Γενικότερα, ο αλγόριθμος ταξινόμησης *'Odd/Even Transposition'* σε όλες τις περιπτώσεις ταξινομούσε τους αριθμούς σε λιγότερο από ένα δευτερόλεπτο. Ας δούμε όμως παρακάτω όλους τους χρόνους για τις εννιά περιπτώσεις που χρονομετρήσαμε.

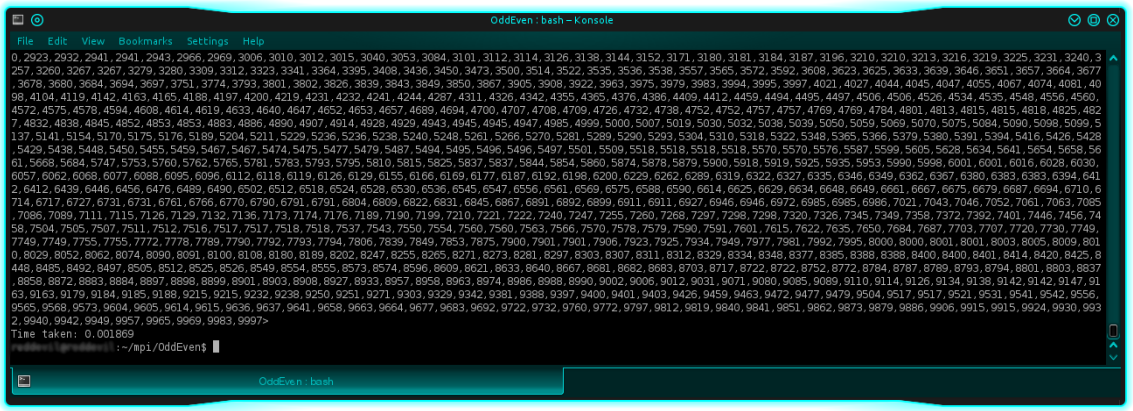
διεργασίες/πλήθος αριθ	1000	4000	16000
1	0.001869	0.002544	0.009367
2	0.002023	0.005023	0.07355
4	0.002961	0.004622	0.012235
8	0.003765	0.003299	0.015574

Πίνακας 4

Πίνακας χρόνων εκτέλεσης του αλγορίθμου Odd/Even Transposition

5.2.6 Σύγκριση με σειριακό αλγόριθμο, υπολογισμός επιτάχυνσης – απόδοσης

Στις επόμενες 3 εικόνες θα χρονομετρήσουμε το πρόγραμμα με την χρήση μίας διεργασίας άρα και σειριακό τρόπο και τα αποτελέσματα που θα εξάγουμε μαζί με τα προηγούμενα θα μας οδηγήσουν στην μέτρηση της απόδοσης του αλγορίθμου *'Odd/Even Transposition'*. Πριν παρουσιάσουμε τους πίνακες με την απόδοση, επιτάχυνση αλλά και φόρτου εργασίας πρέπει να κάνουμε την εξής σημαντική παρατήρηση. Όπως έχουμε ήδη πει η *'mpi'* είναι σχεδιασμένη για να τρέχει σε ένα δίκτυο υπολογιστών με τον μεταγλωτιστή *'lanboot'*. Στην δική μας περίπτωση, η εκτέλεση του προγράμματος έγινε σε ένα πολυπύρηνο υπολογιστικό σύστημα και ουσιαστικά προσομοιώθηκε το *'mpi'* σε σύστημα διαμοιραζόμενης μνήμης. Αυτό έχει σαν αποτέλεσμα να αυξηθούν οι πιθανότητες για αλλοιώσεις των χρονομετρήσεων σε σημαντικό βαθμό.



Εικόνα11.2

Χρόνος ταξινόμησης πίνακα 1000 με σειριακό τρόπο

Ήδη από την πρώτη χρονομέτρηση παρατηρούμε την μεγάλη διαφορά στον χρόνο εκτέλεσης του σειριακού προγράμματος. Φαίνεται πως με την χρήση μίας διεργασίας πετυχαίνουμε καλύτερους χρόνους και αυτό κάποιες φορές είναι λογικό εάν αναλογιστούμε το σύστημα προσομοίωσης δεν είναι το καταλληλότερο όμως ένας από τους σκοπούς της εργασίας είναι να μάθουμε να χρησιμοποιούμε σωστά τις βιβλιοθήκες παράλληλου προγραμματισμού της

γλώσσας *'C'*. Επίσης, το πλήθος των αριθμών είναι σχετικά μικρό και στην συνέχεια που θα ταξινομήσουμε μεγαλύτερο πλήθος θα παρατηρήσουμε πως οι χρόνοι καλυτερεύουν.

Στην συνέχεια ακολουθεί η αποτύπωση σε πίνακα των επιταχύνσεων, των αποδόσεων και των φόρτων για κάθε μία περίπτωση ξεχωριστά.

διεργασίες/πλήθος αριθ	1000	4000	16000
2	0.923875	0.444521	1.273555
4	0.631205	0.550410	0.765590
8	0.353174	0.298486	0.601451

Πίνακας 5

Πίνακας επιταχύνσεων του αλγορίθμου Odd/Even Transposition

διεργασίες/πλήθος αριθ	1000	4000	16000
2	0.461937	0.222260	0.636777
4	0.157801	0.137602	0.191397
8	0.044146	0.037310	0.075181

Πίνακας 6

Πίνακας αποδόσεων του αλγορίθμου Odd/Even Transposition

διεργασίες/πλήθος αριθ	1000	4000	16000
2	0.001089	0.004451	0.002672
4	0.002494	0.003986	0.009894
8	0.005059	0.08205	0.014404

Πίνακας 7

Πίνακας με τα λειτουργικά κόστη των διεργασιών του αλγορίθμου Odd/Even Transposition

5.2.7 Τελικό συμπέρασμα

Η χρονομέτρηση στον *'Odd/Even Transposition'* έγινε συνολικά στην εκτέλεση του προγράμματός όπως έγινε και σε όλους τους αλγορίθμους που υλοποιήσαμε έτσι ώστε να υπάρχει μία ομοιομορφία στο συμπέρασμα. Ας σημειώσουμε σε αυτό το σημείο ότι όλοι οι χρόνοι εκτέλεσης των αλγορίθμων που υλοποιήθηκαν σε *'MPI'* είναι λίγο μεγαλύτεροι από αυτούς που θα προέκυπταν εάν χρησιμοποιούσαμε το *'MPI'* σε ένα δίκτυο υπολογιστών.

5.3 Ο αλγόριθμος Merge Sort σε PTHREADS

Πριν αναλύσουμε τον αλγόριθμο *'mergesort'* να υπενθυμίσουμε ότι η βιβλιοθήκη *'pthreads'* χρησιμοποιεί τα νήματα –threads– του επεξεργαστή και άρα την μνήμη ενός υπολογιστικού συστήματος. Αυτό αυτόματα σημαίνει ότι χρησιμοποιεί την κοινόχρηστη μνήμη του συστήματος *('shared memory systems'*) για να είναι σε θέση να λειτουργήσουν όλα τα νήματα. Οι μετρήσεις έγιναν σε ένα τέτοιο σύστημα, άρα μπορούμε να πούμε πως η βιβλιοθήκη *'pthreads'* του ταιριάζει καλύτερα.

Ας ξεκινήσουμε όμως με τον αλγόριθμο *'Merge Sort'*:

5.3.1 Εισαγωγή - Περιγραφή του αλγορίθμου

Ο αλγόριθμος *'Merge Sort'* είναι ένας αλγόριθμος τύπου *'διαίρει και βασίλευε'* ο οποίος αναπτύχθηκε για πρώτη φορά το 1945 από τον *John von Neumann* και η μέση πολυπλοκότητά του είναι O(nlogn). Αυτό που ουσιαστικά κάνει είναι να διαιρεί μία μη ταξινομημένη λίστα σε υπολίστες μέχρι η κάθε υπολίστα να περιέχει έναν στοιχείο. Στην συνέχεια, συγχωνεύει αυτές τις υπολίστες σε ταξινομημένες υπολίστες μέχρι τελικά να μείνει μία η οποία θα αποτελεί και την ταξινομημένη λίστα.

5.3.2 Ψευδοκώδικας

Η υλοποίηση του αλγορίθμου *'Merge Sort'* με παράλληλο τρόπο έχει πολλές ομοιότητες με την κλασσική υλοποίηση του αλγορίθμου σε σειριακή μορφή. Η μόνη διαφορά είναι ότι οι συναρτήσεις διάσπασης κάθε φορά του πίνακα σε δύο μικρότερους πίνακες αντικαθιστώνται από τα νήματα τα οποία δημιουργούμε.

Ο ψευδοκώδικας παρακάτω μας φανερώνει του λόγου το αληθές:

Αρχικά δηλώνουμε την δομή NODE με τον εξής τρόπο:

```
typedef struct node {
int i;
int j;
} NODE;
```

Στην συνέχεια θα παρουσιάσουμε σε ψευδοκώδικα το κομμάτι εκείνο που παραλληλοποιήσαμε:

```
void *mergeSort (void *a)
{
        NODE *p = (NODE *)a;
        NODE n1, n2;
        int my_i = p->i;  // διαφορετικό το my_i του νήματος με βαθμό 1 από νήμα με βαθμό 2
        int my_j = p->j;  // διαφορετικό το my_j του νήματος με βαθμό 1 από νήμα με βαθμό 2

        int mid = (my_i + my_j) / 2;
        pthread_t thread1, thread2;

        n1.i = my_i;
        n1.j = mid;

        n2.i = mid+1;
        n2.j = my_j;

        if (my_i < my_j)
        {
                pthread_create(thread1, mergeList, &n1);
                pthread_create(thread2, mergeList, &n2);

                pthread_join(thread1, NULL);
                pthread_join(thread2, NULL);
```

```
                merge(my_i, my_j);
                thread exit();
        }
        retutn 0;
}
```

5.3.3 Θεωρητική ανάλυση τμημάτων αλγορίθμου που θα παραλληλοποιηθούν

Πριν ασχοληθούμε με το κομμάτι του αλγορίθμου που θα παραλληλοποιήσουμε πρέπει να αναφέρουμε πως – συνήθως - υπάρχουν δύο τρόποι να υλοποιηθούν είτε σειριακά είτε παράλληλα κάποιοι αλγόριθμοι.

1. Με αναδρομή
2. Χωρίς αναδρομή

Εμείς για να δείξουμε και τους δύο αυτούς τρόπους επιλέξαμε σε κάποιους να χρησιμοποιήσουμε αναδρομή και σε κάποιους άλλους όχι. Ο *'MergeSort'* σε αυτό το παράδειγμα κατασκευάστηκε με την μέθοδο της αναδρομής. Τα κομμάτια που θα παραλληλοποιηθούν είναι τα κλαδιά της αναδρομής καθώς τεμαχίζουμε τον πίνακα σε μικρότερα μέρη ώστε μετά να γίνει η τελική συγχώνευση και ταξινόμηση. Για αυτό το σκοπό χρειαζόμαστε δύο λειτουργίες. Η πρώτη είναι η λειτουργία τεμαχισμού σε υποδιαιρέσεις του μήκους του πίνακα μέχρι αυτός να γίνει ένα και η δεύτερη είναι μόλις τελειώσει αυτός ο τεμαχισμός ή ταξινόμησή του με την μέθοδο της συγχώνευσης (merge).

Πλεονεκτήματα αναδρομής

Το κυριότερο πλεονέκτημα της αναδρομής είναι η ευκολία στην χρήση της και στην κατασκευή (ακόμη και σε παράλληλο πρόγραμμα) εάν φυσικά γίνει κατανοητός ο τρόπος εφαρμογής της.

Επίσης, ένα άλλο πλεονέκτημα είναι η μικρή έκταση του κώδικα που καταλαμβάνει για να υλοποιηθεί.

Μειονεκτήματα αναδρομής

Το κυριότερο μειονέκτημα του είναι η χρήση λίγων νημάτων. Για κάθε αναδρομή δύο νέα νήματα δημιουργούνται και επειδή ο συνόλικος αριθμός των νημάτων είναι πεπερασμένος θα υπάρξει περιορισμός στο πόσα νήματα τελικά θα χρησιμοποιηθούν και άρα πόσοι αριθμοί θα ταξινομηθούν. Το δικό μας υπολογιστικό σύστημα αποτελείται σύμφωνα με τον κατασκευαστή του από 8 πυρήνες και θεωρητικά μπορεί να παράξει μέχρι και 16 νήματα (2νήματα/1πυρήνα). Στην πράξη όμως θα δούμε πως μόνο μέχρι 8 νήματα μπορούμε να θέσουμε σε λειτουργία άρα η αντιστοιχία είναι 1νήμα/1πυρήνα. Το συμπέρασμα που βγαίνει από τα παραπάνω είναι πως η ταξινόμηση που μπορεί να συμβεί με τον αλγόριθμο 'mergesort' στο υπολογιστικό μας σύστημα είναι περιορισμένη μέχρι τους 256 αριθμούς δηλαδή 2^8. Για να είμαστε περισσότερο ακριβείς ο αλγόριθμος μπορεί να ταξινομήσει λίγο πάνω από τους 265 αριθμούς σωστά.

5.3.4 Η παραλληλοποίηση του Merge Sort

Ήδη κάναμε φανερό πιο κομμάτι αυτού του αλγορίθμου θα παραλληλοποιηθεί. Αυτό είναι ο τεμαχισμός του πίνακα κάθε φορά σε δύο ίσους υποπίνακες όπου νέος_πίνακας = παλιός_πίνακας / 2. Ας δούμε πως θα γίνει αυτό με την βιβλιοθήκη *'pthreads'*.

Όπως και στην βιβλιοθήκη *'MPI'* έτσι και εδώ θα θεωρήσουμε πως τα βασικά του προγραμματισμού σε C όπως δηλώσεις μεταβλητών, συναρτήσεων και βιβλιοθηκών τα γνωρίζει ο αναγνώστης και έτσι θα αναφερθούμε μόνο σε αυτές που θα χρησιμοποιήσει η *'pthreads'*.

Η πρώτη γραμμή του κώδικα που μας ενδιαφέρει είναι λοιπόν η δήλωση της νέας βιβλιοθήκης. Αυτό γίνεται φυσικά με τον ίδιο τρόπο που γίνονται όλες οι δηλώσεις.

```
#include<pthread.h>
```

Στη συνέχεια ακολουθούν δηλώσεις πινάκων, μεταβλητών, αρχικοποιήσεις κλπ. Μέσα σε αυτές τις δηλώσεις θα πρέπει εμείς να ορίσουμε πόσα νήματα θα δημιουργεί κάθε φορά στην αναδρομή το πρόγραμμα και αυτό θα γίνει με την εξής γραμμή εντολής:

```
#define NUMTHREADS 2
```

Δύο νήματα διαχειρίζονται κάθε φορά το περιεχόμενο της υπολίστας που δημιουργείται μετά από τον διαχωρισμό στα δύο της προηγούμενης υπολίστας. Το ένα νήμα θα κρατάει τους αριθμούς του πίνακα από τον πρώτο μέχρι και τον μεσαίο. Το άλλο νήμα θα κρατάει τους αριθμούς του πίνακα από τον μεσαίο συν ένα αριθμό (mid+1) μέχρι και τον τελευταίο.

Η επόμενη συνάρτηση είναι αυτή της δημιουργίας τυχαίων αριθμών', στην συνέχεια ακολουθεί αυτή της συγχώνευσης των αριθμών όπου γίνεται και η τελική ταξινόμηση και τέλος πριν την συνάρτηση *'main()'* παρουσιάζεται η *'mergeSort()'* η οποία είναι και αυτή που την μετατρέψαμε σε παράλληλη μορφή και θα παρουσιάσουμε.

Η συνάρτηση merge()

Ο κύριος σκοπός της συνάρτησης *'merge()'* είναι, αφού κατατμηθεί ο πίνακας μέχρι να σχηματιστούν ν πίνακες με μέγεθος ένα, να τους ταξινομήσει και συγχωνεύσει όλους ξανά σε έναν. Η συγχώνευση είναι γνωστή λειτουργία και όρος στο πεδίο της πληροφορικής και ειδικότερα του προγραμματισμού και η λογική της είναι η εξής:

Αφού έστω ένας πίνακας με 8 αριθμούς υποδιαιρεθεί σε 8 πίνακες μεγέθους ένα τότε συγχώνευση *'merge'* ονομάζουμε την προσάρτηση όλων αυτών των πινάκων μεταξύ τους σε έναν ενιαίο και με ταξινομημένη αύξουσα ή φθίνουσα την σειρά των αριθμών που περιέχει. Αυτό συμβαίνει συγκρίνοντας κάθε στοιχείο μίας λίστας – πίνακα με όλα τα υπόλοιπα από το διπλανό του πίνακα και όσα μικρότερα από τον εαυτό του βρίσκει τα τοποθετεί σε νέα λίστα έως ότου να βρεθεί κάποιο μεγαλύτερο για να τοποθετηθεί αυτό και να συνεχίσουν οι συγκρίσεις με το αμέσως διπλανό του αριθμό.

Πως εκκινούμε τα νήματα

Η εκκίνηση των νημάτων πάντοτε συμβαίνει από την συνάρτηση main(). Εκεί δηλώνουμε ότι θα χρησιμοποιήσουμε νήματα, σε αυτό το σημείο δηλώνουμε και πόσα νήματα θα θέσουμε σε λειτουργία. Πιο ολοκληρωμένα έχουμε:

```
int main()
{
        generate_random_numbers();
        int i;
        NODE m;
        m.i = 0;
        m.j = MAX-1;
        pthread_t threads;
```

```
        pthread_create(&threads, NULL, mergeList, &m);
        pthread_join (threads, NULL);

        for(i = 0; I < MAX; i++)
        {
                printf("%d", a[i]);
        }
        printf("\n");
}
```

Η πρώτη γραμμή είναι γνωστή και καλεί την συνάρτηση *'main()'*. Η πρώτη συνάρτηση που καλείται είναι αυτή της γέννησης τυχαίων αριθμών. Στο συγκεκριμένο παράδειγμα 256, τα οποία τα έχουμε ορίσει κάτω από τις δηλώσεις των βιβλιοθηκών που θα χρησιμοποιήσουμε με την εντολή

```
#define MAX 256
```

Στην συνέχεια ακολουθούν οι δηλώσεις μεταβλητών και μίας δομής *'structure'* με την ονομασία *'struct1'*. Μέσα στο σώμα αυτής της δομής υπάρχουν δύο μεταβλητές η i και η j στις οποίες θα ανατεθούν η αρχή και το τέλος του τμήματος της λίστας που θα ταξινομηθεί σε κάθε πεδίο. Αυτή η δομή έχει δηλωθεί νωρίτερα με τον τύπο *'struct1'*.

Η επόμενη δήλωση είναι αυτή που μας ενδιαφέρει κυρίως. Το *'pthread_t'* είναι ένας τύπος δεδομένων που ορίζει τα νήματα και βρίσκεται στην *'pthread.h'* Η αμέσως επόμενη γραμμή είναι αυτή η οποία ευθύνεται για την έναρξη των νημάτων. Σε αυτό το παράδειγμα του αλγορίθμου *'mergesort'* θα δημιουργήσουμε ένα νήμα το οποίο με την σειρά του θα δημιουργήσει άλλα δύο για τον τεμαχισμό του πίνακα. Το καθένα από αυτά τα δύο θα δημιουργήσει άλλα δύο και αυτό θα συμβαίνει μέχρι ο πίνακας να μην τεμαχίζεται άλλο, δηλαδή να περιέχει μόνο ένα στοιχείο. Η *'pthread_join()'* όπως έχει ήδη προαναφερθεί είναι η υπεύθυνη συνάρτηση για να αναστείλει την εκτέλεση του καλούμενου νήματος.

Τέλος, ο βρόγχος for(){} θα εκτυπώσει στην οθόνη τον πίνακα ο οποίος φυσικά σε αυτή τη φάση θα είναι ταξινομημένος.

Από όλα τα προηγούμενα, πρέπει να μείνουμε στο σημείο της συνάρτησης *'pthread_create()'*. Αυτή η συνάρτηση θα δημιουργήσει ένα νήμα το οποίο θα τρέξει την συνάρτηση *'mergeSort()'* και θα πάρει σαν όρισμα την τον πίνακα m ο οποίος έχει σαν τύπο δεδομένων την δομή δεδομένων *'struct1'* και δεδομένα το *'i'* και το *'j'*.

Συνοψίζοντας έως τώρα έχουμε δει πως δημιουργούνται τα νήματα και πως δηλώνονται αυτά στην αρχή του προγράμματος. Το επόμενο τμήμα που θα αναλυθεί είναι αυτό του τεμαχισμού και ουσιαστικά του σημείου που έχουμε παραλληλοποιήσει στο πρόγραμμα.

```
void *mergeSort(void *a)
{
        NODE *p = (NODE *)a;
        NODE n1, n2;

        int my_i = p->i;
        int my_j = p->j;

        int mid = (my_i + my_j) / 2;
        pthread_t thread1, thread2;

        n1.i = my_i;
        n1.j = mid;

        n2.i = mid+1;
```

```
        n2.j = my_j;

        if (my_i < my_j)
        {
                pthread_create(&thread1, NULL, mergeSort, &n1);
                pthread_create(&thread2, NULL, mergeSort, &n2);

                pthread_join(thread1, NULL);
                pthread_join(thread2, NULL);

                merge(my_i, my_j);
                pthread_exit(NULL);

        }

        return 0;
}
```

Με μία γρήγορη ματιά είναι εύκολο να καταλάβουμε την λειτουργία της κάθε γραμμής του κώδικα. Ας τα δούμε όμως λίγο πιο αναλυτικά. Το νόημα σε αυτή τη συνάρτηση είναι να δέχεται έναν πίνακα ως είσοδο, να τον σπάει σε δύο επιμέρους πίνακες με ίσα στοιχεία, να δημιουργεί δύο νέα νήματα και σε κάθε ένα από αυτό να δίνει από ένα πίνακα ώστε να επιτελεστεί η ίδια λειτουργία ξανά και ξανά.

Η πρώτη γραμμή είναι η δήλωση του τύπου της συνάρτησης και το όνομα αυτής. Στο σώμα της, το πρώτο που βλέπουμε είναι η δήλωση των δεδομένων από την δομή καθώς και την ανάθεση τους σε μεταβλητές. Η μεταβλητή *'mid'* είναι αυτή η μεταβλητή στην οποία βρίσκεται το μεσαίο στοιχείο του πίνακα και είναι επιπλέον αυτή την οποία χρησιμοποιούμε στον έλεγχο της περεταίρω διάσπασης του πίνακα. Στην συνέχεια ακολουθεί η δήλωση ότι θα δημιουργηθούν δύο νήματα - το *'thread1'*και το *'thread2'* -με τύπο δεδομένων*'pthread_t'*. Στην συνέχεια δημιουργούμε δύο νέες δομές δεδομένων*'n1'* και *'n2'* οι οποίες έχουν ως δεδομένα την πρώτη και την τελευταία θέση του κάθε πίνακα και οι οποίες θα περαστούν σαν ορίσματα στις συναρτήσεις που θα 'τρέξουν' τα δύο νέα νήματα που θα δημιουργηθούν στην συνέχεια.

Ακολούθως, ο επόμενος έλεγχος είναι και αυτός που θα διαμελίσει τους πίνακες σε δύο επιμέρους με την μισή χωρητικότητα. Αυτή η διάσπαση θα συμβαίνει μόνο όσο η θέση του τελευταίου στοιχείου έχει μεγαλύτερο βαθμό από την θέση του πρώτου στοιχείου. Η διάσπαση είπαμε πριν ότι θα γίνεται δημιουργώντας δύο νήματα και αυτό είναι που συμβαίνει στις πρώτες τέσσερις γραμμές του ελέγχου *'if'*. Συνεχίζοντας παρακάτω και αφού ολοκληρωθεί η διαδικασία διαμέλισης ακολουθεί η συγχώνευση των επιμέρους πινάκων σε έναν ταξινομημένο με την συνάρτηση *'merge()'*. Θυμίζουμε πως η συνάρτηση αυτή έχει περιγραφεί νωρίτερα. Τέλος, η τελευταία γραμμή του ελέγχου είναι η *'pthread_exit(NULL)'* η οποία όπως είδαμε στην περιγραφή της βιβλιοθήκης *'pthreads'* τερματίζει το καλούμενο νήμα.

5.3.5 Η μέτρηση του χρόνου εκτέλεσης

Οι αριθμοί που θα ταξινομήσουμε θα πρέπει να φτάνουν μέχρι τους 265. Εμείς θα συγκρίνουμε την συμπεριφορά του παράλληλου προγράμματος ταξινομώντας 64, 128 και 256 αριθμούς. Ο σημαντικότερος παράγοντας σε αυτή τη προσπάθεια καταμέτρησης του χρόνου είναι το μικρό πλήθος των αριθμών. Όταν το πλήθος των αριθμών είναι αυτής της τάξης τότε τα αποτελέσματα είναι παραπλανητικά. Είναι τόσο μικρός ο χρόνος εκτέλεσης που η πιθανότητα να μην υπάρχει ουσιαστικά διαφορά από την μία ομάδα αριθμών με τις άλλες είναι πολύ πιθανόν. Ειδικά αν αναλογιστούμε πως ένας υπολογιστής συνεχώς 'τρέχει' διάφορες διεργασίες και υπηρεσίες στο παρασκήνιό του. Παρ' όλα αυτά - και έπειτα από 20 διαδοχικές εφαρμογές του προγράμματος καθώς και αποθήκευση του γρηγορότερου χρόνου για την κάθε περίπτωση – μπορούμε να εξάγουμε κάποια συμπεράσματα. Όταν οι αριθμοί ήταν 64 ο χρόνος ταξινόμησής τους ήταν πολύ μικρός (σχετικά μηδαμινός). Στην περίπτωση που το πλήθος των αριθμών ήταν 128 ο

χρόνος σε σχέση με την προηγούμενη ομάδα ήταν πολύ κοντά (με μια μικρή απόκλιση) στην διπλάσια τιμή του.

Τέλος, στο πλήθος των 256 αριθμών η διαφορά σε σχέση με την προηγούμενή του ομάδα σύγκρισης ήταν αρκετά εμφανής. Τις 18 από τις 20 φορές ήταν υπερδιπλάσια η τιμή του και μόνο τις 2 ήταν μικρότερη από διπλάσια. Ας δούμε όμως συγκεντρωμένα όλα τα αποτελέσματα

Νήματα/Πλ. αριθμων	64	128	256
64	0.000006		
128		0.000012	
256			0.000029

Πίνακας 8

Πίνακας χρόνων εκτέλεσης του αλγορίθμου Merge Sort για 2,4 και 8 νήματα

5.3.6 Σύγκριση με σειριακό αλγόριθμο, υπολογισμός επιτάχυνσης - απόδοσης

Για τον αλγόριθμο *'MergeSort'* και συγκεκριμένα για την υλοποίησή του με την μέθοδο της αναδρομής είναι αδύνατο να βρούμε την απόδοσή του σε σχέση με την λειτουργία ενός σειριακού προγράμματος. Ο λόγος για τον οποίο συμβαίνει αυτό είναι το μικρό πλήθος των αριθμών που συγκρίναμε. Είναι τόσο μικροί οι χρόνοι που και σε έναν σειριακό αλγόριθμο δεν θα διαφέρουν τόσο πολύ ώστε να μπορέσουμε να καταλάβουμε που ωφείλεται αυτή η διαφορά. Όπως έχουμε ήδη αναφέρει αυτή η διαφορά σε μια χρονομέτρηση σε τόσο χαμηλά επίπεδα ταξινόμησης μπορεί να οφείλεται σε μία διεργασία η οποία δεν έχει σχέση με το πρόγραμμα. Ωστόσω, σε όλους τους υπόλοιπους αλγορίθμους θα καταμετρηθεί η απόδοση και τα συμπεράσματα που θα βγουν θα είναι πολύ χρήσιμα.

5.3.7 Τελικό συμπέρασμα

Το παράλληλο πρόγραμμα που υλοποιήσαμε με βάση τον αλγόριθμο *'MergeSort'* συμπεριφέρεται έτσι όπως περιμέναμε να συμπεριφερθεί. Ακόμη και αν το πλήθος των αριθμών που συγκρίναμε ήταν πολύ μικρό, η διαφορά στις χρονομετρήσεις ήταν εμφανής. Το ιδανικότερο σε αυτή την περίπτωση θα ήταν να αναφέρουμε μία διαφορετική υλοποίηση η οποία θα κάλυπτε ένα μεγαλύτερο πλήθος αριθμών όμως, επειδή το καταφέραμε στους επόμενους δύο αλγορίθμους (Quicksort και RadixSort) θεωρήσαμε σωστό να καλύψουμε και μία παράλληλη αναδρομή.

5.4 Ο αλγόριθμος Quicksort

5.4.1 Εισαγωγή - Περιγραφή του αλγορίθμου

Ο *'quicksort'* είναι ο πιο κοινός αλγόριθμος ταξινόμησης. Αναπτύχθηκε από τον *'Tony Hoare'* το 1960 και κατά μέσο όρο χρειάζεται *'n*logn'* συγκρίσεις για n αριθμούς. Όπως φανερώνει και το όνομά του είναι ένας γρήγορος αλγόριθμος και υπο προϋποθέσεις, γρηγορότερος από όλους τους υπόλοιπους που έχουν την ίδια πολυπλοκότητα με αυτόν. Η φήμη που ακολουθεί τον συγκεκριμένο αλγόριθμο είναι πολύ μεγάλη. Τον συναντούμε σε *'Unix'* συστήματα σαν μία εξ' ορισμού βιβλιοθήκη για ταξινόμηση αριθμών καθώς και στις βιβλιοθήκες της C γλώσσας προγραμματισμού ως *'qsort()'* συνάρτηση.

Ο δικός μας αλγόριθμος είναι μία παραλλαγή του αρχικού *'Quicksort'* πιο απλός στην υλοποίηση αλλά με μεγαλύτερη πολυπλοκότητα(*'n^2*logn'*) και πιο αργός στους χρόνους εκτέλεσης.

Σε αυτό το παράδειγμα αρχικά ταξινομήσαμε λίστες με πλήθος 1000, 2000 και 4000 αριθμών και χρησιμοποιήσαμε 2, 4 και 8 νήματα για κάθε λίστα. Σε αυτές τις περιπτώσεις οι χρόνοι που πήραμε για κάθε λίστα αυξάνονταν όσο αυξάνονταν και τα νήματα. Όταν όμως ταξινομήσαμε πολύ μεγάλο πλήθος αριθμών συνέβαινε το ακριβώς αντίστροφο. Για κάθε λίστα ταξινόμησης όσο αυξάναμε τα νήματα τόσο η ταξινόμηση ήταν γρηγορότερη. Πιο συγκεκριμένα με την χρήση 2, 4 και 8 νημάτων ταξινομήσαμε λίστες πλήθους 120000, 240000 και 480000 αριθμών.

Όπως ο αλγόριθμος *'MergeSort'* έτσι και ο *'Quicksort'* είναι ένας αλγόριθμος που χρησιμοποιεί την μέθοδο 'διαίρει και βασίλευε'. Η διαφορά του με τον *'MergeSort'* είναι ότι ο *'Quicksort'* δεν χωρίζει τον πίνακα σε δύο ίσους 'υποπίνακες' αλλά σε δύο πίνακες όπου ο ένας περιέχει μικρότερους αριθμούς από κάποιον τυχαίο αριθμό του πίνακα που εμείς ή το πρόγραμμα έχουμε επιλέξει και ο άλλος μεγαλύτερους. Αυτός ο αριθμός που χωρίζει κάθε φορά τον πίνακα σε δύο μικρότερους πίνακες ονομάζεται *'pivot'* και με το πέρας της διάσπασης αυτός ο αριθμός τοποθετείται στον πίνακα που είναι προς ταξινόμηση στην σωστή του θέση. Για παράδειγμα αν έχουμε 4 αριθμούς (3,1,7,4) και επιλέξουμε το 3 σαν *'pivot'* τότε με το πέρας της διαδικασίας θα προκύψει ο προσωρινός πίνακας 1,3,7,4 και το 3 αμέσως θα μετατεθεί στον τελικό πίνακα στην σωστή του θέση δηλαδή στην θέση 2. Μετά από αυτό θα δημιουργηθούν δύο πίνακες ο πρώτος θα περιέχει τον αριθμό 1 και ο δεύτερος τους αριθμούς 7 και 3. Ο αριθμός 1 θεωρείται ταξινομημένος και παίρνει την θέση του στον τελικό πίνακα στην θέση νούμερο 1 ενώ παράλληλα στον δεύτερο πίνακα όποιον αριθμό και να επιλέξουμε για *'pivot'* μετά την πρώτη σύγκριση θα ανταλλάξουν και αυτά θέσεις και εν τέλει θα τοποθετηθούν και αυτά στην σωστή τους θέση στον τελικό πίνακα.

Ο αλγόριθμος που παραλληλοποιήσαμε είναι μία παραλλαγή του *'Quicksort'* πιο απλός και ευκολότερα 'παραλληλοποιήσιμος'. Η λειτουργίες που ουσιαστικά επιτελούνται στο σώμα του αλγορίθμου είναι η επιλογή του pivot, η σύγκριση με τους αριθμούς τους οποίους έχει αναλάβει κάθε νήμα να συγκρίνει, μία κοινή μεταβλητή η οποία θα παίρνει ως τιμή το άθροισμα των θέσεων του πίνακα των οποίων οι αριθμοί είναι μικρότεροι του *'pivot'* και τέλος η ταξινόμηση του πίνακα.Τέλος, η πολυπλοκότητα αυτού του αλγορίθμου είναι μεγαλύτερη από του *'quicksort'.*

Κατά την υλοποίηση, παρατηρήσαμε πως ο αλγόριθμος ήταν λίγο πιο περίπλοκος από τους προηγούμενους που είδαμε και πιο συγκεκριμένα τον *'MergeSort'*.

5.4.2 Ανάλυση του αλγορίθμου

Ουσιαστικά αυτή την φορά είναι ένα κομμάτι το οποίο θα το κάνουμε να τρέχει παράλληλα σε όλα τα νήματα που θα θέσουμε σε λειτουργία. Άρα η συνάρτηση *'quicksort()'* που είναι και το μεγαλύτερο μέρος του προγράμματος θα γραφεί εξ' ολοκλήρου παράλληλα και θα εξεταστεί η συμπεριφορά της.

Όλα τα νήματα θα διαβάζουν το *'pivot'* και θα το συγκρίνουν με τους αριθμούς της υπολίστας που έχουν αναλάβει να διαχειριστούν. Κάθε φορά που θα συναντούν έναν μικρότερο αριθμό από το *'pivot'* θα αυξάνουν μία τοπική μεταβλητή. Αφού διατρέξουν όλους τους αριθμούς της υπολίστας τους θα προσθέσουν την τοπική μεταβλητή σε μία μεταβλητή κοινής ορατότητας. Τέλος, το *'pivot'* θα αποθηκευτεί στον ταξινομημένο πίνακα και στην θέση με αριθμό την τιμή της κοινής μεταβλητής.

5.4.3 Ψευδοκώδικας

Παρακάτω στον ψευδοκώδικα φαίνεται καθαρά η λειτουργία του αλγορίθμου. Αρχικά, επιλέγεται το πρώτο στοιχείο του πίνακα ως pivot και αυτό συγκρίνεται με όλα τα υπόλοιπα και κάθε φορά που βρίσκει ένα μικρότερο στοιχείο προσθέτει 1 στην μεταβλητή *'local_left_sum'*. Στην συνέχεια όλες οι μεταβλητές *'local_left_sum'* αθροίζονται στην κοινή μεταβλητή *'global_left_sum'.* Τέλος, εάν στη θέση του τελικού πίνακα στην οποία προοριζόταν να τοποθετηθεί το *'pivot'* υπάρχει

κάποιο άλλο στοιχείο τότε αυτό ολισθένει μία θέση προς τα δεξιά μέχρι κάποια θέση του πίνακα να είναι κενή.

```
quicksort()
{
        for(i = 0; i < MAX; i++)
        {
                pivot = a[i]
                synchronize threads()
                for each thread
                {
                        for (i=thread_local_first_value; i=thread_local_last_value; i++)
                                if (i < pivot)
                                        local_left_sum++
                }
                pthread_mutex_lock()
                        global_left_sum = global_left_sum + local_left_sum
                pthread_mutex_unlock()

                synchronize threads()
                if (i am the first thread)
                        while(orderedArray[global_left_sum] == pivot)
                                global_left_sum = global_left_sum++
                        orderedArray[global_left_sum] = pivot
                synchronize threads()
                }
        }
}
```

5.4.4 Η παραλληλοποίηση του Quicksort

Η αρχή είναι ίδια σε κάθε πρόγραμμα. Δήλωση βιβλιοθηκών οι οποίες θα μας δώσουν τις συναρτήσεις που θα χρειαστούμε στην πορεία.

Στην συνέχεια θα χρειαστούμε έναν πίνακα ο οποίος θα είναι ο αρχικός και ορατός από όλα τα νήματα και θα περιέχει τους αριθμούς τους οποίους θα ταξινομήσουμε. Η δήλωσή του έγινε με την μέθοδο του δείκτη *'pointer'* η οποία μας εξασφάλισε μικρότερο και κομψότερο κώδικα όπως επίσης δυναμικότητα στον τρόπο συμπλήρωσης του και περιορισμό στην σπατάλη της μνήμης. Το σύνολο των θέσεων του πίνακα προς ταξινόμηση αποτυπώνεται στην μεταβλητή MAX και όπως αναφέραμε θα παίρνει τις τιμές 120000, 240000, 480000 για δύο, τέσσερα και οκτώ νήματα. Από αυτόν τον πίνακα θα επιλέγεται και το νούμερο το οποίο θα αναλαμβάνει τον ρόλο του *'pivot'*.

Ας δούμε όλες τις δηλώσεις έτσι ακριβώς όπως τις έχουμε αναφέρει στο πρόγραμμα και με μορφή σχολίων θα επεξηγούμε όπου χρειάζεται την χρήση της κάθε μίας.

```
int *A;                 // δήλωση του αρχικού πίνακα που περιέχει τους αριθμούς προς ταξινόμηση
int *orderedArray; // δήλωση του τελικού πίνακα που περιέχει τους ταξινομημένους αριθμούς

pthread_mutex_t lock            // δήλωση του αμοιβαίου αποκλεισμού από μία μεταβλητή
pthread_barrier_t our_barrier;  // δήλωση του φράγματος το οποίο θα χρησιμοποιήσουμε
                                   προκειμένου να συγχρονιστούν όλα τα νήματα
```

```
struct thread_data                                  /***************************/
{                                                   /*    ο βαθμός του      */
        int thread_id;                              /*    κάθε νήματος      */
};                                                  /*                      */
struct thread_data thread_data_array[NUMTHREADS];   /***************************/

void* quicksort(void *rank)             // έναρξη συνάρτησης quicksort που χρησιμοποιεί κάθε
νήμα
{                                       // έναρξη σώματος συνάρτησης
        int my_first;           // η πρώτη θέση του πίνακα του κάθε νήματος
        int my_last;            // η τελευταία θέση του πίνακα του κάθε νήματος
        int local_offset;       // το πλήθος των αριθμών που θα ελέγχει κάθε νήμα
        int my_rank;            // το αναγνωριστικό του νήματος
        int i,j,q;              // διάφορες μεταβλητές που χρειαζόμαστε
        int pivot;              // το ενδιάμεσο σημείο κατά το οποίο θα διασπαστεί ο πίνακας
        int my_left;            // τοπική μεταβλητή που μετράει τους μικρότερους αριθμούς

        struct thread_data *my_data;// η δομή δεδομένων που παίρνει τα δεδομένα από την
main()
        my_data = (struct thread_data *) rank;    // πέρασμα των δεδομένων στο my_data
        my_rank = my_data->thread_id; // πέρασμα τιμής από την δομή στην μεταβλητή
                                        my_rank

        local_offset = MAX/NUMTHREADS;      // ανάθεση μήκους ταξινόμησης σε κάθε νήμα
        my_first = my_rank * local_offset;          // ανάθεση σημείου εκκίνσης πίνακα σε κάθε
                                                      νήμα
        my_last = ((my_rank+1)*local_offset)–1  // αντίστοιχα ανάθεση σημείου τερματισμού

        for (i=0; i<MAX; i++)                       // διατρέχουμε όλα τα στοιχεία του πίνακα A
        {                                           // έναρξη σώματος βρόγχου for()
                pivot = A[i];                       // ανάθεση τιμής στην μεταβλητή pivot
                for (j=my_first; j<=my_last; j++)  // διατρέχουμε τους αριθμούς που ανήκουν σε
                                                      κάθε νήμα
                {                                   // έναρξη σώματος βρόγχου for()
                        if(A[j]<pivot) //έλεγχος στοιχείου που διατρέχουμε αν είναι μικρότερο
                                        από pivot
                        {                           // έναρξη σώματος ελέγχου if()
                                my_left_sum =my_left_sum+1;// αυξηση τοπικής μεταβλητής
                                                              κατά ένα
                        }                                   // λήξη σώματος ελέγχου if()
                }                           // λήξη σώματος βρόγχου for()
                pthread_mutex_lock(&lock);      // έναρξη αμοιβαίου αποκλεισμού
                        left_sum = left_sum + my_left_sum;// αύξηση κοινόχρηστης μεταβλητής
                pthread_mutex_unlock(&lock);   // λήξη αμοιβαίου αποκλεισμού
                pthread_barrier_wait(&our_barrier);             // συγχρονισμός των νημάτων
                if (my_rank == 0)       //ταξινόμηση θα γίνει από το ένα νήμα (νήμα με βαθμό 0)
```

```
                {                       // έναρξη σώματος ελέγχου if()
                        while (orderedArray[left_sum] == pivot)   // όσο βρίσκει στη θέση που
                                                                     θέλει να ταξινομήσει ίσο
                                                                     αριθμό με pivot
                        {                                   // έναρξη σώματος βρόγχου while ()
                                left_sum = left_sum+1;              // αύξησε την θέση κατά μία
                        }                                   // λήξη σώματος βρόγχου while()
                        orderedArray[left_sum] = pivot;   // ανάθεση τιμής του pivot στην σωστή
                                                             θέση του τελικού πίνακα
                }                                           // λήξη σώματος ελέγχουif()
                pthread_barrier_wait(&our_barrier);                 // συγχρονισμός των νημάτων
                left_sum = 0;                       // μηδενισμός της left_sum μέσα στο βρόγχο
        }                                           // λήξη σώματος βρόγχου for()
}                                                   // λήξη σώματος συνάρτησης

int main(int argc, char *argv[])    // εκκίνηση της κύριας συνάρτησης του προγράμματος (main())
{
        int i;
        pthread_barrier_init(&our_barrier, NULL, NUMTHREADS);  // αρχικοποίηση φράγματος

        orderedArray = (int *) malloc(MAX * sizeof(int*));// δέσμευση μνήμης για orderedArray
        A = (int *) malloc(MAX * sizeof(int*));             // δέσμευση μνήμης για αρχικό πίνακα A

        generateRandom();  // η συνάρτηση που χρησιμοποιούμε για να αναθέσει τυχαίους
                              αριθμούς προς ταξινόμηση
        pthread_t* threads;                         // ορισμός των νημάτων
        pthread_mutex_init(&lock, NULL);          //αρχικοποίηση αμοιβαίου αποκλεισμού

        begin = clock();       // παίρνουμε την ώρα την στιγμή που καλείται η συνάρτηση clock()
        for(i=0; i<NUMTHREADS; i++)   // ξεκινάμε να δημιουργούμε τα νήματα
        {
                thread_data_array[i].thread_id = i;             // ανάθεση τιμής στο thread_id
                threads=malloc(i*sizeof(pthread_t));      //δέσμευση μνήμης για δημιουργία
                                                            νημάτων
                pthread_create(&threads[i], NULL, quicksort, (void *) &thread_data_array[i]);
        }
        for(i=0; i<NUMTHREADS; i++)
        {
                pthread_join(threads[i], NULL);
        }
        end = clock();              // παίρνουμε την ώρα την στιγμή που καλείται η συνάρτηση
                                       clock()
        time_spent = (double) (end-begin) / CLOCKS_PER_SEC; // η μεταβλητή που κρατάει
                                                               τον χρόνο εκτέλεσης

        for(i=0; i<MAX; i++)
```

```
        {
                printf("%d", orderedArray[i];       // εμφάνιση του πίνακα ταξινομημένο
        }
        printf("\nTime taken: %f\n, time_spent"); // εμφάνιση του χρόνου εκτέλεσης του
                                                    παράλληλου τμήματος του προγράμματος
        return 0;                                   // έξοδος από πρόγραμμα
}
```

Έχοντας ήδη περιγράψει τι κάνει ο συγκεκριμένος αλγόριθμος, σε αυτό το σημείο θα επικεντρωθούμε στα κομμάτια τα οποία μας προκάλεσαν πρόβλημα και έπρεπε να επιλύσουμε. Ουσιαστικά, αυτά τα κομμάτια ήταν δύο. Το πρώτο πρόβλημα εντοπίστηκε όταν χρειάστηκε τα νήματα να ενημερώσουν την τιμή της μεταβλητής *'left_sum'* έτσι ώστε στην συνέχεια αυτή να τοποθετήσει στον τελικό πίνακα – orderedArray[] –το *'pivot'*. Το δεύτερο παρουσιάστηκε όταν έπρεπε να γίνει ο έλεγχος του *'pivot'* με την τιμή της θέσης του ταξινομημένου πίνακα στην οποία επρόκειτο να καταχωρηθεί.

Και τα δύο αυτά προβλήματα είναι εμφανή. Το κάθε νήμα για να αυξήσει την τιμή της μεταβλητής *'left_sum'* κατά ένα θα πρέπει να ελέγχει την δεδομένη στιγμή μόνο του την μεταβλητή. Σε διαφορετική περίπτωση μπορεί κάποιο άλλο νήμα να αλλάξει την τιμή της και όλα τα υπόλοιπα να μην ενημερωθούν για αυτή την αλλαγή με αποτέλεσμα η μεταβλητή να είναι λανθασμένη. Επίσης, το ίδιο πρόβλημα θα συναντήσουμε όταν η τιμή του *'pivot'* υπάρχει ήδη στην θέση που είναι για να τοποθετηθεί και εμείς θα κληθούμε να την μετακινήσουμε προς τα δεξιά κατά μία θέση. Το πρώτο πρόβλημα λύνεται με την χρήση του αμοιβαίου αποκλεισμού (mutex). Δηλαδή μόνο ένα νήμα την φορά θα κληθεί να επεξεργαστεί την τιμή της *'left_sum'*. Το δεύτερο πρόβλημα είναι λίγο πιο απλό. Δεν χρειάζεται όλα τα νήματα να κάνουν τον έλεγχο για το αν υπάρχει η τιμή του *'pivot'* στην θέση του πίνακα που είναι να ταξινομηθεί. Με έναν έλεγχο περιορίζουμε την εργασία αυτή στο πρώτο νήμα. Προϋπόθεση για να γίνει σωστά ο έλεγχος είναι να έχουν φτάσει στο σημείο του όλα τα νήματα έτσι ώστε το πρώτο νήμα να διαβάσει την σωστή τιμή της μεταβλητής. Αυτό το πετυχαίνουμε με τον συγχρονισμό των νημάτων χρησιμοποιώντας την γραμμή εντολής **pthread_barrier_wait(&our_barrier);**.

Τέλος, ο συγχρονισμός των νημάτων είναι απαραίτητος και σε άλλα σημεία. Κάθε φορά που επιλέγεται ένα καινούριο *'pivot'* πρέπει τα νήματα να είναι συγχρονισμένα. Σε διαφορετική περίπτωση θα βγουν λανθασμένα αποτελέσματα. Με την ίδια γραμμή εντολής που προαναφέραμε μπορούμε να προσπελάσουμε και αυτού του είδους τα προβλήματα.

5.4.5 Μέτρηση χρόνου εκτέλεσης

Υπενθυμίζουμε πως οι μετρήσεις έγιναν για 120000, 240000, 480000 αριθμούς με δύο, τέσσερα και οκτώ νήματα αντίστοιχα. Στον παρακάτω πίνακα μπορούμε να δούμε τι χρόνους εκτέλεσης πήραμε.

Νήμα/Πλ. αριθμών	120000	240000	480000
2	32.192	140.028	555.428
4	25.282	85.492	406.274
8	23.033	67.337	228.038

Πίνακας 9

Συγκεντρωτικός πίνακας με χρόνους εκτέλεσης σε seconds για 2, 4 και 8 νήματα της παραλλαγής του αλγορίθμου quicksort

Παρατηρούμε μία σημαντική μείωση του χρόνου ταξηνόμησης όσο προσθέτουμε νήματα για το ίδιο πλήθος αριθμών. Στο συγκεκριμένο παράδειγμα οι πρώτες συγκρίσεις που κάναμε για μέχρι 10 000 αριθμούς έδειξαν ότι όσο προσθέταμε νήματα ο χρόνος ταξινόμησης μεγάλωνε εξαιτείας του Toverhead που χρειάζονται αυτά για να επικοινωνήσουν. Από ένα σημείο και μετά

όμως αυξάνοντας το πλήθος των αριθμών των νημάτων το Toverhead άρχισε να μην γίνεται τόσο σημαντικό όσο το μέγεθος του πίνακα προς ταξινόμηση.

5.4.6 Σύγκριση με σειριακό αλγόριθμο, υπολογισμός επιτάχυνσης - απόδοσης

Ο *'Quicksort'* που υλοποιήσαμε όσο μένει σταθερό το μέγεθος του προβλήματος και προσθέτουμε νήματα τόσο γρηγορότερα κάνει την ταξινόμηση. Ας δούμε τι χρόνους πετυχαίνει κάνοντας την ίδια ταξινόμηση με σειριακό τρόπο.

Νήμα/Πλ.αριθμών	120000	240000	480000
1	59.855	137.647	945.914

Πίνακας 10

Ταξινόμηση της παραλλαγής του Quicksort με σειριακό τρόπο

Ο χαμηλότερος χρόνος ταξινόμησης ήταν τα 59.855 δευτερόλεπτα. Από τις 10 χρονομετρήσεις αυτή ήταν και η μοναδική κάτω του ενός λεπτού. Είπαμε προηγουμένως ότι με την χρήση δύο νημάτων για τον ίδιο πίνακα ακεραίων η μέτρηση ήταν στα 32.192 ενώ με χρήση τεσσάρων στα 25.282 δευτερόλεπτα. Αμέσως μετά βλέπουμε και τις αντίστοιχες σειριακές χρονομετρήσεις για 240000 και 480000 ακεραίους καθώς και τους πίνακες επιτάχυνσης, απόδοσης και φόρτου.

Παρατηρούμε πλέον χρειαζόμαστε 15 λεπτά για να ταξινομήσουμε έναν πίνακα με μέγεθος περίπου μισό εκατομμυρίου ακεραίων.

Η **επιτάχυνση** λοιπόν του αλγορίθμου *'Quicksort'* για δύο νήματα και πλήθος 120000 ακεραίων είναι η εξής:

$$S = \frac{T_{ser}}{T_{par}} \Rightarrow S = \frac{59.855}{32.192} \Rightarrow S = 1.85$$

Η **αποδοτικότητα** για την ίδια περίπτωση είναι:

$$E = \frac{S}{p} \Rightarrow E = \frac{1.85}{2} \Rightarrow E = 0.925$$

Τέλος ο **φόρτος εργασίας** για δύο νήματα υπολογίζεται ως εξής:

$$T_{par} = \frac{T_{ser}}{p} + T_{overhead} \Rightarrow T_{overhead} = T_{par} - \frac{T_{ser}}{p} \Rightarrow$$

$$\Rightarrow T_{overhead} = 32.192 - 29.927 = 2.265$$

Με τον ίδιο τρόπο υπολογίσαμε και τις επόμενες περιπτώσεις και τις συνοψίσαμε στους παρακάτω πίνακες:

Νήματα / πλήθος αριθ.	120000	240000	480000
2	1.85	1.697	1.703
4	2.367	2.779	2.328
8	2.59	3.529	4.148

Πίνακας 7

Συνολική απεικόνιση των επιταχύνσεων όλων των περιπτώσεων

Νήματα / πλήθος αριθ.	120000	240000	480000
2	0.925	0.848	0.851
4	0.591	0.694	0.582
8	0.323	0.441	0.518

Πίνακας 8

Συνολική απεικόνιση των αποδόσεων όλων των περιπτώσεων

Νήματα / πλήθος αριθ.	120000	240000	480000
2	2.265	21.205	82.471
4	10.319	26.081	169.796
8	15.552	37.632	109.799

Πίνακας 9

Συνολική απεικόνιση των φόρτων των νημάτων για όλες τις περιπτώσεις

5.4.7 Τελικό συμπέρασμα

Οι χρονομετρήσεις που προήλθαν από την παραλλαγή του αλγορίθμου *'Quicksort'* μας βοήθησαν να βγάλουμε κάποια βασικά συμπεράσματα για την λειτουργία του. Το πρώτο συμπέρασμα είναι ότι αρχικά, όταν το μέγεθος του προβλήματος ήταν μικρό, δεν συνέφερε η παραλληλοποίηση του αλγορίθμου. Κάθε φορά που προσθέταμε νήματα αυτό καθυστερούσε την ταξινόμησή λόγω του φόρτου της επικοινωνίας των νημάτων. Από την άλλη, όταν ξεκινήσαμε να αυξάνουμε το μέγεθος του προβλήματος παρατηρήσαμε ότι από ένα σημείο και μετά καθώς προσθέταμε νήματα στο πρόγραμμα αυτό ταξινομούσε τον πίνακα γρηγορότερα. Μία άλλη σημαντική παρατήρηση έχει να κάνει με τον τρόπο συγγραφής του κώδικα. Έτσι περιορίσαμε όσο μπορούσαμε τις κοινές μεταβλητές σε μία και με αυτό το τρόπο επιτύχαμε να μην υπάρχει γενικά μεγάλος φόρτος επικοινωνίας. Αυτή είναι μία τακτική που πρέπει πάντα να την υπολογίζουμε όταν γράφουμε πρόγραμμα με παράλληλη μορφή.

5.5 Ο αλγόριθμος Radix Sort

5.5.1 Εισαγωγή - Περιγραφή του αλγορίθμου

Ο αλγόριθμος *'RadixSort'* είναι ένας ιδιαίτερος αλγόριθμος ο οποίος διαφέρει σε αρκετά σημεία από τους περισσότερους αλγορίθμους που υπάρχουν. Για πρώτη φορά αναπτύχθηκε το 1887 από τον **Herman Hollerith** και η χειρότερη μορφή της πολυπλοκότητάς του είναι της τάξης του O(kN). Ο *'RadixSort'* μπορεί να υλοποιηθεί με δύο τρόπους. Ο πρώτος τρόπος ξεκινώντας την ταξινόμηση από το λιγότερο σημαντικό ψηφίο των δεδομένων πηγαίνοντας προς το μεγαλύτερο

και ο δεύτερος τρόπος ακριβώς με την αντίθετη φορά. Η πρώτη περίπτωση ονομάζεται LSD – Less Significant Digit ενώ ο δεύτερος MSD – Most Significant Digit.

Η υλοποίηση του αλγορίθμου ήταν η πιο πολύπλοκη όλων εξαιτίας της φιλοσοφίας του αλλά και της προσπάθειας που καταβάλλαμε για να τον παραλληλοποιήσουμε όσο περισσότερο μπορούσαμε το οποίο μας βοήθησε να κατανοήσουμε και κάποια άλλα κομμάτια του παράλληλου προγραμματισμού με την βιβλιοθήκη *'pthreads'* όπως το *'busy-waiting'* το οποίο αναφέραμε στην παρουσίαση της βιβλιοθήκης σε προηγούμενο κεφάλαιο.

5.5.2 Θεωρητική ανάλυση τμημάτων αλγορίθμου που θα παραλληλοποιηθούν

Όπως θα φανεί στην συνέχεια, ο *'RadixSort'* διαφέρει ως προς την βασική νοοτροπία της ταξινόμησης. Δηλαδή δεν χρησιμοποιεί συγκρίσεις αριθμών για να επιφέρει το επιθυμητό αποτέλεσμα. Αυτό που ουσιαστικά κάνει είναι να δημιουργεί ένα ιστόγραμμα για κάθε ψηφίο και να αποθηκεύει σε αυτό το σύνολο των αριθμών που περιέχουν αυτό το ψηφίο στην εκάστοτε φάση του. Εννοείται πως αυτή η εργασία γίνεται συγχρονισμένα σε κάθε φάση της εκτέλεσης του αλγορίθμου. Έτσι στην πρώτη φάση του, δημιουργούμε το ιστόγραμμα για το λιγότερο σημαντικό ψηφίο (τελευταίο ψηφίο). Με την ολοκλήρωση αυτής της φάσης έχουμε αποθηκευμένες τις ποσότητες των αριθμών που θα ταξινομήσουμε σε μία μεταβλητή για κάθε ψηφίο τους. Στην συνέχεια ακολουθεί η πρόσθεση των αθροισμάτων και αποθήκευση σε ένα άλλο ιστόγραμμα που αυτή τη φορά κρατάει το σύνολο των αριθμών που προηγούνται από το ψηφίο. Τέλος, με βάση αυτή την τελευταία μεταβλητή γίνεται και η ταξινόμηση του πίνακα.

Μόλις ολοκληρωθεί η πρώτη φάση ταξινόμησης αδειάζουμε τις εγγραφές των πινάκων και εκτελούμε ακριβώς την ίδια λειτουργία στον πίνακα που προέκυψε από την πρώτη φάση. Αυτό γίνεται μέχρι να φτάσει το πρόγραμμα να διατρέξει το πρώτο ψηφίο του μεγαλύτερου αριθμού του πίνακα προς ταξινόμηση. Εμείς θα ορίσουμε αυτόν τον αριθμό από την αρχή έτσι ώστε να γνωρίζει ο αλγόριθμος που να τελειώνει τις λειτουργίες του.

5.5.3 Ψευδοκώδικας

Από την *'main()'* συνάρτηση θα δημιουργήσουμε τον πίνακα προς ταξινόμηση, θα βρούμε τον μεγαλύτερο αριθμό και τέλος θα δημιουργήσουμε τα νήματα και θα καλέσουμε την συνάρτηση *'Radix Sort'*.

Ψευδοκώδικας *'main()'* συνάρτησης.

```
int main()
{
        randomGenerator()
        findTheLargestNumber()

        Σημείωση: Βρίσκουμε τον μεγαλύτερο αριθμό ο οποίος ταυτόχρονα θα αποτελείται και από τα
                  περισσότερα ψηφία

        for (i = 0; i < NUMBEROFTHREADS; i++)
        {
                threads_create(&threads[i], radixSort, &thread_data_array[i])
        }
        for (i = 0; i < NUMBEROFTHREADS; i++)
        {
                threads_join(threads[i], NULL)
        }
```

```
}
```

Μέσα στην συνάρτηση *'radixsort()'* θα δημιουργήσουμε το ιστόγραμμα το οποίο αποθηκεύει τα ψηφία κάθε αριθμού σε κάθε φάση, βρίσκει το σύνολο των αριθμών που περιέχουν το εκάστοτε ψηφίο και δημιουργεί έναν νέο πίνακα προς ταξινόμηση. Αυτό συμβαίνει μέχρι να φτάσει στο πρώτο ψηφίο του μεγαλύτερου αριθμού.

Ψευδοκώδικας *'radixsort()'* συνάρτησης

```
void radixSort(void *rank)
{
	int exp; // Εκθέτης ο οποίος παίρνει αρχική τιμή 1 και δεκαπλασιάζεται σε κάθε βήμα ο
		// οποίος μας βοηθάει να παίρνουμε το ψηφίο που επιθυμούμε για να γίνεται η
		// ταξινόμηση σε κάθε φάση του αλγορίθμου
	int local_m = MAX/NUMBEROFTHREADS
	int my_first = local_m * my_rank;
	int my_last = (my_rank+1) * local_m - 1;

	for(digit = rightmost; digit=first_digit; digit--)
	{
		for(i = my_first; i = my_last; i++)
		{
			build Histogram();
		}
		synchronize threads()
		for (j = 0; j< 10; j++)
		{
			global_Histogram = global_Histogram[i-1] + Histogram[i]
		}
		synchronize threads()
		for (i = my_last; i >= my_first; i--)
		{
			orderedArray[--global_Histogram[(a[i] / exp) % 10]] = a[i];
		}
		synchronize threads()

	}
}
```

5.5.4 Η παραλληλοποίηση του RadixSort

Εφαρμόζοντας την ίδια τεχνική παραλληλοποίησης με αυτή της παραλλαγής του *'Quicksort'* θα παρακολουθήσουμε γραμμή-γραμμή τις εντολές του αλγορίθμου, θα επεξηγούμε όπου χρειάζεται με μορφή σχολίων την λειτουργία κάθε γραμμής και θα παρακάμπτουμε το πρόγραμμα όπου χρειαστεί για να αναλύουμε τις τακτικές που χρησιμοποιήσαμε ώστε να προσπεράσουμε τα όποια προβλήματα δημιουργήθηκαν. Ας ξεκινήσουμε όμως με την επεξήγηση της κάθε γραμμής του κώδικα ξεχωριστά.

Οι πρώτες γραμμές είναι οι δηλώσεις των βιβλιοθηκών από τις οποίες θα πάρουμε τις συναρτήσεις της C που θα χρειαστούμε στην πορεία του προγράμματος. Για άλλη μια φορά έχουμε χρησιμοποιήσει τις εξής:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

#define MAX 8000
#define NUMTHREADS 8

long *a;
void *generateRandom()                  /*********************************************************/
{                                       /*                                                       */
        srand(time(NULL));              /*      Αυτή είναι η συνάρτηση που                       */
        inti;                           /*      χρησιμοποιήσαμε σε όλους τους                    */
                                        /*      αλγορίθμους για την συμπλήρωση του               */
        for(i=0; i<MAX; i++)            /*      πίνακα προς ταξινόμηση με αριθμούς               */
        {                               /*      από το 0 έως το 10 000.                          */
                a[i] = rand()%10001;    /*                                                       */
        }                               /*********************************************************/
}

float *threadTime;          // Αυτός είναι ένας πίνακας ο οποίος θα κρατάει τον χρόνο εκτέλεσης
                               κάθε νήματος.
```

Αυτή την φορά θα χρονομετρήσουμε κάθε νήμα ξεχωριστά και σαν χρόνο εκτέλεσης των νημάτων θα πάρουμε τον μεγαλύτερο χρόνο από τους υπόλοιπους. Επίσης, αυτός είναι ένας άλλος τρόπος χρονομέτρησης εκτέλεσης προγράμματος με την χρήση νημάτων και είναι σωστό να περιγραφεί.

```
int *b;                 // προσωρινός πίνακας

int flag = NUMTHREADS-1;        // αρχικοποιεί την μεταβλητή flag η οποία θα χρειαστεί για την
                                   προστασία της μεταβλητής –bucketcounter

pthread_barrier_tour_barrier;     // ορισμός του φράγματος για τον συγχρονισμό των νημάτων

struct thread_data          // δήλωση της δομής δεδομένων η οποία θα περάσει τον βαθμό
{                           // του κάθε νήματος στην συνάρτηση radixsort() καθώς και τον
        int thread_id;      // μέγιστο αριθμό ο οποίος θα χρησιμοποιηθεί σαν σημείο
        int m;              // αναφοράς για την λήξη των ελέγχων
}
struct thread_data thread_data_array[NUMTHREADS];

void *radixSort(void *rank)
{
        int * bucket; // πίνακας που κρατάει το άθροισμα των ψηφίων σε κάθε φάση εκτέλεσης
```

```
pthread_barrier_init(&our_barrier, NULL, NUMTHREADS); // αρχικοποίηση φραγμάτων

int i, exp, my_rank;
struct thread_data *my_data; //ανάθεση ορισμάτων από την main() μέσα στο *my_data
my_data = (struct thread_data *) rank;
my_rank = my_data -> thread_id; // χρήση της μεταβλητής thread_id και ανάθεση τιμής
int my_m = my_data->m; // κάθε νήμα αναθέτει στην my_m την μεγαλύτερη τιμή του

int local_offset = MAX/NUMTHREADS; // το μήκος των αριθμών που θα χειραγωγήσει
                                      το κάθε νήμα
int my_first = local_offset * my_rank;        // η θέση του πρώτου αριθμού κάθε νήματος
int my_last = (my_rank+1) * local_offset-1; // η θέση του τελευταίου αριθμού κάθε
                                              νήματος

for(exp = 1; my_m/exp>0; exp*=10) // κάνουμε τις επαναλήψεις μέχρι να διατρέξουμε
{                                 // όλα τα ψηφία του μεγαλύτερου αριθμού my_m
        flag=NUMTHREADS-1 // αρχικοποίηση μεταβλητής flag σε κάθε επανάληψη
        bucket = (int *) malloc(local_offset * sizeof (int)); // δεσμεύουμε μνήμη
        sumBucket = (int *) malloc(local_offset * sizeof (int)); // δεσμεύουμε μνήμη

        for (i = my_first; i <= my_last; i++)          // κάθε νήμα διατρέχει τα στοιχεία του
        {
                bucket[(a[i]/exp)%10]++; // τεχνική για να απομονώσουμε το ψηφίο που
                                            και να αυξάνουμε κατά μία μονάδα το ιστόγραμμα
        }
        pthread_barrier_wait(&our_barrier);       //συγχρονισμός νημάτων

        for (i = 0; i< 10; i++) // πρόσθεση όλων των περιεχομένων του πίνακα bucket[]
        {                       // σε έναν νέο πίνακα tovsumBucket[]
                if (i==0)      // έλεγχος για την πρώτη περίπτωση
                {
                      sumBucket[i] = bucket[0];    // ανάθεση της πρώτης τιμής του
                                                      bucket[]
                }
                else
                {
                       sumBucket[i]=sumBucket[i-1]+bucket[i]; // ανάθεση υπολοίπων
                                                                 τιμών
                }
        }
        pthread_wait_barrier(&our_barrier); // συγχρονισμός των νημάτων με φράγμα

        while(flag!=my_rank);    // busy-wait μέθοδος για εκτέλεση με την σωστή σειρά
                for (i = my_last; i >= my_first; i--)// των πράξεων από το κάθε νήμα
                {
                        b[--sumBucket[(a[i] / exp) %10]] = a[i]; // μείωση της θέσης του
```

```
                                                        //sumBucket κατά μία και
                        }                               // ανάθεση τιμής στον b[]
                        flag--;                         // πέρασμα στο επόμενο
νήμα
                pthread_barrier_wait(&our_barrier);     // συγχρονισμός νημάτων

                for(i = my_first; i <= my_last; i++)
                {
                        a[i] = b[i];            // ταξινόμηση
                        bucket[i] = 0;          // αρχικοποίηση του πίνακα bucket[]
                        sumBucket[i] = 0;       // αρχικοποίηση του πίνακα sumBucket[]
                }
                pthread_barrier_wait(&our_barrier); // συγχρονισμός νημάτων
        }
}

int main(int argc, char *argv[])
{
        int *sumBucket;    // πίνακας που κρατάει prefix sum του πίνακα bucket

        pthread_t* threads; // ορισμός των νημάτων που θα δημιουργηθούν - χρησιμοποιηθούν
        generateRandom();       // η γεννήτρια τυχαίων αριθμών προς ταξινόμηση
        int i, m=0;
        a = (int *) malloc(MAX * sizeof (int *));     // δέσμευση μνήμης και χώρου για πίνακαa[]
        b = (int *) malloc(MAX * sizeof (int *));     // δέσμευση μνήμης και χώρου για πίνακαb[]

        for (i =0; i<MAX; i++)      // βρίσκουμε τον μεγαλύτερο αριθμό ο οποίος ταυτόχρονα έχει
        {                           // και μεγαλύτερο ή ίσο μήκος σε σχέση με τους υπόλοιπους
                if (a[i] >m)
                {
                        m = a[i];           // καταχώρηση μέγιστης τιμής στην μεταβλητή m
                }
        }

        for (i = 0; i<NUMTHREADS; i++)          // εκκίνηση των νημάτων στο συγκεκριμένο
        {                                       //  παράδειγμα δημιουργούμε οκτώ νήματα
                threads = malloc(i * sizeof (thread_t)); // δέσμευση μνήμης και χώρου για τα
                                                        νήματα
                thread_data_array[i].thread_id = i;
                thread_data_array[i].m = m;
                pthread_create(&threads[i], NULL, radixsort, (void *) &thread_data_array[i]);
        }
        for(i=0; i<NUMTHREADS; i++)
        {
                pthread_join(threads[i], NULL);
        }
```

```
        printf("\nSORTED :");
        for (i=0; i<MAX; i++)
        {
                printf("&ld ", a[i]);          // εκτύπωση στην οθόνη του ταξινομημένου πίνακα
        }
        return 0;                       // έξοδος προγράμματος
}
```

5.5.5 Μέτρηση χρόνου εκτέλεσης

Πριν την απεικόνιση των αποτελεσμάτων από τις χρονομετρήσεις που κάναμε στον *'RadixSort'* μπορούμε να εξάγουμε μία σημαντική παρατήρηση. Με αυτόν τον αλγόριθμο ταξινομήσαμε πολύ μεγάλα πλήθη αριθμών χρησιμοποιώντας ακόμη και τα 16 νήματα που μας δίνει ο επεξεργαστής μας. Εμείς παρ' όλα αυτά θα αναφερθούμε στη χρήση μέχρι και 8 νημάτων αλλά και στις δοκιμές με 16 νήματα τα αποτελέσματα ήταν σχεδόν εντυπωσιακά. Επίσης, οι ομάδες αριθμών που χρονομετρήσαμε και θα παρουσιάσουμε είχαν πλήθος 160000 και 320000 αριθμών αντίστοιχα ενώ σε δοκιμές για 2000000 και 20000000 αριθμούς ο αλγόριθμος σχετικά αποδεκτούς χρόνους εκτέλεσης.

Τέλος, αξίζει να υπενθυμίσουμε ότι όπως και στις προηγούμενες υλοποιήσεις έτσι και σε αυτή μετρήσαμε τους πραγματικούς χρόνους εκτέλεσης του προγράμματος ταξινόμησης όπου και παρουσιάζονται παρακάτω.

Νήματα/Πλ.αριθμών	160000	320000
2	5.829	15.915
4	5.715	13.591
8	5.627	11.503

Πίνακας 12

Συγκεντρωτικός πίνακας χρόνων εκτέλεσης σε seconds και ταξινόμησης του *'Radix Sort'*

Παρατηρούμε πως όσο αυξάνουμε τον αριθμό των νημάτων για το ίδιο πλήθος αριθμών τόσο μειώνεται και ο χρόνος ταξινόμησης και αυτό φαίνεται ξεκάθαρα από το παράδειγμα με τους 320000 αριθμούς προς ταξινόμηση.

Τέλος, να υπενθυμίσουμε και εδώ πως πήραμε μετρήσεις και για άλλες περιπτώσεις χρησιμοποιώντας πλήθος εκατομυρίων αριθμών. Όσο καταφέρναμε να ταξινομούμε πίνακες με γρήγορους ρυθμούς τόσο μεγαλώναμε και το μέγεθος του προβλήματος για να δούμε τα όρια του. Ακόμη και με έναν πίνακα με 20 000 000 000 αριθμούς που δοκιμάσαμε το πρόγραμμα έδειχνε ότι έκανε την ταξινόμηση όμως δεν το αφήσαμε να ολοκληρώσει μέχρι τέλους και να δούμε τι χρόνους θα έδινε γιατί όπως ήταν φυσικό έπαιρνε αρκετές ώρες πιθανώς και ημέρες.

5.5.6 Σύγκριση με σειριακό αλγόριθμο υπολογισμός επιτάχυνσης και απόδοσης

Προηγουμένως είδαμε τις χρονικές μετρήσεις έξι περιπτώσεων. Όσες περισσότερες συγκρίσεις κάνουμε τόσο πλησιάζουμε στο να εξάγουμε πιο ασφαλή συμπεράσματα. Ας μετρήσουμε τώρα τις αποδόσεις, τις επιταχύνσεις και τους φόρτους εργασίας των νημάτων όταν εκτελούνται παράλληλα. Πρωτού συμβεί αυτό ας δούμε τι αποτέλεσματα πήραμε όταν τρέξαμε το πρόγραμμα σειριακά για τις δύο ομάδες αριθμών προς ταξινόμηση.

Νήμα/Πλήθος αριθμών	160000	320000
1	6.938	17.645

Πίνακας 13

Χρόνοι εκτέλεσης σε seconds του Radix sort με σειριακό τρόπο

Η μέτρηση για ταξινόμηση 160000 αριθμών είναι 6.928 δευτερόλεπτα. Ο χρόνος αυτός ήταν ο μικρότερος που πήραμε από 20 διαδοχικές δοκιμές και όπως φαίνεται είναι σημαντικά μεγάλος σε σχέση με όλες τις περιπτώσεις παράλληλοποίησης που δοκιμάσαμε.

νήματα / πλήθος αριθ	160000	320000
2	1.188	1.108
4	1.212	1.298
8	1.231	1.533

Πίνακας 14

Συνολική απεικόνιση των επιταχύνσεων όλων των περιπτώσεων

νήματα / πλήθος αριθ	160000	320000
2	0.0594	0.554
4	0.303	0.324
8	0.153	0.191

Πίνακας 15

Συνολική απεικόνιση των αποδόσεων όλων των περιπτώσεων

νήματα / πλήθος αριθ	160000	320000
2	2.365	7.093
4	3.983	9.18
8	4.761	9.298

Πίνακας 16

Συνολική απεικόνιση του φόρτου εργασίας όλων των περιπτώσεων

5.5.7 Τελικό συμπέρασμα

Η συμπεριφορά του αλγορίθμου *'Radix Sort'* έχει ακριβώς την συμπεριφορά που θα περιμέναμε να έχει. Δηλαδή, στην αρχή η απόδοση και η επιτάχυνση της ίδιας ομάδας μειώνεται όσο αυξάνονται τα νήματα ενώ ο συνολικός φόρτος των νημάτων αυξάνεται όσο αυξάνονται και αυτά. Από ένα σημείο και μετά όμως και πιο συγκεκριμένα από πλήθος 160000 αριθμών και μετά η συμπεριφορά του ήταν θετική. Όσο περισσότερο αυξάναμε το μέγεθος του προβλήματος και προσθέταμε νήματα τόσο γρηγορότερα έκανε τις ταξινομήσεις. Τέλος, το πρόγραμμα εκμεταλλεύτηκε όλα τα νήματα τα οποία δίνει ο κατασκευαστής του επεξεργαστή ενώ δεν 'κρέμασε' ακόμη και όταν το πλήθος των αριθμών που θέσαμε προς ταξινόμηση ήταν πάρα πολύ μεγάλο.

6. Συνολικά συμπεράσματα

Τώρα που έχουμε φτάσει στο τέλος της συγγραφής έφτασε η στιγμή να θέσουμε τα πιο σημαντικα ερωτήματα που δημιουργήθηκαν καθώς συγγράφαμε το πόνημα ώστε να είναι διαθέσιμα για περαιτέρω έρευνα. Το πρώτο και κυριότερο ερώτημα έχει να κάνει με το αν

χρησιμοποιώντας περισσότερους πυρήνες – διεργασίες – νήματα γίνεται ένα πρόγραμμα αποτελεσματικότερο. Σε όλες τις υλοποιήσεις των αλγορίθμων οι οποίες αφορούσαν μικρό πλήθος αριθμών οι χρόνοι εκτέλεσης των προγραμμάτων με πολλούς πυρήνες σε σχέση με την χρήση ενός είχαν διαφορά. Μπορεί η διαφορά να ήταν στις περισσότερες υλοποιήσεις πολύ μικρή όμως σε όλες ήταν υπαρκτή. Αυξάνοντας το πλήθος των προς ταξινόμηση αριθμών η προηγούμενη συμπεριφορά μεταβαλλόταν. Η καθυστέρηση που προκύπτει από την επικοινωνία των νημάτων είναι πλέον μικρότερη από τον χρόνο που χρειάζεται κάθε νήμα να ταξινομήσει το τμήμα του πίνακα που έχει αναλάβει και ο χρόνος αυτός με την σειρά του είναι φυσικά μικρότερος από τον χρόνο που χρειάζεται το ίδιο πρόγραμμα με χρήση λιγότερων νημάτων. Έτσι, καταλήγουμε στο συμπέρασμα πως η χρήση του παράλληλου προγραμματισμού είναι συμφέρουσα όσο μεγαλύτερο είναι το μέγεθος του προβλήματος.

Μετά από αυτή την μελέτη στον παράλληλο προγραμματισμό μπορούμε να εξάγουμε κάποια συμπεράσματα. Έτσι, είναι φανερό ότι πλέον ο τρόπος προγραμματισμού κατευθύνεται προς την παράλληλη μορφή του. Πλατφόρμες, κάρτες γραφικών, λογισμικό αλλά και υλικό δημιουργούνται και κατασκευάζονται κάθε χρόνο τα οποία απαιτούν την χρήση προγραμμάτων σε παράλληλη μορφή ώστε να αποδεσμεύσουν με την σειρά τους την πλήρη τους ισχύς. Άλλο ένα στοιχείο που ‘μαρτυρεί’ την χρησιμότητα του παράλληλου προγραμματισμού είναι ο μεγάλος όγκος των δεδομένων που πρέπει πλέον να επεξεργάζονται οι μηχανές. Ο παράλληλος προγραμματισμός χρειάζεται και είναι αναγκαίος για να εκμεταλλευτούμε την υπολογιστική ισχύ που συνεχώς αυξάνεται σε αυτούς τους μεγάλους όγκους δεδομένων χωρίς αυτό να σημαίνει ότι είναι η γρηγορότερη λύση (τουλάχιστον σε επεξεργασία σχετικά μικρού όγκου δεδομένων).

Πότε όμως είναι σκόπιμο να παραλληλοποιούμε έναν αλγόριθμο; Έχοντας κάνει εκατοντάδεις δοκιμές σε κάθε τμήμα κάθε αλγορίθμου για να κατανοήσουμε την συμπεριφορά όλων των υλοποιήσεων καταλήξαμε στο συμπέρασμα πως για να είναι γρηγορότερο ένα πρόγραμμα πρέπει οι διεργασίες – πυρήνες – νήματα να χρειάζεται να συνεργαστούν όλο και λιγότερες φορές ενώ το μέγεθος του προβλήματος να είναι το μεγαλύτερο δυνατό. Για παράδειγμα, εάν 4 νήματα χρειαστεί να ενημερώσουν μία μεταβλητή αυτό δεν μπορεί να συμβεί παράλληλα γιατί θα προκύψει σφάλμα. Θα γίνει οπωσδήποτε σειριακά θέτοντας τα νήματα σε μία σειρά. Η ειδοποίηση του ενός νήματος ότι τέλειωσε τις εργασίες του στην μεταβλητή ώστε να ξεκινήσει το επόμενο τις δικές του προσδίδουν χρόνο ο οποίος τελικά δεν συμφέρει από άποψη ταχύτητας εκτέλεσης ειδικά σε μικρά προβλήματα. Ας το δούμε αυτό στην πράξη στην συνέχεια για να το καταλάβουμε καλύτερα.

Όποτε χρονομετρήσαμε ενέργειες οι οποίες δεν συμπεριλάμβαναν την συνεργασία διεργασιών ή νημάτων τότε τα αποτελέσματα ήταν καλύτερα. Όποτε δηλαδή χωρίσαμε τον όγκο δεδομένων σε ισομερή τμήματα και αναθέσαμε αυτά τα τμήματα σε νήματα ή διεργασίες και τα χρονομετρήσαμε, οι χρόνοι που πήραμε ήταν γρηγορότεροι από τους αντίστοιχους σειριακούς ακόμη και σε μικρά μεγέθη προβλήματος. Ας δούμε ένα παράδειγμα για να καταλάβουμε καλύτερα. Τον αλγόριθμο *‘Radix Sort’* ξεκινήσαμε να τον χρονομετρούμε από την δημιουργία των νημάτων μέχρι και το τέλος λειτουργίας τους. Οι χρόνοι εκτέλεσης συνολικά ήταν μεγαλύτεροι από ότι τμηματικά. Χρονομετρώντας όμως τα νήματα σε κάθε ένα από τα οποία έχει ανατεθεί ένα τμήμα του προβλήματος, οι χρόνοι που πήραμε ήταν καλύτεροι σε σχέση με την ανάθεση του προβλήματος σε ένα πυρήνα. Πιο συγκεκριμένα το τμήμα που χρονομετρήσαμε είναι το τμήμα όπου βρίσκουμε τα ιστογράμματα του κάθε ψηφίου.

Πλ.Αριθμών/Νήματα	1	2	4	8	16
4000	0.000033	0.000028	0.000020	0.000012	0.000004

Πίνακας 17

Ταξινόμηση 4000 αριθμών με χρήση 1,2,4,8 και 16 νημάτων και χρονομέτρηση παράλληλου τμήματος του κώδικα

Παρατηρούμε ότι ο χρόνος εκτέλεσης του σημείου κατά τον οποίο ο *‘Radix Sort’* δημιουργεί σε κάθε νήμα από ένα ιστόγραμμα είναι 0.000033’’ ενώ όσο προσθέτουμε νήματα ο χρόνος μειώνεται. Να θυμήσουμε εδώ πως σε αυτό το πλήθος αριθμών συνολικά το πρόγραμμα

εκτελείται πιο αργά όσο προσθέτουμε νήματα. Για καλύτερα συμπεράσματα κάναμε το ίδιο και για πλήθος 40.000 αριθμών όπου παρουσιάζουμε παρακάτω.

Πλ.Αριθμών/Νήματα	1	2	4	8	16
40000	0.000572	0.000162	0.000082	0.000042	0.000024

Πίνακας 18

Ταξινόμηση 40000 αριθμών με χρήση 1,2,4,8 και 16 νημάτων και χρονομέτρηση παράλληλου τμήματος του κώδικα

Από τον παραπάνω πίνακα γίνεται αντιληπτό ότι η παραλληλοποίηση ενός αλγορίθμου πρέπει να αφορά συγκεκριμένα σημεία. Η βέλτιστη λογική (και επικρατούσα) είναι η παραλληλοποίηση συγκεκριμένων κομματιών τα οποία απλά αναθέτουν σε διεργασίες ή νήματα ένα μέρος του συνολικού όγκου και το κάθε νήμα εκτελεί τους δικούς του υπολογισμούς πάνω στο τμήμα αυτό. Θυμίζουμε ότι αυτά τα συστήματα ονομάζονται *'MIMD'.* Από αυτό το σημείο και μετά θα πρέπει να λαμβάνεται υπ'όψιν και η αρχιτεκτονική του υπολογιστικού συστήματος. Θα πρέπει να προσέχεται η χρήση της μνήμης σε πολύ μεγαλύτερο βαθμό και ένας αλγόριθμος πρέπει να σχεδιάζεται και υλοποιείται αποκλειστικά με βάση την διαθέσιμη μνήμη.

Αυτό μπορούμε να πούμε ότι είναι η 'χρυσή λεπτομέρεια' και κάνει την διαφορά στην ποιότητα συγγραφής του κώδικα. Είναι πολύ ενδιαφέρων τομέας και φυσικά εμβαθύνει την γνώση ενός προγραμματιστή και γενικότερα ενός επιστήμονα πληροφορικής.

Ας δούμε ένα παράδειγμα για να διαπιστώσουμε εάν επιβεβαιώνονται όλα τα παραπάνω που αναφέραμε. Στην παραλλαγή του αλγορίθμου *'Quicksort'* και σε όλες τις μετρήσεις ο χρόνος της ταξινόμησης με παράλληλο τρόπο αυξανόταν πάντοτε έχοντας σταθερό το μέγεθος του προβλήματος και αυξάνοντας απλά τα νήματα. Όταν όμως χρονομετρήσαμε μόνο το κομμάτι στο οποίο κάθε νήμα καταγράφει τους μικρότερους αριθμούς του *'pivot'* σε μία τοπική του νήματος μεταβλητή και όχι μαζί με αυτό όπου κάνει την τοποθέτηση οι χρόνοι ήταν γρηγορότεροι όσο προσθέταμε νήματα.

Στην συνέχεια παρουσιάζονται οι πίνακες με τις αντίστοιχες χρονομετρήσεις μόνο των παράλληλων κομματιών του κώδικα για τον αλγόριθμο *'Quicksort'*.

Πλ.Αριθμών/Νήματα	1	2	4	8
4000	0.000031	0.000016	0.000008	0.000006

Πίνακας 19

Ταξινόμηση 4000 αριθμών με χρήση 1,2,4,8 και 16 νημάτων και χρονομέτρηση παράλληλου τμήματος του κώδικα

Πλ.Αριθμών/Νήματα	1	2	4	8
8000	0.000111	0.000054	0.00032	0.000026

Πίνακας 20

Ταξινόμηση 8000 αριθμών με χρήση 1,2,4,8 και 16 νημάτων και χρονομέτρηση παράλληλου τμήματος του κώδικα

Πλ.Αριθμών/Νήματα	1	2	4	8
16000	0.000201	0.000110	0.00044	0.000033

Πίνακας 21

Ταξινόμηση 16000 αριθμών με χρήση 1,2,4,8 και 16 νημάτων και χρονομέτρηση παράλληλου τμήματος του κώδικα

Παρατηρούμε πως και στον *'Radix Sort'* αλγόριθμο και στον *'Quicksort'* όταν δεν καλούμε τα νήματα να επικοινωνήσουν μεταξύ τους ώστε να ενημερώσουν την τιμή μίας μεταβλητής τότε οι χρόνοι των παράλληλων κομματιών ακόμη και σε μικρά μεγέθη προβλημάτων είναι μικροτεροι σε σχέση με τους αντίστοιχους σειριακούς. Αυτό θα έχει ως αντίκτυπο και στις υπόλοιπες μετρήσεις όπως η επιτάχυνση. Συγκεκριμένα η επιτάχυνση θα αυξάνεται όσο μειώνονται οι χρόνοι της παράλληλης υλοποίησης και το αποτέλεσμα που θα βγαίνει θα είναι μεγαλύτερο της μονάδας (>1).

Αυτό μας δείχνει πότε μας συμφέρει να χρησιμοποιούμε τον παράλληλο προγραμματισμό. Θα πρέπει να είμαστε πολύ προσεκτικοί καθώς παραλληλοποιούμε κομμάτια σειριακού κώδικα καθώς μπορεί τελικά να μην πετύχουμε τα αποτελέσματα που επιθυμούμε. Αυτός είναι ένας σημαντικός παράγοντας τον οποίο πρέπει να λαμβάνουμε πάντα υπ' όψιν μας ανάμεσα σε πολλούς άλλους τους οποίους συναντήσαμε κατά την συγγραφή.

Τέλος, ένας τελευταίος αλλά όχι ασήμαντος παράγοντας ο οποίος μετράει στην εξαγωγή ενός συμπεράσματος και τον χρησιμοποιήσαμε και εμείς, είναι το πλήθος των δοκιμών στις περισσότερες δυνατές περιπτώσεις και σε ομοιόμορφες καταστάσεις. Εμείς πραγματοποιήσαμε εκατοντάδες δοκιμές σε όλες τις υλοποιήσεις και σε όλες τις πιθανές καταστάσεις που μπορέσαμε να σκεφτούμε και παρουσιάσαμε τα αποτελέσματα που εμφανίστηκαν.

7. Επίλογος

Σκοπός του συγγράματος ήταν να παρουσιαστούν οι βασικές λειτουργίες των δύο βιβλιοθηκών – '*MPI'* και '*PTHREADS'* – και με την χρήση αυτών των βιβλιοθηκών να υλοποιηθούν κάποιοι βασικοί αλγόριθμοι. Οι αλγόριθμοι που παρουσιάσαμε περιορίστηκαν στην ταξινόμηση ακέραιων αριθμών με όσο το δυνατόν καλύτερες συνθήκες. Όμως οι συνθήκες δεν είναι πάντα ιδανικές. Για παράδειγμα, ο όγκος των δεδομένων προς ταξινόμηση μπορεί να μην ισομερίζεται προκειμένου κάθε διεργασία – νήμα να εκμεταλλεύεται το ίδιο πλήθος. Επίσης, τα δεδομένα δεν είναι πάντα ακέραιοι αριθμοί για να ταξινομηθούν. Εκτός ότι μπορεί να είναι άλλης μορφής αριθμοί (δεκαδικοί κλπ) μπορεί να είναι και αλφαριθμητικά τα δεδομένα και σε εκείνη την περίπτωση να χρειάζονται παραπάνω ενέργειες για την ταξινόμησή τους.

Όλα τα προηγούμενα είναι μία καλή ευκαιρία για περισσότερη ανάπτυξη του θέματος. Επίσης, όλα τα προηγούμενα θέματα μπορούν να ολοκληρώσουν το παρόν πόνημα και να μας οδηγήσουν σε μία πιο ολοκληρωμένη αντίληψη στο πως λειτουργούν τα παράλληλα προγράμματα σε πολυπύρηνα και πολυνηματικά υπολογιστικά συστήματα.

8. ΒΙΒΛΙΟΓΡΑΦΙΑ

8.1 Διαδικτυακές πηγές

Οι πηγές που χρησιμοποιήσαμε βρίσκονται στο Διαδίκτυο και είναι διαθέσιμες προς όλους τους χρήστες. Αυτές είναι:

http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/
http://www.drdobbs.com/parallel/207402986
http://llpanorama.wordpress.com/2010/06/18/cuda-gets-easier/
http://www.cplusplus.com/doc/tutorial
http://www.cs.usfca.edu/~galles/visualization/Algorithms.html
https://research.nvidia.com/content/designing-efficient-sorting-algorithms-manycore-gpus

8.2 Επιστημονικά συγγράμματα

1. An introduction to parallel programming by P. Pacheco (Jan 21, 2011)

2. An Introduction to Parallel Computing 2nd Edition. by Ananth Grama, George Karypis, Vipin Kumar and Anshul Gupta, (Jan 26, 2003)

3. Implicit radix sorting on GPUs by Linh Ha, Jens Kruger, Claudio T. Silva, (July 27, 2010)

4. Parallel Programming with MPI by Peter Pacheco, (Oct 1, 1996)

5. A Practical Quicksort Algorithm by Daniel Cederman and Philippas Tsigas, (Department of Computer Science and Engineering Chalmers University of Technology)

6. An efficient parallel sorting compatible with the standard qsort by Duhu Man, Yasuaki Ito and Koji Nakano (2009 International Conference on Parallel and Distributed Computing, Applications and Technologies, Department of Information Engineering, Hiroshima University)

7. Engineering a Multi-core Radix Sort by Jan Wassenberg and Peter Sanders (Karlsruhe Institute of Technology, Karlsruhe, Germany)

8. PARALLEL QUICKSORT IMPLEMENTATION USING MPI AND PTHREADS by PUNEET KATARIA, (10 Dec, 2008)

9. Scalable Distributed-Memory External Sorting by Mirko Rahn, Peter Sanders, Johannes Singler, (Karlsruhe Institute of Technology, ICDE conference 2010)

10. Markgraf, Joey D. (2007), *The Von Neumann bottleneck*, retrieved August 24, 2011

11. Hennessy, John L.; Patterson, David A. *Computer Architecture: A Quantitative Approach*.

12. Quinn, Michael J. (2004). *Parallel Programming in C with MPI and openMP*.

13. Introduction to “The First Draft Report on the EDVAC” by John von Neumann

14. N. Haberman (1972) "Parallel Neighbor Sort (or the Glory of the Induction Principle)," CMU Computer Science Report (available as Technical report AD-759 248, National Technical Information Service, US Department of Commerce, 5285 Port Royal Rd Sprigfield VA 22151).

15. S. Lakshmivarahan, S. K. Dhall, and L. L. Miller (1984), Franz L. Alt and Marshall C. Yovits, ed., "Parallel Sorting Algorithms", *Advances in computers* (Academic Press) **23**: 295–351, ISBN 978-0-12-012123-6