ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Παράλληλα και Κατανεμημένα Συστήματα Υπολογιστών

Άσκηση 1

ΧΑΤΖΗΘΩΜΑ ΑΝΤΡΕΑΣ

ΑΕΜ: 8026

antreasc@ece.auth.gr

Υπεύθυνος καθηγητής: κ. Πιτσιάνης Νίκος
Ημερομηνία Παράδοσης: 12/11/2015

ΕΙΣΑΓΩΓΗ

Το ζητούμενο της εργασίας αυτής ήταν η υλοποίηση ενός παράλληλου προγράμματος για την διάταξη Ν ακέραιων αριθμών σε αύξουσα σειρά, κάνοντας χρήση του αλγόριθμου Bitonic Sort. Για τον σκοπό αυτό, μας δόθηκαν 2 σειριακές εκδόσεις του αλγορίθμου, η αναδρομική και η επαναληπτική. Εμείς καλούμασταν να φτιάξουμε 2 παράλληλες υλοποιήσεις του αλγόριθμου χρησιμοποιώντας στην πρώτη υλοποίηση *pthreads* και στην δεύτερη *openMP* επιλέγοντας την αναδρομική ή την επαναληπτική έκδοση που δόθηκε. Ο τρόπος λειτουργίας του αλγόριθμου Bitonic Sort εξηγήθηκε και αναλύθηκε στο μάθημα.

Μετά από σκέψη και δοκιμές, κρίθηκε ότι η υλοποίηση με pthreads θα ήταν καλύτερη στην αναδρομική έκδοση του αλγορίθμου σε αντίθεση με την openMP όπου υλοποιήθηκε στην επαναληπτική έκδοση του αλγορίθμου. Το πρόγραμμα ζητάει από το χρήστη δυο ακέραιους αριθμούς, q και p, ξεκινάει 2^p νήματα (threads) και φτιάχνει έναν πίνακα τυχαίων ακεραίων, μήκους 2^q.

Για τον έλεγχο καλής λειτουργείας του αλγόριθμου που υλοποιήσαμε με pthreads αλλά και με openMP, χρησιμοποιήθηκε η συνάρτηση test() η οποία ήταν ήδη υλοποιημένη από τον κ. Πιτσιάνη και ελέγχει κατά πόσο ο τελικός πίνακας είναι ταξινομημένος κατά αύξουσα σειρά. Για την απόδοση του κάθε αλγόριθμου που υλοποιήσαμε, μετρήθηκε ο χρόνος εκτέλεσης του για διάφορα q και p. Αρχικά οι αλγόριθμοι εκτελέστηκαν στο δικό μας σύστημα αλλά για τα αποτελέσματα που θα παρουσιάσουμε παρακάτω, οι αλγόριθμοι εκτελέστηκαν στο σύστημα diades απ' όπου πάρθηκαν και οι μετρήσεις.

Για την απόδοση των παράλληλων αλγορίθμων που υλοποιήσαμε, συγκρίθηκαν οι χρόνοι εκτέλεσης του κάθε αλγόριθμου με τους αντίστοιχους σειριακούς. Ακολούθως προστέθηκε ο αλγόριθμος quick sort για τον έλεγχο ορθότητας των αποτελεσμάτων μας. Παρατηρήθηκε ότι ο quick sort αν και σειριακός, ωστόσο οι χρόνοι εκτέλεσης του ήταν πολύ καλοί και μάλιστα σε ορισμένες περιπτώσεις ήταν κοντά στους χρόνους των παράλληλων υλοποιήσεων. Για τον λόγο αυτό, συνδυάστηκε η μέθοδος quick sort με την υλοποίηση pthreads έτσι ώστε να πετύχουμε ακόμη καλύτερους χρόνους εκτέλεσης.

Τέλος, τρέξαμε όλους τους αλγόριθμους για p = [1 : 8] και q = [16 : 24] και πήραμε τους χρόνους εκτέλεσης του κάθε αλγόριθμου. Χρησιμοποιώντας τους χρόνους αυτούς και μέσω του προγράμματος MS Office Excel 2013, δημιουργήσαμε διάφορα διαγράμματα με τα οποία συγκρίνουμε τους χρόνους εκτέλεσης της κάθε υλοποίησης για διάφορα μήκη πίνακα αλλά και αριθμό νημάτων.

Για το compile και την εκτέλεση των προγραμμάτων χρησιμοποιήθηκε ο προσομοιωτής Cygwin.

Γενικά – Συνάρτηση main

Για την υλοποίηση του ζητούμενου προγράμματος χρησιμοποιήθηκαν οι ακόλουθες κοινές (global) μεταβλητές:

- **N_threads:** το σύνολο των διαθέσιμων threads το οποίο ισούτε με 2^p.
- **A_threads:** ο τρέχων αριθμός των ενεργών νημάτων.
- **N:** το μήκος του πίνακα το οποίο ισούτε με 2^q.
- ***a:** pointer ο οποίος δείχνει στον πίνακα που δεσμεύαμε ανάλογα με το N.

Δημιουργήθηκαν οι παρακάτω συναρτήσεις:

- **fileWrite:** γράφει σε αρχείο κειμένου τους χρόνους κάθε εκτέλεσης.
- **cmpfuncAsc, cmpfuncDes**: βοηθητικές συναρτήσεις για την υλοποίηση της συνάρτησης qsort.Οι συναρτήσεις δεν υλοποιήθηκαν από εμάς. (ΠΗΓΗ)

Επίσης, χρησιμοποιήθηκαν οι ήδη υλοποιημένες συναρτήσεις:

- **test**: ελέγχει αν ταξινομήθηκε σωστά ο πίνακας και εκτυπώνει το αντίστοιχο μήνυμα
- **init**: δίνει τυχαίες (random) τιμές στον πίνακα που δεσμεύτηκε.
- **exchange**: ανταλλάσσει αμοιβαία 2 στοιχεία του πίνακα.
- **compare**: συγκρίνει 2 στοιχεία του πίνακα βάσει την κατεύθυνση της ταξινόμησης.

Αφού υλοποιήσαμε τις μεθόδους που ζητήθηκαν και εφόσον δεν υπάρχει κανένα σφάλμα στον κώδικα, κάνουμε compile τον κώδικα και τρέχουμε το πρόγραμμα δίνοντας 2 αριθμούς οι οποίοι αντιστοιχούν στα p και q. Μετά, μέσω της συνάρτηση main η μεταβλητή N παίρνει την τιμή 2^q και δεσμεύεται ο αντίστοιχος χώρος στη μνήμη, και η μεταβλητή N_threads παίρνει την τιμή 2^p.

Ακολούθως, εκτελείται η παρακάτω διαδικασία για όλες τις υλοποιημένες μεθόδους της Bitonic Sort, είτε σειριακές είτε παράλληλες. Μέσω της συνάρτησης init ο πίνακας παίρνει τυχαίες τιμές. Αφού γίνει αυτό, μέσω της συνάρτησης gettimeofday της βιβλιοθήκης sys/time.h παίρνουμε τον χρόνο στη συγκεκριμένη χρονική στιγμή. Αμέσως μετά εκτελείται η αντίστοιχη συνάρτηση για την υλοποίηση κάποιας μεθόδου. Μόλις τελειώσει η ταξινόμηση του πίνακα και η συγχώνευση του (σε ορισμένες υλοποιήσεις), το πρόγραμμα επιστρέψει στην main, και μέσω πάλι της gettimeofday, παίρνουμε τον εκάστοτε χρόνο έτσι ώστε να υπολογίσουμε τον χρόνο εκτέλεσης του αλγορίθμου. Ακολούθως ελέγχουμε την ορθότητα του αλγορίθμου μέσω της συνάρτησης test και εκτυπώνουμε στην κονσόλα τα αποτελέσματα. Τέλος καλούμε τη συνάρτηση fileWrite έτσι ώστε να περάσουμε τους χρόνους εκτέλεσης του κάθε αλγόριθμου σε αρχεία κειμένου που φέρουν το όνομα της κάθε μεθόδου.

Όλη η διαδικασία από την αρχή μέχρι το τέλος, εκτελείτε κάθε φορά που δίνουμε διαφορετικά p και q.

Υλοποίηση pthreads

Για την υλοποίηση του αλγόριθμου με pthreads, χρησιμοποιήθηκε η αναδρομική έκδοση του αλγορίθμου που μας δόθηκε, έτσι ώστε κάθε αναδρομική κλήση της συνάρτησης για κάθε υποπίνακα, να εκτελείτε από διαφορετικό νήμα. Αυτό εφαρμόστηκε και στη συνάρτηση Bitonic Sort αλλά και στην Bitonic Merge της σειριακής έκδοσης.

Όλες οι συναρτήσεις για την υλοποίηση με pthreads, βρίσκονται στο αρχείο του κώδικα που φέρει το όνομα antreascBitonic.c. Επίσης χρησιμοποιήθηκε μια δομή, η threads_args, για να περάσουμε στις συναρτήσεις τα διάφορα ορίσματα που χρησιμοποιήθηκαν.

Οι συναρτήσεις που υλοποιήθηκαν είναι:

- **pthSort**: Ορίζει αρχικές τιμές στα ορίσματα και καλεί την pthBitonicSort.
- **pthBitonicSort**: Ταξινομεί διτονικά τον πίνακα και καλεί την pthBitonicMerge.
- **pthBitonicMerge**: Συγχωνεύει τους πίνακες.

Η δομή threads_args έχει τα ακόλουθα ορίσματα:

- **lo**: δείχνει στο 1ο στοιχείο κάθε υποπίνακα.
- **cnt**: έχει τον αριθμό των στοιχείων του κάθε υποπίνακα.
- **dir**: φέρει 1 για αύξουσα ταξινόμηση και 0 για φθίνουσα.

Πιο αναλυτικά, όταν η συνάρτηση pthBitonicSort καλείται, και εφόσον έχουμε περάσει τα ορίσματα μέσω της δομής, γίνεται έλεγχος της μεταβλητής cnt που όπως αναφέραμε, αντιπροσωπεύει τον αριθμό των στοιχείων του πίνακα στα οποία θα εκτελεστεί η ταξινόμηση. Αν ο αριθμός αυτός είναι μικρότερος ή ίσος με 1 τότε τερματίζεται η λειτουργία της συνάρτησης διότι δεν υπάρχει νόημα να γίνει ταξινόμηση σε υποπίνακα με 1 στοιχείο. Αν είναι μεγαλύτερος από 1 τότε ελέγχουμε τον αριθμό των ενεργών νημάτων έτσι ώστε να τηρούμε το όριο που τέθηκε. Ακολούθως αυξάνουμε τον αριθμό των ενεργών νημάτων κατά 2 διότι πρόκειται να ενεργοποιηθούν 2 καινούρια νήματα. Αυτό γίνεται με τη χρήση κλειδώματος και ξεκλειδώματος μιας μεταβλητής mutex έτσι ώστε να διασφαλίσουμε την ορθότητα του μετρητή των ενεργών νημάτων λόγω των μαζικών και ταυτόχρονων αλλαγών που μπορεί να υποστεί. Ακολούθως, καλούμε την pthBitonicSort παράλληλα, μια φορά για το 1ο μισό του πίνακα και μια φορά για το 2ο μισό του πίνακα. Αυτό γίνεται ενεργοποιώντας 2 νήματα και δημιουργώντας 2 αντικείμενα της δομής threads_args ούτως ώστε να περάσουμε τα επιθυμειτά ορίσματα στις 2 αυτές κλήσεις της συνάρτησης.

Στο τέλος αυτού του block καλούμε τη συνάρτηση pthread_join για κάθε νήμα που ενεργοποιήθηκε έτσι ώστε το νήμα που κάλεσε τα υπόλοιπα νήματα να περιμένει τα νήματα να τελειώσουν και κατά κάποιο τρόπο να ενωθούν και να περάσουν τις τιμές τους στον πίνακα. Αν τα ενεργά νήματα ξεπεράσουν το όριο των νημάτων, τότε καλούμε την σειριακή υλοποίηση της Bitonic Sort η οποία ήταν ήδη υλοποιημένη με το όνομα recBitonicSort.

Τέλος, όταν οι αναδρομικές κλήσεις της pthBitonicSort φτάσουν στο τέλος τους, σειρά έχει η pthBitonicMerge. Εφόσον περάσουμε τα ορίσματα στην pthBitonicMerge .και την καλέσουμε, ελέγχουμε και πάλι το μέγεθος του πίνακα με την μεταβλητή cnt. Όπως και πριν, αν ο αριθμός αυτός είναι μικρότερος ή ίσος με 1 τότε τερματίζεται η λειτουργία της συνάρτησης, αν είναι μεγαλύτερος από 1 τότε μέσω της ήδη υλοποιημένης συνάρτησης compare γίνεται η σύγκριση των στοιχείων των υποπινάκων και ανάλογα της κατεύθυνσης γίνεται εναλλαγή ή όχι. Ακολούθως, γίνεται έλεγχος των ενεργών νημάτων. Αν δεν ξεπεράστηκε τι όριο τότε όπως και με την pthBitonicSort, ενεργοποιούνται 2 νήματα και καλούμε την pthBitonicMerge. Η λογική είναι ακριβώς ίδια με την pthBitonicSort. Αν δεν υπάρχουν διαθέσιμα νήματα, τότε εκτελείται η σειριακή έκδοση της Bitonic Merge καλώντας την ήδη υλοποιημένη συνάρτηση recBitonicMerge.

Όταν ο αλγόριθμος τελειώσει και από την τελευταία αναδρομική BitonicMerge είτε παράλληλη είτε σειριακή, τότε η ταξινόμηση του πίνακα έχει τελειώσει.

Υλοποίηση openMP

Για την υλοποίηση του αλγόριθμου με openMP, χρησιμοποιήθηκε η επαναληπτική έκδοση του αλγορίθμου που μας δόθηκε, διότι χάρει στην οδηγία διαμοιρασμού εργασίας για επαναληπτικές μεθόδους for που έχει το πρότυπο openMP, η υλοποίηση της επαναληπτικής έκδοσης του αλγορίθμου είναι πολύ απλή σε σχέση με την αναδρομική έκδοση.

Για την υλοποίηση της μεθόδου χρησιμοποιήθηκε η ήδη υλοποιημένη συνάρτηση impBitonicSort και η παράλληλη υλοποίηση βρίσκεται στη συνάρτηση ompBitonicSort, η οποία καλείται κατευθείαν από τη συνάρτηση main.

Η παραλληλοποίηση του αλγόριθμου εφαρμόστηκε στον 3ο κόμβο επανάληψης for. Αυτό γίνεται προσθέτοντάς την οδηγία που θέλουμε ακριβώς πριν την έναρξη του κόμβου επανάληψης που θέλουμε να παραλληλοποιήσουμε, στην περίπτωση μας, αμέσως πριν το 3ο for.

Αυτό πραγματοποιήθηκε προσθέτοντας την οδηγία-εντολή:

#pragma omp parallel for num_threads(numOfThreads)

Όπου **numOfThreads =** αριθμός νημάτων = **N_threads**

Μετά από δοκιμές καταλήξαμε στην εντολή:

#pragma omp parallel for schedule(dynamic, chuck)

Όπου **chuck = N/N_threads**

με την οποία εντολή καταφέραμε να πετύχουμε ακόμη καλύτερους χρόνους εκτέλεσης.

Ο λόγος είναι διότι με την 1η οδηγία, ορίζουμε ένα συγκεκριμένο αριθμό νημάτων που μπορούν να χρησιμοποιηθούν. Εναλλακτικά, με την παράμετρο της 2ης οδηγίας, οι επαναλήψεις χωρίζονται σε μέρη μεγέθους chunk. Κάθε φορά που ένα νήμα ολοκληρώνει την εκτέλεση του μέρους που του έχει ανατεθεί, του ανατίθεται δυναμικά ένα άλλο μέρος. Έτσι γίνεται καλύτερη κατανομή εργασιών με αποτέλεσμα να πετυχαίνουμε καλύτερους χρόνους εκτέλεσης.

Στην περίπτωση της 2ης εντολής, χρειάστηκε να ορίσουμε το μέγιστο πλήθος των νημάτων που θα δημιουργηθούν, εισάγωντας στην αρχή της συνάρτησης την εντολή:

omp_set_num_threads(N_threads);

Τέλος, ο λόγος που επιλέχθηκε η 3η for για παραλληλοποίηση είναι επειδή ήταν η μόνη που στο εσωτερικό της είχε διάφορες εκτελέσεις πράξεων οι οποίες απαιτούν κάποιο χρόνο και θα μπορούσαν να εκτελεστούν παράλληλα. Αν επιλέγαμε να παραλληλοποιήσουμε κάποιον από τους άλλους 2 κόμβους επανάληψης, τότε υπήρχε ο κίνδυνος λάθους εγγραφής δεδομένων.

Πράγματι, όταν τρέξαμε δοκιμαστικά τον αλγόριθμο, παρατηρήσαμε μείωση του χρόνου εκτέλεσης σε σχέση με την αντίστοιχη σειριακή έκδοση.

Υλοποίηση pthreads & Quick Sort

Μετά από διάφορες δοκιμές και αφού τρέξαμε τους αλγόριθμους αρκετές φορές, παρατηρήθηκε ότι η σειριακή ταξινόμηση Quick Sort είχε σχετικά μικρούς χρόνους εκτέλεσης. Συγκριτικά με την Bitonic Sort είχε πάντα μικρότερους χρόνους εκτέλεσης και από την αναδρομική υλοποίηση και από την επαναληπτική. Ειδικότερα, όταν το μήκος του πίνακα ήταν σχετικά μικρό ή είχαμε μικρό αριθμό διαθέσιμων νημάτων, οι χρόνοι εκτέλεσης της Quick Sort ήταν κοντά στους χρόνους εκτέλεσης των παράλληλων υλοποιήσεων. Αυτό μας προβλημάτισε αρκετά, οπότε σκεφτήκαμε ότι αν συνδυάζαμε την Quick Sort μαζί με μια παράλληλη υλοποίηση ίσως να μικραίναμε ακόμη περισσότερο τους χρόνους εκτέλεσης του αλγορίθμου. Εξάλλου, στην εκφώνηση της άσκησης, ο συνδυασμός της Quick Sort μαζί με μια παράλληλη υλοποίηση προτείνεται στα Extra Credit.

Στην εκφώνηση της άσκησης προτείνεται η συνδυασμένη χρήση της συνάρτησης qsort, για την κατάταξη μικρού μεγέθους υποπινάκων. Αφού δοκιμάστηκε η υλοποίηση pthreads μαζί με Quick Sort, παρατηρήθηκε ότι όντως ο χρόνος εκτέλεσης μειώθηκε, άλλα όχι ικανοποιητικά.

Αργότερα, δοκιμάσαμε να χρησιμοποιήσουμε την qsort στη συνάρτηση pthBitonicSort εκεί στο σημείο που όταν τελειώσουν τα διαθέσιμα threads εκτελείται η σειριακή έκδοση της Bitonic Sort. Αντί λοιπόν να καλούμε την recBitonicSort, αντικαταστήσαμε τη συνάρτηση με την qsort. Με αυτό τον τρόπο πετύχαμε ακόμη μικρότερους χρόνους εκτέλεσης και είδαμε ικανοποιητική βελτίωση. Ειδικότερα, έχουμε μεγάλη βελτίωση όταν ο αριθμός των νημάτων είναι μικρός ή/και όταν το μέγεθος του πίνακα είναι μεγάλο. Η βελτίωση αυτή θα φανεί και στα διαγράμματα που θα ακολουθήσουν στις επόμενες σελίδες.

Αποτελέσματα – Παρατηρήσεις – Συμπεράσματα

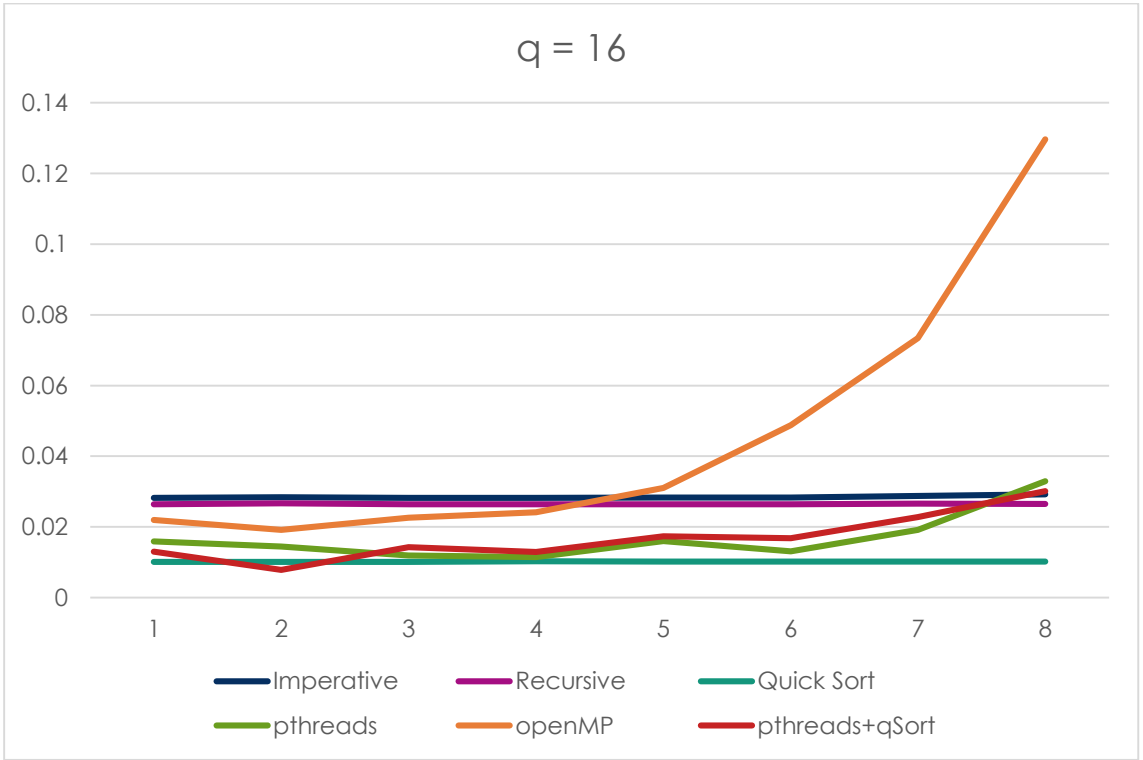
Όπως αναφέρθηκε και πιο πάνω, καθ' όλη τη διάρκεια των εκτελέσεων, μέσω της συνάρτησης fileWrite, αποθηκεύονταν οι χρόνοι εκτέλεσης κάθε αλγόριθμου σε αρχείο txt, ένα για την κάθε υλοποίηση. Αφού συλλέξαμε όλους τους χρόνους, τους περάσαμε στο πρόγραμμα MS Office Excel (στο αρχείο «Παράρτημα Α – Χρόνοι Εκτέλεσης») και σχηματίσαμε διάφορα διαγράμματα για να είναι ευκολότερη η μελέτη της απόδοσης της κάθε υλοποίησης αλλά και να μπορούμε να τις συγκρίνουμε.

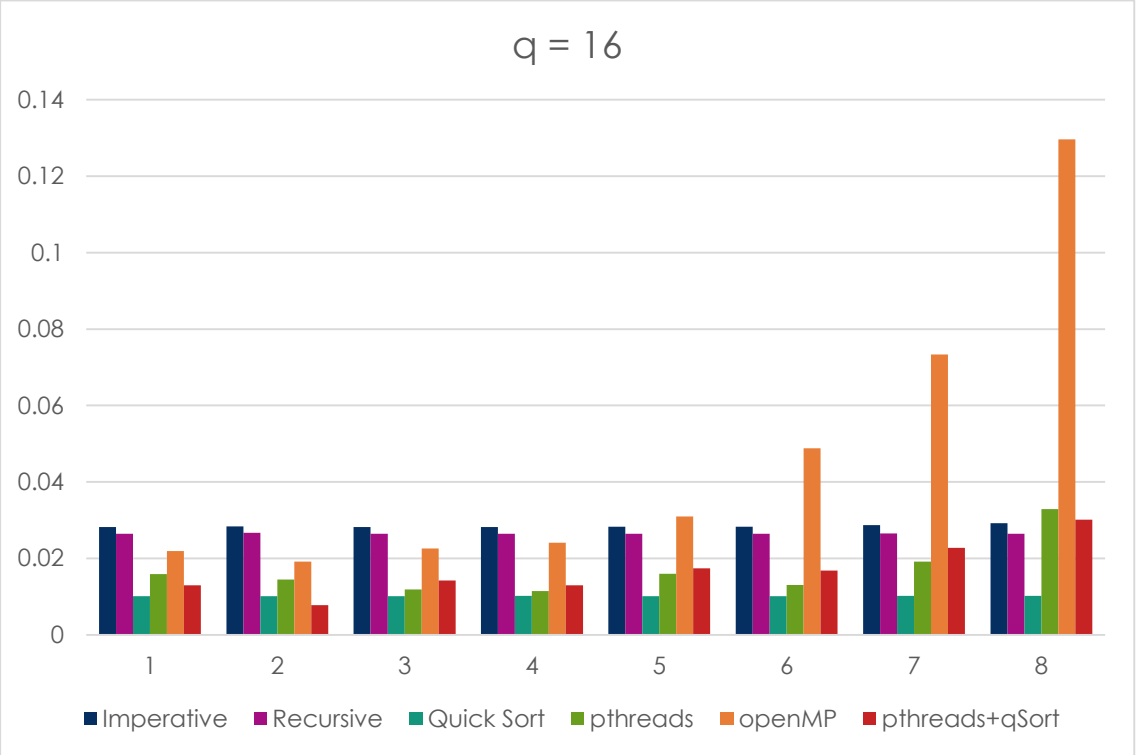
Για το κάθε διάγραμμα, κρατούσαμε σταθερό το N το οποίο αντιπροσωπεύει το μήκος του πίνακα και μεταβάλλαμε τον αριθμό των νημάτων, ο οποίος φαίνεται στον οριζόντιο άξονα. Στον κατακόρυφο άξονα τοποθετήθηκε ο χρόνος εκτέλεσης.

Κάποια διαγράμματα δεν παρουσιάζονται εδώ, για λόγους οικονομίας χώρου, ωστόσο βρίσκονται όλα αποθηκευμένα στο αρχείο «Παράρτημα Β – Διαγράμματα.pdf».

Όπως αναφέραμε και στην αρχή, όλες οι μετρήσεις πάρθηκαν στο σύστημα diades.

Στην αρχή, τρέχουμε το πρόγραμμα μας κρατώντας το μήκος του πίνακα σταθερό και ίσο με 16, αλλάζοντας τον αριθμό των νημάτων, δηλαδή το N_threads = 2^q, μεταβάλλοντας την μεταβλητή q από 1 μέχρι 8. Το διάγραμμα που προκύπτει φαίνεται παρακάτω:





Όπως φαίνεται από τα παραπάνω διαγράμματα, οι σειριακές υλοποιήσεις είναι αρκετά αργές και δεν επηρεάζονται από τον αριθμό των νημάτων, πράγμα το οποίο ήταν αναμενόμενο.

Ακολούθως παρατηρούμε ότι η Quick Sort είναι πολύ γρήγορη και μάλιστα πιο γρήγορη και από τις παράλληλες υλοποιήσεις.

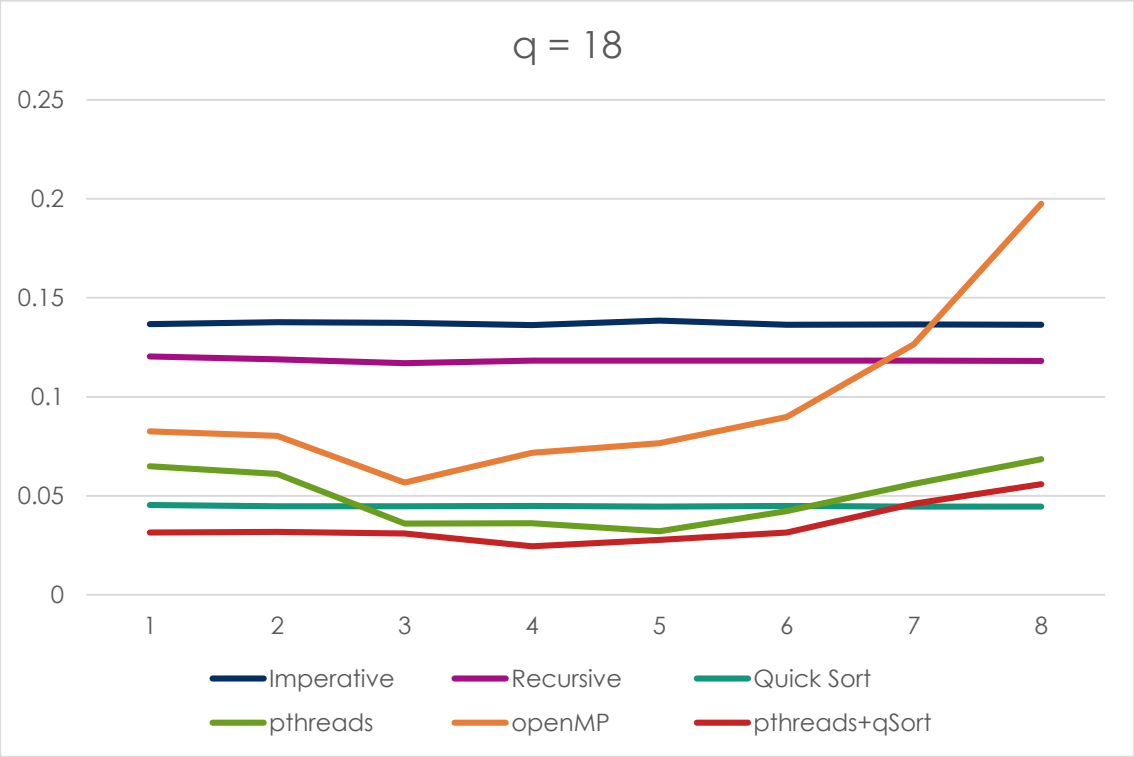
Για την pthreads, βλέπουμε ότι για 2^1 νήματα είναι γρηγορότερη από την αντίστοιχη σειριακή, παρόλα αυτά η quick sort είναι γρηγορότερη. Αυτό οφείλεται στο μικρό μέγεθος του πίνακα και στα λίγα νήματα που χρησιμοποιούνται, οπότε ο αλγόριθμος εισέρχεται σε σειριακή λειτουργία πολύ σύντομα και δεν βλέπουμε μεγάλη βελτίωση στο χρόνο εκτέλεσης συγκριτικά με την αναδρομική σειριακή έκδοση. Καθώς αυξάνουμε τον αριθμό των νημάτων ελάχιστα, βελτιώνεται ο χρόνος εκτέλεσης της pthreads, παρόλα αυτά η quick sort παραμένει πιο γρήγορη. Αν αυξήσουμε ακόμη περισσότερο τον αριθμό των νημάτων, βλέπουμε πως ο χρόνος εκτέλεσης της pthreads αυξάνεται. Αυτό οφείλεται στο ότι η ενεργοποίηση των νημάτων απαιτεί κάποιο χρόνο, χρόνος ο οποίος κοστίζει στον τελικό χρόνο εκτέλεσης και σε μικρής τάξης προβλήματα όπως εδώ, αυτός ο χρόνος γίνεται αισθητός.

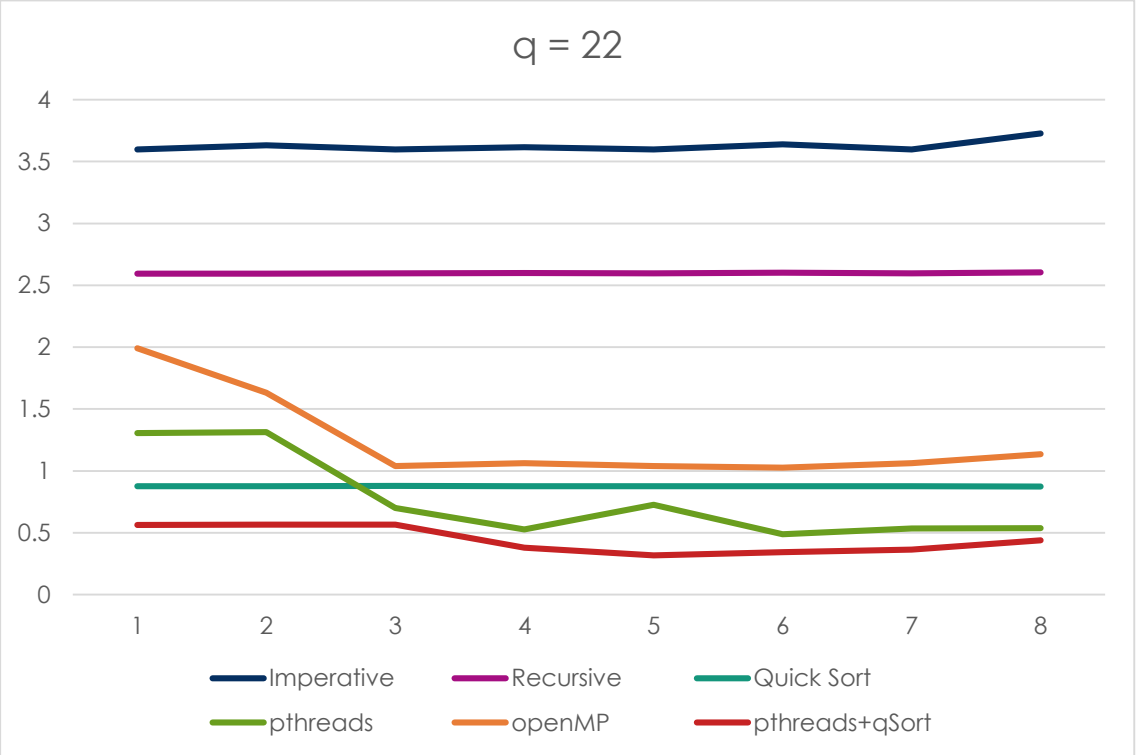
Σχετικά με την υλοποίηση openMP, παρατηρούμε χειροτέρευση στον χρόνο εκτέλεσης καθώς αυξάνουμε τον αριθμό των νημάτων, με αποτέλεσμα, όταν έχουμε μεγάλο αριθμό νημάτων, η openMP να έχει χρόνους μεγαλύτερους και από τις σειριακές υλοποιήσεις. Ο λόγος, όπως εξηγήσαμε και για την pthreads, είναι ο χρόνος που απαιτείται για την έναρξη όλων αυτών των νημάτων ενώ η τάξη μεγέθους του προβλήματος είναι μικρή.

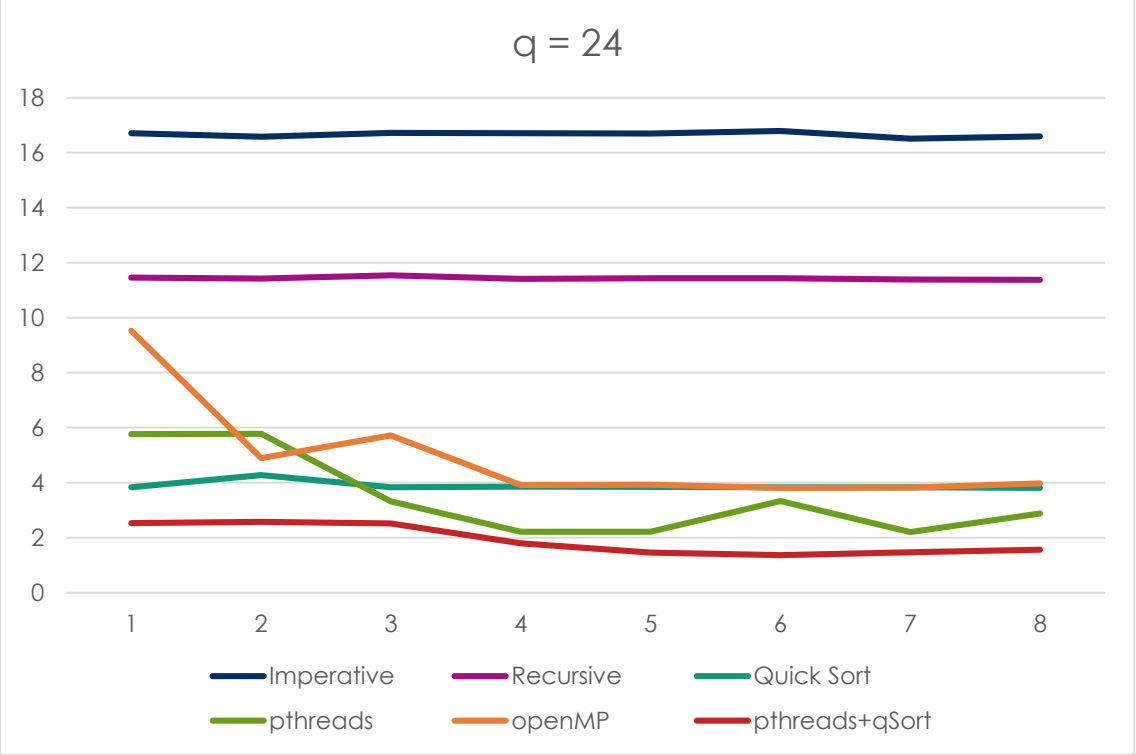
Τέλος, για τον συνδυασμό pthreads με quick sort παρατηρούμε ότι οι χρόνοι εκτέλεσης, σε ορισμένες περιπτώσεις, είναι κοντά στους χρόνους εκτέλεσης της pthreads και σε ορισμένες περιπτώσεις είναι μεγαλύτεροι και σε άλλες μικρότεροι.

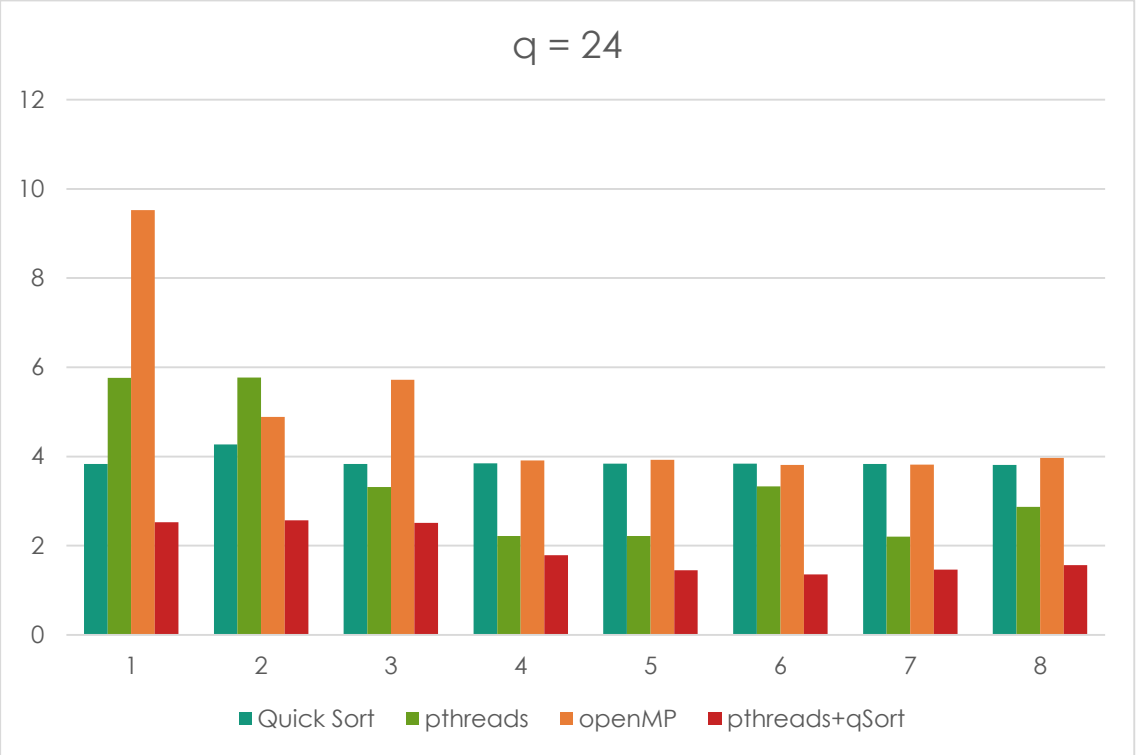
Επειδή όμως το μέγεθος του πίνακα είναι αρκετά μικρό για να βγάλουμε συμπεράσματα και η διαφορά στους χρόνους εκτέλεσης των διαφόρων υλοποιήσεων δεν γίνεται αισθητή, θα μεταπηδήσουμε σε μεγαλύτερα μεγέθη πινάκων παρακάμπτοντας κάποια διαγράμματα. Ωστόσο, όπως αναφέρθηκε και πριν, τα διαγράμματα αυτά υπάρχουν μαζεμένα στο Παράρτημα Β που βρίσκεται στο αρχείο «Παράρτημα Β – Διαγράμματα.pdf».

Ακολούθως θα δούμε τα διαγράμματα για q = 18, 20, 24.




q = 22
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
Imperative
Recursive
Quick Sort
pthreads
openMP
pthreads+qSort


q = 24
18
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
Imperative
Recursive
Quick Sort
pthreads
openMP
pthreads+qSort



Από τα διαγράμματα των σελίδων 10-11, βλέπουμε ξεκάθαρα μια σημαντική βελτίωση του χρόνου εκτέλεσης, χρησιμοποιώντας παράλληλο προγραμματισμό έναντι του σειριακού. Αυτό, γίνεται ακόμη πιο αισθητό όταν αυξάνουμε το μέγεθος του πίνακα, αλλά και καθώς αυξάνουμε τον αριθμό τον διαθέσιμων νημάτων, αφού όπως βλέπουμε, η διαφορά στους χρόνους των σειριακών υλοποιήσεων με τις παράλληλες υλοποιήσεις, όλο και γίνεται μεγαλύτερη.

Στο τελευταίο διάγραμμα, αφαιρέθηκε η αναδρομική σειριακή υλοποίηση καθώς και η επαναληπτική σειριακή υλοποίηση, ούτως ώστε να γίνει καλύτερη σύγκριση των παράλληλων μεθόδων καθώς και της σειριακής quick sort.

Αξίζει να σημειωθεί, ότι η υλοποίηση με το πρότυπο openMP για 2^1 νήματα, σχεδόν για όλα τα μεγέθη πίνακα, έχει τους μεγαλύτερους χρόνους εκτέλεσης από τις υπόλοιπες παράλληλες υλοποιήσεις. Καθώς όμως αυξάνεται ο αριθμός των νημάτων, βελτιώνεται κατά πολύ ο χρόνος εκτέλεσης, παρόλα αυτά, οι χρόνοι της είναι κοντά στους χρόνους της quick sort. Αυτή η βελτίωση παρατηρείται και καθώς αυξάνουμε το μέγεθος του πίνακα.

Εν ολίγοις, η υλοποίηση με openMP αλλά και pthreads, φαίνεται να βελτιώνονται και να αποκτούν μεγαλύτερο νόημα καθώς μεγαλώνουμε το μέγεθος του πίνακα ή/και αυξάνουμε τον αριθμό των νημάτων.

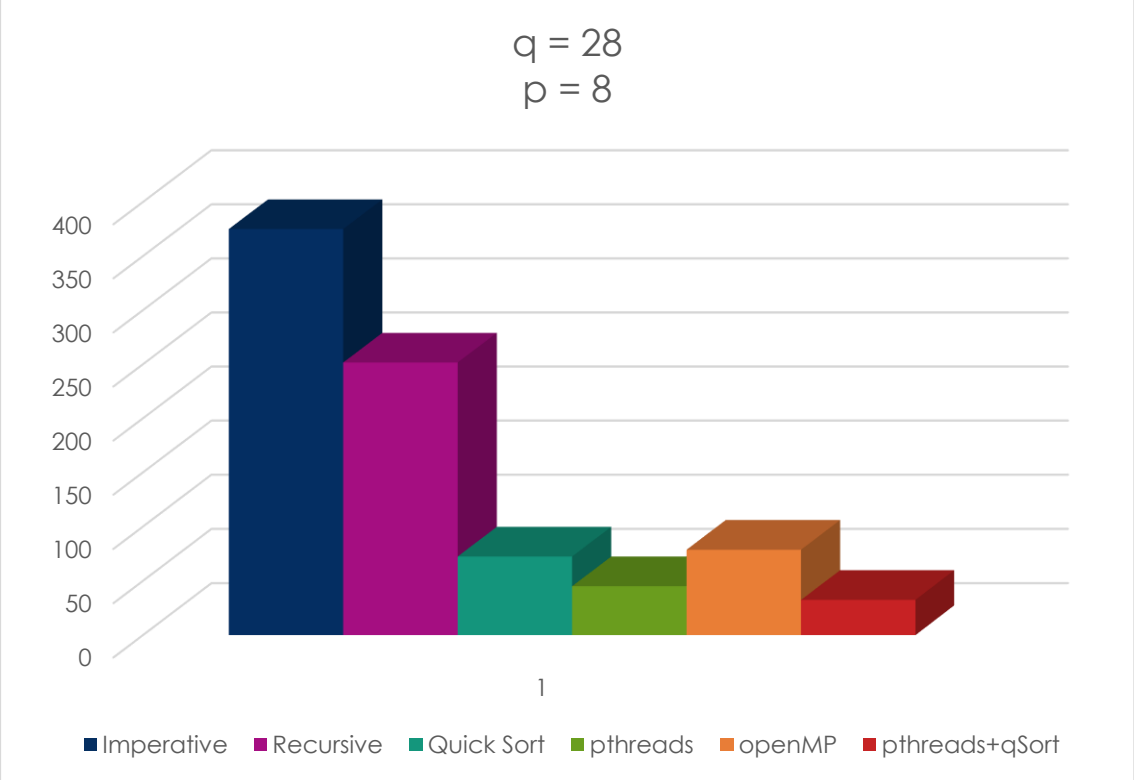
Η quick sort, όσο αυξάνουμε το μέγεθος του πίνακα αλλά και τον αριθμό των νημάτων, φαίνεται να χάνει έδαφος, αφού όσο αυξάνονται αυτά, γίνεται πιο αργή από τις παράλληλες υλοποιήσεις. Αν τρέχαμε τους αλγόριθμους για πίνακες με περισσότερα από 2^{24} στοιχεία αλλά και περισσότερα νήματα, θα βλέπαμε μεγαλύτερη διαφορά της quick sort από τις υπόλοιπες παράλληλες υλοποιήσεις.

Ο συνδυασμός της quick sort με pthreads, φαίνεται να αποδίδει καλά, αφού σχεδόν σε όλες τις περιπτώσεις, έχει τους καλύτερους χρόνους εκτέλεσης, και σε ορισμένες περιπτώσεις οι χρόνοι αυτής της υλοποίησης είναι κοντά στους χρόνους της υλοποίησης pthreads. Όπως και στις υπόλοιπες παράλληλες υλοποιήσεις, βλέπουμε ότι οι χρόνοι εκτέλεσης βελτιώνονται καθώς αυξάνουμε το μέγεθος του πίνακα. Επίσης στα 2 τελευταία διαγράμματα, βλέπουμε ξεκάθαρα ότι η υλοποίηση αυτή, έχει τους καλυτέρους χρόνους εκτέλεσης από όλες τις υπόλοιπες υλοποιήσεις.

Επίσης, να σημειωθεί ότι η συνάρτηση test, η οποία χρησιμοποιήθηκε για τον έλεγχο τον αποτελεσμάτων του κάθε αλγόριθμου, επέστρεψε σε όλες τις περιπτώσεις το μήνυμα ‘‘PASSed’’, δηλαδή ότι η ταξινόμηση πραγματοποιήθηκε με επιτυχία.

Τέλος, τρέξαμε τους αλγόριθμους για μεγαλύτερο μήκος πίνακα, ίσο με 2^{28} και νήματα 2^8 για να πάρουμε μια γεύση του τι γίνεται όταν έχουμε μεγαλύτερης τάξης προβλήματα.

Το διάγραμμα που δημιουργήθηκε είναι το παρακάτω.



Από το διάγραμμα αυτό, παρατηρούμε μια σημαντική βελτίωση στους χρόνους εκτέλεσης, όταν η εκτέλεση του αλγόριθμου γίνεται παράλληλα και όχι σειριακά. Η βελτίωση αυτή, είναι μεγαλύτερη από το 50% του χρόνου εκτέλεσης των σειριακών υλοποιήσεων.

Επίσης, βλέπουμε ότι η παράλληλη υλοποίηση με το πρότυπο openMP, έχει χρόνους εκτέλεσης κοντά στην quick sort. **Θα ήταν καλύτερα αν ενεργοποιούσαμε περισσότερα νήματα διότι το μέγεθος του πίνακα σε αυτή την περίπτωση είναι αρκετά μεγάλο**.

Η υλοποίηση pthreads, όπως φαίνεται και από το διάγραμμα, τα πήγε πολύ καλά έχοντας καλύτερο χρόνο από την υλοποίηση openMP και όλες τις σειριακές.

Τέλος, σε αυτό που ίσως να αξίζει να σταθούμε, είναι η υλοποίηση threads με συνδυασμό quick Sort. Για άλλη μια φορά, έχει τον μικρότερο χρόνο από όλες τις υπόλοιπες υλοποιήσεις, καθιστώντας την ως την πιο γρήγορη παράλληλη υλοποίηση.

Περί diades…. Αξίζει να αναφέρουμε ότι το κάθε σύστημα συμπεριφέρεται διαφορετικά. Αυτό παρατηρήθηκε και από το σύστημα diades σε σχέση με το δικό μας σύστημα. Στο κάθε σύστημα, οι χρόνοι ήταν διαφορετικοί από τους αντίστοιχους χρόνους του άλλου συστήματος. Αυτό, είναι απόλυτα λογικό. Επίσης, σε ώρες αιχμής που πιθανώς κάποιοι συνάδελφοι να έπερναν μετρήσεις στο σύστημα diades, παρατηρήθηκε διαφορετική συμπεριφορά των αλγορίθμων από την συμπεριφορά που παρατηρήθηκε στο τοπικό μας σύστημα, έτσι οι μετρήσεις πάρθηκαν ξανά σε άλλη χρονική στιγμή που κρίθηκε ιδανικότερη.

Παρατηρήθηκε ότι υλοποίηση pthreads+qsort οι χρόνοι εκτέλεσης της και στα 2 συστήματα ήταν πανομοιότυποι παρόλο που οι υπόλοιπες υλοποιήσεις είχαν σχετικά μεγάλες διαφορές. Αυτός ίσως να είναι ένας λόγος που θα μπορούσαμε να χαρακτηρίσουμε τη μέθοδο αυτή ως την πιο αξιόπιστη και πιο σταθερή.

Για τις παρατηρήσεις και τα συμπεράσματα, βοηθητικό είναι το Παράρτημα Α το οποίο εμπεριέχει όλους τους χρόνους εκτέλεσης για p = [1 : 8] και q = [16 : 24] οι οποίοι πάρθηκαν από το σύστημα diades αλλά και από τον προσωπικό μας υπολογιστή. Το παράρτημα Α βρίσκεται στο αρχείο «Παράρτημα Α – χρόνοι εκτέλεσης» το οποίο παραδίδεται μαζί με τα υπόλοιπα αρχεία της εργασίας.

Σύνοψη

Μετά από όλες αυτές τις δοκιμές που κάναμε, τις μετρήσεις που πήραμε, αλλά και τα διαγράμματα που δημιουργήσαμε, καταλήξαμε ότι η παράλληλη υλοποίηση pthreads με συνδυασμό quick sort είναι η γρηγορότερη παράλληλη υλοποίηση από τις υπόλοιπες που υλοποιήσαμε. Αυτό όμως, δεν την καθιστά γρηγορότερη μέθοδο ταξινόμησης για όλα τα μεγέθη πίνακα, αφού όπως είδαμε για μικρούς πίνακες η σειριακή ταξινόμηση με την μέθοδο quick sort απαιτούσε μικρότερο χρόνο εκτέλεσης.

Η υλοποίηση με το πρότυπο openMP, αποδίδει καλύτερα όταν έχουμε όσο πιο πολλά διαθέσιμα νήματα αλλά και όταν ο πίνακας είναι σχετικά μεγάλος. Όπως είδαμε, για πολλά νήματα αλλά μικρό μέγεθος πίνακα, η υλοποίηση openMP είχε μεγάλους χρόνους εκτέλεσης.

Η υλοποίηση pthreads, αποδίδει καλύτερα για μεγάλους πίνακες αλλά όπως και στην openMP, αποδίδει καλύτερα αν έχουμε περισσότερα νήματα, με την προϋπόθεση ότι ο πίνακας είναι μεγάλος. Όπως είδαμε και από το 1ο διάγραμμα της σελίδας 8, για μικρούς πίνακες και μεγάλο αριθμό νημάτων, η pthreads όπως και η openMP, είναι σχετικά αργή διότι απαιτείται κάποιος χρόνος ενεργοποίησης των νημάτων, και για μικρού μεγέθους πίνακες όπου η ταξινόμηση γίνεται γρήγορα ακόμη και με μια σειριακή υλοποίηση, ο χρόνος αυτός (της ενεργοποίησης των νημάτων), καθυστερεί τον συνολικό χρόνο εκτέλεσης. Η pthreads επίσης, είχε γενικά καλύτερους χρόνους από την openMP.

Τέλος, η υλοποίηση pthreads με συνδυασμό quick sort, όπως αναφέραμε ξανά, είναι η γρηγορότερη μέθοδος στις περισσότερες περιπτώσεις. Ιδίως όταν ο πίνακας που έχουμε είναι μεγάλος, η υλοποίηση αυτή είναι η γρηγορότερη από τις υπόλοιπες υλοποιήσεις, ανεξάρτητα από τον αριθμό των threads.

Βλέποντας όλες αυτές τις παρατηρήσεις, καταλήγουμε στο συμπέρασμα ότι σε γενικές γραμμές η καλύτερη υλοποίηση μπορεί να θεωρηθεί η pthreads + quick sort. Παρόλα αυτά, όταν θα έρθει η στιγμή που θα πρέπει να επιλέξουμε κάποια μέθοδο για την επίλυση κάποιου προβλήματος, πριν κάνουμε την επιλογή μας, θα πρέπει να σκεφτούμε πολύ σοβαρά το μέγεθος του προβλήματος το οποίο έχουμε να αντιμετωπίσουμε αλλά και των αριθμό των νημάτων που θα έχουμε στη διάθεση μας, αφού όπως είδαμε, κάθε μέθοδος συμπεριφέρεται διαφορετικά ανάλογα με αυτές τις 2 παραμέτρους που αναφέραμε. Επομένως, ο παράλληλος προγραμματισμός δεν είναι πάντα και ο πιο βέλτιστος.